Jointly Learning Prices and Product Features

Ehsan Emamjomeh-Zadeh [*], **Renato Paes Leme** [†], **Jon Schneider** [‡], **Balasubramanian Sivan** [§]

Abstract

Product Design is an important problem in marketing research where a firm tries to learn what features of a product are more valuable to consumers. We study this problem from the viewpoint of online learning: a firm repeatedly interacts with a buyer by choosing a product configuration as well as a price and observing the buyer's purchasing decision. The goal of the firm is to maximize revenue throughout the course of T rounds by learning the buyer's preferences. We study both the case of a set of discrete products and the case of a continuous set of allowable product features. In both cases we provide nearly tight upper and lower regret bounds.

1 Introduction

In "The Ketchup Conundrum" [Gladwell, 2004] Malcolm Gladwell describes the history of market research in the food industry and the sophisticated scientific techniques used by companies like Campbell Soup and Pepsi to optimize their products to the taste of customers. In his tale about mustard, Gladwell describes how the mustard brand Grey Poupon dominated the market in the 1980s and was able to charge more than twice the price of its competitors by understanding the properties of mustard that were most desired by the market and offering a mustard product that was substantially different from its competitors.

Pricing is an important component of revenue optimization, but one equally important component is to optimize the product being offered to increase its value to consumers: How sweet should soda be? How big should the fonts be in a website to maximize engagement? What color of orange juice is the best to boost sales? Often, the design space is quite complex. Gladwell describes the design space of pasta sauce as follows: "*These were designed to differ in every conceivable way: spiciness, sweetness, tartness, saltiness, thickness, aroma, mouth feel, cost of ingredients, and so forth.*"

[*]Facebook Research, ehsanez@fb.com

[†]Google Research, renatoppl@google.com

[‡]Google Research, jschnei@google.com

[§]Google Research, balusivan@google.com

In this paper we explore the question of optimizing revenue by optimizing both the pricing policy and the features of the product itself. We look at this question from the viewpoint of online learning, or more specifically, online contextual pricing. In contextual pricing a seller is presented with differentiated products represented by feature vectors and then proceed to sell them adaptively. When deciding on a price to sell each product, the seller faces a trade-off usually referred as learn/earn: lower prices are more likely to sell and guarantee revenue but more aggressive prices will allow the seller to learn more about the buyer's value and this knowledge can boost future revenues. There has been a lot of activity on contextual pricing recently [Amin *et al.*, 2014; Bastani and Bayati, 2016; Cohen *et al.*, 2016; Javanmard and Nazerzadeh, 2016; Lobel *et al.*, 2017; Javanmard, 2017; Leme and Schneider, 2018; Mao *et al.*, 2018; Liu *et al.*, 2020] but in all those papers the seller optimizes the pricing strategy with no control of which products he is selling.

In our model, we assume that there is a space $X \subseteq \mathbb{R}^d$ of products that can be offered by the firm represented by feature vectors, e.g. X is the set of all the potential flavors of pasta sauce that can be produced and the features correspond to things like spiciness, sweetness, tartness, etc. The seller then repeatedly interacts with a buyer with valuation function $\nu : X \to \mathbb{R}$ by choosing a product $x_t \in X$ to offer and a price p_t and observes whether the buyer purchases the product ($p_t \leq \nu(x_t)$) or not ($p_t > \nu(x_t)$). The buyer is here assumed to be myopic and have a fixed valuation.

The learning task we consider is to optimize the revenue achieved by the seller while learning. Formally, we minimize regret during a course of T rounds defined as the difference between the seller's revenue and the best possible revenue in T rounds that can be extracted if the seller knew ν.

Our Results We start by studying the case where there are d independent products, e.g., $X = \{e_1, \ldots, e_d\}$ where e_i the i-th vector in the standard basis for $\mathbb{R}^d$. The simplest algorithm for this problem is to do binary search to determine the value of each item within $1/T$ precision and then offer the highest value product at the lower estimate. The total regret in this approach is $O(d \log T)$. Using a surprising idea by [Kirkpatrick and Gao, 1990] it is possible to identify the best product together with its value with only $O(d + \log T)$ queries. We also show that, using a different algorithm, we can exponentially improve the dependence on T. More pre-

cisely, we obtain a regret bound of $O(d \log\log T)$ using a parallelized version of the algorithm of [Kleinberg and Leighton, 2003] for a single item. Since our benchmark is the value of the best product, it is not enough to run separate instances of Kleinberg and Leighton's algorithm for each product. Instead we need to synchronize different instances to make sure the regret bounds are valid across products.

While our upper bound results are relatively straightforward applications/extensions of existing algorithms, our main question is whether one can do better. Namely, given that it is possible to obtain regret $O(d \log\log T)$ and $O(d + \log T)$, it is natural to ask whether it is possible to obtain the best qualities of both bounds: a $\log\log$ dependency on T *and* an additive dependency on d. In other words, is $O(d+\log\log T)$ achievable for this problem? We answer this in the negative.

Our main result and technical contribution is a lower bound on regret showing that if $T = 2^{2^d}$, then any deterministic algorithm must incur $\Omega(d^2/\log d)$ regret. In other words, for large T the parallel version of Kleinberg-Leighton is optimal up to $\log$ factors. The construction for the lower bound is quite intricate.

In the second part of the paper, we consider a general set $X \subseteq \mathbb{R}^d$. We assume that we have an oracle that given a direction $v \in \mathbb{R}^d$, we can efficiently compute a point in $\operatorname{argmax}_{x\in X} \langle x, v\rangle$ and obtain an algorithm with regret $O(d \log d \log\log T)$. We obtain this result by a reduction to the Contextual Pricing problem [Leme and Schneider, 2018; Liu *et al.*, 2020].

1.1 Related Work

Product Design is a formulation of a traditional problem in marketing research from the perspective of online learning. A large body of literature in marketing has been dedicated to developing techniques to understand the preferences of consumers. Particularly close to our problem is the technique known as *cojoint analysis* [Green and Srinivasan, 1978]. The goal of this technique is to design surveys to be sent to consumers and analyze those in order to infer what product attributes are more influential in purchase decisions. We refer to the excellent paper by Toubia, Hauser and Simester [Toubia *et al.*, 2004] as a great introduction to the field. Methods for cojoint analysis have been derived from polyhedral techniques [Toubia *et al.*, 2004; Toubia *et al.*, 2003], convex programming [Evgeniou *et al.*, 2007], statistics [Ding *et al.*, 2005] and more recently machine learning [Chapelle and Harchaoui, 2005; Viappiani and Boutilier, 2010].

We depart from this line of literature in two major ways: (i) we focus on revenue and on the tradeoffs between learning and earning, while the traditional focus of marketing has been in learning the weights that consumers attribute to each feature; (ii) our methodological approach is rooted in online learning: we have clearly specified loss functions and obtain provable guarantees on what is achievable for such loss functions.

Our methodological approach is closely related to the literature on contextual pricing, in which differentiated products are presented to a seller who then chooses a price based on the observable features of the product. Different techniques have been applied to this problem leading to different regret bounds: stochastic gradient descent [Amin *et al.*, 2014], statistical learning [Bastani and Bayati, 2016; Javanmard and Nazerzadeh, 2016; Javanmard, 2017], iterative partition refinement [Mao *et al.*, 2018] and high-dimensional convex geometry [Cohen *et al.*, 2016; Lobel *et al.*, 2017; Leme and Schneider, 2018]. We differ from this line of work both in the benchmark against which we compute regret and more importantly in the decision variable. While in contextual pricing the decision maker is only asked to choose a price in each step, in our setting we choose a price and a product configuration.

2 Problem Definition

Product Design is an online decision making problem in which a firm repeatedly interacts with a buyer to sell an item from a fix set of items. The firm knows the set of products that it is capable of producing. This set of products will be represented by a set $X \subseteq \mathbb{R}^d$ where each product $x \in X$ is identified by a set of d features. The buyer's value for each product is given by a function $\nu : X \to \mathbb{R}$ that is fixed but unknown to the firm.

At every period $t = 1, 2, \ldots, T$, the firm chooses a product x_t together with a price p_t and offers x_t at price p_t to the buyer. The buyer accepts the offer if $p_t \le \nu(x_t)$ and rejects otherwise. The goal of the firm is to minimize its regret with respect to the maximum obtainable revenue, which is: $T \cdot \max_{x\in X} \nu(x)$. Formally,

$$\mathsf{Regret} = T \cdot \max_{x\in X} \nu(x) - \sum_{t=1}^{T} p_t \cdot \mathbf{1}\{p_t \le \nu(x_t)\}$$

We will make the common assumption that the function $\nu(x)$ is linear in the feature space, i.e., $\nu(x) = \langle v, x\rangle$ for a certain vector $v \in \mathbb{R}^d$. To fix the scale of the problem we will also assume that $\|v\|_2 \le 1$. As usual, this can be generalized to more complex functions by mapping the features to a different space[1].

First we consider the case of *Independent Products* in which $X = \{e_1, \ldots, e_d\}$ and e_i is the i-th coordinate vector. In this case what we learn fro pricing a certain product says nothing about the value of the buyer for other products. Next we consider *General Products* in which X is an arbitrary set with $\|x\|_2 \le 1, \forall x \in X$ for which we have an oracle that returns $\operatorname{argmax}_{x\in X} \langle v, x\rangle$ for any $v \in \mathbb{R}^d$.

A natural algorithm is to try to find the best product as fast as possible and the price it at its lower estimate.

Lemma 1. *If it is possible to find a pair $(\hat{x}, \hat{\nu})$ such that*

$$\nu^* - \epsilon \le \hat{\nu} \le \nu(\hat{x}) \le \nu^* \text{ where } \nu^* = \max_{x\in X} \nu(x)$$

in $f(\epsilon)$ iterations, then it is possible to obtain an $1 + f(1/T)$ regret algorithm for Product Design.

[1] In particular, this is easily generalizable to any functions of the form $\nu(x) = f(\langle \phi(x), v\rangle)$ for $\phi : \mathbb{R}^d \to \mathbb{R}^{d'}$ and $f : \mathbb{R} \to \mathbb{R}$. Those include polynomials in x, logistic regression $v = [1 + \exp(\langle v, x\rangle)]^{-1}$, the log-log model $\log v = \sum_i v_i \log(x_i)$, the semilog model $\log v = \sum_i v_i x_i$, among others.

Proof. Use the first $f(1/T)$ iterations to obtain a pair $(\hat{x}, \hat{\nu})$ incurring regret at most $f(1/T)$. From this point on, choose $(\hat{x}, \hat{\nu})$ in all remaining steps, always having the offer accepted and incurring regret at most $T \cdot (\nu^* - \nu(\hat{x})) \leq T \cdot 1/T = 1$. □

In many cases, however, it is possible to obtain exponentially better regret for Product Design than what is possible to obtain via a direct reduction to Product Finding.

3 Independent Products

In this section we explore algorithms for the Product Design problem when there are d "independent" products such that the value of any subset of products does not reveal any information about any other product. In the language of this paper, one can assume that the products are (for example) the d unit vectors in the standard basis of $\mathbb{R}^d$. We derive both upper bounds (relatively straightforward extensions) and lower bound (main result, involved) on the regret.

3.1 Upper Bounds

One naive approach to this problem is to estimate the value of each product to within $1/T$ additive error in $\log T$ rounds (using a binary search), and after that, always sell the (approximately) best product at its estimated price. Because it takes $d \log T$ rounds to approximate the value of each item, this algorithm achieves a regret of $d \log T + 1$. We present two algorithms which improve on this upper bound in incomparable ways: one achieves regret $O(d \log \log T)$ while the other one achieves an upper bound of $O(d + \log T)$.

We begin by showing how to get $O(d \log \log T)$ total regret by parallelizing an algorithm of Kleinberg and Leighton [Kleinberg and Leighton, 2003] which achieves a regret upper bound of $O(\log \log T)$ for a single item.

Kleinberg-Leighton (KL) Algorithm for One Item

In the KL setting, there is only one item, and the buyer has a fixed unknown value $v^* \in [0, 1]$ for this item. Within a course of T rounds, the algorithm chooses a price p_t in each round t, and observes whether the offer is accepted ($p_t \leq v$) or not ($p_t > v$). The regret is defined as $\mathsf{Regret} = Tv - \sum_{t=1}^{T} p_t \cdot \mathbf{1}[p_t \leq v]$. We describe the KL algorithm and the proof of the regret it obtains in the supplement for completeness. The main takeaway we need from the KL algorithm is that it obtains a regret of $O(\log \log T)$.

First Algorithm: Parallelization of the Kleinberg-Leighton Algorithm

We now show how we generalize this algorithm for d independent items $X = \{e_1, \ldots, e_d\}$. A naive idea is to run the KL algorithm on each item individually until we find its approximate value and then offer the best item for the rest of the rounds. This approach incurs linear regret since the benchmark is the value of top item and any accepted offer for a suboptimal item generates regret at least equal to its difference compared to the optimal item.

Instead, we run the KL algorithm for all items in lockstep. To be precise, we keep a set S of item that are alive, initially containing all items. We also keep an instance of the (single-item) KL algorithm which we will use as a black-box. The algorithm will proceed in phases. Each phase will correspond to one step of the black-box KL algorithm which will correspond to $|S|$ price offerings in the actual algorithm. In each phase s, we get a price p_s from the KL black-box and offer this price to each item in S. Then we proceed as follows:

- If all offers are rejected, we feed the KL algorithm with "rejected" and move to the next phase without changing S.
- If at least one offer is accepted, we feed the KL algorithm with "accepted", remove from S all items for which the offers were rejected and proceed to the next phase.

Theorem 2. *The parallelized KL algorithm achieves $O(d \log \log T)$ total regret on the Product Design problem with independent items.*

We prove Theorem 2 in the online supplement.

Second Algorithm: Binary Search Based Algorithm

The second algorithm is based on a clever procedure by Kirkpatrick and Gao [Kirkpatrick and Gao, 1990]. Their algorithm considers the following problem: given d integers $x_1, \ldots, x_d$ between 1 and n we want to determine $\max_i x_i$ using only queries of the type $x_i \geq p$. Their algorithm finds the maximum in $O(d + \log n)$ steps. This is essentially the Product Finding problem for independent product and their algorithm automatically implies a $O(d + \log(1/\epsilon))$ mistake bound, which translates to an $O(d + \log(T))$ regret bound for Product Design via Lemma 1. We sketch the idea behind the algorithm of Kirkpatrick and Gao in the supplement.

3.2 Lower Bound

In the previous section we demonstrated algorithms for the Product Design problem which achieve regret $O(d \log \log T)$ and $O(d + \log T)$ respectively. Given these two upper bounds, a natural question is whether it is possible to obtain the best properties of both bounds, i.e., an algorithm that achieves a regret of $O(d + \log \log T)$. In this section we show that a $O(d + \log \log T)$ bound is *not* achievable by any deterministic algorithm: we show this by considering a setting with d products, $T = 2^{2^d}$ and showing that any deterministic algorithm must incur $\Omega(d^2 / \log d)$ regret. This is the main result of the paper and goes through an involved construction.

We establish some facts about arbitrary deterministic algorithms for the Product Design problem in the form of Lemmas 3 and 4. Since regret is additive, we focus on the case where all values are in $[1, 2]$ for the lower bound (unlike the upper bound sections where we focused on values in $[0, 1]$).

Deterministic algorithms as binary trees. A deterministic algorithm for the Product Design problem is simply a sequence of queries of the form "is $v(x_i) \geq p$?", where each query is possibly dependent on the sequence of queries previously asked and responses received (where the i and p are decided by the algorithm). This is readily portrayed as a

binary tree. Each non-leaf node of this binary tree represents the query (i.e. the pair (i, p)), and every such node has at least one and at most two child edges: YES and NO[2]. Every path in this tree is an execution of this algorithm on a given input instance. Note that two instances which are sufficiently close could result in the same path being realized, but a given input instance results in a unique path in this binary tree.

Adversary's task. The adversary's task is to construct an input instance with regret $\Omega(d^2/\log d)$. Note that the adversary has full knowledge of the algorithm, i.e., the binary tree. Thus adversary's task boils down to the following: for every binary tree that represents an algorithm, picking a path from root to leaf with resulting regret $\Omega(d^2/\log d)$.

Node labeling in the binary tree. Each node v in the binary tree is naturally labeled as an $(n+1)$-tuple $(a(v), b_1(v), b_2(v), \ldots, b_n(v))$, where $a(v)$ is the maximum price for which we have gotten a YES response so far before querying node v and b_i is the minimum price at which we have gotten a NO response for item i before querying node v. Whenever $b_i(v) < a(v)$, we will set $b_i(v) = \emptyset$, representing that item i can no longer be the most valuable item. The label summarizes all the relevant information obtained from the queries asked so far. Without loss of generality we only focus on algorithms that query an item i with $b_i(v) \neq \emptyset$ (for algorithms that do query such null items i, the adversary can simply answer in any manner that is not inconsistent with previous answers; the algorithm only wastes regret by making such queries).

For any deterministic algorithm $\mathcal{A}$, we let $\mathrm{Reg}(\mathcal{A})$ denote the worst-case regret of $\mathcal{A}$ over T rounds. The following lemmas provide constraints on any protocol with low regret.

Lemma 3 (NO edges)**.** *In any length T (or lesser) path in a deterministic algorithm $\mathcal{A}$'s binary tree, the number of NO edges is at most* $\mathrm{Reg}(\mathcal{A})$.

Proof. Consider any problem instance whose execution results in the path at hand. Each NO query contributes 1 to algorithm $\mathcal{A}$'s regret (since the lowest value is 1), so the total number of NO queries is at most $\mathrm{Reg}(\mathcal{A})$. □

Lemma 4 (Large-gap nodes)**.** *In any length T (or lesser) path in a deterministic algorithm $\mathcal{A}$'s binary tree, for any $d > 0$ the number of nodes v where $\max_i b_i(v) - a(v) \geq d$ is at most* $\mathrm{Reg}(\mathcal{A})/d$.

Proof. Suppose there exists a path with strictly more than $\mathrm{Reg}(\mathcal{A})/d$ nodes v with $\max_i b_i(v) - a(v) \geq d$. Consider the last node in the path (i.e, the node farthest from the root in that path) and let $(a^*, b_1^*, \ldots, b_n^*)$ be this node's label. For every earlier node v in this path $a(v) \leq a^*$ and $b_i(v) \geq b_i^*$. At this point the adversary could, without contradicting any of the responses it gave earlier, set $\max_i b_i^*$ to be the value of the maximum valued item. In this case, every previous query where the answer was a NO resulted in regret of $\max_i b_i^*$ and every previous query where YES was the answer resulted in regret of $\max_i b_i^* - a(v) \geq \max_i b_i^* - a^* \geq d$. Thus the total regret is at least d times the number of "large-gap nodes", i.e., nodes v in the path where the gap $\max_i b_i(v) - a(v)$ strictly exceeds $\mathrm{Reg}(\mathcal{A})/d$. This contradicts $\mathcal{A}$'s regret to be $\mathrm{Reg}(\mathcal{A})$. □

[2]Sometimes there need not be two child edges because one of YES or NO answers is inconsistent given the responses received so far. For example if $v(x_2) >= 5$ was received from an earlier query and the latter question is whether $v(x_2) >= 4$, the answer has to always be YES.

Adversarial instance construction. As discussed before, we set $T = 2^{2^d}$. We show that given any deterministic algorithm $\mathcal{A}$'s binary tree, the adversary can construct an instance, i.e., a path in the binary tree, with total regret at least $\Omega(d^2/\log d)$ (whereas an algorithm with regret $O(d + \log\log T)$ would incur at most $O(d)$ regret). The adversary constructs this path by constructing and concatenating many smaller paths, which we refer to as small-paths. In particular, assume to the contrary that $\mathrm{Reg}(\mathcal{A}) < R = \lceil \frac{d^2}{8\log(d+2)} \rceil = O(d^2/\log d)$. We construct R small paths, each with exactly one NO edge, resulting in a total regret of at least R, contradicting the fact that $\mathrm{Reg}(\mathcal{A}) < R$. To keep track of the progress made so far, we annotate each node with a triple that captures a relevant summary of the history so far.

Definition 5 (Summary triple)**.** *A node v in an algorithm's binary tree is (d_1, d_2, k)-bounded if both:*

1. $\min_i b_i(v) - a(v) \geq d_1$.
2. *for at least k different indices i s.t.,* $b_i(v) - a(v) \geq d_1 + d_2$.

By definition 5, the root node where $a = 1$ and $b_i = 2$ for all i is $(1, 0, 0)$ bounded. We now establish two important small-path constructing mechanisms for the adversary. By stringing together these two constructions, we show that the total path length is at most T consisting of R small-paths with each having exactly one NO edge, leading to a regret of at least R.

Lemma 6 (Small-path construction Case-1)**.** *Starting from a node v that is (d_1, d_2, k)-bounded with $k > 0$, the adversary is always able to find a small-path between v and v' of length $\frac{R}{d_2}$ such that v' is $(\frac{d_1 d_2}{R}, d_2, k-1)$-bounded.*

Proof. Let $L = \frac{R}{d_2}$. We assume L is an integer because our invocation of this lemma will involve $\frac{1}{d_2}$ always being an integer, and R is an integer by definition. Let $v = v_0$ be the current node with label $(a(v_0), b_1(v_0), \ldots, b_n(v_0))$. The adversary constructs the small-path as follows. Let $v_1, v_2, \ldots$ be the nodes encountered by the adversary when it proceeds as specified below, and let $q(v_i)$ be the price the algorithm asks at node v_i. For notation consistency let $q(v_{-1}) = a(v_0)$.

1. At node v_i: for queries that have $q(v_i) - q(v_{i-1}) < \frac{d_1}{L}$, the adversary answers YES.
2. At node v_i: for queries that have $q(v_i) - q(v_{i-1}) \geq \frac{d_1}{L}$ the adversary answers NO and immediately terminates the small-path after the first NO answer.

Without loss of generality, we will assume that $q(v_j) \geq q(v_{j-1})$, i.e. that the algorithm never asks a price lower than a previous YES query. (If the algorithm does so, we just pretend that the algorithm asked $q(v_j) = q(v_{j-1})$ for the purpose of analysis.)

We claim that for at least one $j \in \{1, \ldots, L\}$, we have $q(v_j) - q(v_{j-1}) \geq \frac{d_1}{L}$. If not we have would have had L large-gap nodes $v_1, \ldots, v_L$ where for each v_j, $\max_i b_i(v_j) - a(v_j) \geq d_2$ where as by Lemma 4 the number of large-gap nodes with gap d_2 should be at most $\frac{\mathrm{Reg}(\mathcal{A})}{d_2} < \frac{R}{d_2} = L$.

Let j^* be the smallest j at which $q(v_j) - q(v_{j-1}) \geq \frac{d_1}{L}$, and thus our v' is v_{j^*}. In fact, $\min\left(q(v_{j^*}), a(v_0) + d_1\right) - q(v_{j^*-1}) \geq \frac{d_1}{L}$

From here, it is straightforward to see why v' is $(\frac{d_1 d_2}{R}, d_2, k-1)$-bounded. For d_1' note that:

$$\min_i b_i(v_{j^*}) - a(v_{j^*}) \geq a(v_0) + d_1 - q(v_{j^*-1}) \geq \frac{d_1}{L} = \frac{d_1 d_2}{R}$$

Since there was exactly one NO answer, at most one b_i decreased. Also $a(v_{j^*}) + d_1/L \leq a(v_0) + d_1$. Thus at the same value of $d_2' = d_2$ we have k decreasing by one, giving $k' = k - 1$.

□

Lemma 7 (Small-path construction Case-2)**.** *Starting from a node that is (d_1, d_2, k)-bounded with $k = 0$, the adversary is always able to find a small-path between v and v' of length $\frac{2R}{d_1}$ such that v' is $(\frac{d_1^2}{4R}, \frac{d_1}{2}, d-1)$-bounded.*

Proof. We proceed similarly as in case 1, Lemma 6. Let $L = \frac{2R}{d_1}$. We assume L is an integer because our invocation of this lemma will involve $\frac{1}{d_1}$ always being an integer, and R is an integer by definition. Let $v = v_0$ be the current node with label $(a(v_0), b_1(v_0), \ldots, b_n(v_0))$. The adversary constructs the small-path as follows. Let $v_1, v_2, \ldots$ be the nodes encountered by the adversary when it proceeds as specified below, and let $q(v_i)$ be the price the algorithm asks at node v_i. Let again $q(v_{-1}) = a(v_0)$ for notation consistency.

1. At node v_i: for queries that have $q(v_i) - q(v_{i-1}) < \frac{d_1}{2L}$, the adversary answers YES.
2. At node v_i: for queries that have $q(v_i) - q(v_{i-1}) \geq \frac{d_1}{2L}$ the adversary answers NO and immediately terminates the small-path after the first NO answer.

We again focus on the case where $q(v_j) \geq q(v_{j-1})$, pretending $q(v_j) = q(v_{j-1})$ whenever $q(v_j) < q(v_{j-1})$.

We claim that for at least one $j \in \{1, \ldots, L\}$, we have $q(v_j) - q(v_{j-1}) \geq \frac{d_1}{2L}$. If not we have would have had L large-gap nodes $v_1, \ldots, v_L$ where for each v_j, $\max_i b_i(v_j) - a(v_j) \geq \frac{d_1}{2}$ where as by Lemma 4 the number of large-gap nodes with gap $\frac{d_1}{2}$ should be at most $\frac{2\,\mathrm{Reg}(\mathcal{A})}{d_1} < \frac{2R}{d_1} = L$.

Let j^* be the smallest j at which $q(v_j) - q(v_{j-1}) \geq \frac{d_1}{2L}$, and thus our v' is v_{j^*}. Since $a(v_0) + \frac{d_1}{2} - a(v_0) \geq \frac{d_1}{2}$, we have $\min\left(q(v_{j^*}), a(v_0) + \frac{d_1}{2}\right) - q(v_{j^*-1}) \geq \frac{d_1}{2L}$.

From here, it is straightforward to see why v' is $(\frac{d_1^2}{4R}, \frac{d_1}{2}, d-1)$-bounded since:

$$\min_i b_i(v_{j^*}) - a(v_{j^*}) \geq a(v_0) + \frac{d_1}{2} - q(v_{j^*-1}) \geq \frac{d_1}{2L} = \frac{d_1^2}{4R}$$

Since there was exactly one NO answer, at most one b_i decreased, and thus while all other b_i's are at least $a(v_0) + d_1$ and hence:

$$b_i \geq a(v_0) + d_1 \geq a(v_0) + \frac{d_1}{2} + \frac{d_1}{2} \geq a(v_{j^*}) + \frac{d_1^2}{4R} + \frac{d_1}{2}$$

□

We are now ready to prove the main lower bound.

Theorem 8. *No deterministic algorithm for the Product Design problem with independent items can achieve a regret of $O(d + \log\log T)$.*

Proof. Recap: As discussed earlier, set $T = 2^{2^d}$. We show that for any deterministic algorithm $\mathcal{A}$, there is an instance of the Product Design problem where they must incur at least $\Omega(d^2/\log d)$ total regret (whereas an algorithm with regret $O(d + \log\log T)$ would incur at most $O(d)$ regret). The adversary constructs this path by constructing and concatenating many smaller paths, which we refer to as small-paths. In particular, assume to the contrary that $\mathrm{Reg}(\mathcal{A}) < R = \frac{d^2}{8\log(d+2)} = O(d^2/\log d)$. We construct R small paths, each with exactly one NO edge, resulting in a total regret of at least R, contradicting the fact that $\mathrm{Reg}(\mathcal{A}) < R$.

We start at the root node which is $(1, 0, 0)$-bounded and construct R small-paths as follows:

- Construct a Case-2 small-path, i.e. apply Lemma 7 once.
- Construct $d-1$ Case-1 small-paths in succession, i.e., $d-1$ consecutive applications of Lemma 6.
- Repeat until R paths are constructed

It is easy to verify that conditions applicable for applying the respective lemmas are met. In particular, after every d small-paths constructed, we have $k = 0$ making it fit to invoke Case-2 Lemma 7, and for the next $d-1$ rounds we have k decreasing by one from $d-1$ to 0 making it fit to invoke Lemma 6. Also note that whenever we invoke Lemma 6 we have $d_2 = \frac{1}{2^r}$ for some non-negative integer r, making $\frac{1}{d_2}$ integral, and thus making the $\frac{R}{d_2}$ used in Lemma 6 integral. Similarly $\frac{1}{d_1}$ is always integral when we invoke Lemma 7, making $\frac{2R}{d_1}$ used in that lemma integral (d_1 is 1 at the first invocation of Lemma 7, and at the subsequent i-th invocation it will be of the form $R \cdot \left(\frac{1}{2R}\right)^{(d+1)^i} \cdot 2^{-\frac{d+1}{d} \cdot [(d+1)^{i-1} - 1]}$.

Note that after every d paths contructed by the adversary, the state summary changes

$$(d_1, d_2, 0) \to \left(R\left(\frac{d_1}{2R}\right)^{d+1}, \frac{d_1}{2}, 0\right)$$

Each such "cycle" (i.e., d consecutive paths) has a total path length of $\frac{2R}{d_1} + (d-1)\frac{R}{d_2} = \frac{2Rd}{d_1}$, because $d_2 = d_1/2$ here. So the total length of all the paths when we complete η cycles is:

$$\begin{aligned}\mathsf{length}_\eta &= 2Rd + \sum_{i=1}^{\eta-1} \frac{2Rd}{R \cdot (\frac{1}{2R})^{(d+1)^i} \cdot 2^{-\frac{d+1}{d}\cdot[(d+1)^{i-1}-1]}} \\ &\leq 2Rd\eta(4R)^{(d+1)^{\eta-1}}. \end{aligned} \tag{1}$$

We claim that when $\eta = \frac{R}{d} = \frac{d}{8\log(d+2)}$, the RHS above namely $2Rd\eta(4R)^{(d+1)^{\eta-1}}$ is at most $T = 2^{2^d}$. To see this, the logarithm of the upper bound in (1) is given by

$$\begin{aligned}\log(\mathsf{length}_\eta) &\leq \log(2Rd\eta) + (d+1)^{\eta-1}\log(4R) \\ &\leq \log(2d^4) + (d+1)^{\eta-1}\log(4d^2) \\ &\leq 5\log d + 2(d+1)^{\eta-1}\log d \leq (d+1)^\eta \\ &= 2^{\eta\log(d+1)} = 2^{d/8} \leq 2^d = \log(T),\end{aligned}$$

where we used the fact that d is large enough in a few inequalities. It follows that the total length of our feasible path was at most T and it has at least R no edges because of R small-paths with one NO edge each, thus violating Lemma 3 and contradicting our assumption that $\mathsf{Regret}(\mathcal{A}) < R$. □

4 General Products

Finally, we study the Product Design problem for a generic set $X \subseteq \mathbb{R}^d$. We will solve this by a reduction to a contextual learning problem. Contextual pricing is an online decision making problem very similar to ours with two major differences: the products to be sold in each iteration are chosen by an adversary and the regret is computed with respect to the value of products chosen. Formally, there is a fixed unknown valuation $v \in \mathbb{R}^d$, $\|v\|_2 \leq 1$. In each iteration an adversary chooses a product $x_t \in \mathbb{R}^d$, $\|x_t\| \leq 1$ and the decision maker then chooses a price p_t. After choosing the price, the decision maker learns whether $p_t \leq \langle v, x_t\rangle$ or not. The regret of contextual pricing is given by:

$$\mathsf{Regret}_{\mathsf{CP}} = \sum_t \langle v, x_t\rangle - p_t \cdot \mathbf{1}\{p_t \leq \langle v, x_t\rangle\}$$

Note that the benchmark in Contextual Pricing is $\sum_t \langle v, x_t\rangle$ while the benchmark in Product Design is $T \cdot \max_{x\in X}\langle v, x\rangle$. The algorithms in [Cohen *et al.*, 2016; Lobel *et al.*, 2017; Leme and Schneider, 2018; Liu *et al.*, 2020] provide a gurantee with respect to a stronger benchmark, which is based on the notion of the width of the knowledge set. The knowledge set is the set of all values $\hat{v}$ that are consistent with the observations so far. To be precise, if $\sigma_t \in \{-1, +1\}$ corresponds to the feedback at time t (i.e. $\sigma_t = +1$ if $p_t \leq \langle v, x_t\rangle$ and $\sigma_t = -1$ otherwise) then the knowledge set is given by:

$$\begin{aligned}K_t = \{&\hat{v} \in \mathbb{R}^d; \|\hat{v}\|_2 \leq 1 \text{ and} \\ &\sigma_\tau \cdot (p_\tau - \langle \hat{v}, x_\tau\rangle) \leq 0 \text{ for } \tau = 1...t-1\}\end{aligned}$$

Given a certain vector $x \in \mathbb{R}^d$ we define the width of the knowledge set in direction x as:

$$\mathsf{width}(K_t, x) = \max_{\hat{v}\in K_t}\langle \hat{v}, x\rangle - \min_{\hat{v}\in K_t}\langle \hat{v}, x\rangle$$

Using those notions we can define an upper bound on the Contextual Pricing regret, which we call width-based regret:

$$\mathsf{WRegret}_{\mathsf{CP}} = \sum_t \mathbf{1}\{\sigma_t = -1\} + \mathbf{1}\{\sigma_t = +1\}\cdot\mathsf{width}(K_t, x_t)$$

A contextual pricing algorithm is *reasonable* if $p_t \geq \underline{p}_t := \min_{\hat{v}\in K_t}\langle x_t, \hat{v}\rangle$ since the price $\underline{p}_t$ is always guaranteed to sell. For any reasonable algorithm, the following bound holds: $\mathsf{Regret}_{\mathsf{CP}} \leq \mathsf{WRegret}_{\mathsf{CP}}$ since the regret in a no-sale event can always be upper bounded by 1 and the regret of a sale can always be bounded by $\langle v, x_t\rangle - p_t \leq \max_{\hat{v}\in K_t}\langle \hat{v}, x_t\rangle - \min_{\hat{v}\in K_t}\langle x_t, \hat{v}\rangle = \mathsf{width}(K_t, x_t)$.

We say that an algorithm has width-based regret R if $\mathsf{WRegret}_{\mathsf{CP}} \leq R$. We note that all the algorithms we have referenced obtain regret bounds by bounding the width-based regret. In particular, the result of Liu, Paes Leme, and Schneider implies that:

Theorem 9. *There is a reasonable algorithm for contextual pricing such that* $\mathsf{WRegret}_{\mathsf{CP}} \leq O(d\log d\log\log T)$.

We now use that result to obtain an algorithm for Product Design with the same regret:

Theorem 10. *Given a reasonable algorithm for contextual pricing with width-based regret R, it is possible to obtain an algorithm for Product Design with regret R.*

Proof. Design an algorithm as follows: in period t, find $(x_t, \hat{v}) \in \operatorname{argmax}_{x\in X, \hat{v}\in K_t}\langle x_t, \hat{v}\rangle$ and feed x_t to the Contextual Pricing algorithm and obtain p_t. Now, feed (x_t, p_t) to the Product Design algorithm.

To show that the regret of the product discovery in at most $\mathsf{WRegret}_{\mathsf{CP}}$ observe that in the case of a no-sale the regret incurred is $\max_{x\in X}\langle v, x\rangle$ which is at most 1 which is the amount attributed to a no-sale event by $\mathsf{WRegret}_{\mathsf{CP}}$. In the case of a sale, the regret incurred is $\max_{x\in X}\langle v, x\rangle - p_t$. Note that $\max_{x\in X}\langle v, x\rangle \leq \max_{x\in X, \hat{v}\in K_t}\langle \hat{v}, x\rangle = \max_{\hat{v}\in K_t}\langle \hat{v}, x_t\rangle$ and $p_t \geq \min_{\hat{v}\in K_t}\langle \hat{v}, x_t\rangle$ since the contextual pricing algorithm is reasonable, therefore, the regret incurred by product discovery is at most $\mathsf{width}(K_t, x_t)$. Putting it all together:

$$\mathsf{Regret} \leq \mathsf{WRegret}_{\mathsf{CP}} \leq R$$

□

Corollary 11. *There is a $O(d\log d\log\log T)$ regret algorithm for Product Design for general product sets X.*

Although the Contextual Pricing algorithm of [Liu *et al.*, 2020] is a polynomial-time algorithm, we rely on an NP-hard problem to choose the product x_t in each step. Given two convex sets K_1, K_2, finding the pair of point that maximizes the dot product $\max_{x_1\in K_1, x_2\in K_2}\langle x_1, x_2\rangle$ is NP-hard. When $K_1 = K_2$ this becomes the maximum-norm problem that was shown to be NP-hard by Bodlaender et al [Bodlaender *et al.*, 1990]. We leave as an open problem how to obtain this regret guarantee in polynomial-time for a generic X having access to an oracle that for every $\hat{v}$ returns $\hat{x} \in \operatorname{argmax}_{x\in X}\langle x, \hat{v}\rangle$.

References

[Amin *et al.*, 2014] Kareem Amin, Afshin Rostamizadeh, and Umar Syed. Repeated contextual auctions with strategic buyers. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 622–630, 2014.

[Bastani and Bayati, 2016] Hamsa Bastani and Mohsen Bayati. Online decision-making with high-dimensional covariates. *Working paper, Stanford University*, 2016.

[Bodlaender *et al.*, 1990] Hans L. Bodlaender, Peter Gritzmann, Victor Klee, and Jan Van Leeuwen. Computational complexity of norm-maximization. *Combinatorica*, 10(2):203–225, 1990.

[Chapelle and Harchaoui, 2005] Olivier Chapelle and Zaid Harchaoui. A machine learning approach to conjoint analysis. In *Advances in neural information processing systems*, pages 257–264, 2005.

[Cohen *et al.*, 2016] Maxime C Cohen, Ilan Lobel, and Renato Paes Leme. Feature-based dynamic pricing. In *Proceedings of the 2016 ACM Conference on Economics and Computation*, pages 817–817. ACM, 2016.

[Ding *et al.*, 2005] Min Ding, Rajdeep Grewal, and John Liechty. Incentive-aligned conjoint analysis. *Journal of marketing research*, 42(1):67–82, 2005.

[Evgeniou *et al.*, 2007] Theodoros Evgeniou, Massimiliano Pontil, and Olivier Toubia. A convex optimization approach to modeling consumer heterogeneity in conjoint estimation. *Marketing Science*, 26(6):805–818, 2007.

[Gladwell, 2004] Malcolm Gladwell. The ketchup conundrum. *The New Yorker*, 2004.

[Green and Srinivasan, 1978] Paul E Green and Venkatachary Srinivasan. Conjoint analysis in consumer research: issues and outlook. *Journal of consumer research*, 5(2):103–123, 1978.

[Javanmard and Nazerzadeh, 2016] Adel Javanmard and Hamid Nazerzadeh. Dynamic pricing in high-dimensions. *Working paper, University of Southern California*, 2016.

[Javanmard, 2017] Adel Javanmard. Perishability of data: dynamic pricing under varying-coefficient models. *The Journal of Machine Learning Research*, 18(1):1714–1744, 2017.

[Kirkpatrick and Gao, 1990] David G Kirkpatrick and Feng Gao. Finding extrema with unary predicates. In *International Symposium on Algorithms (SIGAL)*, pages 156–164. Springer, 1990.

[Kleinberg and Leighton, 2003] Robert Kleinberg and Tom Leighton. The value of knowing a demand curve: Bounds on regret for online posted-price auctions. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 594–605. IEEE, 2003.

[Leme and Schneider, 2018] Renato Paes Leme and Jon Schneider. Contextual search via intrinsic volumes. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 268–282, 2018.

[Liu *et al.*, 2020] Allen Liu, Renato Paes Leme, and Jon Schneider. Contextual search for general hypothesis classes. *arXiv preprint arXiv:2003.01703*, 2020.

[Lobel *et al.*, 2017] Ilan Lobel, Renato Paes Leme, and Adrian Vladu. Multidimensional binary search for contextual decision-making. *Operations Research*, 2017.

[Mao *et al.*, 2018] Jieming Mao, Renato Paes Leme, and Jon Schneider. Contextual pricing for lipschitz buyers. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.*, pages 5648–5656, 2018.

[Toubia *et al.*, 2003] Olivier Toubia, Duncan I Simester, John R Hauser, and Ely Dahan. Fast polyhedral adaptive conjoint estimation. *Marketing Science*, 22(3):273–303, 2003.

[Toubia *et al.*, 2004] Olivier Toubia, John Hauser, and Duncan Simester. Polyhedral methods for adaptive choice-based conjoint analysis. *Journal of Marketing Research*, 41:116–131, 2004.

[Viappiani and Boutilier, 2010] Paolo Viappiani and Craig Boutilier. Optimal bayesian recommendation sets and myopically optimal choice query sets. In *Advances in neural information processing systems*, pages 2352–2360, 2010.

A Appendix

A.1 The KL algorithm and its regret

Recall that in the KL setting, there is only one item, and the buyer has a fixed unknown value $v^* \in [0,1]$ for this item. Within a course of T rounds, the algorithm chooses a price p_t in each round t, and observes whether the offer is accepted ($p_t \leq v$) or not ($p_t > v$). The regret is defined as $\mathsf{Regret} = Tv - \sum_{t=1}^{T} p_t \cdot \mathbf{1}[p_t \leq v]$. We describe the KL algorithm for completeness and prove that it obtains a regret of $O(\log \log T)$.

Here is how the KL algorithm works: In each round t, the algorithm keeps a *feasible interval* $\mathcal{I}_t = [a_t, b_t]$ based on all the feedback it has received so far. One strategy, inspired by classic binary search, is to offer the item in round t at price $(a_t + b_t)/2$. Even though binary search can (approximately) find v^* *as quickly as possible*, it does not guarantee optimal regret throughout the T rounds. Notice that when the item is overpriced ($p_t > v^*$), the offer is declined and the algorithm makes no revenue in round t; if the item is underpriced ($p_t < v^*$), on the other hand, the offer is accepted and the revenue is p_t. This observation, also pointed out by [Kleinberg and Leighton, 2003], explains the intuition behind the KL Algorithm: even though binary search extracts "most amount of information" in every round, in order to optimize the regret, the algorithm has to tend to underprice.

In addition to the feasible interval $[a_t, b_t]$, the KL algorithm keeps a *step size* Δ_t in each round t. The algorithm then offers the item at price $p_t = a_t + \Delta_t$. There are two possible outcomes:

(a) If the offer is accepted, then the feasible interval is updated to $[a_t + \Delta_t, b_t]$, i.e., $a_{t+1} = a_t + \Delta_t$ and $b_{t+1} = b_t$. In this case, the algorithm uses the same step size for the next round ($\Delta_{t+1} = \Delta_t$); hence, the next offer will be at price $p_{t+1} = a_{t+1} + \Delta_{t+1} = p_t + \Delta_t$.

(b) If the offer is declined, then the feasible interval is updated to $[a_t, a_t + \Delta_t]$, i.e., $a_{t+1} = a_t, b_{t+1} = p_t$, and the step size shrinks to $\Delta_{t+1} = \Delta_t^2$.

Initially, $a_1 = 0, b_1 = 1, \Delta_1 = 1/2$. The algorithm continues as long as $b_t - a_t > 1/T$. Once $b_t - a_t \leq 1/T$, a_t is a good enough approximation of v^*. At this point, the algorithm keeps offering the item at the same price a_t for the rest of the rounds.

Lemma 12 ([Kleinberg and Leighton, 2003], restated)**.** *The KL algorithm has a regret $O(\log \log T)$.*

Proof of Lemma 12. The total regret from steps in which $b_t - a_t \leq 1/T$ is at most $T \cdot 1/T = 1$. For the remaining phases we note that Δ_t has the form 2^{-2^j} and that $\Delta_t \leq b_t - a_t$, so it can assume only $O(\log \log T)$ distinct values before $b_t - a_t \leq 1/T$. Now we bound the total regret incurred for each given value of the step size Δ_t.

For each given value there is at most one no-sale which incurs in regret at most 1. To bound the regret on sale events, note that when Δ_t becomes 2^{-2^j} the value of $b_t - a_t$ is $2^{-2^{j-1}} = \sqrt{\Delta_t}$ and therefore there will be at most $\sqrt{\Delta_t}/\Delta_t$ sales for that step size. Each of the sales incur regret at most the original value of $b_t - a_t$ which is $\sqrt{\Delta_t}$. The total regret incurred on sales for that step size is therefore at most $\sqrt{\Delta_t} \cdot \sqrt{\Delta_t}/\Delta_t = 1$. □

We now prove that the parallelized KL algorithm we designed in Section 3.1 obtains a regret of $O(d \log \log T)$.

Restatement of Theorem 2. *The parallelized KL algorithm achieves $O(d \log \log T)$ total regret on the Product Design problem with independent items.*

Proof of Theorem 2. The optimal item i^* is never eliminated from S since it cannot be the case that the same price is rejected for i^* but accepted for some other item. If we restrict our attention to offers to i^*, this corresponds to an execution of the single item KL algorithm for i^* which has regret $O(\log \log T)$.

For any other item $i \neq i^*$, the regret incurred in each round, as long as item i is still alive, is the exact same regret incurred when we offered the same price to i^*, hence the total regret is at most $O(d \log \log T)$. □

A.2 The Kirkpatrick and Gao algorithm, and their mistake bound

The idea of the algorithm of Kirkpatrick and Gao is to keep a tuple $(a, b_1, \ldots, b_n)$ where a is the maximum price that led to an accepted across all items and b_i is an upper bound on the minimum price which led to a rejected offer for item i. The items that are alive (i.e. can potentially be the optimal item) are given by $S = \{i; b_i \geq a + \epsilon\}$. The algorithm offers price $p = \frac{1}{2}(a + \min_{i \in S} b_i)$ for the item with largest b_i.

- If the offer is rejected, we simply update $b_i = p$ for that item.
- If the offer is accepted, then keep offering price $p_t = \min_{i \in S} b_i$ and updating a and S while the offers are accepted. Stop when there is a rejection. At that point, just set $p_t = \min_{i \in S} b_i$.

At some point, all items will be rejected. At that point return the item with the best accepted price as the best item. Since an item is only elminated from S when we can show that its value is at most the best accepted offer plus ϵ we are guarantees that once all items are eliminated, the best accepted offer is ϵ-optimal.

For completeness, we provide a rough proof sketch of why the Kirkpatrick-Gao procedure terminates in $O(d + \log(1/\epsilon))$ iterations (for details, we recommend the reader consult the proof of Theorem 3.2 in [Kirkpatrick and Gao, 1990]).

Our goal will be to examine the potential function $\phi := \min_{i \in S} b_i - a$ equal to the gap between the largest YES query and the smallest NO query. Note that this gap starts equal to 1, and is always at least ϵ (by the definition of S).

Now, if a price offer is rejected, then the gap is halved. If the offer is accepted and then immediately rejected, then the gap is again halved. If the offer is accepted more than once before the first rejection, then some items are removed from S, but the gap ϕ can increase since $\min_{i \in S} b_i$ increases. The clever trick is that the b_i are chose in such a way that for each item that is removed from S, the gap ϕ can at most double. Therefore, over time, the gap will increase at most d times and by at most a 2^d factor overall. Hence, the number

of gap decreases until no item is left is at most $\log(2^d/\epsilon) = d + \log(1/\epsilon)$.

Since the gap must either increase or decrease every two queries, this means that the total number of queries is at most $2(d + (d + \log(1/\epsilon)) = O(d + \log(1/\epsilon))$, as desired.