Delayed Gaussian Processes with Time Dependencies and Context

Ari Fiorino

CMU-CS-22-110
May 2022

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Aarti Singh
Jeff Schneider

Submitted in partial fulfillment of the requirements
for the Fifth Year Master's Program

Keywords: Additive Manufacturing, Gaussian Processes, Contextual Gaussian Processes, Time Series Optimization, Delayed Rewards

Abstract

This thesis presents a method to find a series of actions that optimizes a series of rewards. This is with the assumption that the rewards are only known periodically after a series of actions are taken, and that there is a time dependency between actions and rewards. This setting is motivated by an additive manufacturing problem where we first create an object (make actions), and then measure its properties (observe rewards). The method makes use of Contextual Gaussian Processes to make efficient and informative predictions from past training data. The method is shown to work on synthetic data and is compared to four other algorithms designed to solve the same problem. A greedy variation is described which performs much faster than the full version and has close to optimal performance. Finally, the method is applied to a COVID dataset to predict a sequence of COVID deaths given a sequence of COVID cases. The algorithm presented in this thesis is applicable in many other fields, and is capable of finding a quicker and better optimum than similar methods.

Acknowledgments

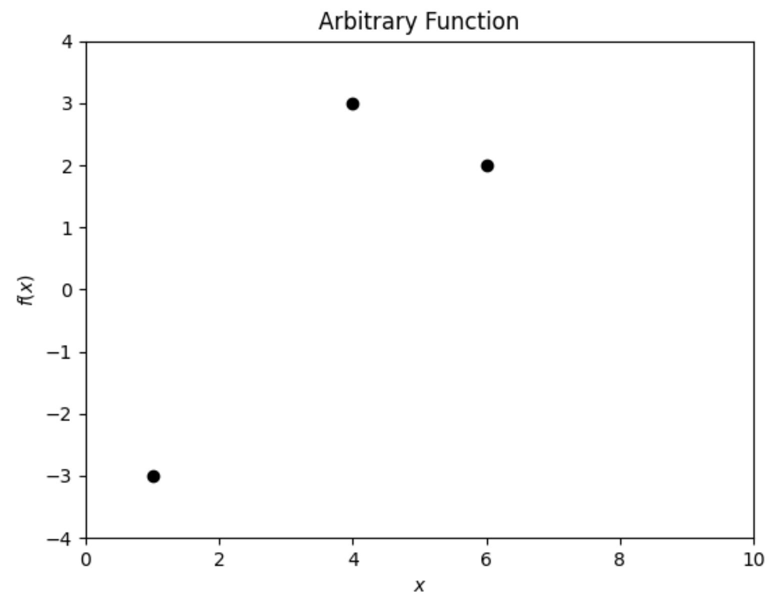
I would like to acknowledge Aarti Singh for guiding me through this research. She was tremendously helpful for suggesting ideas and next steps along the way. I would like to thank Ojash Neopane for meeting with me and Aarti weekly and for contributing ideas for algorithmic improvements. I also thank Morgan Chen for working with me on his chemical synthesis robot which led me to begin researching GPs. I also thank Reeja Jayan for guiding my research with Morgan. I also thank Benji Maruyama at AFRL for inviting me to visit the lab and allowing me to experiment with the AM robot. Finally I would like to thank Jim Denault for building the AM robot which led to this gaussian process method to be developed.

Contents

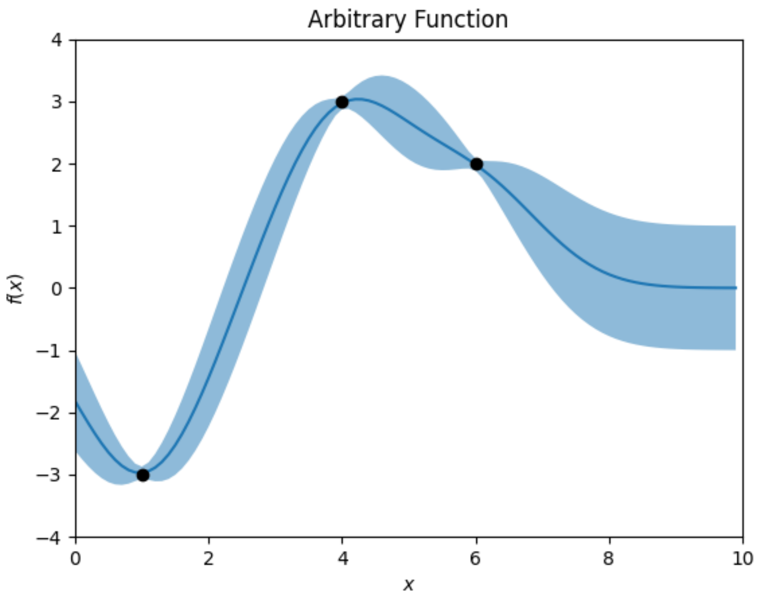
Chapter 1

Introduction

A Gaussian Process (GP) [4] takes as input a set of observations $(x, f(x))$ for some unknown function f:



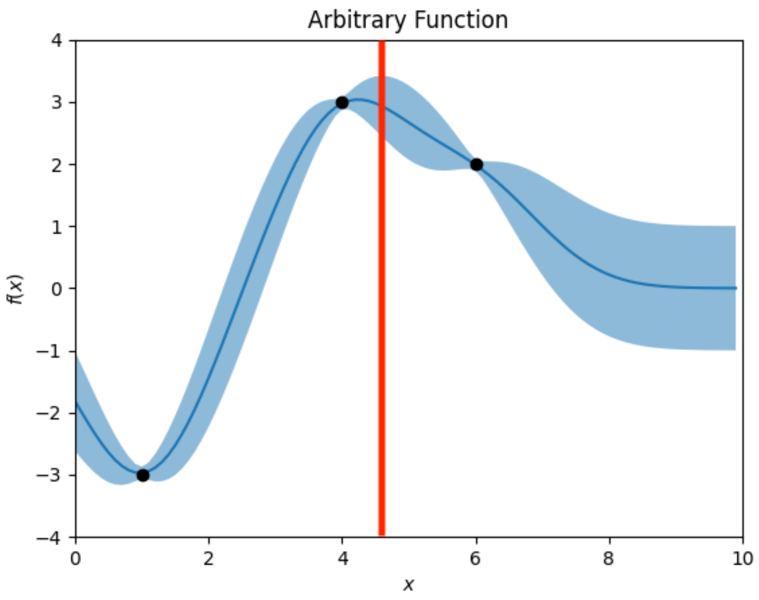
Then it creates an approximation of f, and the mean and uncertainty of its prediction f_μ, f_σ:



Note that f_σ is small around points that have been observed already.
Now, say we want to optimize a function f using a GP. This is where the Gaussian Process - Upper Confidence Bound (GP-UCB) algorithm comes into play. This algorithm picks as its next point:

$$\arg\max_x f_\mu(x) + m \cdot f_\sigma(x)$$

The m parameter is the exploitation-exploration tradeoff. It is picking a point that it knows will have a high value for f, but also a point that it is uncertain of, and wants to explore if it has a high value. For our f, it would choose this point as it's next query $(m = 1)$:



So, GP-UCB will be able to maximize any arbitrary function in this manner.
But there is a modification to GP-UCB where for each round the GP-UCB algorithm is tasked with finding the maximum, it is given a different context c and it has to find the maximum of $f(c, x)$ [3]. In this scenario, the Contextual GP-UCB (CGP-UCB) algorithm will approximate $f(c, x)$ as f_μ, f_σ and then choose:

$$\arg\max_x f_\mu(c, x) + m \cdot f_\sigma(c, x)$$

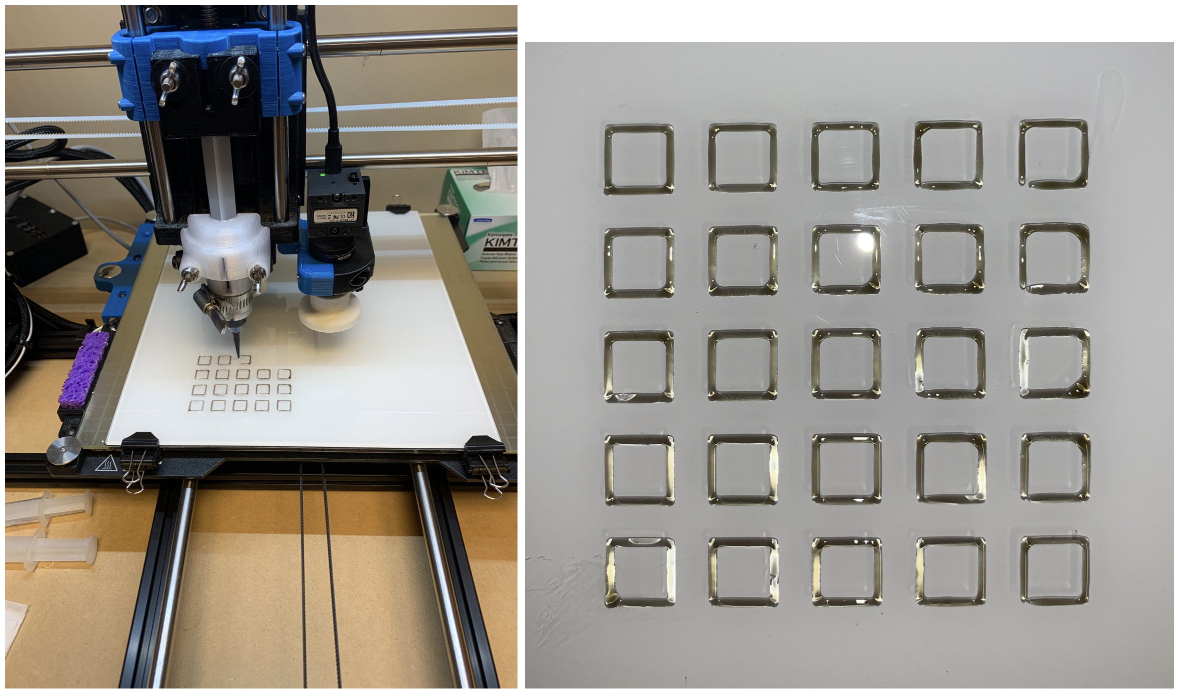
As its next point.

Chapter 2

Algorithm Description

2.1 Motivating Problem

The goal is to use Gaussian Processes to optimize additive manufacturing problems. Specifically, in the Wright-Patterson Air Force Base, there is a robot that which draws 2D shapes out of a viscous material in a syringe [1]:

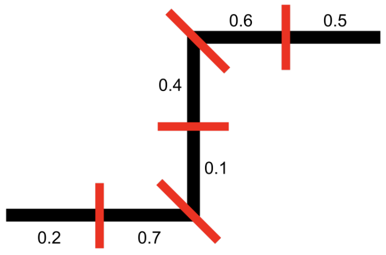


One can specify how much material to extrude along the shape and the robot has a camera which can take a picture of the finished shape. Some challenges are that there is a delay between extruding the viscous material and when the viscous material is actually extruded. This delay is hard to predict and would require predicting a different delay for different materials. This delay also changes depending on which shape we are drawing. Therefore an algorithm is presented which uses CGP-UCB to draw any shape with any material.

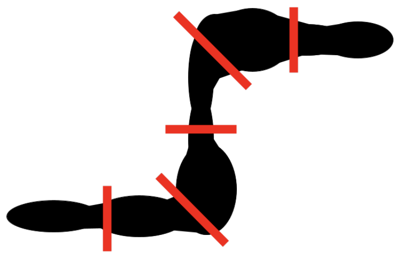
The algorithm is very broad, and only assumes a time dependency between actions and rewards, so it is able to run on a COVID dataset to predict a sequence of deaths from a sequence of cases.

2.2 Problem Formulation

Overall the robot will convert a sequence of extrusion amounts along the shape:



To a sequence of rewards according to how well each section was drawn:



This is formalized as follows:
In each episode,

1. The algorithm receives contexts $c_1, \dots, c_T$ which relate to the current shape we are drawing. This is the angle between the current section and the previous section. For example, in above it is $0, 0, -90°, 0, +90°, 0$.
2. The algorithm chooses actions $a_1, \dots, a_T$. This is the amount to extrude in each line segment.
3. The robot draws the shape
4. The algorithm receives rewards $r_1, \dots, r_T$ which is how well each segment is drawn.

2.3 Algorithm overview

For the algorithm, the assumption is that $r_t = f(a_t, \dots, a_{t-k}, c_t, \dots, c_{t-k})$ for some unknown function f and delay parameter k. One can approximate f with a GP on past data and get f_μ, f_σ.

Then the goal is to maximize the sum of rewards using CGP-UCB. Therefore the update is

$$\hat{a} = \arg\max_{a_1,\ldots,a_T} \sum_{t=1}^{T} f_\mu(a_t, \ldots, a_{t-k}, c_t, \ldots, c_{t-k}) + m \cdot f_\sigma(a_t, \ldots, a_{t-k}, c_t, \ldots, c_{t-k})$$

2.4 Baseline Algorithms

In order to prove the algorithm is optimal, four other algorithms are presented to compare against.

Algorithm 1 Here we assume $r_t = f(a_t, c_t)$ for some unknown f. We approximate f with a GP and end up with f_μ, f_σ. Then as our update, we pick

$$\hat{a}_t = \arg\max_{a_t} f_\mu(a_t, c_t) + m \cdot f_\sigma(a_t, c_t)$$

Algorithm 1' Here we assume there are T separate functions $f_1, \ldots, f_T$ and that $r_t = f_t(a_t, c_t)$. Each f_t only takes as input the action and context at time t and outputs the reward at time t. We approximate $f_1, \ldots, f_T$ with distinct GPs and end up with $f_{1,\mu}, f_{1,\sigma}, \ldots, f_{T,\mu}, f_{T,\sigma}$. Then as our update, we pick

$$\hat{a}_t = \arg\max_{a_t} f_{t,\mu}(a_t, c_t) + m \cdot f_{t,\sigma}(a_t, c_t)$$

Algorithm 2 Here we assume $\sum_{t=1}^{T} r_t = f(a_1, \ldots, a_T, c_1, \ldots, c_T)$ for an unknown function f. We approximate f with a GP and end up with f_μ, f_σ. Then as our update, we pick

$$\hat{a} = \arg\max_{a_1,\ldots,a_T} f_\mu(a_1, \ldots, a_T, c_1, \ldots, c_T) + m \cdot f_\sigma(a_1, \ldots, a_T, c_1, \ldots, c_T)$$

Algorithm 2' Here we assume $r_t = f(a_t, \ldots, a_{t-k})$ for some unknown function f and delay parameter k. One can approximate f with a GP on past data and get f_μ, f_σ. Then the goal is to maximize the sum of rewards using GP-UCB. Therefore the update is

$$\hat{a} = \arg\max_{a_1,\ldots,a_T} \sum_{t=1}^{T} f_\mu(a_t, \ldots, a_{t-k}) + m \cdot f_\sigma(a_t, \ldots, a_{t-k})$$

Algorithm 3 This is the algorithm described in section 2.3.

2.5 Greedy Algorithm

Algorithm 4 An issue observed with algorithm 3 is that it is slow to calculate the next actions. This is because we are doing an $\arg\max$ over all $a_1, \ldots, a_T$. Therefore the runtime is $O(c^T)$

for some constant c. Therefore there is a greedy version of algorithm 3 which first finds the best $a_1, \ldots, a_k$ to maximize

$$\underset{a_1, \ldots a_k}{\arg\max} f_\mu(a_k, \ldots, a_1, c_k, \ldots, c_1) + m \cdot f_\sigma(a_k, \ldots, a_1, c_k, \ldots, c_1)$$

Once $a_1, \ldots, a_k$ are found, a_{k+1} is calculated as

$$\underset{a_{k+1}}{\arg\max} f_\mu(a_{k+1}, \ldots, a_2, c_{k+1}, \ldots, c_2) + m \cdot f_\sigma(a_{k+1}, \ldots, a_2, c_{k+1}, \ldots, c_2)$$

Then a_{k+2} is calculated as

$$\underset{a_{k+2}}{\arg\max} f_\mu(a_{k+2}, \ldots, a_3, c_{k+2}, \ldots, c_3) + m \cdot f_\sigma(a_{k+2}, \ldots, a_3, c_{k+2}, \ldots, c_3)$$

And so on...

This makes algorithm 3 run in $O(c^k)$ while algorithm 4 runs in $O(c^T)$ for some constant c.

This makes algorithm 4 much faster to run than algorithm 3, over 1000x faster in practice. But there is a performance loss associated with this greedy algorithm, it is not actually finding the optimal sequence of actions to optimize the cumulative UCB. This is just an approximation. Therefore another algorithm is introduced which is slightly slower than algorithm 4, but has better performance:

Algorithm 4' First run algorithm 4, then find if changing any one action increases the cumulative UCB. To check action a_t, check if the sum of all the UCB components that include a_t is higher than the previous sum. There are k of these components to check. Therefore a runtime of $O(kT)$ is added. Since this is a polynomial amount of time, the added time is negligible. Algorithm 4' is still around 1000x faster than algorithm 3.

Chapter 3

Synthetic Results

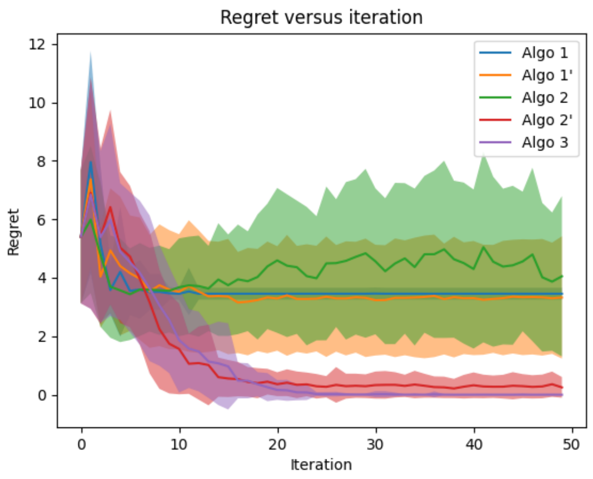
3.1 Experimental Setup

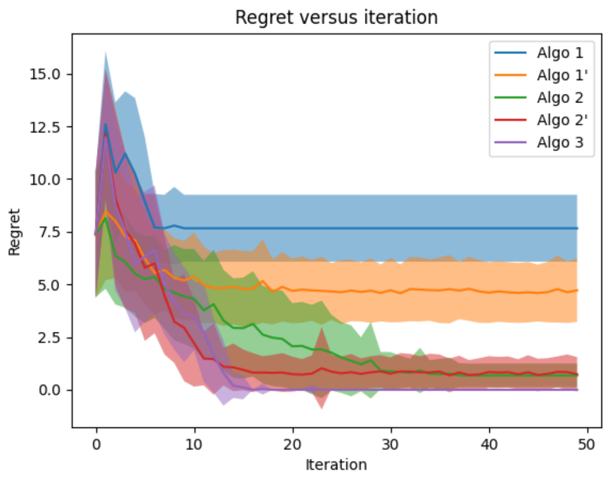
Experiments were ran on synthetic data to compare the algorithms. These were the experimental parameters: $T = 5, \{c_1, \ldots, c_5\} = \{1, 2, 1, 2, 1\},\ k = 3,\ m = 5,\ a_t \in \{1, \ldots, 5\}$. The kernel function for the GP is an RBF kernel with $\sigma = 2$. We first generate a random function $f : \{1, \ldots, 5\}^k \times \{0, 1\} \to \mathbb{R}$. This function is represented as a vector of length $5^3 \cdot 2$, because every input needs to have an output defined. Then a covariance matrix was created using the RBF kernel between every possible input of f. Then `np.random.multivariate_normal` is used with this covariance matrix to generate f as a vector. Then g is created, the function to optimize. $g : \{1, \ldots, 5\}^T \to \mathbb{R}^T$. g is defined as $(g(a_1, \ldots, a_T))_t = f(a_t, \ldots, a_{t-k+1}, c_t)$. A modulus is used so that $a_0 = a_5$.

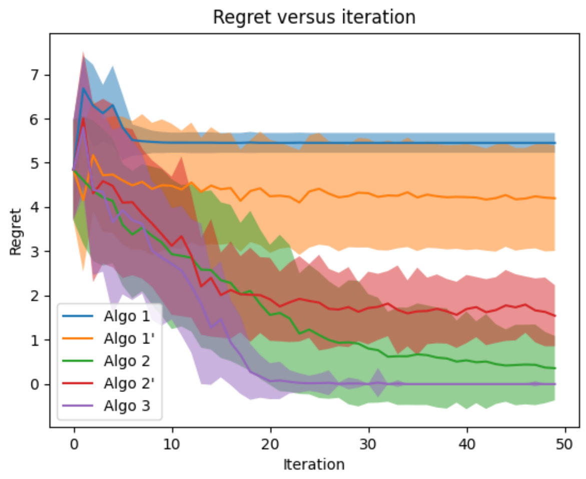
The experimental setup for comparing the algorithms is as follows. First pick a random starting point for the algorithms. This means setting random values for $a_1, \ldots a_5$. Then run all algorithms starting with the same starting point for 100 iterations each. Once this is done, pick another starting point and repeat the process. After going through the process 100 times, the mean and standard deviation of the regret per iteration is plotted for the algorithms.

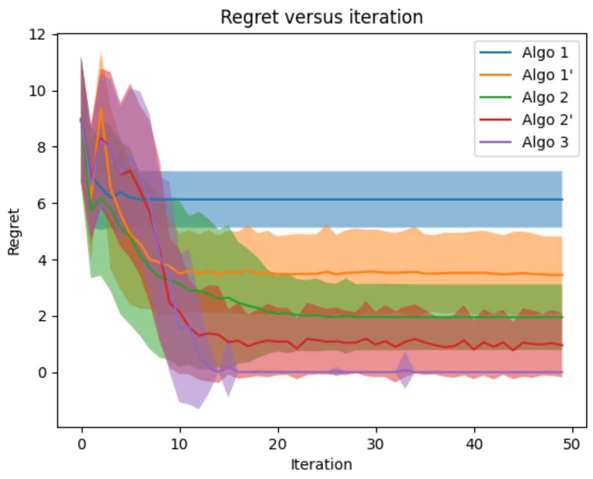
3.2 Results

These are the plots comparing the performance of the algorithms. Each plot has a randomly generated target function.

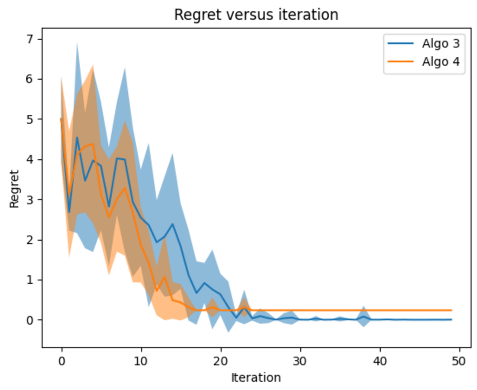


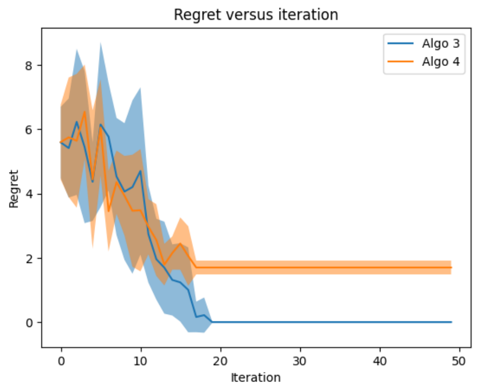


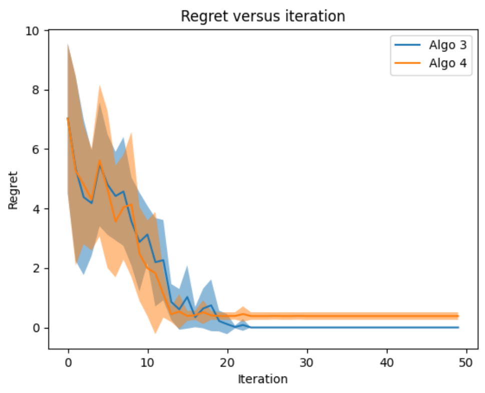


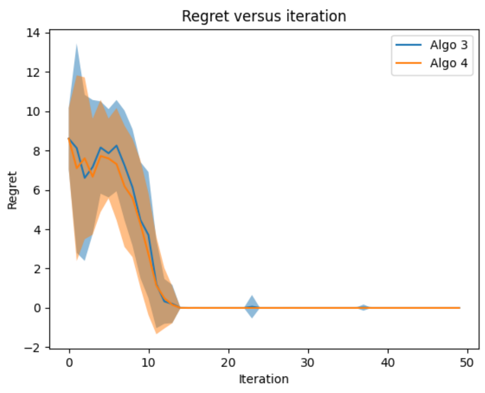


This is the comparison from algorithm 3 to algorithm 4 on four different target functions.

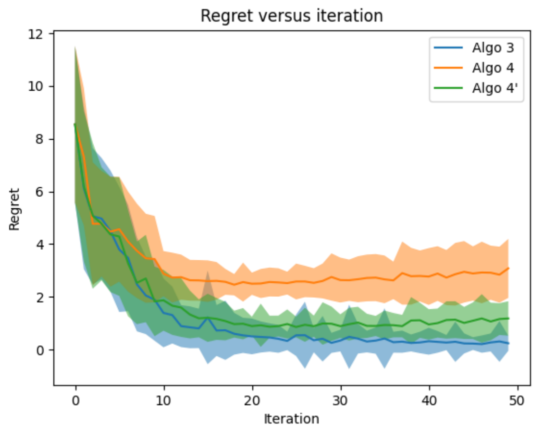








This is the comparison from algorithm 4 to algorithm 4'.



For this experiment, algorithm 3 took on average 133.93s, algorithm 4 took 0.11s, and algorithm 4' took 0.15s to run.

3.3 Explanation of Results

We see that Algorithm 3 performs much better than Algorithms 1, 1', 2, and 2'. Here are the downsides of those algorithms, and why they perform worse:

Algorithm 1 This algorithm only assumes that, $r_t = f(a_t, c_t)$ for some function f. Since this f is the same for all t, if $c_t = c_{t'}$ for some t, t', then $a_t = a_{t'}$. This means that in the additive manufacturing context, all segments with the same context will have the same extrusion amount. This limits the set of possible action sequences, and often excludes the optimal action sequence.

Algorithm 1' This algorithm assumes that $r_t = f_t(a_t, c_t)$ for some functions $f_1, \ldots, f_T$. In this case, there are T separate GPs, one for each action. Let's see an example where we $T = 5$ and we use a constant context $c_1 = c_2 = \cdots = c_5$ and our past experiments are $[a_1, \ldots, a_5] = [1, 2, 3, 4, 5] \rightarrow [r_1, \ldots, r_5] = [3, 3, 3, 3, 3]$ and $[a_1, \ldots, a_5] = [1, 3, 3, 4, 4] \rightarrow [r_1, \ldots, r_5] = [4, 4, 4, 4, 4]$. In this case, conflicting training points are passed into the GPs. For example, the

GP for a_1 sees $1 \to 3$ and $1 \to 4$ as its training points. These conflicting training points for GPs cause this algorithm not to work optimally.

Algorithm 2 In this algorithm, $\sum_{t=1}^{T} r_t = f(a_1, \ldots, a_T, c_1, \ldots, c_T)$. This assumes a full dependency between actions and rewards. Therefore a_5 can affect r_1 even though action 5 is taken after reward 1. This is a very large GP that will take a long time to converge, making it inefficient.

Algorithm 2' In this algorithm, $r_t = f(a_t, \ldots, a_{t-k})$. The only downside to this algorithm is that it doesn't take into account the context associated with each action. This context adds information about the shape being drawn. Without the context, it assumes all shapes should be drawn the same way.

We see that the performance of Algorithm 4 depends on which target function we are optimizing over. This is expected of a greedy algorithm. Also Algorithm 4 is much more scalable as we can scale T to be very large while having a constant k. Algorithm 4' is always performs better or the same as Algorithm 4, and it only has a slightly higher runtime.

3.4 Related Work

In order to solve our problem formulation, we used Gaussian Processes to approximate our time dependency function. Gaussian processes as originally formulated [4] were useful, but we needed to include a context in order to draw different shapes. This is where Contextual Gaussian Processes [3] were used and they were modified to allow for delayed rewards and for time dependencies. We passed our shape parameters as the context and optimized drawing our shape based on that context.

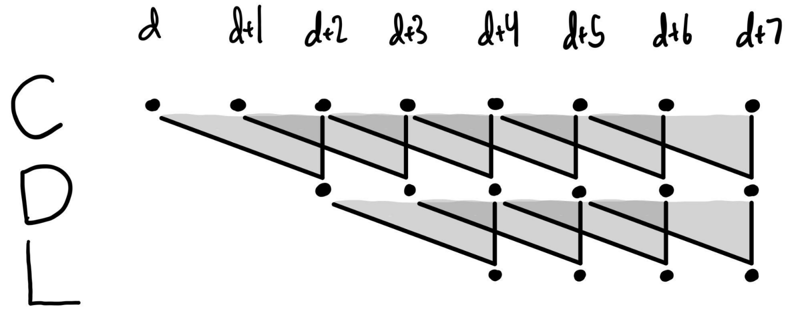
Another related line of work is reinforcement learning (RL) [5], which is a class of machine learning algorithms with the following problem formulation. There is a current state s. The algorithm picks an action a for the state s. The algorithm receives a reward r for taking that action for that state. Then the algorithm receives a new state s'. Then the algorithm continues at state s'. We can formulate our problem as an RL problem. We have as our state the past actions we've taken in the episode and the current context. After taking actions $a_1, \ldots, a_{T-1}$, we receive a reward of $\vec{0}$. Then after taking action a_T, we receive a reward of $[r_1, \ldots, r_T]$. We need to formulate it in this way because we only receive the rewards after taking all the actions. We need to have a vector of rewards because we need to include the time dependencies of each reward. There is a limited amount of work on reinforcement learning with a vector of rewards. Possibly this algorithm could be applied in a reinforcement learning setting.

Chapter 4

COVID Dataset

4.1 Experimental setup

After running synthetic experiments, the goal was to apply the algorithm on a real world dataset. The dataset from the CDC of COVID cases and deaths per day was used. This dataset was chosen because there is proven lag between cases and deaths of 8.053 ± 4.116 days [2]. In our case, we will be predicting a sequence of deaths from a sequence of cases. The dependencies are demonstrated in this image:



The deaths on day d depend on the past 3 days of cases. Say on day d we predict deaths x_d, the actual deaths are $\hat{x}_d$, and our loss is ℓ_d. Then we have

$$\ell_d = |x_d - \hat{x}_d| + |x_{d-1} - \hat{x}_{d-1}| + |x_{d-2} - \hat{x}_{d-2}|$$

This loss function was chosen to include a time dependency associated with it. This assumption is necessary for our algorithm to work. Then experimental setup is as follows:

- Pick a random day d
- Algorithm receives cases for $d, \dots, d+7$
- Algorithm predicts deaths for $d+2, \dots, d+7$
- Algorithm receives losses for $d+4, \dots, d+7$

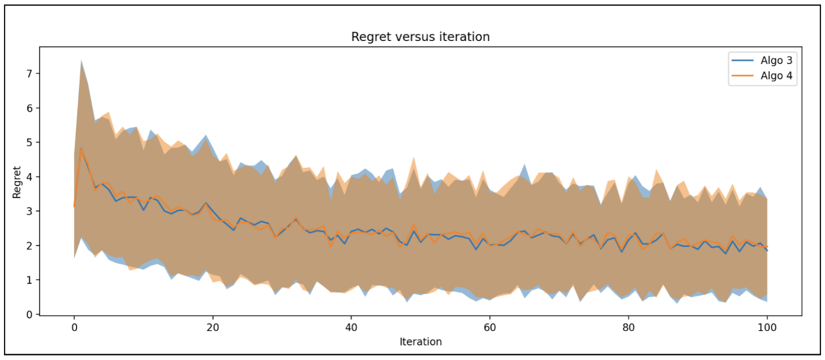
The algorithm knows that $\ell_d = f(x_d, x_{d-1}, x_{d-2}, \hat{x}_d, \hat{x}_{d-1}, \hat{x}_{d-2})$ It also knows that $\hat{x}_d = g(c_d, c_{d-1}, c_{d-2})$ where c is cases. Therefore it uses this assumption

$$\ell_d = h(x_d, x_{d-1}, x_{d-2}, c_d, c_{d-1}, c_{d-2}, c_{d-3}, c_{d-4})$$

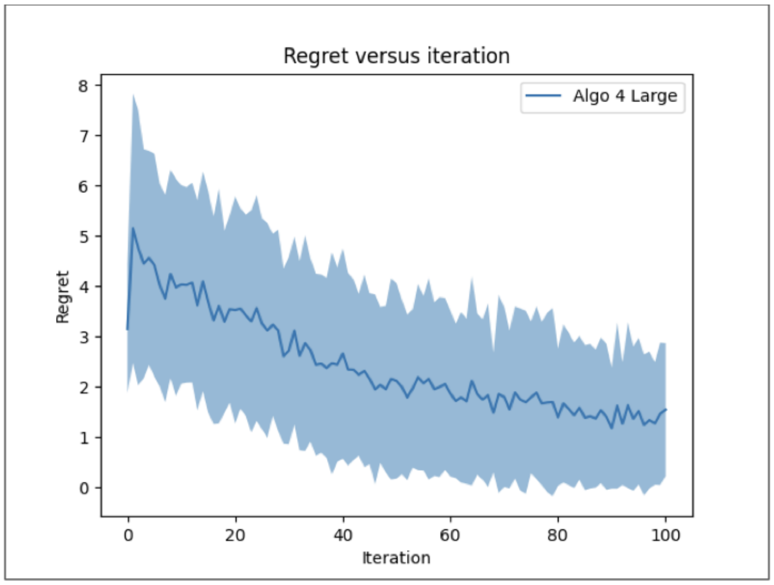
It approximates h with a GP and then minimizes it. Also, the deaths are split into bins so the algorithm only optimizes over a small number of deaths. For example the algorithm can only choose deaths $\{0, 500, 1000, 1500, 2000\}$.

4.2 Results

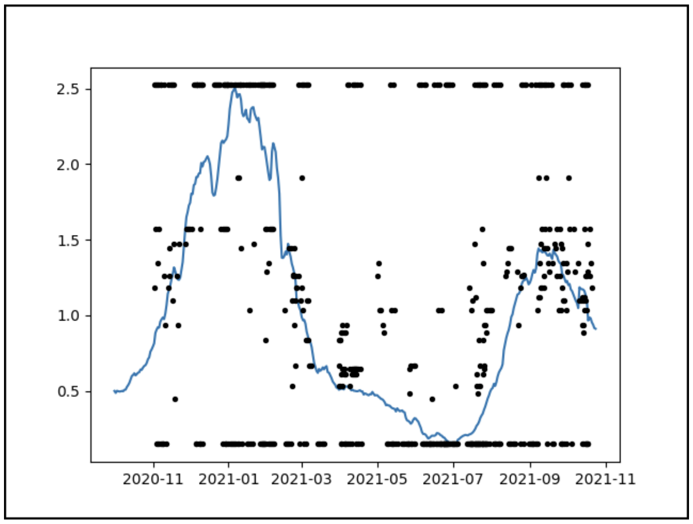
First, algorithms 3 and 4 were ran on the covid dataset ($k = 3$ with 5 bins):



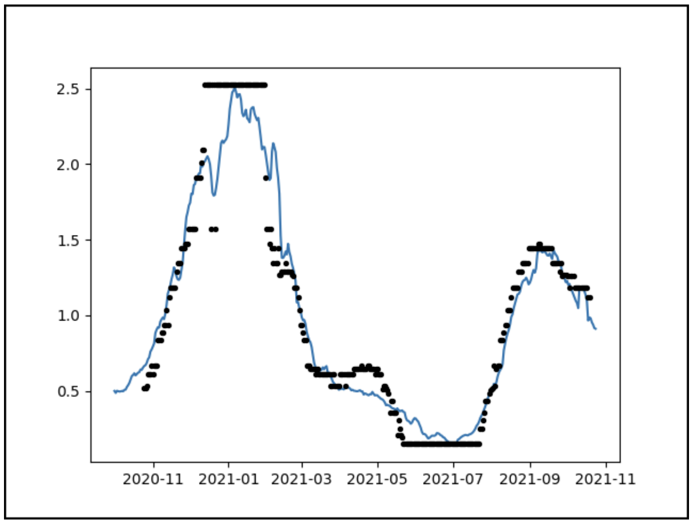
Since they performed equally well but algorithm 4 is much faster to run, algorithm 4 was used from this point forward. Algorithm 4 was ran on a larger scale ($k = 24$ with 40 bins):



The loss was halved from random which is a good result. In this graph, the blue line is actual deaths and the black dots are deaths the algorithm queried while running GP-UCB:



And here are the final deaths it predicted:



We can see that it closely resembles the actual deaths on those days.

Chapter 5

Conclusion

In conclusion, this thesis presented a method of optimizing a series of rewards from a series of actions using GP-UCB. This method is shown to find the optimal faster than four status quo algorithms on synthetic data. Then a greedy version of the method is presented which is much faster to run and has only slightly worse performance. Then the algorithm was applied to a covid dataset to predict a sequence of deaths from a sequence of cases and the algorithm was shown to work optimally. Future work would include running the algorithm on an additive manufacturing robot or another problem with the time dependency assumptions. Future work also includes finding faster algorithms like algorithm 4'.

The full code is available at: `https://github.com/arifiorino/Modified-GP`

Bibliography

[1] James Deneault, Jorge Chang, Jay Myung, Daylond Hooper, Andrew Armstrong, Mark Pitt, and Benji Maruyama. Toward autonomous additive manufacturing: Bayesian optimization on a 3d printer. *MRS Bulletin*, 46, 04 2021. doi: 10.1557/s43577-021-00051-1. 2.1

[2] Raymond Jin. The lag between daily reported covid-19 cases and deaths and its relationship to age. *Journal of Public Health Research*, 10(3):jphr.2021.2049, 2021. doi: 10.4081/jphr.2021.2049. URL `https://doi.org/10.4081/jphr.2021.2049`. PMID: 33709641. 4.1

[3] Andreas Krause and Cheng Ong. Contextual gaussian process bandit optimization. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011. URL `https://proceedings.neurips.cc/paper/2011/file/f3f1b7fc5a8779a9e618e1f23a7b7860-Paper.pdf`. 1, 3.4

[4] Niranjan Srinivas, Andreas Krause, Sham M. Kakade, and Matthias W. Seeger. Gaussian process bandits without regret: An experimental design approach. *CoRR*, abs/0912.3995, 2009. URL `http://arxiv.org/abs/0912.3995`. 1, 3.4

[5] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018. ISBN 0262039249. 3.4