Robotica (2018) volume 36, pp. 141–166. © Cambridge University Press 2017
doi:10.1017/S0263574717000212

# Selection of trajectory parameters for dynamic pouring tasks based on exploitation-driven updates of local metamodels

Joshua D. Langsfeld†, Krishnanand N. Kaipa‡ and Satyandra K. Gupta*§

†*Maryland Robotics Center, Institute for Systems Research, University of Maryland, College Park, MD, USA. E-mail: jdlangs@umd.edu*
‡*Department of Mechanical and Aerospace Engineering, Old Dominion University, Norfolk, VA, USA. E-mail: kkaipa@odu.edu*
§*Center for Advanced Manufacturing, Department of Aerospace and Mechanical Engineering, University of Southern California, Los Angeles, CA, USA*

(Accepted April 1, 2017. First published online: May 8, 2017)

**SUMMARY**
We present an approach that allows a robot to generate trajectories to perform a set of instances of a task using few physical trials. Specifically, we address manipulation tasks which are highly challenging to simulate due to complex dynamics. Our approach allows a robot to create a model from initial exploratory experiments and subsequently improve it to find trajectory parameters to successfully perform a given task instance. First, in a *model generation* phase, local models are constructed in the vicinity of previously conducted experiments that explain both task function behavior and estimated divergence of the generated model from the true model when moving within the neighborhood of each experiment. Second, in an *exploitation-driven updating* phase, these generated models are used to guide parameter selection given a desired task outcome and the models are updated based on the actual outcome of the task execution. The local models are built within adaptively chosen neighborhoods, thereby allowing the algorithm to capture arbitrarily complex function landscapes. We first validate our approach by testing it on a synthetic non-linear function approximation problem, where we also analyze the benefit of the core approach features. We then show results with a physical robot performing a dynamic fluid pouring task. Real robot results reveal that the correct pouring parameters for a new pour volume can be learned quite rapidly, with a limited number of exploratory experiments.

KEYWORDS: Trajectory generation; Locally weighted learning; Adaptive function approximation; Robot dynamics; Robot pouring.

## 1. Introduction

Programming robots to perform real-world manipulation tasks is challenging and time consuming. For a large class of tasks, the robot arm must be provided with a precise trajectory to follow, prior to execution. For most currently deployed robots, these trajectories are defined manually by a skilled human operator. However, in the case of tasks with complex dynamics (e.g., manipulation of deformable materials and/or fluids), manual construction of analytical models that accurately capture the task dynamics is often not feasible, requiring a different method to generate the robots trajectory without resorting to trial and error on the part of the operator.

Such tasks often consist of a set of *instances*, with each instance defining a slightly different goal state and, therefore, the trajectory that the robot must follow. These different trajectories can

* Corresponding author. E-mail: guptask@usc.edu

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

be generated automatically, either through direct synthesis or by selecting the parameter values for a manually parameterized trajectory. In this paper, we focus on the latter method of trajectory generation. Many tasks have a natural parameterization that a human can specify which can cover the full range of the robots ability. For example, a ball throwing task can be specified by having the arm move in a circular arc with a small set of parameters such as the radius, speed, and release point fully controlling the throwing distance. We are interested in the problem of how the parameter values can be determined for each task instance automatically in order to generate the final trajectory to be executed by the robot.

A common approach taken to find the parameter values for a particular task instance is to use a model of the task and search for valid parameter values using the predictive ability of the model. To obtain the highest prediction accuracy, it is typically ideal to use a physics-based simulator that can capture the details of most, if not all, of the task dynamics. However, for complex tasks, such simulators are either unavailable or generally far too computationally expensive to be able to use for parameter selection in reasonably short time periods. This is especially true for tasks involving deformable materials and fluids, which would require full finite-element simulations. Instead of simulators, such tasks can be modeled with surrogate models created from the data that the robot collects as it gains experience while attempting the task. This framework requires the robot to go through a learning period where it may perform task instances incorrectly until it collects enough data to find valid parameter values for all the desired instances.

An important aspect of the types of tasks we address, which is reflected in our approach, is that many satisficing parameter values are available for a given task instance. We do not address the problem of finding the single optimum set of parameters or searching for valid parameters when they are clustered together in small regions of the space. Instead, our approach is designed for tasks where valid parameter sets are found throughout the space and it is sufficient to find a single one. The focus is to exploit the available knowledge to find satisficing parameters with as few task attempts as possible.

Our proposed approach has the goal of rapidly finding valid solutions in the parameter space corresponding to new task variations with sparse initial data. We model the task abstractly as a set of parameters existing in a finite-dimensional space where each point in the space defines a trajectory to perform a single task variation. First, in a *model generation* phase, local models are constructed in the vicinity of the previously conducted experiments that explain both the task function behavior and the estimated divergence of the generated model from the true model when moving within the neighborhood of each experiment. Second, in an *exploitation-driven updating* phase, these models are used to guide parameter selection given a desired task outcome and the models are updated based on the actual outcome of the task execution. Our approach exploits local information available in adaptively chosen neighborhoods effectively to incrementally build and update multiple local models, thereby allowing the algorithm to capture arbitrarily complex function landscapes.

We validate our approach by testing it both on synthetic non-linear functions and on a physical robot. For physical experiments, we consider a dynamic pouring task, in which the robot is tasked to pour a certain volume of liquid from a bottle it is holding into a container placed on a rotating turntable. Moving the bottle to track the moving container while simultaneously tilting it to pour creates complex fluid dynamics. This makes the mapping from pouring-trajectory parameters to poured volume highly non-linear and infeasible to model analytically. Our results reveal that the correct pouring parameters for a new pour volume can be learned quite rapidly, with a small number of exploratory experiments.

## 2. Related Work

### *2.1. Trajectory generation for robot manipulation*

There are a diversity of different approaches to specifying robot trajectories with complex model dynamics in the literature. We focus on illustrating past work involving robot manipulation tasks using model and trajectory parameterizations and approaches to generate trajectories that correctly perform specific tasks.

Many researchers have used trajectory optimization to find solutions for robot manipulation tasks, typically optimizing a given geometric path with or without a model of the task dynamics.[30,42]

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

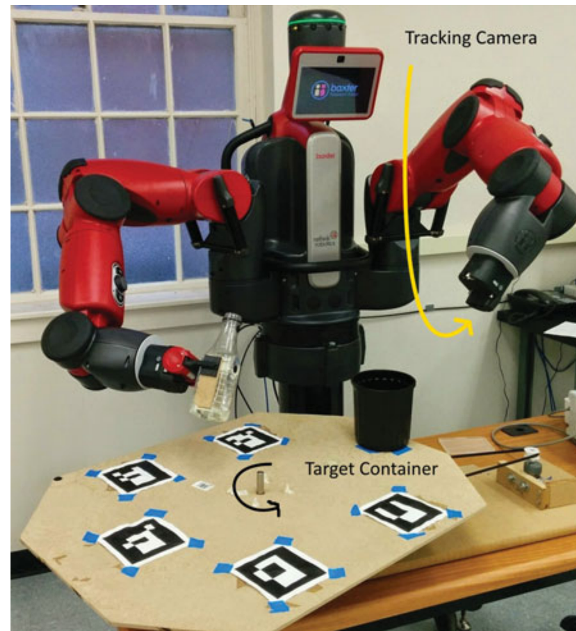


Fig. 1. Experimental setup used for the pouring task. The right hand holds the bottle and performs the pouring while the camera in the left hand monitors the visual markers on the table to measure its current position and speed.

Mordatch and Todorov demonstrated the use of trajectory optimization to help train a neural network-based manipulation control policy[27] for 3d reaching movements. However, in general, using a standard trajectory optimizer is quite expensive if the task dynamics are complex. This remains true with parallelizable optimization methods such as genetic algorithms.[2] Kim *et al.* [20] developed a trajectory generation method which is able to learn a model of the task dynamics and compensate for unobservable elements in simple object manipulation tasks. Their approach formulates the trajectory creation as a policy in the state space which is approximated by multiple local trajectory generators. A similar approach involves sequencing the solutions of low-dimensional optimization sub-problems.[31]

Posa and Tedrake address the problem of optimizing the planner parameters of a geometric path for robustness in manipulation tasks involving frictional contact[35] where the trajectory parameters are the contact reaction forces. Luo and Hauser provide an extension to transform the parameters of the trajectory into a fast linear programming optimization problem.[24] This allows the robot to perform tasks such as sliding a stack of blocks across a rough surface without any falling down.

Zhang *et al.* [43] used a sampling-based motion planner to find trajectories for a ball throwing task[1] by searching for valid intermediate dynamic states. They were able to iteratively adapted the planned trajectory to match expected behavior with relatively few attempts. Learning is also possible by storing full paths in a trajectory library[11] and adjusting the most appropriate path to work when given a new task instance,[7] or by having a planner compensate for different instances having learned the relevant invariant constraints.[9]

### 2.2. *Learning pouring tasks*

An initial result in pouring is from Akgun *et al.* [3] where a robot learns to scoop and pour quantities of coffee beans directly from human demonstrations. The robot can learn the task elements from a small number of demonstrations but does not generalize the task to changes in the environment. Pastor *et al.* [32], showed an initial result of learning a pouring task using Dynamic Motion Primitives (DMPs). Their system shows generalization in the pouring target location where the robot proceeds to empty its held container into the target container without spilling.

Later results used reinforcement learning techniques to pour either an entire bottle of fluid[41] or a target volume[28] while simultaneously learning to avoid spilling. Again, DMPs are used to encode the robot motions and the learning is done on the DMP parameters. The core contribution is using a

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

statistical analysis to reduce the dimensionality of the learning space, thereby greatly speeding up the learning rate. In simulation of a stationary pouring task, Nemec *et al.* [28] achieve no spillage after an average of 7–8 attempts, with a pouring error around 10 mL. The learned policy, however, cannot be used for different poured volumes.

Rozo *et al.* [38] demonstrate a force-based approach that enables the robot to pour volumes of fluid with different initial volumes inside the bottle being held. The pouring behavior is encoded with a parametric hidden Markov model. They successfully showed the robot could learn to pour a desired volume of liquid into multiple target containers starting from a new initial volume with just a single prior demonstration.

The work of Brandi, *et al.* [10] shows the generalization of a pouring task to new objects (fluid source and target) not seen in the initial training phase given by kinesthetic demonstrations. The robot can determine a correspondence between the new object and the training object by finding the warping between the object geometries. This warping can then be applied to the manipulator motion to successfully pour out a full quantity of fluid into a new object, but not specific volumes.

In summary, researchers have successfully been able to implement learning algorithms for specific pouring tasks but there are still open questions in terms of the algorithm abilities to generalize to new task instances. There does not appear to be existing work that addresses the problem of specifying a procedure for generating trajectories for multiple task instances, allowing for precisely pouring different volumes of fluid.

### *2.3. General learning methods*

The core of our approach makes use of a set of models that approximate the task dynamics. In the robotics community, two commonly used function approximation algorithms are Gaussian Process Regression (GPR) and Locally-Weighted Projection Regression (LWPR), an extension of locally weighted learning[6] for higher dimensions. A thorough overview of modeling algorithms for robotics can be found in ref. [29]. GPR is a standard approach for estimating a non-linear function from sparse samples and has been used successfully in many motion learning tasks. Whereas this approach requires the function to be learned globally in a batch process, we focus on developing an approach based on local models. More recent adaptations which are efficient enough for incremental learning[17] also have a global fitting behavior.

In contrast, LWPR provides a method of incrementally learning non-linear functions through the use of linear models that have local influence only. LWPR is non-parametric and generates new local models as needed, but works best with high volumes of data. Other approaches that focus on performing local learning include ref. [5], and more recently, refs. [18, 22, 34].

Reinforcement learning is often used to generate trajectories for robots as the system learns models, typically representing control policies[21] or a set of controllers.[4,16] Generally, the problem formulation is distinct in that data arrives in continuous state-action trajectories rather than discrete samples, but some of the modeling approaches are similar enough to take inspiration from. Common approaches involve learning by using policy gradients,[15,33] where the parameters of the policy model are adjusted locally. We implement similar local moves in our adjustments but test adjustments from all regions of the parameter space. In ref. [19], an approach similar to ours is used where the behavior of a continuous state-action pair space is assumed to be approximated by local models surrounding a finite set of samples. They prove bounds on the optimality of such a model but the problem of defining the ideal finite covering set remains specific to the application.

In ref. [25], a hybrid approach is used where an agent undergoing reinforcement learning can request relocation to a different region of the state space and can actively select the most promising state to reduce its task cost. In a sense, our approach includes this feature, but we expand it by continually evaluating the benefit of relocation to any previous point for each task trial.

When faced with the constraint of real-time task learning, a learning algorithm generally must make some tradeoff between model exploration and exploitation.[23] Several learning approaches rely on exploratory movements to find the model structure prior to direct learning, even using a different experimental platform than the final learning one.[8] Examples where exploration is used heavily during the learning process include finding a helicopter control policy,[26] during reinforcement learning of movement primitives,[13] and in developing an initial sensor model.[12] In contrast, ref. [37] shows a maximum exploitation strategy to learn additional material properties from what information is already available, which is similar to the approach we present.

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

## 3. Problem Formulation

### *3.1. Problem statement*

In this paper, we utilize a framework where a robot performs a task by executing a finite-length trajectory $\tau$. The task is characterized by an unknown dynamics function that produces a real-valued output, or score, for a particular trajectory:

$$y = f(\tau), \quad y \in \mathbb{R}. \tag{1}$$

Note that this task function is different from typical robot dynamics functions that only map the current state to either a future state, or the derivative of the state. Here, the task output is a function of the entire trajectory.

Rather than learn the trajectory directly, we assume the trajectory to be the output of a task-specific planner $\pi$ with an input of a parameter vector $\mathbf{x}$ of length $p$. We assume that the structure of the planner can be defined manually, leaving the detail of producing the correct task output to the learning algorithm, which will find the corresponding parameter values. For example, in a task such as throwing a ball at a certain distance, a planner might be specified to move the robot end-effector in a circular arc. This trajectory may comprise parameters like radius, speed, and release point, the values of which determine the final throwing distance. This method can enable faster task learning by combining an intelligent human task parameterization with an automated learning algorithm. The algorithm then operates on the direct mapping from task parameters to outputs, which is the composition of the planner with the task dynamics function:

$$\tau = \pi(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^p$$
$$y = h(\mathbf{x}) = (f \circ \pi)(\mathbf{x}) = f(\tau)$$

The overall task itself is represented by a set of task *instances*, each describing a specific desired task output value $T_i$. Each task instance has a corresponding non-negative cost function $C_i$ and tolerance amount $\delta_i$ such that, if $y_i$ is a successful output of the $i$th instance, then $C_i(y_i) < \delta_i$. For the remainder of the paper, we consider the simple cost function $C_i(y) = |y - T_i|$. When the robot is assigned a specific task instance, it conducts *attempts* to solve the instance. Each attempt consists of selecting the parameters, executing the resulting trajectory, and measuring the resulting cost function.

The problem addressed here can be formulated as follows: Specify a computational procedure $\Psi$ that the robot will execute to solve a task instance, given the knowledge acquired from previous instances and attempts. The procedure consists of an evaluation loop where a candidate parameter vector is determined from $m$ previous attempts and the corresponding trajectory is executed. The previous attempts ($\mathcal{X}_m = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$, $\mathcal{Y}_m = \{y^{(1)}, \ldots, y^{(m)}\}$) may come from previous instances, previous attempts of the current instance, or prior knowledge obtained elsewhere such as random samples. If the corresponding task output for the candidate is successful, the procedure terminates. Otherwise, the procedure will iterate with a new candidate selected after adding the previous attempt to $\mathcal{X}_m$ and $\mathcal{Y}_m$.

$$\mathbf{x}_{m+1} \leftarrow \Psi(T_i, \mathcal{X}_m, \mathcal{Y}_m) \tag{2}$$

In general, the user will assign not just a single task instance during the robot training, but will be interested to have the robot learn a set of many different instances. We assume that such a set is not known beforehand and may possibly not even have a known final size when the robot begins to learn the first instances. Without knowledge of future instances to learn, the system focuses on each instance separately, with each one being solved before addressing the next.

In this framework, we desire to find a procedure that can solve each instance with a minimal number of attempts. Therefore, we use the total number of attempts made to solve $N_t$ task instances as the overall cost metric $\mathcal{C}$:

$$\mathcal{C}(T) = \sum_{i=1}^{N_t} \eta_i, \tag{3}$$

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

where $T = \{T_1, \ldots, T_{N_t}\}$, and $\eta_i$ is the number of attempts taken to solve the $i$th instance. Since the task instances are solved independently, the ideal procedure can directly minimize each $\eta_i$ without regard for the other instances.

While the optimal procedure that produces $\eta_i = 1$ for all instances is impossible to achieve, we do aim to specify a procedure that can asymptotically reduce $\eta_i$ to 1, while simultaneously trying to minimize the number of attempts for the earlier instances. In practical robotics scenarios, it is important to explicitly address the cost of learning all task instances as $N_t$ must be small enough so that the complete learning is feasible and the costs for the early instances cannot be discounted.

### *3.2. Evaluation of existing approaches for application to trajectory parameter selection problem*

Before explaining our approach in detail, it is useful to briefly discuss how existing techniques can be applied to this particular problem and what are some of the problems they experience. We discuss two major alternatives for addressing our problem and provide illustrative comparative results for two representational algorithms.

*3.2.1. Reinforcement learning based policy search.* Our problem can be posed to work under the standard reinforcement learning framework used in robotics. The framework is composed of unknown system dynamics, which are influenced by a controller, whose output is determined by a parameterized control policy.

$$
\begin{aligned}
s_{t+1} &= f(s_t, u_t) \\
u_t &= \pi_{\mathbf{x}}(s_t)
\end{aligned}
\tag{4}
$$

A task cost function $J(\mathbf{x})$ is defined for the policy parameters, which is evaluated over one or more trajectories (i.e., $\tau = \{s_1, \ldots, s_{t_f}\}$) followed by the system with a controller executing the policy. The goal of any learning algorithm is to find the set of policy parameters $\mathbf{x}^*$ that minimize

$$
J(\mathbf{x}) = \sum_{i=1}^{t_f} c(s_i),
\tag{5}
$$

where $c$ is a user-specified cost function that penalizes the current state of the system. Typical reinforcement learning problems involve specifying non-zero costs or rewards in only a subset of the state space rather than uniformly, and the goal of the learning algorithm is to find policy parameters that minimize the expected total costs of the states the system will be in.

While superficially different, the formulation of our problem in the previous section can be considered a special case of the general RL framework. The state space is taken as simply the task score itself, $y \in \mathbb{R}$. Instead of a multi-step trajectory determined by the system dynamics, we can consider just a single step from an initial state to the final output. The control policy is taken to output the task parameters themselves. Dropping the initial state from the model, as it is always the same, these simplifications reduce the general system dynamics in Eq. (4) to match Eq. (1) in section 3.1.

*3.2.2. Adapting the Probabilistic Inference for Learning Control (PILCO) Algorithm.* The PILCO algorithm[14] provides an approach to simultaneously learn the system dynamics and find preferred policy parameters for the general reinforcement learning framework. The core of the algorithm is to approximate $f(x, u)$ with a Gaussian process regression model, where the current state and control input are used as model inputs and the future state (or change in state) is output as a prediction. The GP outputs a probability distribution for its output, which can be propagated through the system dynamics in Eq. (4). This allows the algorithm to compute an expression for the expected cost of following a particular policy, with the current knowledge of the system dynamics. To improve the policy, the derivative of the cost function with respect to the policy parameters can be directly computed, with slight assumptions on the structure of the policy and cost function. The derivative allows standard local optimization algorithms, such as the Broyden–Fletcher–Goldfarb–Shanno (BFGS) method, to converge to a local optimum for the policy parameters.

When adapted to our problem, the complex computations used in PILCO for the general case are dramatically simplified. With only a single state update, the cost function only depends on a single

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

Table I. Comparison results with the adapted PILCO algorithm.

| Algorithm | Mean # iters | % within 10 iters |
|---|---|---|
| Adapted PILCO | 12.7 | 48% |
| Local linear models | 5.6 | 90% |

output for the GP, which is a normal distribution. The expected value of this output is therefore just the mean of the prediction and using a simple squared-error state cost function $c(y) = (y - y^*)^2$, the policy cost reduces to

$$J(\mathbf{x}) = (\mu(\mathbf{x}) - y^*)^2. \tag{6}$$

Here, $\mu(\mathbf{x})$ is the mean of the Gaussian process prediction of the task score $y$. Computing the derivative of $J(\mathbf{x})$ is then straightforward, given a smooth choice of covariance function for the GP.

This adapted formulation of the PILCO approach was compared against our current method of searching for task parameters, using a set of local linear models, on a synthetic five-dimension non-linear test function (see Section 5 for a full description). For both experiments, the same initial set of 50 random task parameter samples and their corresponding outputs were given to the learning algorithms. One hundred target outputs, uniformly distributed over the range of the test function, were given individually to the algorithms, with the known data set reset to the initial 50 samples after each target was found. For the PILCO policy search, the initial parameters were set to the mean of the initial data set. Searching with linear models does not require any initialization.

Primarily due to local updating, the adapted PILCO policy search has noticeably worse performance than our approach using linear models search as seen in Table I, which shows the results of 100 different target searches. For PILCO, out of 100 targets attempted, 40 could still not be found after 20 samples of the test function, most likely indicating that the policy parameters became stuck in local minima, which is something our approach is not vulnerable to. To conclude the comparison, when the learning problem is formulated in the framework we have developed our approach for, existing reinforcement learning techniques, represented by PILCO, address the problem in a simplified manner. Reducing the problem to performing only local updates of the policy parameters leads to much worse performance than can be achieved by our proposed approach.

*3.2.3. Bayesian Optimization Framework.* An alternate approach to solving the problem is with an optimization framework. For a desired task score $y^*$, the same cost function for an attempt in the reinforcement learning can be used:

$$c(\mathbf{x}) = (y - y^*)^2 = (h(\mathbf{x}) - y^*)^2. \tag{7}$$

A search algorithm can then attempt to minimize this cost function to find a set of task parameters that provide the desired task score. The algorithm terminates when a minimum value within the task tolerance of zero is found.

A Bayesian optimization framework allows an algorithm to select the next best point to test in the space based on a current probability model which consists of a prior that is conditioned on information acquired during the search. A Gaussian process is a straightforward model that can be used in this Bayesian manner.

*3.2.4. GP-UCB Algorithm.* Given a probability model conditioned on known information about the task, the question arises of how to select the next point to test, which requires some tradeoff of exploration and exploitation. A simple algorithm that balances these factors is the Gaussian Process Upper Confidence Bound algorithm.[40] Essentially, given a set of candidate points $D$ where the next test point can be taken, the candidate selected is

$$\mathbf{x}_{m+1} = \underset{\mathbf{x} \in D}{\operatorname{argmax}}\, \mu_m(\mathbf{x}) + \beta_m^{1/2} \sigma_m(\mathbf{x}), \tag{8}$$

where $\mu_m(\mathbf{x})$ and $\sigma_m(\mathbf{x})$ are the mean and the standard deviation of the Gaussian process at $\mathbf{x}$ when conditioned on $m$ previous samples. The weighting parameter $\beta_m$ is set to $2\log(|D|m^2\pi^2/6\delta)$ for a

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

Table II. Comparison results with the GP-UCB algorithm.

| Algorithm | Mean # iters | % within 10 iters |
|---|---|---|
| GP-UCB | 22.1 | 30% |
| Local linear models | 2.0 | 95% |

user chosen value of $\delta$, which allows for provable bounds on the regret incurred during the optimization process. In practice, however, $\beta_m$ can be tuned for faster convergence, which is detailed in the previous citation.

This algorithm was adopted for our problem and cost function, flipping the signs in Eq. (8) for minimization. We also did not use a fixed candidate set $D$, but rather used Eq. (8) with an off-the-shelf optimization library which searched for the minimizing value within the valid parameter bounds. We discovered that our adaptation of GP-UCB was effective in quickly searching the parameter space but had substantial problems finding solutions with very high precision. This would happen because while the algorithm quickly found regions with zero-error parameters nearby, it had little incentive to search closeby to the newly tested points to lower the error further. This is due to the Gaussian process uncertainty decreasing dramatically in the region of the new test point, while other regions have much higher uncertainty, and are therefore weighted higher by the GP-UCB heuristic. To get decent performance with the search, we had to loosen the task tolerance considerably but this only improved the performance of our own approach as well, which was far better. This distinction is shown in Table II, which contains the results from 20 target searches while using a very high task tolerance value. Given this behavior, we concluded that using a standard optimization method was not ideal for this problem as (1) we were trying to find parameters where the objective function was exactly zero and not just minimal, and (2) information about where zeros may lie was being discarded by using a squared error term instead of retaining the sign.

*3.2.5. Observation.* To conclude this section, we observed that our particular problem has characteristics that make it challenging for existing methods that are broadly applicable. Since our task involves only a one-step update directly to the final score, the full power of reinforcement learning algorithms cannot be made use of, and they are reduced to simple local updates of the policy parameters. And because we are searching for parameters that are exactly zeros of a non-negative cost function, a Bayesian optimization algorithm is not ideal, as it moves to other more uncertain regions of the parameter spaces once it learns that a particular area is not going to go any lower than zero. These difficulties will be explicitly addressed by the features in our approach to greatly improve the learning performance for this particular type of problem.

### *3.3. Overview of approach*

This paper presents an approach and procedure to find solutions to new task instances by iteratively improving a model of the task dynamics function $h(\mathbf{x})$. Since the individual task instances are presented to the robot sequentially, and it is not known *a priori* what the total number will be, we adopt the overall minimization strategy of solving each instance with as few attempts as possible. However, to reduce the number of attempts for possible later task instances, the learning algorithm continually gains experience of the task dynamics in order to exploit available knowledge and find faster solutions.

An important property of tasks that can learned efficiently with this strategy is that many solutions exist in the parameter space for each task instance, and hence can be found easily. The fact that large solution sets exist throughout the space allows a strategy of selecting test points by adjusting a single parameter value. This helps to maximize the information gained through multiple attempts and find solutions rapidly. If the solution set is instead sparse and clustered into small regions, using single parameter adjustments is likely to perform worse, as it becomes less likely that a solution exists somewhere on the line obtained by allowing a single parameter to be free. Many robot tasks have natural parameterizations, however, that show this behavior of being able to find solutions by adjusting a single parameter only.

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

The general outline of the approach involves the following components. First, at each known sample point, a neighborhood is defined and used to compute a local linear model to approximate the function behavior. Second, an error model at each point is computed to approximate the region where the linear model is considered accurate. Candidate parameter adjustments which are expected to solve the task instance in one step are generated from all current sample points, and are compared to select the one with the highest confidence. The new point in the parameter space is sent to the robot for execution and if the resulting task output is not a success, the system repeats the full strategy with the new information.

## 4. Local Linear Metamodel-Based Approach

The primary guiding principle of the proposed approach is to iteratively search the task parameter space by creating local models of parametric neighborhood and finding a movement within a carefully chosen neighborhood that has the minimum uncertainty. First, in an initialization phase, a set of training samples is obtained from either human demonstrations or robot executions of randomly selected points in the parameter space. Each sample is a tuple comprising a parameter vector and its mapping to a task score. Second, in a *model generation* phase, the algorithm builds local models based on the current training set. Third, in an *exploitation-driven model updating* phase, the algorithm finds a new point by using the current set of local models, evaluates its task score (for example, in the context of the robot pouring task, this corresponds to the robot using the found tilt parameters to execute the task and measuring the poured amount), adds the newly found point to the training set, and updates the set of local models. The cycle of model generation and model exploitation repeats until a point is found whose task score is within the success tolerance of that of the desired task.

### *4.1. Initialization*

Let $\mathcal{S} = \{(\mathbf{x}^{(i)}, y^{(i)}) : \mathbf{x}^{(i)} \in \mathcal{X}, y^{(i)} \in \mathcal{Y}, i = 1, 2, \ldots, m\}$ be the initial set of training samples, where $\mathcal{X}$ is the set of sample points in the parameter space, $\mathbf{x}^{(i)} \in \mathbb{R}^n$ is the $i$th point in $\mathcal{X}$, $\mathcal{Y}$ is the set of corresponding task scores, $y^{(i)} \in \mathbb{R}$ is the $i$th task score in $\mathcal{Y}$, and $m$ is the number of training samples. Let $x_j^{(i)} \in \mathbb{R}$ represent the $j$th element of $\mathbf{x}^{(i)}$. Further, let $\mathbf{x} \in \mathbb{R}^n$ represent a general point in the parameter space.

### *4.2. Model generation*

This phase is achieved in three steps: (1) adaptive neighborhood selection, (2) planar model approximation, and (3) error-divergence model approximation.

*4.2.1. Adaptive neighborhood selection.* For each point $\mathbf{x}^{(i)} \in \mathcal{X}$, the algorithm builds a local linear model by selecting the points from $\mathcal{X}$ residing within an adaptive box-neighborhood $\mathcal{N}^{(i)} \subset \mathcal{X}$:

$$\mathcal{N}^{(i)}(k) = \{\mathbf{x}^{(j)} \in \mathcal{X} : ||\mathbf{x}^{(i)} - \mathbf{x}^{(j)}||_\infty \leq \delta_k^{(i)}\} \tag{9}$$

where $k$ is the number of neighbors in the neighborhood and $\delta_k^{(i)}$ is the neighborhood size, which is given by

$$\delta_k^{(i)} = \max_{\mathbf{x}^{(j)} \in \mathcal{N}^{(i)}(k)} ||\mathbf{x}^{(i)} - \mathbf{x}^{(j)}||_\infty \text{ s.t. } |\mathcal{N}^{(i)}(k)| = k \tag{10}$$

According to Eqs. (9) and (10), note that $\delta_k^{(i)}$ is assigned to the minimum possible size that results in $k$-nearest neighbors. Since the density of points in the data set can be highly variable, it is important to find a $\delta_k^{(i)}$ that results in a neighborhood with sufficient points to generate a relatively accurate local approximation, but not so many that the non-linear behavior of the underlying function deteriorates the approximation. This is done by using a leave-one-out cross-validation technique to estimate the optimal neighborhood size $\delta_{k^*}^{(i)}$. In particular, for each $\mathbf{x}^{(j)} \in \mathcal{N}^{(i)}(k)$, a plane is fit using least-squares on the set $\mathcal{N}^{(i)}(k)/\mathbf{x}^{(j)}$ and the linear-fit error at $\mathbf{x}^{(j)}$ is computed. Now, the mean of linear-fit errors $\eta_k$ over all $\mathbf{x}^{(j)} \in \mathcal{N}^{(i)}(k)$ is used as a fitness to evaluate the neighborhood size.

We consider $k = n + 1$ as the least number of desired points in $\mathcal{N}^{(i)}(k)$ since $n$ points are needed for a unique plane fit, plus an additional point for cross-validation error measurement. Accordingly, the

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

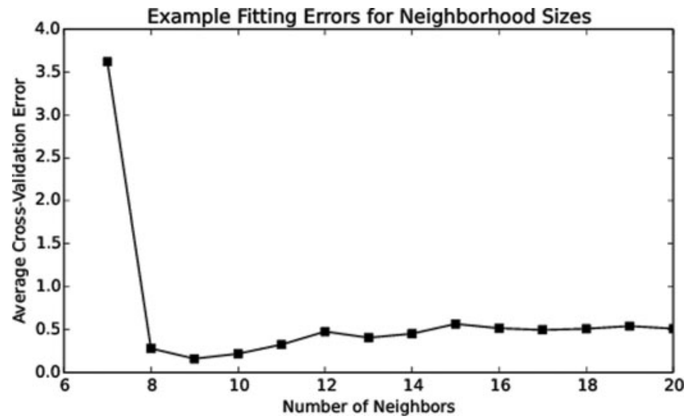


Fig. 2. An example of neighborhood cross-validation error fits as the neighborhood size increases. This particular example behaves nicely and the algorithm will select the optimal size at $N$=9.

neighborhood size is initialized to $\delta^{(i)}_{n+2}$ and the corresponding $\eta_k$ is computed. Next, $k$ is incremented by one and $\eta_{k+1}$ is computed. If $\eta_k < \eta_{k+1}$, then $\delta^{(i)}_k$ is reported as optimal. Otherwise, the search continues to find a better neighborhood size.

In general, if sample density around $\mathbf{x}^{(i)}$ is moderate, the successive error values will decrease as the plane fits are less sensitive to noise induced from few samples, but then increase again as the neighborhood begins to include the non-linearity of the sampled function. Finding the neighborhood-size when the error first increases is used as a rough heuristic to find the balance between these two factors while also not evaluating more neighborhood sizes than necessary. Figure 2 shows an actual computed example where this heuristic happens to give the optimal neighborhood size.

**Algorithm 1** Adaptive neighborhood selection

1: **Input:** Sample set $\mathcal{S}$, index of base point $i$
2: $k \leftarrow n + 2$
3: $\mathcal{N}^{(i)}(k) \leftarrow$ GenerateNeighborhoodSet($k$) using(9) & (10)
4: $\eta \leftarrow 0$
5: **for** $j = 1 : k$ **do**
6: $\quad A \leftarrow$ FitHyperplane($\mathcal{N}^{(i)}(k)/\mathbf{x}^{(j)}$)
7: $\quad \eta \leftarrow \frac{1}{k}\left(\eta + \text{FindLinearFitError}(A, \mathbf{x}^{(j)})\right)$
8: **end for**
9: $k^* \leftarrow k$
10: $\eta_{prev} \leftarrow \eta$
11: **while** (true) **do**
12: $\quad k \leftarrow k + 1$
13: $\quad \mathcal{N}^{(i)}(k) \leftarrow$ GenerateNeighborhoodSet($k$) using(9) & (10)
14: $\quad \eta \leftarrow 0$
15: $\quad$ **for** $j = 1 : k$ **do**
16: $\quad\quad A \leftarrow$ FitHyperplane($\mathcal{N}^{(i)}(k)/\mathbf{x}^{(j)}$)
17: $\quad\quad \eta \leftarrow \frac{1}{k}\left(\eta + \text{FindLinearFitError}(A, \mathbf{x}^{(j)})\right)$
18: $\quad$ **end for**
19: $\quad$ **if** $\eta_{prev} < \eta$ **then**
20: $\quad\quad k^* \leftarrow k - 1$ **break**
21: $\quad$ **end if**
22: $\quad \eta_{prev} \leftarrow \eta$
23: **end while**
24: **return** $\left[\mathcal{N}^{(i)}(k^*), k^*\right]$

Figure 3 shows the resulting behavior of this algorithm, which is to shrink neighborhood sizes in regions of high sample density so as to maintain model accuracy. This algorithm does generate

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

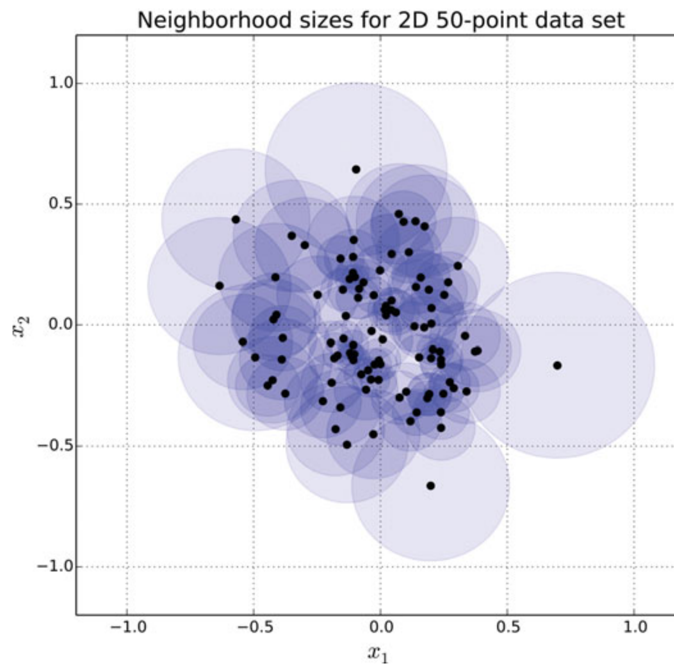


Fig. 3. Example neighborhood sizes computed using the adaptive heuristic for normally distributed 2D data.

similar neighborhood sizes as the simpler heuristic of just using the nearest $N$ neighbors, but, as will be shown later, accounting for the accuracy of the plane fit leads to better target search results.

*4.2.2. Planar model approximation.* An affine hyperplane $\mathcal{F}^{(i)}(\mathbf{x})$ is fit to points in the neighborhood $\mathcal{N}^{(i)}(k^*)$, obtained from the previous phase, by using a least-squares method:

$$\mathcal{F}^{(i)}(\mathbf{x}) = A^{(i)}(\mathbf{x} - \mathbf{x}^{(i)}) + y^{(i)} \tag{11}$$

where $A^{(i)} = \left[a_1^{(i)}\ a_2^{(i)}\ \cdots\ a_n^{(i)}\right]$ is a row vector of planar model coefficients. If the plane is not uniquely determined (e.g., all the points are collinear), then the neighborhood size is incrementally expanded until a set of points is found that uniquely determines the plane.

*4.2.3. Divergence-model approximation.* The plane obtained using Eq. (11) is assumed to be an approximation of the tangent plane of the true task function in the vicinity of the point in question. This approximation is expected to diverge substantially as we move away from the fitted neighborhood. Therefore, we then estimate how quickly this divergence occurs by computing the absolute error $e^{(j)}$ between the predicted task score (using the plane approximation at $\mathbf{x}^{(i)}$) and the actual measured score for every point $\mathbf{x}^{(j)}$ in an annular-box-neighborhood $\mathcal{M}^{(i)}$:

$$\mathcal{M}^{(i)}(\beta) = \{\mathbf{x}^{(j)} : \mathbf{x}^{(j)} \in \mathcal{X} / \mathcal{N}^{(i)} \wedge ||\mathbf{x}^{(i)} - \mathbf{x}^{(j)}||_\infty \leq \beta\} \tag{12}$$

where $\beta > \delta$ is the size of the neighborhood.

$$e^{(j)} = |y^{(j)} - \mathcal{F}^{(i)}(\mathbf{x}^{(j)})|\ \forall\ \mathbf{x}^{(j)} \in \mathcal{M}^{(i)} \tag{13}$$

where $\mathcal{F}^{(i)}$ is the affine hyperplane corresponding to $\mathbf{x}^{(i)}$.

For each $\mathbf{x}^{(i)}$, we then construct an error estimate function $\mathcal{E}^{(i)}(\Delta\mathbf{x})$ that estimates the upper bound on these absolute error values. Our formulation uses a quadratic function with different weights $\omega_j^{(i)}$

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

for each parameter axis and whose minimum lies at $\mathbf{x}^{(i)}$ where the plane is fit:

$$\mathcal{E}^{(i)}(\Delta\mathbf{x}) = \sum_{j=1}^{n} \omega_j^{(i)} \Delta\mathbf{x}_j^2 \tag{14}$$

where $\Delta\mathbf{x} = \mathbf{x} - \mathbf{x}^{(i)}$. The weights $\omega_j^{(i)}$ are found by solving the following optimization problem:

$$\text{Minimize } \sum_{j=1}^{n} \left(\omega_j^{(i)}\right)^2 \tag{15}$$

$$\text{subject to } \mathcal{E}^{(i)}(\Delta\mathbf{x}^{(j)}) \geq e_j \; \forall \, \mathbf{x}^{(j)} \in \mathcal{M}^{(i)} \tag{16}$$

A quadratic programming solver was used for this purpose.

At the end of the model generation phase, we have $\mathcal{N} = \{\mathcal{N}^{(i)} : i = 1, 2, \ldots, m\}$, $\mathcal{F} = \{\mathcal{F}^{(i)} : i = 1, 2, \ldots, m\}$, and $\mathcal{E} = \{\mathcal{E}^{(i)} : i = 1, 2, \ldots, m\}$ representing the sets of $m$ adaptive neighborhoods, $m$ hyperplanes, and $m$ error estimate functions, respectively, corresponding to each sample point in $\mathcal{S}$.

### *4.3. Exploitation-driven model updating*

Given a desired task score $y_d$, the sets $\mathcal{S}$, $\mathcal{F}$, and $\mathcal{E}$ are used to find a new point in the parameter space. As we want to search the parameter space quite conservatively, we would like to query a new point that will provide the smallest uncertainty in task score with respect to a local error estimate function. This is performed by selecting an existing sample point in $\mathcal{S}$ as a base point and by selecting only a single parameter for modification at that base point, which precludes the need to measure distances involving changes in multiple parameters and minimizes the possibility of error arising from unknown cross-effects between the parameters.

These two selections are made by conducting a search at each base point $\mathbf{x}^{(i)}$ in the following way. For each parameter $x_j^{(i)}$, the desired corrective movement $\Delta x_j^{(i)}$ is calculated by finding a point in the direction parallel to that parameter axis whose task score based on the plane approximation is equal to the desired amount $y_d$. The parameter change is saturated if the corresponding error estimate function rises above a given threshold $e_{max}$ before reaching the new point. Accordingly, the parameter change is given by

$$\Delta x_j^{(i)} = \text{sgn}\left(\frac{y_d - y^{(i)}}{a_j^{(i)}}\right) \min\left(\left|\frac{y_d - y^{(i)}}{a_j^{(i)}}\right|, \sqrt{\frac{e_{max}}{\omega_j^{(i)}}}\right) \tag{17}$$

$\forall \, j = 1, 2, \ldots, n.$

The saturation limit on parameter change used in Eq. (17) prevents the system from testing points that have the potential for large error, possibly resulting in trials outside the proper operating range which would give no new information.

Note that the search in the parameter space is deliberately restricted to individual parameter directions. This results in generation of new sets of points called *line-sets*, where all points in a line-set $\mathcal{L}_j^{(i)}$ lie on a line parallel to single parameter axis $j$, $j = 1, 2, \ldots, n$:

$$\mathcal{L}_j^{(i)} = \{\mathbf{x}^{(k)} \in \mathcal{X} : |x_\ell^{(i)} - x_\ell^{(k)}| \neq 0 \text{ only for } \ell = j\} \tag{18}$$

Therefore, whenever such a line set is available for a base point $\mathbf{x}^{(i)}$, the algorithm makes use of a line approximation over the points in the line set, instead of using the planar approximation in Eq. (11), to compute the parameter change at that point. That is, for each parameter $j$ where $|\mathcal{L}_j^{(i)}| \neq 0$, the algorithm computes $b_j^{(i)}$ as the slope of the best fit line through the points in $\{x^{(i)}, \mathcal{L}_j^{(i)}\}$ using a one-dimensional least squares computation. Accordingly, $b_j^{(i)}$ replaces $a_j^{(i)}$ in Eq. (17) during the computation of $\Delta x_j^{(i)}$.

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

**Algorithm 2** Model generation and movement selection

1: **Input:** $\mathcal{S} = \{(\mathbf{x}^{(i)}, y^{(i)}) : \mathbf{x}^{(i)} \in \mathcal{X}, y^{(i)} \in \mathcal{Y}, i = 1, 2, \ldots, m\}$,
2: target score $y_d$, success tolerance $\epsilon$, time out $t_{max}$,
3: number of new sample points after each time out $n_c$
4: **while** $(\nexists\ y^{(i)} \in \mathcal{Y}$ s.t. $|y_d - y^{(i)}| < \epsilon)$ **do**
5: $t \leftarrow 0$
6: **while** $(t \le t_{max})$ **do**
7: **for** $i = 1 : m$ **do**
8: $\left[\mathcal{N}^{(i)}, k\right] \leftarrow$ AdaptiveNeighborhoodSelection$(i, \mathcal{S})$
9: $A^{(i)} \leftarrow$ FitHyperplane $\left(\{(\mathbf{x}^{(j)}, y^{(j)}) : \mathbf{x}^{(j)} \in \mathcal{N}^{(i)}(k)\}\right)$ (using (11))
10: $\mathcal{M}^{(i)}(\beta) \leftarrow \{\mathbf{x}^{(j)} : \mathbf{x}^{(j)} \in \mathcal{X} / \mathcal{N}^{(i)} \wedge ||\mathbf{x}^{(i)} - \mathbf{x}^{(j)}||_\infty \le \beta\}$
11: **for** $j = 1 : |\mathcal{M}^{(i)}|$ **do**
12: $e^{(j)} \leftarrow |y^{(j)} - \mathcal{F}^{(i)}(\mathbf{x}^{(j)})|$
13: **end for**
14: $\omega^{(i)} \leftarrow$ FindWeights $\left(\mathcal{E}^{(i)}(\Delta\mathbf{x}), f(\omega^{(i)}), \{e^{(j)} \in \mathcal{M}^{(i)}\}\right)$ (using (14)–(16))
15: **end for**
16: **for** $i = 1 : m$ **do**
17: **for** $j = 1 : n$ **do**
18: $\mathcal{L}_j^{(i)} \leftarrow \{\mathbf{x}^{(k)} \in \mathcal{X} : |x_\ell^{(i)} - x_\ell^{(k)}| \ne 0$ only for $\ell = j\}$
19: $c \leftarrow a_j^{(i)}$
20: **if** $|\mathcal{L}_j^{(i)}| \ne 0$ **then**
21: $c \leftarrow FitLine\left(\{\mathbf{x}^{(i)}, \mathcal{L}_j^{(i)}\}\right)$
22: **end if**
23: $\Delta x_j^{(i)} \leftarrow \text{sgn}\left(\frac{y_d - y^{(i)}}{c}\right) \min\left(\left|\frac{y_d - y^{(i)}}{c}\right|, \sqrt{\frac{e_{max}}{\omega_j^{(i)}}}\right)$
24: **end for**
25: **end for**
26: $\Delta x_{j^*}^{(i^*)} \leftarrow \min\limits_i \left(\arg\min\limits_j \mathcal{E}^{(i)}(\Delta x_j^{(i)})\right)$
27: $[i^*, j^*] \leftarrow \arg\ \Delta x_{j^*}^{(i^*)}$
28: $\hat{x}_j \leftarrow \begin{cases} x_j^{(i^*)} + \Delta x_j^{(i^*)} & \text{if } j = j^* \\ x_j^{(i^*)}, & \text{otherwise} \end{cases}$
29: $\hat{y} \leftarrow$ Evaluate$(\hat{\mathbf{x}})$
30: $\mathcal{S} \leftarrow$ Append$(\hat{\mathbf{x}}, \hat{y})$
31: $t \leftarrow t + 1$
32: **end while**
33: **end while**

Now, the error estimate function is computed for $\Delta x_j^{(i)}$ for all $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$ and the parameter change with the least error estimate is found as below:

$$\Delta x_{j^*}^{(i^*)} = \min_i \left(\arg\min_j \mathcal{E}^{(i)}(\Delta x_j^{(i)})\right) \tag{19}$$

Therefore, this optimal amount of change will depend both on its magnitude, which is a function of the model coefficients $a_j^{(i)}$ or $b_j^{(i)}$, and how quickly the error function rises, which is a function of the quadratic surface weight $\omega_j^{(i)}$.

Now, the new test point $\hat{\mathbf{x}}$ is determined as follows:

$$[i^*, j^*] = \arg\ \Delta \mathbf{x}_{j^*}^{(i^*)} \tag{20}$$

$$\hat{x}_j = \left\{ \begin{matrix} x_j^{(i^*)} + \Delta x_j^{(i^*)} \text{ if } j = j^* \\ x_j^{(i^*)},\ \text{otherwise} \end{matrix} \right\} \tag{21}$$

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

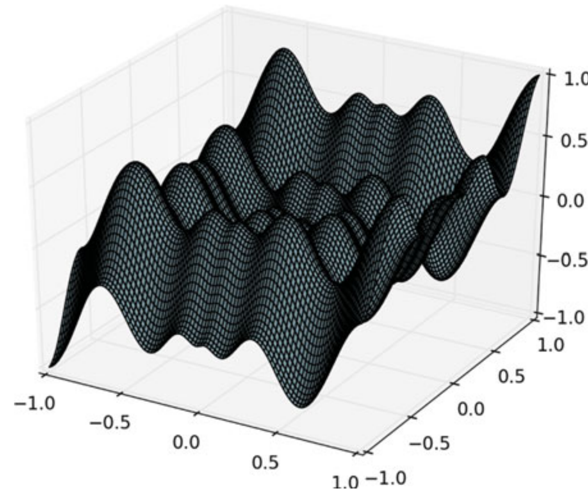


Fig. 4. A cross-section of the Schwefel test function ($c$=200). Two parameters were varied from −1 to 1 and three were fixed at zero.

The new point $\hat{\mathbf{x}}$ is then sent to the trajectory generation module, which then provides the robot with a new trial. The robot performs the trial and the new task score $\hat{y}$ is recorded. Assuming the trial execution still results in failure, the new sample $(\hat{\mathbf{x}}, \hat{y})$ is appended to $\mathcal{S}$ and the process is repeated.

## 5. Results

### *5.1. Synthetic non-linear task function*

Our primary results are shown using a synthetic task of learning the behavior of a directly specified non-linear function through sampling of the parameter space. An $N$-dimensional variation of the Schwefel function, whose landscape presents complexities including high non-linearity and multiple local extrema was chosen for the purpose.

$$\text{Schwefel} : z_s(\mathbf{x}) = \frac{1}{2} \sum_{i}^{N} x_i \sin \sqrt{|cx_i|} \tag{22}$$

Figure 4 shows the landscape of a two-dimensional cross-section of the Schwefel test function, which indicates the complexity of the function and the challenge in learning its behavior. Our results were obtained with the five-dimensional version of the function, with each parameter in the range −1 to 1. For further comparison, we additionally obtained results on a five-dimensional parabola with the same parameter domain.

*5.1.1. Linear models exploitation performance.* We computed results for the approach over multiple trials, varying both the number of initial randomly generated sample points and the task tolerance. Each trial consists a large set of target values to find, but trials used to find earlier targets are not retained, so that each target is found from the same initial data set. The results are presented per trial, with basic information about the distribution of the number of trials needed for each target: the mean over the whole target set, the range of the median 90% of the numbers of iterations, and the full range from the minimum to the maximum number of iterations needed.

Figure 5 shows the performance as a function of the initial random data set size. As we expect to see, there is an indication of asymptotic behavior where the algorithm is able to find solutions to the large majority of targets in a single iteration with sufficiently dense sampling of the parameter space. However, it is notable that the majority of the decrease in estimated number of iterations occurs at small data sizes (around 100), indicating good performance is attainable with only minor cost-intensive initial sampling. The number of iterations needed for the worst performing target in each trial does show some decrease with larger initial data sets, but it is clear that some outliers

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

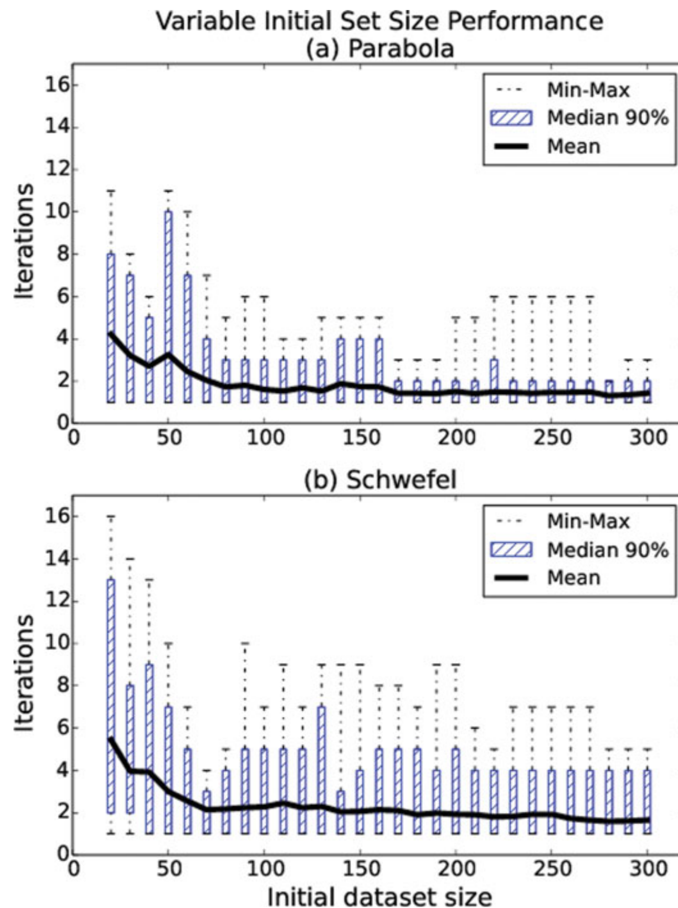


Fig. 5. A comparison of the algorithm performance on both the (a) parabola and (b) Schwefel test functions, with varying sizes of the initial randomly sampled data set. All trials were performed with a tolerance of 0.005. Notice that, by design, the algorithm will always perform at least one iteration to minimize the error.

remain that need several more iterations than typical, revealing the complexity of the test function. However, all targets are found without a substantially large deviation from the mean.

Figure 6 shows the performance of our algorithm when tested with different values of task success tolerance. The algorithm is able to rapidly find solutions on both functions. The Schwefel function shows greater variability in the range of iterations needed, as would be expected given the much greater degree of non-linearity exhibited. However, the maximum number of iterations needed is not significantly greater than the parabola result.

Finally, Fig. 7 shows the performance of the algorithm for a long-term learning scenario where all previous points are not discarded when starting the search for a new target, but are instead retained in the library. A set of 4500 targets was given to the algorithm with a sufficiently high tolerance that no existing points in the data set were already solutions, and were sequentially provided to the algorithm in a random order. As we would expect, the performance of the learning algorithm approaches a single iteration when searching for a new target. While there are still a few outliers that take 3–6 iterations to find solutions, we would expect that number to reduce as even more samples were taken and more details about the function landscape were obtained.

*5.1.2. Exploitation updates with a GPR model.* As previously mentioned, it is possible to use many features of our exploitation updates approach in a regression-model-agnostic way. We therefore show the performance of our iterative strategy with an equivalent implementation on a Gaussian process regression model. A Gaussian process is trained on the same initial data set and can therefore make predictions both of the function value and the estimated error for new points. To implement the same exploitation-driven update strategy as described in Section 4.3, we generate a set of parameter movements that are expected to achieve the desired target value and are based at the previously

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

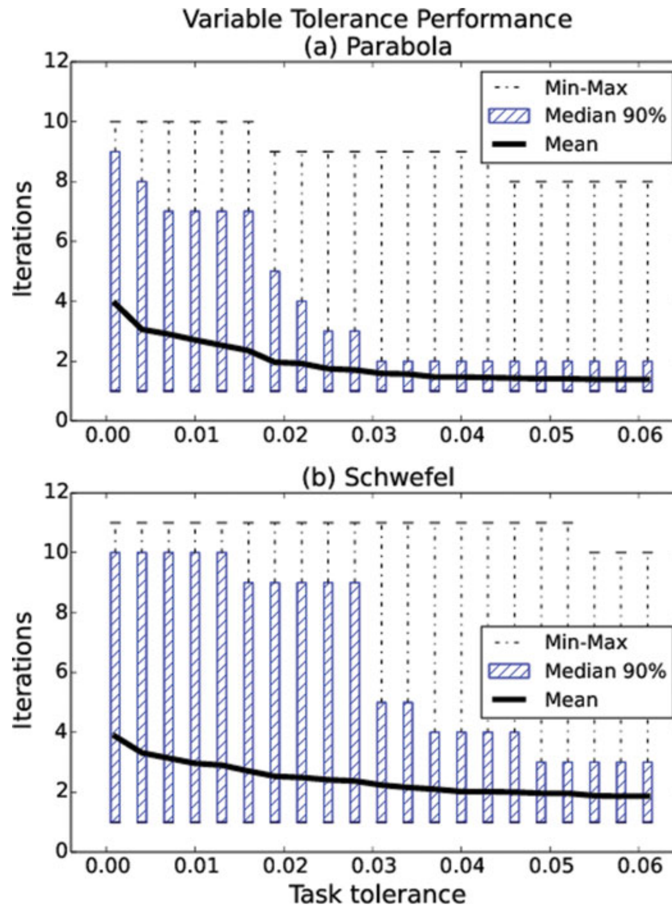


Fig. 6. A similar comparison of the two test functions, but using different task tolerance values. All trials were performed with an initial data set of 50 sampled points.

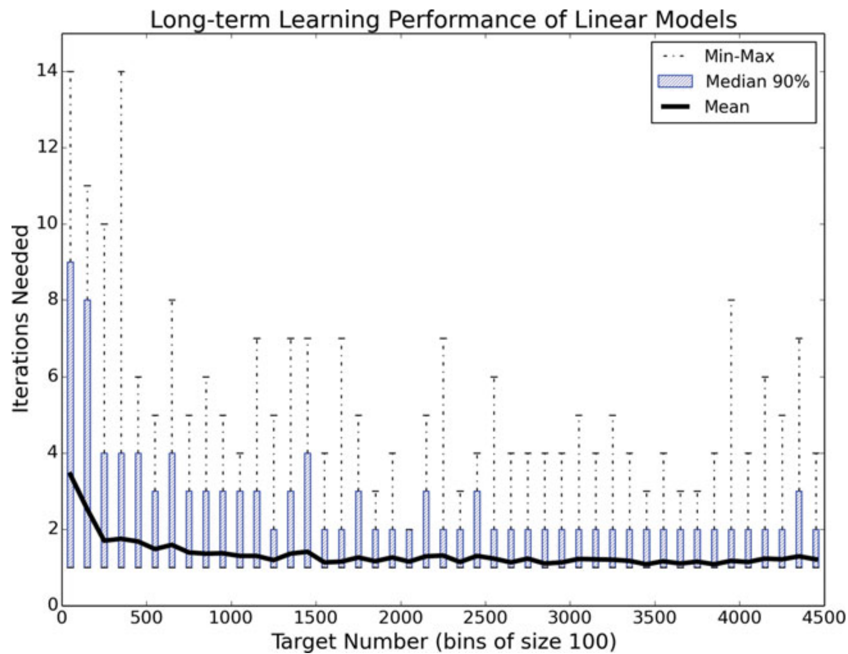


Fig. 7. The algorithm performance for a set of 4500 randomly ordered targets where every trial is saved for future learning. The distribution of the number of iterations is shown for each bin of 100 consecutive targets to show the trend in performance.

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

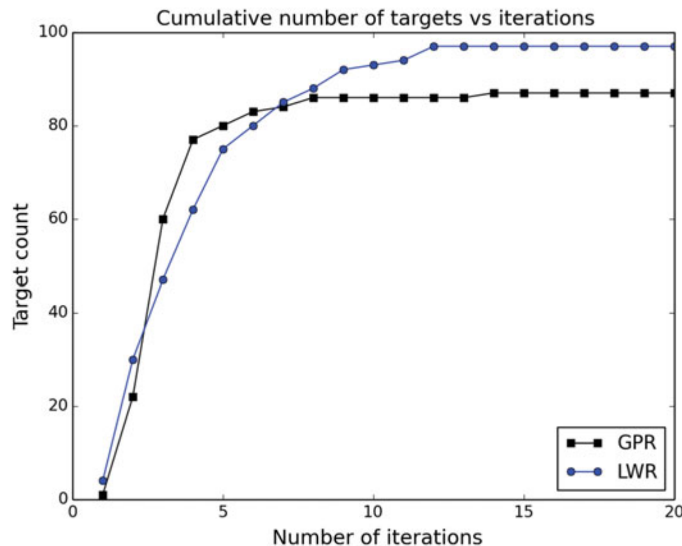


Fig. 8. Comparison of the exploitation-update strategy using local linear models and a global GPR model in terms of the numbers of targets found by each iteration. The task tolerance was $10^{-3}$.

sampled data points. The only difference is that to find the move distances, an iterative search is required, as no invertible analytical model is available of the function behavior in the vicinity of the base point. All of the moves from the base points in all parameter directions are compared and the move with the smallest expected error is selected for execution. As before, if the target is not achieved within the specified tolerance, the true point is added to the library and the process is restarted.

To compare the two modeling algorithms, we gave each the same set of 50 data points and 100 targets to achieve. Unsuccessful trials were added to the set of points, but the data set was reset after each target was found. Figure 8 shows the number of these 100 targets achieved depending on how many iterations were run. Our approach with linear models has more targets found faster at 1–2 iterations but falls slightly behind for slightly longer runs of 3–6 iterations. Note that both approaches fail to achieve a few targets within 20 iterations (3 for our approach and 13 using GPR). For our approach, this occurs because of a time-out in the updating algorithm, but we would expect the algorithm to eventually find solution points as more data samples were acquired. In the case of the GPR model, however, the missing targets are due to the search algorithm not finding any points with the expected target value, even when searching in all parameter directions from all known base points. In such cases, the updating strategy has no option but to exit and report that the target cannot be found. Finally, the computational cost of our model was significantly lower as there is no need to search systematically for the estimate target. Due to this faster performance, initial performance benefit at low iterations, and the additional robustness for finding targets, we report the remaining results in this paper for our model only. However, we still emphasize that the update strategy performed effectively with the GPR model and it may be more appropriate in other circumstances.

### 5.2. *Algorithm features characterization*

Here, we present results demonstrating the value of the key algorithm features detailed in the previous section in terms of the overall performance. The three main features in our approach are (1) using single parameter adjustments, (2) using an upper-bounding quadratic error model, and (3) adaptively selecting the neighborhood size of each local model.

*5.2.1. Single parameter adjustments.* One of the more novel features in our approach is that new test points are generating by only adjusting a single parameter from a previously tested point. While initially seeming overly restrictive, this has the benefit of generating multiple data points along a single line. We can then exploit this to do model fitting of that line directly, rather than only trying fit an $N$-dimensional plane to a neighborhood. This direct fit allows movements along the line to have much higher accuracy than along the plane in an arbitrary direction. Corresponding, the performance of the algorithm in terms of iterations needed to find desired task outputs is greatly improved.

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

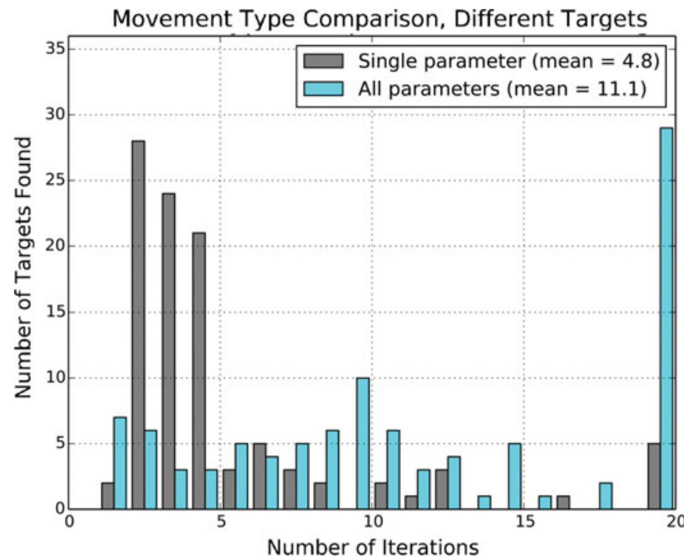


Fig. 9. Performance of the learning algorithm using different parameter adjustment methods as a comparative histogram. The algorithm started with an initial data set of 50 random points and searched for 100 different targets.

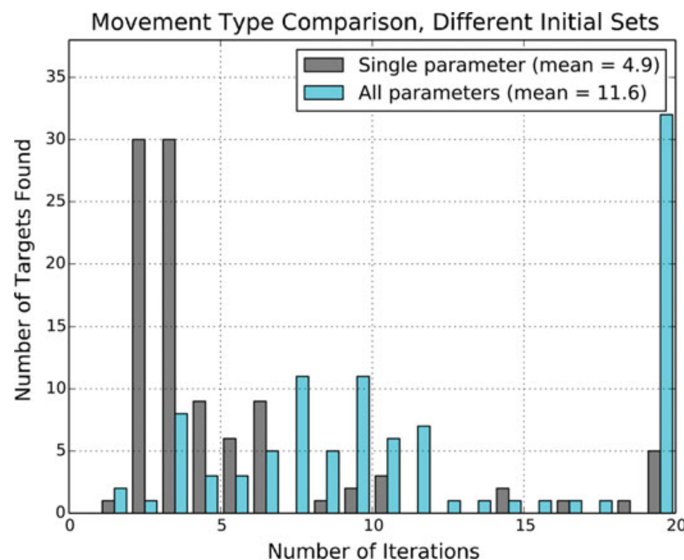


Fig. 10. Learning algorithm performance with different parameter adjustment methods as in the previous figure. Here, the same target was found 100 times, but with a different random initial data set each time.

We compare our approach with a simpler method where the movement from each known data point is parallel to the gradient of the plane fitted to the selected neighborhood. The adjustments are ranked by the expected error using the same error model and the movement distance is saturated if the expected error rises above a given threshold. So the only difference in the comparison is the direction of the adjustment and the possible use of the line model in the case of single parameter adjustments. Figures 9 and 10 show the results from the comparison. The single parameter adjustment strategy comes out substantially better, reducing the average number of iterations by over 50%.

*5.2.2. Quadratic error model.* For comparison with the quadratic error model, we used an algorithm that specifies a region around each local model as error free and any prediction outside of the region to have potentially infinite error, inspired by a trust-region approach.[39] The size of the region was determined by the distance to the furthest sample point whose measured error with respect to the plane fit to the local neighborhood remained below a threshold. If no neighboring point had an error

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

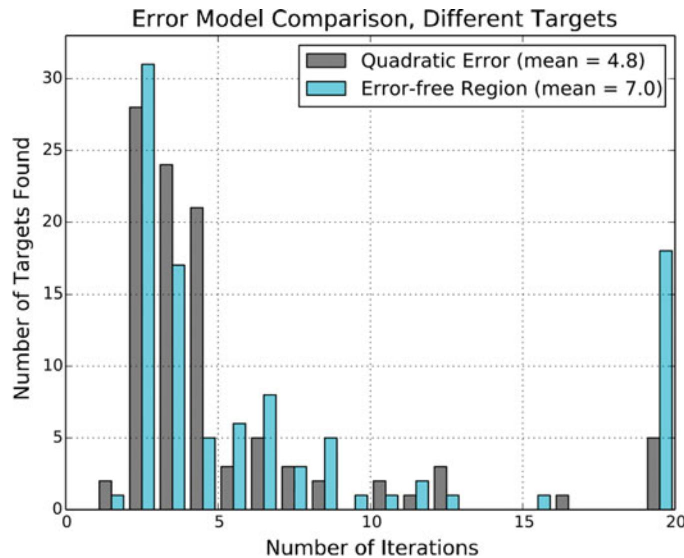


Fig. 11. Performance of the learning algorithm using different error model algorithms as a comparative histogram. The algorithm started with an initial data set of 50 random points and searched for 100 different targets.

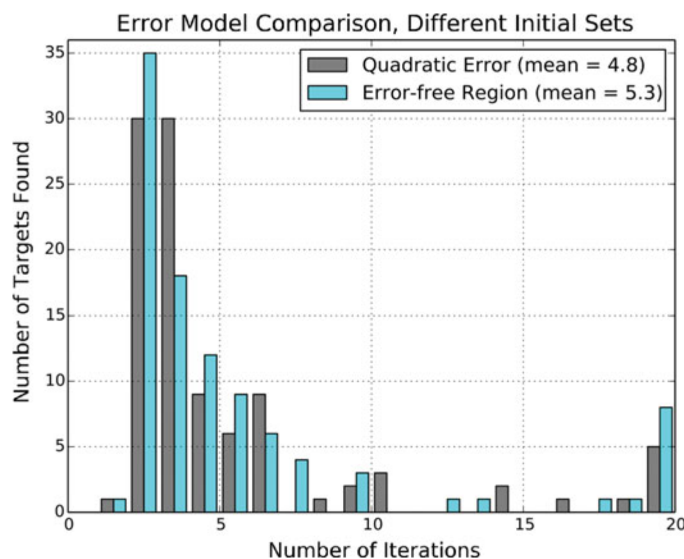


Fig. 12. Learning algorithm performance with different error model algorithms as in the previous figure. Here, the same target was found 100 times, but with a different random initial data set each time.

below this threshold, the region radius was set to zero. The threshold can be set fairly arbitrary and we found that the model had good performance when it was set to the same value as $e_{max}$, described in Section 4.3. When selecting the parameter adjustments from the different local models, the shortest move that remained within the error-free region was selected. The comparison results with the quadratic bounding error model are shown in Figs. 11 and 12, which shows better performance for the quadratic model on average. Interestingly, the error-free region model has better performance in the number of task instances solved after just two iterations, but then immediately drops off in performance while the quadratic model finds many other solutions in three and four iterations.

*5.2.3. Adaptive neighborhood heuristic.* The comparison for the adaptive neighborhood heuristic was done with two other much simpler heuristics. The $N$-closest heuristic uses the minimal number of neighboring points to fit the local plane, which is $N$ for an $N$-dimensional task parameter space. If, as happens in the case of single parameter adjustments, the $N$ closest points form a linearly dependent vector space, the neighborhood was expanded incrementally until a unique plane could be fit to the

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

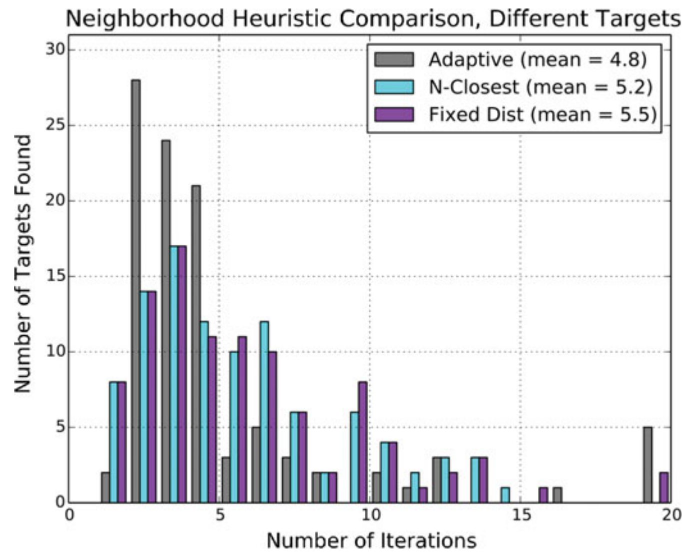


Fig. 13. Performance of the learning algorithm using three different neighborhood heuristics as a comparative histogram. The algorithm started with an initial data set of 50 random points and searched for 100 different targets.

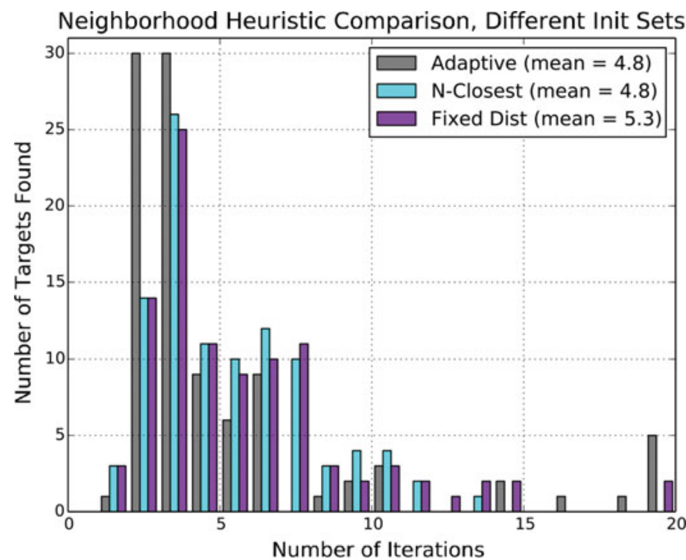


Fig. 14. Learning algorithm performance with three neighborhood heuristics as in the previous figure. Here, the same target was found 100 times, but with a different random initial data set each time.

points. For the fixed distance heuristic, all points within a specified distance were selected for the neighborhood. If there were insufficient points within the given distance, the results from the $N$-closest heuristic were used instead. Figures 13 and 14 show the results when compared with the adaptive neighborhood heuristic. On average, the adaptive heuristic has better performance, especially with the number of solutions found within three iterations.

### *5.3. Robot dynamic pouring experiments*

To validate our approach on a physical robot, we used a dynamic pouring task, where a Baxter robot is tasked to pour a specific amount of liquid into a container which is placed on a rotating table. This task is intended to be representative of a manufacturing scenario where the robot may be asked to perform new tasks or task variations with limited time available for learning. Additionally, the task of pouring liquid into a moving container is highly amenable to autonomous learning as it is extremely difficult to model accurately without an existing data set.

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

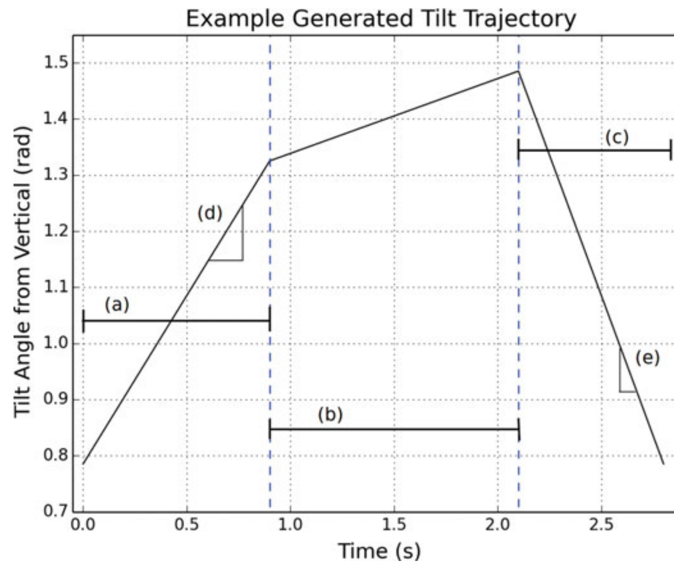


Fig. 15. An example of a generated tilt trajectory which is controlled by five parameters: (a) the tilt-forward time, (b) the intermediate time between the two tilts, (c) the tilt-backward time, (d) the tilt-forward rate, and (e) the tilt-backward rate. To reduce the amount of overall rotation needed, the base tilt angle is not zero but 45°.

Fig. 16. An example execution of the pouring task for illustration. The four images show (a) the robot holding the bottle and waiting for the trigger to begin pouring, (b) the initial tilting phase where the liquid starts to begins to rush out, (c) the middle pouring phase where the flow is more laminar, and (d) the untilting phase where the flow is cut off by the rising bottle edge.

We selected the tilt trajectory of the bottle held by the robot as the learning target with the complementary motions of the robot's joints found using standard planning algorithms. In particular, the tilt profile robot's end-effector action consists of tilting the bottle in one direction (forward tilt) for some duration, keeping the tilt steady for some time, and untilting the bottle (reverse tilt). Accordingly, the tilt profile was parameterized by five real-valued parameters, which were manually defined as relevant physical features of the pouring action: (1) forward tilt time $t_f$, (2) forward tilt rate $\dot{\theta}_f$, (3) intermediate pouring time $t_s$, (4) reverse tilt time $t_r$, and (5) reverse tilt rate $\dot{\theta}_r$. A value for each of these parameters defines a point in the five-dimensional parameter space in which the learning algorithm operates. An example plot of what the generated tilt trajectory looks like can be seen in Fig. 15. The parameters are all normalized in order to handle different units and prevent changes in one parameter for dominating all others. This normalization can be done after an initial data set is obtained from random trials, or by knowing reasonable physical limits in the case of a manually defined parameter space.

The experimental setup used is shown in Fig. 1. The rotating table has several visual markers used in augmented reality applications that provide the robot knowledge about the current table angle. After measuring the table rotation speed, a manually defined planning algorithm takes the tilt profile to execute and generates a path for the arm which ensures the tip of the held bottle will remain above the target container and in a proper pouring location during the pouring motion. This involves simply computing the desired pose of the robot end-effector and using the inbuilt inverse kinematics to compute corresponding joint angles at several points along the arc followed by the target container.

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

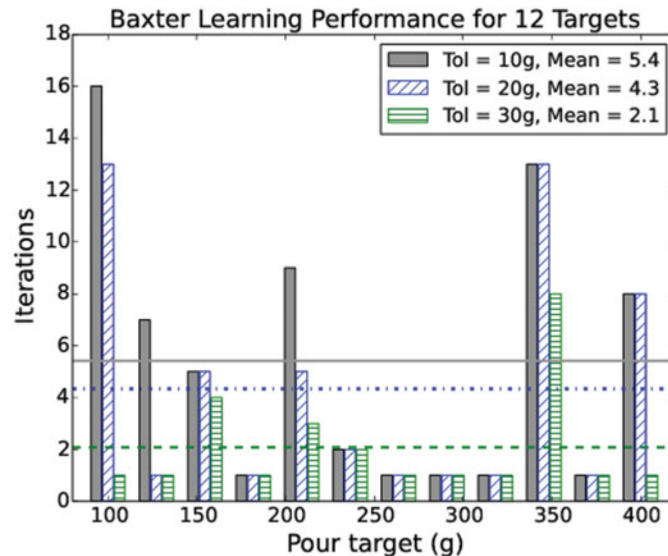


Fig. 17. Baxter learning performance for a uniformly distributed set of targets. Each group of three bars is the number of iterations needed for a single target, with the gray, blue, and green bars corresponding to tolerances of 10, 20, and 30 g, respectively. All trials were done with the initial set of 37 random points.

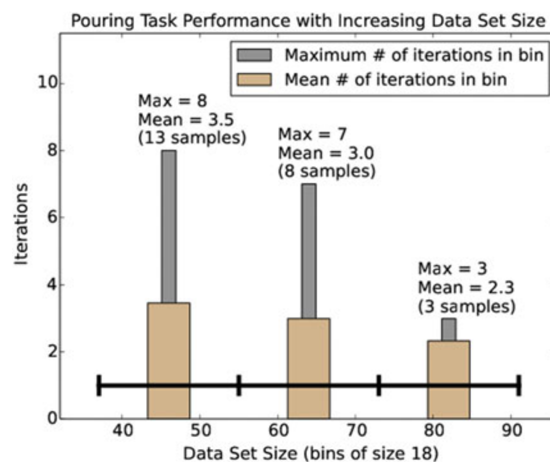


Fig. 18. Baxter learning performance depending on initial size of the data set when starting each new target. For this data, the robot was given the same initial set of 37 samples, but as it progressed through the 12 targets, all trials were saved in the data set so later targets had more samples to learn from. The full experiment was then repeated starting from the initial set once again to obtain 24 total target samples. As there was substantial variation in the samples, the results were divided into three bins so that a rough mean could be estimated. The success tolerance was 20 g for all targets.

An example execution of the task is shown in Fig. 16 (using the tilt-trajectory in Fig. 15), which shows the breakdown of the pouring motion into the three phases controlled by the learned parameters.

After each pour, the amount of fluid poured was measured manually using a scale with a precision of 2 g. Initial exploratory tests showed that due to the various system errors, including measuring the table position and robot trajectory tracking, the amount poured for a single set of parameters has a variation of up to $\pm 5$ g from a mean poured volume. This indicates a small degree of stochastic behavior that can affect the number of iterations needed to find a pouring trajectory which falls within a strict tolerance of task success. For context, it should be noted that the variation in poured volumes when the task is performed by humans, even with practice, is much higher that the robot variation.

An initial library of 40 points was generated by evaluating randomly generated points in the parameter space. Three were removed where either the entire bottle of fluid was poured (450 g) or no fluid was poured. With this initial data set, 12 targets were given uniformly from 100 to 400 g.

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

Figure 17 shows the performance on these targets for three different task success tolerance values. In keeping with our results from function approximation, the algorithm has faster convergence with less restrictive tolerance. Figure 18 shows a second experiment where the same targets were repeated but the initial data set used included the points tested in all previous targets. Again, paralleling the results from the previous section, there is a visible trend of a decreasing number of iterations needed as the initial library size grows. Also, note that all the means for the long-term learning in Fig. 18, using a 20 g tolerance, are lower than the mean for the 20 g tolerance case in Fig. 17.

## 6. Conclusions

We presented an approach that allows a robot to generate task trajectories using a set of linear metamodels. These models were successfully learned from sparse initial exploratory experiments and enabled the system to learn to perform a complex task instance with very few attempts needed. Our approach made use of multiple features which we demonstrated to improve the overall performance of the algorithm, including single parameter adjustments, local quadratic error models, and adaptive neighborhood selection. Experimental results using both synthetic and real robot tasks revealed that the algorithm converges faster as more experiments were conducted, suggesting that the algorithm supports lifelong learning. We also discussed how alternative well-known approaches were not ideal for solving this particular problem, which our's handles quite well. We conclude that it is primarily useful for motion planning problems where many similar task instances must be solved and in which model prediction by simulating the underlying physics involving the trajectory variables and task behavior is very difficult.

Future work will focus on extending the approach to be feasible for more complex problems. In particular, how the complexity and computational limits of the algorithm scale to higher dimensions is still an open question. Currently, we update all local models as new data is acquired but the formulation naturally extends to an incremental learning scheme where only the models in the vicinity of the new data point are adjusted. Additionally, other modeling algorithms can also be used with the exploitation strategy effectively. This suggests we could expand the approach to become model agnostic and even use a hybrid approach where different regions of the parameter space could be modeled by different algorithms, depending on the function behavior. The linear models could be used in regions of sparse data with high computational performance, a Gaussian process can be used to provide more accurate predictions in regions of high variability, and LWPR could be used when sufficient samples had been acquired to slow down the other algorithms. These improvements may enable the approach to be competitive for much higher dimensional problems as well.

## Acknowledgements

This work was supported in part by Office of Naval Research under grant N000141310597. The information in this paper does not necessarily reflect the position or policy of the sponsors.

## References

1. E. Aboaf, C. G. Atkeson and D. J. Reinkensmeyer, Task Level Robot Learning: Ball Throwing. Technical report, MIT, Cambridge, MA, 1987.
2. F. J. Abu-Dakka, F. J. Valero, J. Luis Suner and V. A, "Mata direct approach to solving trajectory planning problems using genetic algorithms with dynamics considerations in complex environments," *Robotica* **33**(3), 669–683 (2015).
3. B. Akgun, M. Cakmak, K. Jiang and A. L. Thomaz, "Keyframe-based learning from demonstration," *Int. J. Soc. Robot.* **4**(4), 343–355 (2012).
4. H. F. N. Al-Shuka, B. Corves and W.-H. Zhu, "Function approximation technique-based adaptive virtual decomposition control for a serial-chain manipulator," *Robotica* **32**(3), 375–399 (2014).
5. M. Arif, T. Ishihara and H. Inooka, "Incorporation of experience in iterative learning controllers using locally weighted learning," *Automatica* **37**(6), 881–888 (2001).
6. C. G. Atkeson, A. W. Moore and S. Schaal, "Locally weighted learning," *Artif. Intell.* **11**, 11–73 (1997).
7. D. Berenson, P. Abbeel and K. Goldberg, "A Robot Path Planning Framework that Learns from Experience," *Proceedings of the International Conference on Robotics and Automation (ICRA)*, St. Paul, MN, USA (2012) pp. 3671–3678.

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

8. B. Bocsi, L. Csato and J. Peters, "Alignment-Based Transfer Learning for Robot Models," *Proceedings of the International Joint Conference on Neural Networks*, Dallas, TX (2013) pp. 1–7.
9. C. Bowen, G. Ye and R. Alterovitz, "Asymptotically optimal motion planning for learned tasks using time-dependent cost maps," *IEEE Trans. Autom. Sci. Eng.* **12**(1), 171–182 (2015).
10. S. Brandi, O. Kroemer and J. Peters, "Generalizing Pouring Actions Between Objects using Warped Parameters," *Proceedings of the 14th IEEE-RAS International Conference on Humanoid Robots*, Madrid, Spain (2014) pp. 616–621.
11. M. S. Branicky, R. A. Knepper and J. J. Kuffner, "Path and Trajectory Diversity: Theory and Algorithms," *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Pasadena, CA, USA (2008) pp. 1359–1364.
12. A. Broun, C. Beck, T. Pipe, M. Mirmehdi and C. Melhuish, "Bootstrapping a robot's kinematic model," *Robot. Auton. Syst.* **62**(3), 330–339 (2014).
13. B. Castro da Silva, G. Konidaris and A. G. Barto, "Learning Parameterized Skills," *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, Edinburgh, Scotland (2012) pp. 1679–1686.
14. M. P. Deisenroth and C. E. Rasmussen, "PILCO: A Model-Based and Data-Efficient Approach to Policy Search," *Proceedings of the 28th International Conference on Machine Learning*, Bellevue, WA, USA (2011) pp. 465–472.
15. A. El-Fakdi and M. Carreras, "Two-step gradient-based reinforcement learning for underwater robotics behavior learning," *Robotics and Autonomous Systems* **61**(3), 271–282 (2013).
16. H. Esfandiar, S. Daneshmand and R. D. Kermani, "On the control of a single flexible arm robot via Youla-Kucera parameterization," *Robotica* **34**(01), 150–172 (2016).
17. D. H. Grollman and O. C. Jenkins, "Sparse Incremental Learning for Interactive Robot Control Policy Estimation," *Proceedings of the IEEE International Conference on Robotics and Automation*, Pasadena, CA, USA (2008) pp. 3315–3320.
18. L. Jamone, B. Damas and J. Santos-Victor, "Incremental Learning of Context-Dependent Dynamic Internal Models for Robot Control," *Proceedings of the IEEE International Symposium on Intelligent Control (ISIC)*, Antibes, France (2014) pp. 1336–1341.
19. S. M. Kakade, M. J. Kearns and J. Langford, "Exploration in Metric State Spaces," *Proceedings of the 20th International Conference on Machine Learning (ICML)*, Washington, D.C., USA (2003) pp. 306–312.
20. B. Kim, A. Kim, H. Dai, L. Kaelbling and T. Lozano-perez, "Generalizing over Uncertain Dynamics for Online Trajectory Generation," *Proceedings of the International Symposium on Robotics Research (ISRR)*, Sestri Levante, Italy (2015) pp. 1–16.
21. J. Kober, A. Wilhelm, E. Oztop and J. Peters, "Reinforcement learning to adjust parametrized motor primitives to new situations," *Auton. Robots* **33**, 361–379 (2012).
22. C. Lehnert and G. Wyeth, "Locally Weighted Learning Model Predictive Control for Nonlinear and Time Varying Dynamics," *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Karlsruhe, Germany (2013) pp. 2619–2625.
23. C. Lovell, G. Jones, K.-P. Zauner and S. R. Gunn, "Exploration and Exploitation with Insufficient Resources," *JMLR: Workshop and Conference Proceedings*, Bellevue, WA, USA, vol. 26 (2012) pp. 37–61.
24. J. Luo and K. Hauser, "Robust Trajectory Optimization Under Frictional Contact with Iterative Learning," Lydia E. Kavraki, David Hsu, and Jonas Buchli, editors. *Robotics: Science and Systems (RSS)*, Rome, Italy (2015) ISBN 978-0-9923747-1-6.
25. L. Mihalkova and R. Mooney, "Using Active Relocation to Aid Reinforcement Learning," *Proceedings of the 19th International FLAIRS Conference*, Melbourne Beach, FL, USA (2006) pp. 580–585.
26. T. Mihai Moldovan, S. Levine, M. I. Jordan and P. Abbeel, "Optimism-Driven Exploration for Nonlinear Systems," *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Seattle, WA, USA (2015) pp. 3239–3246.
27. I. Mordatch and E. Todorov, "Combining the Benefits of Function Approximation and Trajectory Optimization," Dieter Fox, Lydia E. Kavraki and Hanna Kurniawati, editors. *Robotics: Science and Systems (RSS)*, Berkeley, CA USA (2014) ISBN 978-0-9923747-0-9.
28. B. Nemec, D. Forte, R. Vuga, M. Tamosiunaite, F. Worgotter and A. Ude, "Applying Statistical Generalization to Determine Search Direction for Reinforcement Learning of Movement Primitives," *IEEE-RAS International Conference on Humanoid Robots*, Osaka, Japan (2012) pp. 65–70.
29. D. Nguyen-Tuong and J. Peters, "Model learning for robot control: A survey," *Cognitive Science* **12**(4), 319–40 (2011).
30. G. Pajak and I. Pajak, "Sub-optimal trajectory planning for mobile manipulators," *Robotica* **33**(06), 1181–1200 (2015).
31. C. Park, J. Pan and D. Manocha, "High-DOF robots in dynamic environments using incremental trajectory optimization," *Int. J. Humanoid Robot.* **11**(02) (2014).
32. P. Pastor, H. Hoffmann, T. Asfour and S. Schaal, "Learning and Generalization of Motor Skills by Learning from Demonstration," *Proceedings of the IEEE International Conference on Robotics and Automation, ICRA '09*, Kobe, Japan (May 2009) pp. 763–768.
33. J. Peters and S. Schaal, "Reinforcement learning of motor skills with policy gradients," *Neural Netw.* **21**, 682–697 (2008).


https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

34. T. Petrič, A. Gams, L. Žlajpah and A. Ude, "Online Learning of Task-Specific Dynamics for Periodic Tasks," *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2014)*, Chicago, IL, USA (2014) pp. 1790–1795.
35. M. Posa and R. Tedrake, "Direct Trajectory Optimization of Rigid Body Dynamical Systems Through Contact," **In**: *Algorithmic Foundations of Robotics X* (E. Frazzoli, T. Lozano-Perez, N. Roy, D. Rus, eds.), volume 86 (Springer Berlin Heidelberg, 2013) pp. 527–542.
36. C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning* (MIT Press, Boston, Massachusetts, United States, 2006).
37. C. Rosales, A. Ajoudani, M. Gabiccini and A. Bicchi, "Active Gathering of Frictional Properties from Objects," *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2014)*, Chicago, IL, USA (Sep. 2014) pp. 3982–3987.
38. L. Rozo, P. Jimenez and C. Torras, "Force-Based Robot Learning of Pouring Skills using Parametric Hidden Markov Models," *International Workshop on Robot Motion and Control, RoMoCo*, Wasowo, Poland (Jul. 2013) pp. 227–232.
39. J. Schulman, S. Levine, M. Jordan and P. Abbeel, "Trust Region Policy Optimization," *Proceedings of the International Conference on Machine Learning (ICML)*, Lille, France (2015) pp. 1889–1897.
40. N. Srinivas, A. Krause, S. M. Kakade and M. Seeger, "Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design," *Proceedings of the 27th International Conference on Machine Learning (ICML 2010)*, Haifa, Israel (2010) pp. 1015–1022.
41. M. Tamosiunaite, B. Nemec, A. Ude and F. Wörgötter, "Learning to pour with a robot arm combining goal and shape learning for dynamic movement primitives," *Robot. Auton. Syst.* **59**(11), 910–922 (2011).
42. E. Theodorou, J. Buchli and S. Schaal, "Learning Policy Improvements with Path Integrals," *International Conference on Artificial Intelligence and Statistics (AISTATS)*, Sardinia, Italy (2010).
43. Y. Zhang, J. Luo and K. Hauser, "Sampling-Based Motion Planning with Dynamic Intermediate State Objectives: Application to Throwing," *IEEE International Conference on Robotics and Automation (ICRA)*, St. Paul, MN, USA (2012) pp. 2551–2556.

## Appendix A. Gaussian Process Regression Method

GPR fits a Gaussian process over a set of training samples comprised of multi-dimensional inputs and real-valued outputs, and produces a predicted normal distribution for the output of any point in the input space. While a Gaussian process can be thought of as an infinite-dimensional normal distribution, when making predictions for new input points the formulation depends only on the training data, which makes the algorithm simple to implement. The standard training and prediction algorithm is derived in ref. [36] and gives efficient prediction for specified points in the input space. However, it does not provide any guidance on how to select test points. Here, we extend the basic algorithm to be applicable for our use case, where we are searching the input space for a point that corresponds to a desired output.

The standard algorithm takes a data set of previous trials, with inputs $\mathcal{X} \subset \mathbb{R}^n$, and outputs $\mathcal{Y} \subset \mathbb{R}$, and returns a predicted mean ($\mu_*$) and variance ($\sigma_*$) of the output for a test point, $x^{(*)}$. The most important component of the GP is the covariance function, $k : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, which controls the similarity of neighboring points in the input space and enforces the smoothness of functions that are fit to the training data. We use the standard squared exponential covariance function of the form:

$$k(\mathbf{x}, \mathbf{y}) = \sigma_f e^{(\mathbf{x}-\mathbf{y})^T D(\mathbf{x}-\mathbf{y})}. \tag{A1}$$

Here, $D$ is a symmetric, positive-definite matrix that defines a distance metric. In our experiments, we used a diagonal matrix with each diagonal element as a free parameter that controls how quickly the influence of two points drops as the distance between them increases along a particular axis. The other free parameter used is $\sigma_f$, which is a uniform scaling factor for the covariance.

The covariance function is used to compute correlations between the training set and the test point. The matrix $K \in \mathbb{R}^{m\times m}$ is the covariance matrix of the training data, computed by $K_{ij} = k(x^{(i)}, x^{(j)})$ for all $m$ points in $\mathcal{X}$. The covariance vector $k_*$ is computed by pairing $x^{(*)}$ with each $x^{(i)}$ from the training data. The predicted mean and variance of the output corresponding to $x^{(*)}$ are then given by

$$\begin{aligned} \mu_* &= k_*^T(K + \sigma_n^2 I)^{-1}\mathcal{Y} \\ \sigma_* &= k(x^{(*)}, x^{(*)}) - k_*^T(K + \sigma_n^2 I)^{-1}k_* \end{aligned} \tag{A2}$$

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press

Notice that the majority of the computation is inverting the matrix $K_n = K + \sigma_n^2 I$ which can be quite expensive if its dimensions are large. However, as $K_n$ is only dependent on the training data, it can be computed once and cached for future predictions of test points. This is crucial for efficient use of the prediction in our extension.

After a training set of data is obtained, prediction accuracy can be increased by finding a set of hyperparameters that maximizes the likelihood of the training data. In our case, the hyperparameters consist of the diagonal elements of $D$ and $\sigma_f$, forming the parameter vector $\theta$. The log-likelihood of the hyperparameters for a given training set can be analytically derived as

$$logp(\mathbf{y}|X, \theta) = -\frac{1}{2}\mathbf{y}^T K^{-1}\mathbf{y} - \frac{1}{2}log|K| - \frac{n}{2}log2\pi \tag{A3}$$

To maximize this function, a standard gradient ascent algorithm in the space of the parameter vector $\theta$ is used as the derivative can be analytically computed. For each parameter $\theta_i$, this derivative is

$$\frac{\partial}{\partial\theta_i}logp(\mathbf{y}|X, \theta) = \frac{1}{2}tr\left[\left(K^{-1}\mathbf{y}(K^{-1}\mathbf{y})^T - K^{-1}\right)\frac{\partial K}{\partial\theta_i}\right] \tag{A4}$$

Note that the calculation of $K$ (and therefore $K^{-1}$) is dependent on the values of $\theta$, essentially requiring a full retraining of the GP at each new point in the hyperparameter space. This cost is sufficient that it becomes prohibitive to reoptimize the hyperparameters every time new data is acquired, for example. However, we found it is possible to obtain sufficient prediction accuracy when finding the optimal parameters for the initial training set only.

https://doi.org/10.1017/S0263574717000212 Published online by Cambridge University Press