Journal of Machine Learning Research 17 (2016) 1-47 Submitted 5/15; Revised 1/16; Published 7/16

Monotonic Calibrated Interpolated Look-Up Tables

Maya Gupta MAYAGUPTA@GOOGLE.COM
Andrew Cotter ACOTTER@GOOGLE.COM
Jan Pfeifer JANPF@GOOGLE.COM
Konstantin Voevodski KVODSKI@GOOGLE.COM
Kevin Canini CANINI@GOOGLE.COM
Alexander Mangylov AMANGY@GOOGLE.COM
Wojciech Moczydlowski WOJTEKM@GOOGLE.COM
Alexander van Esbroeck ALEXVE@GOOGLE.COM
Google
1600 Amphitheatre Pkwy
Mountain View, CA 94301, USA

Editor: Saharon Rosset

Abstract

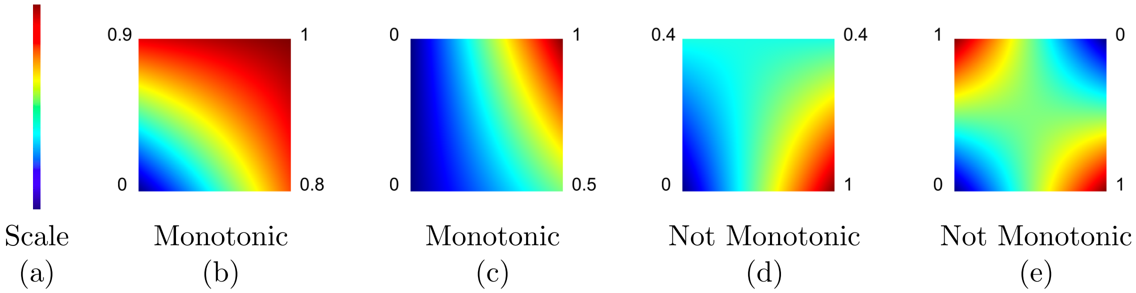
Real-world machine learning applications may have requirements beyond accuracy, such as fast evaluation times and interpretability. In particular, guaranteed monotonicity of the learned function with respect to some of the inputs can be critical for user confidence. We propose meeting these goals for low-dimensional machine learning problems by learning flexible, monotonic functions using calibrated interpolated look-up tables. We extend the structural risk minimization framework of lattice regression to monotonic functions by adding linear inequality constraints. In addition, we propose jointly learning interpretable calibrations of each feature to normalize continuous features and handle categorical or missing data, at the cost of making the objective non-convex. We address large-scale learning through parallelization, mini-batching, and random sampling of additive regularizer terms. Case studies on real-world problems with up to sixteen features and up to hundreds of millions of training samples demonstrate the proposed monotonic functions can achieve state-of-the-art accuracy in practice while providing greater transparency to users.

Keywords: interpretability, interpolation, look-up tables, monotonicity

1. Introduction

Many challenging issues arise when making machine learning useful in practice. Evaluation of the trained model may need to be fast. Features may be categorical, missing, or poorly calibrated. A blackbox model may be unacceptable: users may require guarantees that the function will behave sensibly for all samples, and prefer functions that are easier to understand and debug. In this paper we address these practical issues, without trading-off for accuracy.

We have found that a key interpretability issue in practice is whether the learned model can be guaranteed to be *monotonic* with respect to some features. For example, suppose the goal is to estimate the value of a used car, and one of the features is the number of km it has been driven. If all the other feature values are held fixed, we expect the value of the



Scale (a) Monotonic (b) Monotonic (c) Not Monotonic (d) Not Monotonic (e)

Figure 1: Example 2×2 interpolated look-up table functions over a unit square, with color scale shown in (a). Each function is defined by a 2×2 lattice with four parameters, which are the values of the function in the four corners (shown). The function is linearly interpolated from its parameters (see Figure 2 for a pictorial description of linear interpolation). The function in (b) is strictly monotonically increasing in both features, which can be verified by checking that each upper parameter is larger than the parameter below it, and that each parameter on the right is larger than the parameter to its left. The function in (c) is strictly monotonically increasing in the first feature, and monotonically increasing in the second feature (but not strictly so since the parameters on the left are both zero). The function in (d) is monotonically increasing in the first feature (one verifies this by noting that $1 \geq 0$ and $0.4 \geq 0.4$), but non-monotonic in the second feature: on the left side the function increases from $0 \rightarrow 0.4$, but on the right side the function decreases from $1 \rightarrow 0.4$. The function in (e) is a saddle function interpolating an exclusive-OR, and is non-monotonic in both features.

used car to never increase as the number of km driven increases. But a model learned from a small set of noisy samples may not, in fact, respect this prior knowledge.

In this paper, we propose learning monotonic, efficient, and flexible functions by constraining and calibrating interpolated look-up tables in a structural risk minimization framework. Learning monotonic functions is difficult, and previously published work has only been illustrated on small problems (see Table 1). Our experimental results demonstrate learning flexible, guaranteed monotonic functions on more features and data than prior work, and that these functions achieve state-of-the-art performance on real-world problems.

The parameters of an interpolated look-up table are simply values of the function, regularly spaced in the input space, and these values are interpolated to compute $f(x)$ for any x. See Figures 1 and 2 for examples of 2×2 and 2×3 look-up tables and the functions produced by interpolating them. Each parameter has a clear meaning: it is the value of the function for a particular input, for a set of inputs on a regular grid. These parameters can be individually read and checked to understand the learned function's behavior.

Interpolating look-up tables is a classic strategy for representing low-dimensional functions. For example, backs of old textbooks have pages of look-up tables for one-dimensional functions like $sin(x)$, and interpolating look-up tables is standardized by the ICC Profile for the three and four dimensional nonlinear transformations needed to color manage printers (Sharma and Bala, 2002). In this paper we interpolate look-up tables defined over much

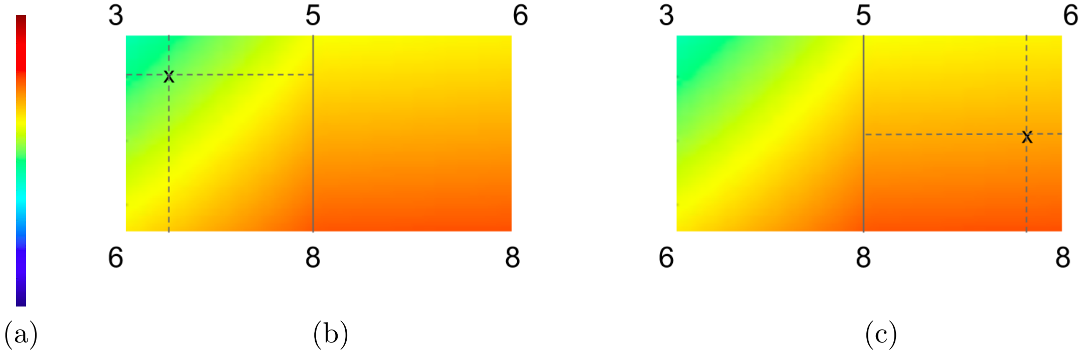


Figure 2: Figure illustrates a 3×2 lattice function and multilinear interpolation, with color scale given by (a) and parameters as shown. The lattice function shown is continuous everywhere, and differentiable everywhere except at the boundary between lattice cells, which is the vertical edge joining the middle parameters 5 and 8. As shown in (b), to evaluate $f(x)$, any x that falls in the left cell of the lattice is linearly interpolated from the parameters at the vertices of the left cell, here 6, 3, 5 and 8. Linear interpolation is linear not in x but in the lattice parameters, that is $f(x)$ is a weighted combination of the parameter values 6, 3, 5, and 8. The weights on the parameters are the areas of the four boxes formed by the dotted lines drawn orthogonally through x, with each parameter weighted by the area of the box farthest from it, so that as x moves closer to a parameter the weight on that parameter gets bigger. Because the dotted lines partition a unit square cell, the sum of these linear interpolation weights is always one. As shown in (c), samples like the marked x that fall in the right cell of the lattice are interpolated from that cell's parameters: 8, 5, 6 and 8.

larger feature spaces. Using an efficient linear interpolation method we refer to as *simplex interpolation*, the interpolation of a D-dimensional look-up table can be computed in $O(D \log D)$ time. For example, we found that interpolating a look-up table defined over $D = 20$ features takes only 2 microseconds on a standard CPU. The number of parameters in the look-up table scales as 2^D, which limits the size of D, but still enables us to learn higher-dimensional flexible monotonic functions than ever before.

Estimating the parameters of an interpolated look-up table using structural risk minimization was proposed by Garcia and Gupta (2009) and called *lattice regression*. Lattice regression can be viewed as a kernel method that uses the explicit nonlinear feature transformation formed by mapping an input $x \in [0, 1]^D$ to a vector of linear interpolation weights $\phi(x) \in \Delta^{2^D}$ over the 2^D vertices of the look-up table cell that contains x, where Δ denotes the standard simplex. Then the function is linear in these transformed features: $f(x) = \theta^T \phi(x)$. We will refer to the look-up table parameters θ as the *lattice*, and to the interpolated look-up table $f(x)$ as the *lattice function*. Earlier work in lattice regression focused on learning highly nonlinear functions over 2 to 4 features with fine-grained lattices, such as a $17 \times 17 \times 17$ lattice for modeling a color printer or super-resolution of spherical images (Garcia et al., 2010, 2012). In this paper, we apply lattice regression to more generic machine learning problems with $D = 5$ to 16 features, and show that 2^D lattices work well for many real-world classification and ranking problems, especially when paired with jointly trained one-dimensional pre-processing functions.

We begin with a survey of related work in machine learning of interpretable and monotonic functions. Then we review lattice regression in Section 3. The main contribution is learning monotonic lattices in Section 4. We discuss efficient linear interpolation in Section 5. We propose an interpretable torsion lattice regularizer in Section 6. We propose jointly learning one-dimensional calibration functions in Section 7, and consider two strategies for supervised handling of missing data for lattice regression in Section 8. In Section 9, we consider strategies for speeding up training and handling large-scale problems and large-scale constraint-handling. A series of case studies in Section 10 experimentally explore the paper's proposals, and demonstrate that monotonic lattice regression achieves similar accuracy as a random forest, and that monotonicity is a common issue that arises in many different applications. The paper ends with conclusions and open questions in Section 11.

2. Related Work

We give a brief overview of related work in interpretable machine learning, then survey related work in learning monotonic functions.

2.1 Related Work in Interpretable Machine Learning

Two key themes of the prior work on interpretable machine learning are *(i)* interpretable function classes, and *(ii)* preferring simpler functions within a function class.

2.1.1 Interpretable Function Classes

The function classes of decision trees and rules are generally regarded as relatively interpretable. Naïve Bayes classifiers can be interpreted in terms of weights of evidence (Good, 1965; Spiegelhalter and Knill-Jones, 1984). Similarly, linear models form an interpretable function class in that the parameters dictate the relative importance of each feature. Linear approaches can be generalized to sum nonlinear components, as in generalized additive models (Hastie and Tibshirani, 1990) and some kernel methods, while still retaining some of their interpretable aspects.

The function class of interpolated look-up tables is interpretable in that the function's parameters are the look-up table values, and so are semantically meaningful: they are simply examples of the function's output, regularly spaced in the domain. Given two look-up tables with the same structure and the same features, one can analyze how their functions differ by analyzing how the look-up table parameters differ. Analyzing which parameters change by how much can help answer questions like "If I add training examples and re-train, what changes about the model?"

2.1.2 Prefer Simpler Functions

Another body of work focuses on choosing simpler functions within a function class, optimizing an objective of the form: minimize empirical error and maximize simplicity, where simplicity is usually defined as some manifestation of Occam's Razor or variant of Kolmogorov complexity. For example, Ishibuchi and Nojima (2007) minimize the number of fuzzy rules in a rule set, Osei-Bryson (2007) prunes a decision tree for interpretability, Rätsch et al. (2006) finds a sparse convex combination of kernels for a multi-kernel support vector machine, and Nock (2002) prefers smaller committees of ensemble classifiers. Similarly, Garcia et al. (2009) measure the interpretability of rule-based classifiers in terms of the number of rules and number of features used. More generally, this category of interpretability includes model selection criteria like the Bayesian information criterion and Akaike information criterion (Hastie et al., 2001), sparsity regularizers like sparse linear regression models, and feature selection methods. Other approaches to simplicity may include simplified structure in graphical models or neural nets, such as the structured neural nets of Strannegaard (2012).

While sparsity-based approaches to interpretability can provide regularization that reduces over-fitting and hence increases accuracy, it has also been noted that such strategies may create a trade-off between interpretability and accuracy (Casillas et al., 2002; Nock, 2002; Yu and Xiao, 2012; Shukla and Tripathi, 2012). We hypothesize this occurs when the assumed simpler structure is a poor model of the true function.

Monotonicity is another way to choose a semantically simpler function to increase interpretability (and regularize). Our case studies in Section 10 illustrate that when applied to problems where monotonicity is warranted true, we do not see a trade-off with accuracy.

2.2 Related Work in Monotonic Functions

A function $f(x)$ is monotonically increasing with respect to feature d if $f(x_i) \geq f(x_j)$ for any two feature vectors $x_i, x_j \in \mathbb{R}^D$ where $x_i[d] \geq x_j[d]$ and $x_i[m] = x_j[m]$ for $m \neq d$.

A number of approaches have been proposed for enforcing and encouraging monotonicity in machine learning. The computational complexity of these algorthims tends to be high, and most methods scale poorly in the number of features D and samples n, as summarized in Table 1.

We detail the related work in the following sections organized by the type of machine learning, but these methods could instead be organized by strategy, which mostly falls into one of four categories:

1. Constrain a more flexible function class to be monotonic, such as linear functions with positive coefficients, or a sigmoidal neural network with positive weights.

2. Post-process by pruning or reducing monotonicity violations after training.

3. Penalize monotonicity violations by pairs of samples or sample derivatives when training.

4. Re-label samples to be monotonic before training.

2.2.1 Monotonic Linear and Polynomial Functions

Linear functions can be easily constrained to be monotonic in certain inputs by requiring the corresponding slope coefficients to be non-negative, but linear functions are not sufficiently flexible for many problems. Polynomial functions (equivalently, linear functions with pre-defined crosses of features) can also be easily forced to be monotonic by requiring all coefficients to be positive. However, this is only a sufficient and not necessary condition: there are monotonic polynomials whose coefficients are not all positive. For example, consider the simple case of second degree multilinear polynomials defined over the unit square $f : [0, 1]^2 \to \mathbb{R}$ such that:

$$f(x) = a_0 + a_1 x[0] + a_2 x[1] + a_3 x[0]x[1]. \tag{1}$$

Restricting the function to a bounded domain the domain $x \in [0, 1]^2$ and forcing the derivative to be positive over that domain, one sees that the complete set of monotonic functions of the form (1) on the unit square is described by four linear inequalities:

$$\begin{array}{ll} a_1 > 0 & a_2 > 0 \\ a_1 + a_3 > 0 & a_2 + a_3 > 0. \end{array}$$

The general problem of checking whether a particular choice of polynomial coefficients produces a monotonic function requires checking whether the polynomial's derivative (also a polynomial) is positive everywhere, which is equivalent to checking if the derivative has any real roots, which can be computationally challenging (see, for example, Sturm's theorem for details).

Functions of the form (1) can be equivalently expressed as a 2×2 lattice interpolated with multilinear interpolation, but as we will show in Section 4, with this alternate parameterization it is easier to check and enforce the complete set of monotonic functions.

2.2.2 Monotonic Splines

In this paper we extend lattice regression, which is a spline method with fixed knots on a regular grid and a linear kernel (Garcia et al., 2012), to be monotonic. There have been a number of proposals to learn monotonic one-dimensional splines. For example, building on Ramsay (1998), Shively et al. (2009) parameterize the set of all smooth and strictly monotonic one-dimensional functions using an integrated exponential form $f(x) = a + \int_0^x e^{b+u(t)} dt$, and showed better performance than the monotone functions estimators of Neelon and Dunson (2004) and Holmes and Heard (2003) for smooth functions. In other related spline work, Villalobos and Wahba (1987) considered smoothing splines with linear inequality constraints, but did not address monotonicity.

2.2.3 Monotonic Decision Trees and Forests:

Stumps and forests of stumps are easily constrained to be monotonic. However, for deeper or broader trees, all pairs of leaves must be checked to verify monotonicity (Potharst and Feelders, 2002b). Non-monotonic trees can be pruned to be monotonic using various strategies that iteratively reduce the non-monotonic branches (Ben-David, 1992; Potharst and Feelders, 2002b). Monontonicity can also be encouraged during tree construction by penalizing the splitting criterion to reduce the number of non-monotonic leaves a split would create (Ben-David, 1995). Potharst and Feelders (2002a) achieved completely flexible monotonic trees using a strategy akin to bogosort (Gruber et al., 2007): train many trees on different random subsets of the training samples, then select one that is monotonic.

	Method	Monotonicity Strategy	Guaranteed Monotonic?	Max D	Max n
Archer and Wang (1993)	neural net	constrain function	yes	2	50
Wang (1994)	neural net	constrain function	yes	1	150
Mukarjee and Stern (1994)	kernel estimate	post-process	yes	2	2447
Ben-David (1995)	tree	penalize splits	yes	8	125
Sill and Abu-Mostafa (1997)	neural net	penalize pairs	no	6	550
Sill (1998)	neural net	constrain function	yes	10	196
Kay and Ungar (2000)	neural net	constrain function	yes	1	100
Potharst and Feelders (2002a)	tree	randomize	yes	8	60
Potharst and Feelders (2002b)	tree	prune	yes	11	1090
Spouge et al. (2003)	isotonic regression	constrain	yes	2	100,000
Duivesteijn and Feelders (2008)	k-NN	re-label samples	no	12	768
Lauer and Bloch (2008)	svm	sample derivatives	no	none	none
Dugas et al. (2000, 2009)	neural net	constrain function	yes	4	3434
Shively et al. (2009)	spline	constrain function	yes	1	100
Kotlowski and Slowinski (2009)	rule-based	re-label samples	yes	11	1728
Daniels and Velikova (2010)	neural net	constrain function	yes	6	174
Riihimäki and Vehtari (2010)	Gaussian process	sample derivatives	no	7	1222
Qu and Hu (2011)	neural net	derivatives / constrain	yes	1	30
Neumann et al. (2013)	neural net	sample derivatives	no	3	625

Table 1: Some related work in learning monotonic functions. Many of these methods *guarantee* a monotonic solution, but some only *encourage* monotonicity. The last two columns gives the largest number of features D and the largest number of samples n used in any of the experiments in that paper (generally not the same experiment).

2.2.4 Monotonic Support Vector Machines

With a linear kernel, it may be easy to check and enforce monotonicity of support vector machines, but for nonlinear kernels it will be more challenging. Lauer and Bloch (2008) encouraged support vector machines to be more monotonic by constraining the derivative of the function at the training samples. Riihimäki and Vehtari (2010) used the same strategy to encourage more monotonic Gaussian processes.

2.2.5 Monotonic Neural Networks

In perhaps the earliest work on monotonic neural networks, Archer and Wang (1993) adaptively down-weighted samples during training whose gradient updates would violate monotonicity, to produce a positive weighted neural net. Other researchers explicitly proposed constraining the weights to be positive in a single hidden-layer neural network with the sigmoid or other monotonic nonlinear transformation (Wang, 1994; Kay and Ungar, 2000; Dugas et al., 2000, 2009; Minin et al., 2010). Dugas et al. (2009) showed with simulations of four features and 400 training samples that both bias and variance were reduced by enforcing monotonicity. However, Daniels and Velikova (2010) showed this approach requires D hidden layers to arbitrarily approximate any D-dimensional monotonic function. In addition to a general proof, they provide a simple and realistic example of a two-dimensional monotonic function that cannot be fit with one hidden layer and positive weights.

Abu-Mostafa (1993) and Sill and Abu-Mostafa (1997) proposed regularizing a function to be more monotonic by penalizing squared deviations in monotonicity for virtual pairs of input samples that are added for this purpose. Unfortunately, it generally does not guarantee monotonicity everywhere, only with respect to those virtual input pairs. (And in fact, to guarantee monotonicity for the sampled pairs, an exact penalty function would be needed with a sufficiently large regularization parameter to ensure the regularization was equivalent to a constraint).

Lauer and Bloch (2008), Riihimäki and Vehtari (2010), and Neumann et al. (2013) encouraged *extreme learning machines* to be more monotonic by constraining the derivative of the function to be positive for a set of sampled points.

Qu and Hu (2011) did a small-scale comparison of encouraging monotonicity by constraining input pairs to be monotonic, versus encouraging monotonic neural nets by constraining the function's derivatives at a subset of samples (analogous to Lauer and Bloch (2008)), versus using a sigmoidal function with positive weights. They concluded the positive-weight sigmoidal function is best.

Sill (1998) proposed a guaranteed monotonic neural network with two hidden layers by requiring the first linear layer's weights to be positive, using hidden nodes that take the maximum of groups of first layer variables, and a second hidden layer that takes the minimum of the maxima. The resulting surface is piecewise linear, and as such can represent any continuous differentiable function arbitrarily well. The resulting objective function is not strictly convex, but the authors propose training such monotonic networks using gradient descent where samples are associated with one active hyperplane at each iteration. Daniels and Velikova (2010) generalized this approach to handle the "partially monotonic" case that the function is only monotonic with respect to some features.

2.2.6 Isotonic Regression and Monotonic Nearest Neighbors

Isotonic regression re-labels the input samples with values that are monotonic and close to the original labels. These monotonically re-labeled samples can then be used, for example, to define a monotonic piecewise constant or piecewise linear surface. This is an old approach; see Barlow et al. (1972) for an early survey. Isotonic regression can be solved in $O(n)$ time if monotonicity implies a total ordering of the n samples. But for usual multi-dimensional machine learning problems, monotonicity implies only a partial order, and solving the n-parameter quadratic program is generally $O(n^4)$, and $O(n^3)$ for two-dimensional samples (Spouge et al., 2003). Also problematic for large n is the $O(n)$ evaluation time for new samples.

Mukarjee and Stern (1994) proposed a suboptimal monotonic kernel regression that is computationally easier to train than isotonic regression. It computes a standard kernel estimate, then locally upper and lower bounds it to enforce monotonicity.

The *isotonic separation* method of Chandrasekaran et al. (2005) is like the work of Abu-Mostafa (1993) in that it penalizes violations of monotonicity by pairs of training samples. Like isotonic regression, the output is a re-labeling of the original samples, the solution is at least $O(n^3)$ in the general case, and evaluation time is $O(n)$.

Ben-David et al. (1989); Ben-David (1992) constructed a monotonic rule-based classifier by sequentially adding training examples (each of which defines a rule) that do not violate monotonicity restrictions.

Duivesteijn and Feelders (2008) proposed re-labeling samples before applying nearest neighbors based on a monotonicity violation graph with the training examples at the vertices. Coupled with a proposed modified version of k-NN, they can enforce monotonic outputs. Similar pre-processing of samples can be used to encourage any subsequently trained classifier to be more monotonic (Feelders, 2010).

Similarly, Kotlowski and Slowinski (2009) try to solve the isotonic regression problem to re-label the dataset to be monotonic, then fit a monotonic ensemble of rules to the re-labeled data, requiring zero training error. They showed overall better performance than the ordinal learning model of Ben-David et al. (1989) and isotonic separation (Chandrasekaran et al., 2005).

3. Review of Lattice Regression

Before proposing *monotonic* lattice regression, we review lattice regression (Garcia and Gupta, 2009; Garcia et al., 2012). Key notation is listed in Table 2.

Let $M_d \in \mathbb{N}$ be a hyperparameter specifying the number of vertices in the look-up table (that is, lattice) for the dth feature. Then the lattice is a regular grid of $M \overset{\triangle}{=} M_1 \times M_2 \times \ldots M_D$ parameters placed at natural numbers so that the lattice spans the hyper-rectangle $\mathcal{M} \overset{\triangle}{=} [0, M_1 - 1] \times [0, M_2 - 1] \times \ldots [0, M_D - 1]$. See Figure 1 for examples of 2×2 lattices, and Figure 2 for an example 3×2 lattice. For machine learning problems we find $M_d = 2$ for all d to often work well in practice, as detailed in the case studies in Section 10. For image processing applications with only two to four features, much larger values of M_d were needed (Garcia et al., 2012).

D	number of features
n	number of samples
$M_d \in \mathbb{N}$	number of vertices in the lattice along the dth feature
$M \in \mathbb{N}$	total number of vertices in the lattice: $M = \prod_d M_d$
$\mathcal{M}$	hyper-rectangular span of the lattice: $[0, M_1 - 1] \times \ldots \times [0, M_D - 1]$
x_i	ith training sample with D components. Domain depends on section.
$y_i \in \mathbb{R}$	ith training sample label
$x[d]$	dth component of feature vector x
$\phi(x) \in [0,1]^M$	linear interpolation weights for x
$\theta \in \mathbb{R}^M$	lattice values (parameters)
$v_j \in \{0,1\}^D$	jth vertex of a 2^D lattice

Table 2: Key Notation

The feature values are assumed to be bounded and linearly scaled to fit the lattice, so that the dth feature vector value $x[d]$ lies in $[0, M_d - 1]$. (We propose learning non-linear scalings of features jointly with the lattice parameters in Section 7.)

Lattice regression is a kernel method that maps $x \in \mathcal{M}$ to a transformed feature vector $\phi(x) \in [0,1]^M$. The values of $\phi(x)$ are the interpolation weights for x for the 2^D indices corresponding to the 2^D vertices of the hypercube surrounding x; for all other indices, $\phi(x) = 0$.

The function $f(x)$ is linear in $\phi(x)$ such that $f(x) = \theta^T \phi(x)$. That is, the function parameters θ each correspond to a vertex in the lattice, and $f(x)$ linearly interpolates the θ for the lattice cell containing x.

Before reviewing the lattice regression objective for learning the parameters θ, we review standard multilinear interpolation to define $\phi(x)$.

3.1 Multilinear Interpolation

Multilinear interpolation is the multi-dimensional generalization of the familiar bilinear interpolation that is commonly used to up-sample images. See Figure 2 for a pictorial explanation.

For notational simplicity, we assume a 2^D lattice such that $x \in [0,1]^D$. For multi-cell lattices, the same math and logic is applied to the lattice cell containing the x. Denote the kth component of $\phi(x)$ as $\phi_k(x)$. Let $v_k \in \{0,1\}^D$ be the kth vertex of the unit hypercube. The multilinear interpolation weight on the vertex v_k is

$$\phi_k(x) = \prod_{d=0}^{D-1} x[d]^{v_k[d]} (1 - x[d])^{1 - v_k[d]}. \tag{2}$$

Note the exponents in (2) are v_k and $1 - v_k[d]$, which either equal 0 and 1, or equal 1 and 0, so these exponents act like selectors that multiply in either $x[d]$ or $1 - x[d]$ for each dimension d. Equivalently, one can write

$$\phi_k(x) = \prod_{i=0}^{D-1} \left((1 - \texttt{bit}[i,k]) \, (1 - x[i]) + \texttt{bit}[i,k] x[i] \right), \tag{3}$$

where $\texttt{bit}[i,k] \in \{0,1\}$ denotes the ith bit of vertex v_k, and can be computed $\texttt{bit}[i,k] = (k \gg i) \, \& 1$ using bitwise arithmetic.

The resulting $f(x) = \theta^T \phi(x)$ is a multilinear polynomial over each lattice cell. For example, a 2×2 lattice interpolated with multilinear interpolation gives:

$$f(x) = \theta[0](1 - x[0])(1 - x[1]) + \theta[1]x[0](1 - x[1]) + \theta[2](1 - x[0])x[1] + \theta[3]x[0]x[1]. \tag{4}$$

Expanding (4), one sees it is a different parameterization of the multilinear function given in (1), where the parameter vectors are related by a linear matrix transform: $a = T\theta$ for $T \in \mathbb{R}^{4\times4}$. But the θ parameterization has the advantage that each parameter is the function value for a feature vector at the vertex of the lattice (see Figure 1), and as we show in Section 4, makes it easier to learn the complete set of monotonic functions.

The linear interpolation is applied per lattice cell. At lattice cell boundaries the resulting function is continuous, but not differentiable. The overall function is piecewise polynomial, and hence a spline, and can be equivalently formulated using a linear basis function. Higher-order basis functions like the popular cubic spline will lead to smoother and potentially slightly more accurate functions (Garcia et al., 2012). However, higher-order basis functions destroy the interpretable localized effect of the parameters, and increase the computational complexity.

The multilinear interpolation weights are just one type of linear interpolation. In general, linear interpolation weights are defined as solutions to the system of $D + 1$ equations:

$$\sum_{k=0}^{2^D} \phi_k(x) v_k = x \text{ and } \sum_{k=0}^{2^D} \phi_k(x) = 1. \tag{5}$$

This system of equations is under-determined and has many solutions for an x in the convex hull of a lattice cell. The multilinear interpolation weights given in (2) are the maximum entropy solution to (5) (Gupta et al., 2006), and thus have good noise averaging and smoothness properties compared to other solutions. We discuss a more efficient linear interpolation in Sec. 5.2.

3.2 The Lattice Regression Objective

Consider the standard supervised machine learning set-up of a training set of randomly sampled pairs $\{(x_i, y_i)\}$ pairs, where $x_i \in \mathcal{M}$ and $y_i \in \mathbb{R}$, for $i = 1, \ldots, n$. Historically, people created look-up tables by first fitting a function $h(x)$ to the $\{x_i, y_i\}$ using a regression algorithm such as a neural net or local linear regression, and then evaluating $h(x)$ on a regular grid to produce the look-up table values (Sharma and Bala, 2002). However, even if they fit the function to minimize empirical risk on the training samples, they did not minimize the *actual* empirical risk because these approaches did not take into account that the trained look-up table would be interpolated at run-time, and this interpolation changes the error on the training samples.

Garcia and Gupta (2009) proposed directly optimizing the look-up table parameters θ to minimize the actual empirical error between the training labels and the interpolated look-up table:

$$\arg\min_{\theta \in \mathbb{R}^M} \sum_{i=1}^{n} \ell(y_i, \theta^T \phi(x_i)) + R(\theta), \tag{6}$$

where ℓ is a loss function such as squared error, $\phi(x_i) \in [0,1]^M$ is the vector of linear interpolation weights over the lattice for training sample x_i (detailed in Section 3.1 and Sec. 5.2), $f(x_i) = \theta^T \phi(x_i)$ is the linear interpolation of x_i from the look-up table parameters θ, and $R(\theta)$ is a regularizer on the lattice parameters. In general, we assume the loss ℓ and regularizer R are convex functions of θ so that solving (6) is a convex optimization. Prior work focused on squared error loss, and used graph regularizers $R(\theta)$ of the form $\theta^T K \theta$ for some PSD matrix K, in which case (6) has a closed-form solution which can be computed with sparse matrix inversions (Garcia and Gupta, 2009; Garcia et al., 2010, 2012).

4. Monotonic Lattices

In this section we propose constraining lattice regression to learn monotonic functions.

4.1 Monotonicity Constraints For a Lattice

In general, simply checking whether a nonlinear function is monotonic can be quite difficult (see the related work in Section 2.2). But for a linearly interpolated look-up table, checking for monotonicity is relatively easy: if the lattice values increase in a given direction, then the function increases in that direction. See Figure 1 for examples. Specifically, one must check that $\theta_s > \theta_r$ for each pair of adjacent look-up table parameters θ_r and θ_s. If all features are specified to be monotonic for a 2^D lattice, this results in $D2^{D-1}$ pairwise linear inequality constraints to check.

These same pairwise linear inequality constraints can be imposed when learning the parameters θ to ensure a monotonic function is learned. The following result establishes these constraints are sufficient and necessary for a 2^D lattice to be monotonically increasing in the dth feature (the result extends trivially to larger lattices):

Lemma 1 (Monotonicity Constraints) *Let $f(x) = \theta^T \phi(x)$ for $x \in [0,1]^D$ and $\phi(x)$ given in (2). The partial derivative $\partial f(x)/\partial x[d] > 0$ for fixed d and any x iff $\theta_{k'} > \theta_k$ for all k, k' such that $v_k[d] = 0$, $v_{k'}[d] = 1$ and $v_k[m] = v_{k'}[m]$ for all $m \neq d$.*

Proof First we show the constraints are necessary to ensure monotonicity. Consider the function values $f(v_k)$ and $f(v_{k'})$ for some adjacent pair of vertices $v_k, v_{k'}$ that differ only in the dth feature. For $f(v_k)$ and $f(v_{k'})$, all of the interpolation weight falls on θ_k or $\theta_{k'}$ respectively, such that $f(v_k) = \theta_k$ and $f(v_{k'}) = \theta_{k'}$. So $\theta_{k'} > \theta_k$ is necessary for $\partial f(x)/\partial x[d] > 0$ everywhere.

Next we show the constraints are sufficient to ensure monotonicity. Pair the terms in the interpolation $f(x) = \theta^T \phi(x)$ corresponding to adjacent parameters $\theta_k, \theta_{k'}$ so that for each k, k' it holds that $v_k[d] = 0, v_{k'}[d] = 1, v_k[m] = v_{k'}[m]$ for $m \neq d$:

$$\begin{aligned}
f(x) &= \sum_{k,k'} \theta_k \phi_k(x) + \theta_{k'} \phi_{k'}(x), \text{ then expand } \phi_k(x) \text{ and } \phi_{k'}(x) \text{ using (2) :} \\
&= \sum_{k,k'} \alpha_k \left(\theta_k x[d]^{v_k[d]} (1 - x[d])^{(1-v_k[d])} + \theta_{k'} x[d]^{v_{k'}[d]} (1 - x[d])^{(1-v_{k'}[d])} \right), \\
&\text{where } \alpha_k \text{ is the product of the } m \neq d \text{ terms in (2) that are the same for } k \text{ and } k', \\
&= \sum_{k,k'} \alpha_k \left(\theta_k (1 - x[d]) + \theta_{k'} x[d] \right) \text{ by the definition of } v_k \text{ and } v_{k'}. \qquad (7)
\end{aligned}$$

The partial derivative of (7) is $\frac{\partial f(x)}{\partial x[d]} = \sum_{k,k'} \alpha_k (\theta_{k'} - \theta_k)$. Because each $\alpha_k \in [0, 1]$, it is sufficient that $\theta_{k'} > \theta_k$ for each k, k' pair to guarantee this partial is positive for all x. ■

4.2 Monotonic Lattice Regression Objective

We relax strict monotonicity to monotonicity by allowing equality in the adjacent parameter constraints (for an example, see the second function from the left in Figure 1). Then the set of pairwise constraints can be expressed as $A\theta \leq 0$ for the appropriate sparse matrix A with one 1 and -1 per row of A, and one row per constraint. Each feature can independently be left unconstrained, or constrained to be either monotonically increasing or decreasing by the specification of A.

Thus the proposed monotonic lattice regression objective is convex with linear inequality constraints:

$$\arg\min_\theta \sum_{i=1}^n \ell(y_i, \theta^T \phi(x_i)) + R(\theta), \text{ s.t. } A\theta \leq b. \qquad (8)$$

Additional linear constraints can be included in $A\theta \leq b$ to also constrain the fitted function in other practical ways, such as $f(x) \in [0, 1]$ or $f(x) \geq 0$.

The approach extends to the standard learning to rank from pairs problem (Liu, 2011), where the training data is pairs of samples x_i^+ and x_i^- and the goal is to learn a function such that $f(x_i^+) \geq f(x_i^-)$ for as many pairs as possible. For this case, the monotonic lattice regression objective is:

$$\arg\min_\theta \sum_{i=1}^n \ell(1, \theta^T \phi(x_i^+) - \theta^T \phi(x_i^-)) + R(\theta), \text{ s.t. } A\theta \leq b. \qquad (9)$$

The loss functions in (6), (8) and (9) all have the same form, for example, squared loss $\ell(y, z) = (y - z)^2$, hinge loss $\ell(y, z) = \max(0, 1 - yz)$, or logistic loss $\ell(y, z) = \log(1 + \exp(y - z))$.

5. Faster Linear Interpolation

Interpolating a look-up table has long been considered an efficient way to specify and evaluate a low-dimensional non-linear function (Sharma and Bala, 2002; Garcia et al., 2012).

But computing linear interpolation weights with (3) requires $O(D)$ operations for each of the 2^D interpolation weights, for a total cost of $O(D2^D)$ to evaluate a sample. In Section 5.1, we show the multilinear interpolation weights of (3) can be computed in $O(2^D)$ operations. Then, in Section 5.2, we review and analyze a different linear interpolation that we refer to as *simplex* interpolation that takes only $O(D \log D)$ operations.

5.1 Fast Multilinear Interpolation

Much of the computation in (3) can be shared between the different weights. In Algorithm 1 we give a dynamic programming solution that loops D times, where the dth loop takes 2^d time, so in total there are $\sum_{d=0}^{D-1} 2^d = O(2^D)$ operations.

Algorithm 1 Computes the multilinear interpolation weights and corresponding vertex indices for a unit lattice cell $[0,1]^D$ and an $x \in [0,1]^D$. Let the lattice parameters be indexed such that $s_d = 2^d$ is the difference in the indices of the parameters corresponding to any two vertices that are adjacent in the dth dimension, for example, for the 2×2 lattice, order the vertices [0 0], [1 0], [0 1], [1 1] and index the corresponding lattice parameters in that order.

```
CalculateMultilinearInterpolationWeightsAndParameterIndices($x$)
    Initialize indices[] = [0], weights[] = [1]
    For $d = 0$ to $D - 1$:
        For $k = 0$ to $2^d - 1$:
            Append $s_d$ + indices[$k$] to indices
            Append $x[d] \times$ weights[$k$] to weights
            Update weights[$k$] = $(1 - (x[d])) \times$ weights[$k$]
    Return indices and weights
```

The following lemma establishes the correctness of Algorithm 1.

Lemma 2 (Fast Multilinear Interpolation) *Under its assumptions, Algorithm 1 returns the indices of the 2^D parameters corresponding to the vertices of the lattice cell containing x:*

$$\mathtt{indices}[k] = \sum_{d=0}^{D-1} \left(\lfloor x[d] \rfloor + \mathtt{bit}_i(k)\right) s_d, \text{ for } k = 1, 2, \ldots, 2^D \tag{10}$$

and the corresponding 2^D multilinear interpolation weights given by (3).

Proof At the end of the D'th iteration over the dimension in Algorithm 1:

$$\begin{aligned}
\mathtt{size}\left(\mathtt{indices}\right) &= \mathtt{size}\left(\mathtt{weights}\right) = 2^{D'+1} \\
\mathtt{indices}[k] &= \sum_{d=0}^{D'} \left(\lfloor x[d] \rfloor + \mathtt{bit}_d(k)\right) s_d \\
\mathtt{weights}[k] &= \prod_{d=0}^{D'} \left(\left(1 - \mathtt{bit}_d(k)\right)\left(1 - (x[d] - \lfloor x_d \rfloor)\right) + \mathtt{bit}_d(k)(x[d] - \lfloor x_d \rfloor)\right).
\end{aligned}$$

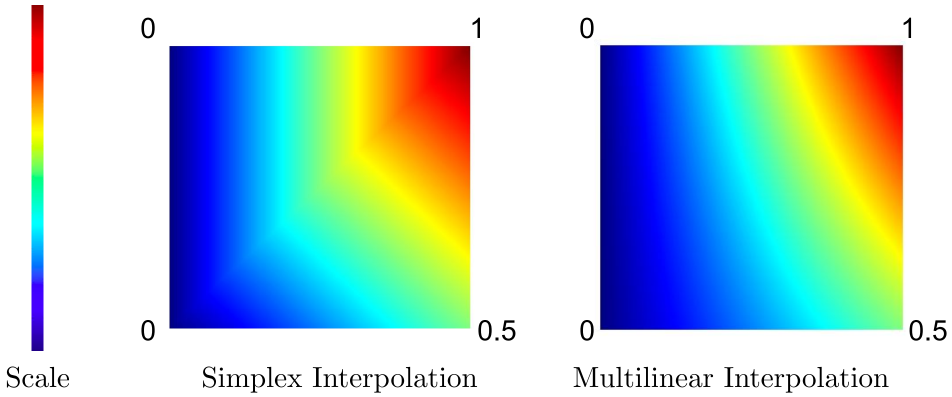


Figure 3: Illustration of two different linear interpolations of the same 2×2 look-up table with parameters $0, 0, 0.5$ and 1. The simplex interpolation splits the unit square into two simplices (the upper and lower triangle) and interpolates within each. The function is continuous because the points along the diagonal are interpolated from only the two corner vertices, and the interpolated function is linear over each simplex. Both interpolations produce monotonic functions over both features.

The above holds for the $D' = 1$ case by the initialization and inspection of the loop. It is straightforward to verify that if the above hold for D', then they also hold for $D' + 1$. Then by induction it holds for $D' = D - 1$, as claimed. ■

5.2 Simplex Linear Interpolation

For speed, we propose using a more efficient linear interpolation for lattice regression that linearly interpolates each x from only $D+1$ of the 2^D surrounding vertices. Many different linear interpolation strategies have been proposed to interpolate look-up tables using only a subset of the 2^D vertices (for a review, see Kang (1997)). However, most such strategies suffer from being too computationally expensive to determine the subset of vertices needed to interpolate a given x. The wonder of *simplex interpolation* is that it takes only $O(D \log D)$ operations to determine the $D+1$ vertices needed to interpolate any given x, and then only $O(D)$ operations to interpolate the identified $D + 1$ vertices. An illustrative comparison of simplex and multilinear interpolation is given in Figure 3 for the same lattice parameters.

Simplex interpolation was proposed in the color management literature by Kasson et al. (1993), and independently later by Rovatti et al. (1998). Simplex interpolation is also known as the *Lovasz extension* in submodular optimization, where it is used to extend a function defined on the vertices of a unit hypercube to be defined on its interior (Bach, 2013).

After reviewing how simplex interpolation works, we show in Section 5.2.3 that it requires the same constraints for monotonicity as multilinear interpolation, and then we discuss how its rotational dependence impacts its use for machine learning in Section 5.2.4. We give example runtime comparisons in Section 10.7.

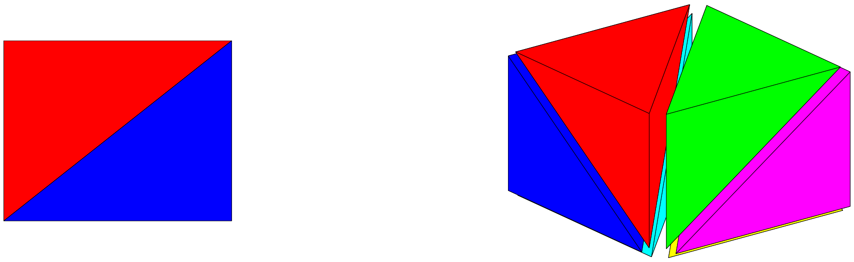


Figure 4: Simplex interpolation decomposes the D-dimensional unit hypercube into $D!$ simplices. **Left:** For the unit square, there are two simplices, one is defined by the three vertices [0 0], [0 1], and [1 1], and the other is defined by the three vertices [0 0], [1 0], and [1 1]. **Right:** For the unit cube there are $3! = 6$ simplices, each defined by four vertices. The first has vertices: [0 0 0], [0 0 1], [0 1 1] , [1 1 1]. The second has vertices: [0 0 0], [0 0 1], [1 0 1], [1 1 1]. And so on. All six simplices have vertices [0 0 0] and [1 1 1], and thus share the diagonal between those two vertices.

5.2.1 Partitioning of the Unit Hypercube Into Simplices

Simplex interpolation implicitly partitions the hypercube into the set of D! congruent simplices that satisfy the following: each simplex includes the all 0's vertex, one vertex is all zeros but has a single 1, one vertex is all zeros but has two 1's, and so on, ending with one vertex that is all 1's, for a total of $D + 1$ vertices in each simplex. Figure 4 shows the partitioning for the $D = 2$ and $D = 3$ unit hypercubes.

This decomposition can also be described by the hyperplanes $x_k = x_r$ for $1 \leq k \leq r \leq D$ (Schimdt and Simon, 2007). Knop (1973) discussed this decomposition as a special case of Eulerian partitioning of the hypercube, and Mead (1979) showed this is the smallest possible equivolume decomposition of the unit hypercube.

5.2.2 Simplex Interpolation

Given $x \in [0, 1]^D$, the $D + 1$ vertices that specify the simplex that contains x can be computed in $O(D \log D)$ operations by sorting the D values of the feature vector x, and then the dth simplex vertex has ones in the first d sorted components of x. For example, if $x =$[.8 .2 .3], the $D + 1$ vertices of its simplex are [0 0 0], [1 0 0], [1 0 1], [1 1 1].

Let V be the $D+1$ by D matrix whose dth row is the dth vertex of the simplex containing x. Then the simplex interpolation weights $\psi(x)$ must satisfy the linear interpolation equations given in (5) such that $\begin{bmatrix} V^T \\ \mathbf{1}^T \end{bmatrix} \psi(x) = \begin{bmatrix} x \\ 1 \end{bmatrix}$. Thus $\psi(x) = \begin{bmatrix} V^T \\ \mathbf{1}^T \end{bmatrix}^{-1} \begin{bmatrix} x \\ 1 \end{bmatrix}$, where because of the highly structured nature of the simplex decomposition the required inverse always exists, and has a simple form such that $\psi(x)$ is the vector of differences of sequential sorted components of x. For example, for a $2 \times 2 \times 2$ lattice, and an x such that $x[0] > x[1] > x[2]$, the simplex interpolation weights $\psi(x) = [1 - x[0], x[0] - x[1], x[1] - x[2], x[2]]$ on the vertices

$[0,0,0], [1,0,0], [1,1,0], [1,1,1]$, respectively. The general formula is detailed in Algorithm 2; for more mathematical details see Rovatti et al. (1998).

Algorithm 2 Computes the simplex interpolation weights and corresponding vertex indices for a unit lattice cell $[0,1]^D$ and an $x \in [0,1]^D$. Let the lattice parameters be indexed such that $s_d = 2^d$ is the difference in the indices of the parameters corresponding to any two vertices that are adjacent in the dth dimension, for example, for the 2×2 lattice, order the vertices [0 0], [1 0], [0 1], [1 1] and index the corresponding lattice parameters in that order.

```
CalculateSimplexInterpolationWeightsAndParameterIndices($x$)
    Compute the sorted order $\pi$ of the components of $x$ such that $x[\pi[k]]$ is the $k$th largest value of $x$,
                that is, $x[\pi[1]]$ is the largest value of $x$, etc.
    Initialize index $= 0$, $z = 1$, indices[] = [], weights[] = []
    For $d = 0$ to $D - 1$:
        Append index to indices
        Append $z - x[\pi[d]]$ to weights
        Update index = index + $s_{\pi[d]}$
        Update $z = x[\pi[d]]$
    Append index to indices
    Append $z$ to weights
    Return indices and weights
```

5.2.3 Simplex Interpolation and Monotonicity

We show that the same linear inequality constraints that guarantee monotonicity for multilinear interpolation also guarantee monotonicity with simplex interpolation:

Lemma 3 (Monotonic Constraints with Simplex Interpolation) *Let $f(x) = \theta^T \phi(x)$ for $\phi(x)$ given in Algorithm 2. The partial derivative $\partial f(x)/\partial x[d] > 0$ iff $\theta_{k'} > \theta_k$ for all k, k' such that $v_k[d] = 0$, $v_{k'}[d] = 1$, and $v_k[m] = v_{k'}[m]$ for all $m \neq d$.*

Proof Simplex interpolation linearly interpolates from $D + 1$ vertices at a time, and thus the resulting function is linear over each simplex. Thus to prove that the function is monotonic everywhere, we need only to show that each locally linear function is monotonically increasing in dimension d, and that the function is continuous everywhere. Each simplex only has one pair of vertices v_k and $v_{k'}$ that differ in dimension d such that $v_k[d] = 0$, $v_{k'}[d] = 1$, and $v_k[m] = v_{k'}[m]$ for all $m \neq d$. In addition, we can verify that for the linear function over this simplex, $\partial f(x)/\partial x[d] = \theta_{k'} - \theta_k$, where θ_k and $\theta_{k'}$ are the parameters corresponding to these vertices. Therefore if $\theta_{k'} > \theta_k$, then the linear function over that simplex must be increasing with respect to d. Conversely, if it does not hold that $\theta_{k'} > \theta_k$, then the linear function over that simplex must have non-positive slope with respect to d. Further, $f(x)$ is continuous for all x, because any x on a boundary between simplices only has nonzero interpolation weight on the vertices defining that boundary. In conclusion, the function is piecewise monotonic and continuous, and thus monotonic everywhere. ■

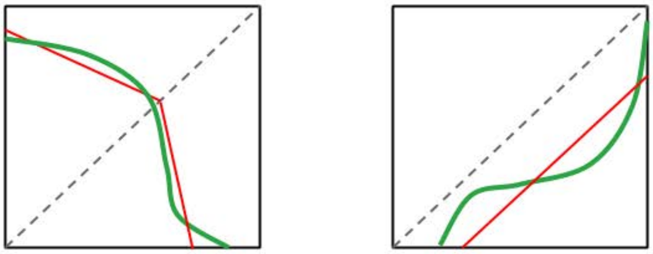


Figure 5: Illustrates rotational dependence of simplex interpolation for a 2 × 2 lattice and its impact on a binary classification problem. Green thick line denotes the true decision boundary of a binary classification problem. Red thin lines denote the piecewise linear decision boundary fit by lattice regression using simplex interpolation. Dotted gray line separates the two simplices; the function is locally linear over each simplex. **Left:** The true decision boundary (green) crosses the two simplices. The simplex decision boundary (red) has two linear pieces and can fit the green boundary well. **Right:** The same green boundary has been rotated ninety degrees, and now lies entirely in one simplex. The simplex decision boundary (in red) is linear within each simplex, and hence has less flexibility to fit the true green decision boundary.

5.2.4 Using Simplex Interpolation for Machine Learning

Simplex interpolation produces a locally linear continuous function made-up of $D!$ hyperplanes oriented around the main diagonal axis of the unit hypercube. Compared to multilinear interpolation, simplex interpolation is not as smooth (though continuous), and it is rotationally-dependent.

For low-dimensional regression problems using a look-up table with many cells, performance of the two interpolation methods has been found to be similar, particularly if one is using a fine-grained lattice with many cells. For example, in a comparison by Sun and Zhou (2012) for the three-dimensional regression problem of color managing an LCD monitor, multilinear interpolation of a $9 \times 9 \times 9$ look-up table (also called trilinear interpolation in the special case of three-dimensions) produced around 1% worse average error than simplex interpolation, but the maximum error with multilinear interpolation was only 60% of the maximum simplex interpolation error. Another study by Kang (1995) using simulations concluded that the interpolation errors of these methods was “about the same.”

However, when using a coarser lattice like 2^D, as we have found useful in practice for machine learning, the rotational dependence of simplex interpolation can cause problems because the flexibility of the interpolated function $f(x)$ differs in different parts of the feature space. Figure 5 illustrates this for a binary classifier on two features.

To address the rotational dependence, we recommend using prior knowledge to define the features positively or negatively in a way that aligns the simplices’ shared diagonal axis along the assumed slope of $f(x)$. If there are monotonicity constraints, this is done by specifying each feature so that it is monotonically increasing, rather than monotonically decreasing. For binary classification, features should be specified so that the feature vector for the most prototypical example of the negative class is the all-zeros feature vector, and the feature vector for the most prototypical example of a positive class is the all-ones feature vector. This should put the decision boundary as orthogonal to the shared diagonal axis as possible, providing the interpolated function the most flexibility to model that decision boundary. In addition, for low-dimensional problems, using a finer-grained lattice will produce more flexibility overall, so that the flexibility within each lattice cell is less of an issue.

Following these guidelines, we surprisingly and consistently find that simplex interpolation of 2^D lattices is roughly as accurate as multilinear interpolation, and much faster for $D \geq 8$. This is demonstrated in the case studies of Section 10 (runtime comparisons given in Section 10.7).

6. Regularizing the Lattice Regression To Be More Linear

We propose a new regularizer that takes advantage of the lattice structure and encourages the fitted function to be more linear by penalizing differences in parallel edges:

$$R_{\text{torsion}}(\theta) = \sum_{d=1}^{D} \sum_{\substack{\tilde{d}=1 \\ \tilde{d} \neq d}}^{D} \sum_{\substack{r,s,t,u \text{ such that} \\ v_r \text{ and } v_s \text{ adjacent in dimension } d, \\ v_t \text{ and } v_u \text{ adjacent in dimension } d, \\ v_r \text{ and } v_t \text{ adjacent in dimension } \tilde{d}}} ((\theta_r - \theta_s) - (\theta_t - \theta_u))^2. \qquad (11)$$

This regularizer penalizes how much the lattice function twists from side-to-side, and hence we refer to this as the *torsion* regularizer. The larger the weight on the torsion regularizer in the objective function, the more linear the lattice function will be over each 2^D lattice cell.

Figure 6 illustrates the torsion regularizer and compares it to previously proposed lattice regularizers, the standard graph Laplacian (Garcia and Gupta, 2009) and graph Hessian (Garcia et al., 2012). As shown in the figure, for multi-cell lattices, the torsion and graph Hessian regularizers make the function more linear in different ways, and may both be needed to closely approximate a linear function. Like the graph Laplacian and graph Hessian regularizers, the proposed torsion regularizer is convex but not strictly convex, and can be expressed in quadratic form as $\theta^T K \theta$, where K is a positive semidefinite matrix.

7. Jointly Learning Feature Calibrations

One can learn arbitrary bounded functions with a sufficiently fine-grained lattice, but increasing the number of lattice vertices M_d for the dth feature multiplicatively grows the total number of parameters $M = \prod_d M_d$. However, we find in practice that if the features are first each transformed appropriately, then many problems require only a 2^D lattice to capture the feature interactions. For example, a feature that measures distance might be better specified as log of the distance. Instead of relying on a user to determine how to best transform each feature, we automate this feature pre-processing by augmenting our function class with D one-dimensional transformations $c_d(x[d])$ that we learn jointly with the lattice, as shown in Figure 7.

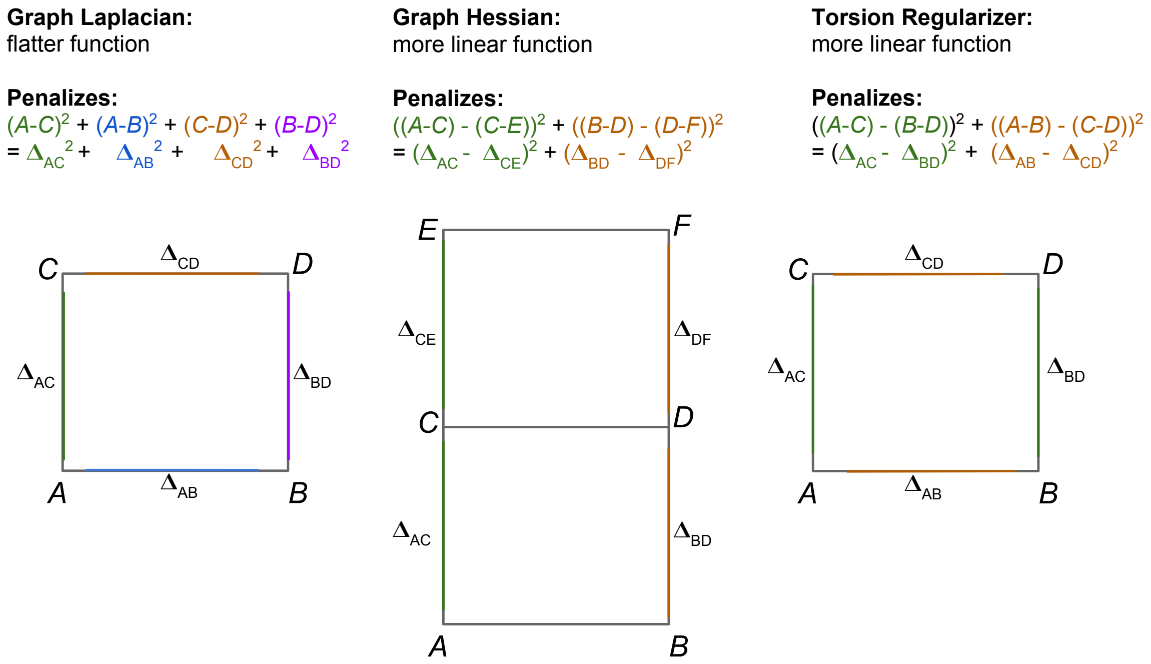

Graph Laplacian:
flatter function
Penalizes:
(A-C)^2 + (A-B)^2 + (C-D)^2 + (B-D)^2
= $\Delta_{AC}{}^2$ + $\Delta_{AB}{}^2$ + $\Delta_{CD}{}^2$ + $\Delta_{BD}{}^2$
C
D
A
B
Δ_{CD}
Δ_{AC}
Δ_{BD}
Δ_{AB}
Graph Hessian:
more linear function
Penalizes:
((A-C) - (C-E))^2 + ((B-D) - (D-F))^2
= (Δ_{AC} - Δ_{CE})^2 + (Δ_{BD} - Δ_{DF})^2
E
F
Δ_{CE}
Δ_{DF}
C
D
Δ_{AC}
Δ_{BD}
A
B
Torsion Regularizer:
more linear function
Penalizes:
((A-C) - (B-D))^2 + ((A-B) - (C-D))^2
= (Δ_{AC} - Δ_{BD})^2 + (Δ_{AB} - Δ_{CD})^2
C
D
Δ_{CD}
Δ_{AC}
Δ_{BD}
A
B
Δ_{AB}

Figure 6: Comparison of lattice regularizers. The lattice parameters are denoted by A, B, C, D, E, and F. The deltas indicate the differences between adjacent parameters along each edge, and thus each delta is the slope of the function along its edge. Each color corresponds to a different additive term in a regularizer. The graph Laplacian regularizer *(left)* minimizes the sum of the squared slopes, producing a flatter function. The graph Hessian regularizer *(middle)* minimizes the change in slope in each direction of a multi-cell lattice, keeping the function from bending too much between lattice cells. The proposed torsion regularizer *(right)* minimizes the change in slope between sides of the lattice, for each direction, minimizing the twisting of the function.

7.1 Calibrating Continuous Features

We calibrate each continuous feature with a one-dimensional monotonic piecewise linear function, as illustrated in Figure 8. Our approach is similar to the work of Howard and Jebara (2007), which jointly learns monotonic piecewise linear one-dimensional transformations and a linear function.

This joint estimation makes the objective non-convex, discussed further in Section 9.3. To simplify estimating the parameters, we treat the number of changepoints C_d for the dth feature as a hyperparameter, and fix the C_d changepoint locations (also called knots) at equally-spaced quantiles of the feature values. The changepoint values are then optimized jointly with the lattice parameters, detailed in Section 9.3.

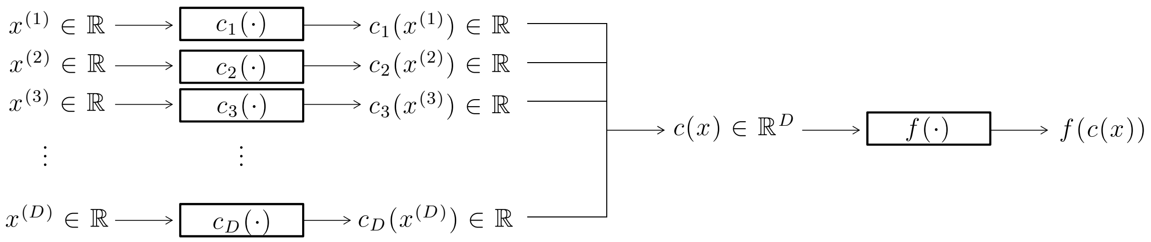
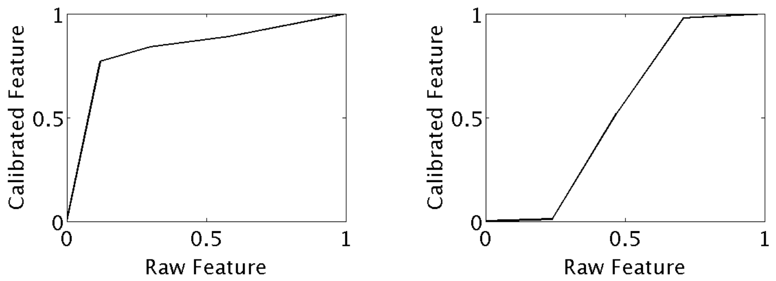


Figure 7: Block diagram showing one-dimensional calibration functions $\{c_d(\cdot)\}$ to pre-process each feature before the lattice $f(\cdot)$ fuses the features together nonlinearly.



Distance Calibration

Address Similarity Calibration

Figure 8: Learned one-dimensional piecewise linear calibration functions for a distance and address-similarity feature for the business-matching case study in Section 10.2. **Left:** The raw distance is measured in meters, and its calibration has a log-like effect. **Right:** The raw address feature is calibrated with a sigmoid-like transformation.

7.2 Calibrating Categorical Features

If the dth feature is categorical, we propose using a calibration function $c_d(\cdot)$ to map each category to a real value in $[0, M_d - 1]$. That is, let the set of possible categories for the dth feature be denoted $\mathcal{G}_d$, then $c_d : \mathcal{G}_d \to [0, M_d - 1]$, adding $|\mathcal{G}_d|$ additional parameters to the model. Figure 9 shows an example lattice with a categorical country feature that has been calibrated to lie on $[0, 2]$. If prior knowledge is given about the ordering of the original discrete values or categories, then partial or full pairwise constraints can be added on the mapped values to respect the known ordering information. These can be expressed as additional sparse linear constraints on pairs of parameters.

8. Calibrating Missing Data and Using Missing Data Vertices

We propose two supervised approaches to handle missing values in the training or test set.

First, one can do a supervised imputation of missing data values by calibrating a missing data value for each feature. This is the same approach proposed for calibrating categorical

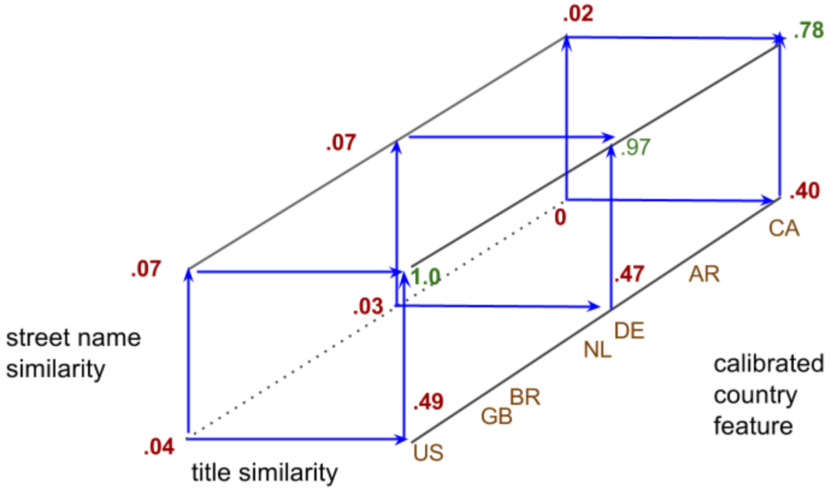
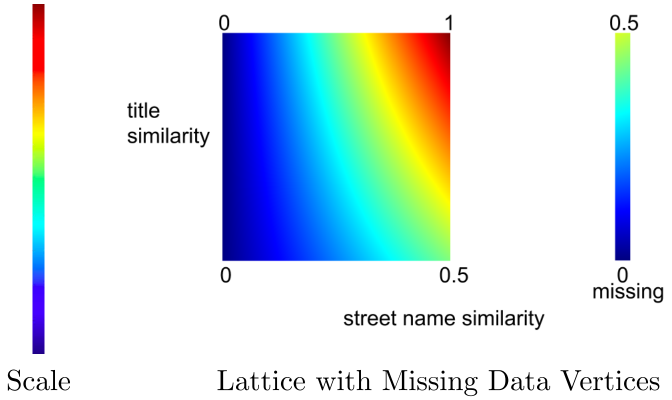


Figure 9: A 2 × 2 × 3 lattice illustrating calibrating a categorical feature. In this example, each sample is a pair of business listings, and the problem is to classify whether the two listings are about the same business, based on the similarity of their street names, titles, and the country. A score $f(x)$ is interpolated from the parameters corresponding to the vertices of the 2 × 2 × 2 lattice cell in which x lies, then thresholded at 0.5. The red parameter values are below the matching threshold of 0.50, and the green parameters are above the matching threshold. The blue arrows denote that the lattice was constrained to be monotonically increasing in the street name similarity and the title similarity. In this toy example, we only show the calibrated values for a few countries: US maps to 0, Great Britain maps to .3, Brazil to .4, Netherlands to .9, Germany to 1, Argentina to 1.5, and Canada to 2. One can interpret this lattice as modeling three classifiers, sliced along the country vertices: a classifier for country = 0, one for country = 1, and one for country = 2. Samples from Argentina (AR) are interpolated equally from the parameters where country = 1 and country = 2. Samples from Great Britain, and Netherlands are interpolated from the two classifiers specified at country = 0 and 1, with Netherlands putting the most weight on the classifier where country = 1. The lattice parameters can be interpreted as showing that both the street name and title features are stronger positive evidence in the US than in Canada.

values in Section 7.2: learn the numeric value in $[0, M_d - 1]$ to impute if the dth feature is missing that minimizes the structural risk minimization obejctive. In this approach, missing data is handled by a calibration function $c_d(\cdot)$, and like the other calibration function parameters. Other researchers have also considered joint training of classifiers and imputations for missing data, for example van Esbroeck et al. (2014) and Liao et al. (2007).

Second, a more flexible option is to give missing data its own *missing data vertices* in the lattice, as shown in Figure 10. This is similar to a decision tree handling a missing data value by splitting a node on whether that feature is missing. For example, the non-



Scale

Lattice with Missing Data Vertices

Figure 10: Illustration of handling missing data by assigning missing data to its own slice of vertices in the lattice. In this example, one feature is a title similarity and is always given, and the other feature is a street name similarity that can be missing. The lattice has $3 \times 2 = 6$ parameters, with the parameter values shown. For example, given a feature vector x with title similarity 0.5 and missing street name similarity, the two parameters corresponding to the missing street name slice of the lattice would be interpolated with equal weights, producing the output $f(x) = 0.25$.

missing feature values can be scaled to $[0, M_d - 2]$, and if the data is missing is it mapped to $M_d - 1$. This increases the number of parameters but gives the model the flexibility to handle missing data differently than non-missing data. For example, missing the street number in a business description may correlate with lower quality information for all the features.

To regularize the lattice parameters corresponding to missing data vertices, we apply the graph regularizers detailed in Section 6. These could be use to tie any of the parameters to the missing data parameters. In our experiments, for the purposes of graph regularization, we treat the missing data vertices as though they were adjacent to the minimum and maximum vertices of that feature in the lattice.

With either of these two proposed strategies, linear inequalities can be added on the appropriate parameters (the calibrator parameters in the first proposal, or the missing data vertex parameters in the second proposal) to ensure that the function value for missing data is bounded by the minimum and maximum function values, that is, that missing $x[d]$ never produces a smaller $f(x)$ than $x[d] = 0$, nor a larger $f(x)$ than $x[d] = M_d$.

9. Large-Scale Training

For convex loss functions $\ell(\theta)$ and convex regularizers $R(\theta)$, any solver for convex problems with linear inequality constraints can be used to optimize the lattice parameters θ in (8). However, for large n and for even relatively small D, training the proposed calibrated monotonic lattices is challenging due to the number of linear inequality constraints, the number of terms in the graph-regularizers, and the non-convexity created by using calibration functions.

In this section we discuss various standard and new strategies we found useful in practice: our use of stochastic gradient descent (SGD), stochastic handling of the regularizers, parallelizing-and-averaging for distributed training, handling the large number of constraints in the context of SGD, and finally some details on how we optimize the non-convex problem of training the calibrator functions and the lattice parameters. Throughout this section, we assume the standard setting of (8); the generalization to the pairwise ranking problem of (9) is straightforward.

9.1 SGD and Reducing Variance of the Subgradients

To scale to a large number of samples n, we used SGD for all our experiments. For each SGD iteration t, a labeled training sample (x_i, y_i) is sampled uniformly from the set of training sample pairs. One finds the corresponding subgradient of (8), and takes a tiny step in its negative gradient direction. (The resulting parameters may then violate the constraints, which we discuss in Section 9.4.)

A straightforward SGD implementation for (8) would use the subgradient:

$$\Delta = \nabla_\theta \ell \left(\theta^T \phi\left(x_i\right), y_i\right) + \nabla_\theta R\left(\theta\right), \tag{12}$$

where the ∇_θ operator finds an arbitrary subgradient of its argument w.r.t. θ. Ideally, these subgradients should be cheap-to-compute, so each iteration is fast. The computational cost is dominated by computing the regularizer, if using any of the graph regularizers discussed in Section 6.

Because the training example (x_i, y_i) in (12) is randomly sampled, the above subgradient is a realization of a stochastic subgradient whose expectation is equal to the true gradient. The number of iterations needed for the SGD to converge depends on the squared Euclidean norms of the stochastic subgradients (Nemirovski et al., 2009), with larger norms resulting in slower convergence. The expected squared norm of the stochastic subgradient can be decomposed into the sum of two terms: the squared expected subgradient magnitude, and the variance. We can do little about the expected magnitude, but we can improve the trade-off between the computational cost of each subgradient and the variance of the stochastic subgradients. In the next two sub-sections, we describe two such strategies.

9.1.1 Mini-Batching

We reduce the variance of the stochastic subgradient's loss term by mini-batching over multiple random samples (Dekel et al., 2012). Let $\mathcal{S}_\ell$ denote a set of k_ℓ training indices sampled uniformly with replacement from $1, \ldots, n$, then the mini-batched subgradient is:

$$\Delta = \frac{1}{k_\ell} \sum_{i \in \mathcal{S}_\ell} \nabla_\theta \ell \left(\theta^T \phi\left(x_i\right), y_i\right) + \nabla_\theta R\left(\theta\right). \tag{13}$$

This simultaneously reduces the variance and increases the computational cost of the loss term by a factor of k_ℓ. For sufficiently small k_ℓ, this is a net win because differentiating the regularizer is the dominant computational term.

9.1.2 Stochastic Subgradients for Regularizers

We propose to reduce the computational cost of each SGD iteration by randomly sampling the additive terms of the regularizer, for regularizers that can be expressed as a sum of terms: $R(\theta) = \sum_{j=1}^{m} r_j(\theta)$. For example, for a 2^D lattice, each calculation of the graph Laplacian regularizer subgradient sums over $m = D2^{D-1}$ terms, and the graph torsion regularizer subgradient sums over $m = D(D-1)2^{D-3}$ terms.

Let $\mathcal{S}_R$ denote a set of k_R indices sampled uniformly with replacement from $1, l \ldots, m$, then define the subgradient:

$$\Delta = \frac{1}{k_\ell} \sum_{i \in \mathcal{S}_\ell} \nabla_\theta \ell \left(\theta^T \phi\left(X_i\right), Y_i\right) + \frac{m}{k_R} \sum_{j \in \mathcal{S}_R} \nabla_\theta r_j\left(\theta\right). \tag{14}$$

While this makes the subgradient's regularizer term stochastic, and hence increases the subgradient variance, we find that good choices of k_ℓ and k_R in (14) can produce a useful tradeoff between the computational cost of computing each subgradient and the number of SGD iterations needed for acceptable converge. For example, in one real-world application using torsion regularization, the choice of $k_R = 1024$ and $k_\ell = 1$ led to a 150× speed-up in training and produced statistically indistinguishable accuracy on a held-out test set.

9.2 Parallelizing and Averaging

For a large number of training samples n, one can split the n training samples into K sets, then independently and in-parallel train a lattice on each of the K sets. Once trained, the vector lattice parameters for the K lattices can simply be averaged. This parallelize-and-average approach was investigated for large-scale training of linear models by Mann et al. (2009). Their results showed similar accuracies to distributed gradient descent, but 1000× less network traffic and reduced wall-clock time for large datasets. In our implementation of the parallelize-and-average approach we do multiple syncs: averaging the lattices, then sending out the averaged lattice to parallelized workers to keep improving with further training. We illustrate the performance and speed-up of this simple parallelize-and-average for learning monotonic lattices in Section 10.6 and Section 10.7. A more complicated implementation of this strategy would use the alternating direction method of multipliers with a consensus constraint (Boyd et al., 2010), but that requires an additional regularization towards a local copy of the most recent consensus parameters.

Note that if calibration functions are used, they must be held fixed during the parallelization of the lattice training, as it does not make mathematical sense to average differently calibrated lattices.

9.3 Jointly Optimizing Lattice and Calibration Functions

To learn a *calibrated* monotonic lattice, we jointly optimize the calibration functions and the lattice parameters. Let x denote a feature vector with D components, each of which is either a continuous or categorical value (discrete features can be modeled either as continuous features or categorical as the user sees fit). Let $c_d(x[d]; \alpha^{(d)})$ be a calibration function that acts on the dth component of x and has parameters $\alpha^{(d)}$.

If the dth feature is continuous, we assume it has a bounded domain such that $x[d] \in [l_d, u_d]$ for finite $l_d, u_d \in \mathbb{R}$. Then the dth calibration function $c_d(x[d]; \alpha^{(d)})$ is a monotonic piecewise linear transform with fixed knots at l_d, u_d, and the $C_d - 2$ equally-spaced quantiles of dth feature over the training set. Let the first and last knots of the piecewise linear function map to the lattice bounds 0 and $M_d - 1$ respectively (as shown in Figure 8), that is, if $C_d = 2$ then $c_d(x[d]; \alpha^{(d)})$ simply linearly scales the raw range $[l_d, u_d]$ to the lattice domain $[0, M_d - 1]$ and there are no parameters $\alpha^{(d)}$. For $C_d > 2$, the parameters $\alpha^{(d)} \in [0, M_d - 1]^{C_d - 2}$ are the $C_d - 2$ output values of the piecewise linear function for the middle $C_d - 2$ knots.

If the dth feature is categorical with finite category set $\mathcal{G}_d$ such that $x[d] \in \mathcal{G}_d$, then the dth calibration function maps the categories to the lattice span such that $c_d(x[d]; \alpha^{(d)}) : \mathcal{G}_d \to [0, M_d - 1]$ and the parameters are the $|\mathcal{G}_d|$ categorical mappings such that $c_d(x[d]; \alpha^{(d)}) = \alpha^{(d)}[k]$ if $x[d]$ belongs to category k and $\alpha^{(d)} \in [0, M_d - 1]^{|\mathcal{G}_d|}$.

Let $c(x; \alpha)$ denote the vector function with dth component function $c_d(x[d]; \alpha^{(d)})$, and note $c(x; \alpha)$ maps a feature vector x to the domain $\mathcal{M}$ of the lattice function. Use e_d to denote the standard unit basis vector that is one for the dth component and zero elsewhere with length D, then one can write:

$$c(x; \alpha) = \sum_{d=1}^{D} e_d c_d(e_d^T x; \alpha^{(d)}), \tag{15}$$

Then the proposed calibrated monotonic lattice regression objective expands the monotonic lattice regression objective (8) to:

$$\arg\min_{\theta, \alpha} \sum_{i=1}^{n} \ell(y_i, \theta^T \phi(c(x_i, \alpha)) + R(\theta) \text{ s.t. } A\theta \leq b \text{ and } \tilde{A}\alpha \leq \tilde{b},$$

where each row of A specifies a monotonicity constraint for a pair of adjacent lattice parameters (as before), and each row of $\tilde{A}$ similarly specifies a monotonicity constraint for a pair of adjacent calibration parameters for one of the piecewise linear calibration functions.

This turns the convex optimization problem (8) into a non-convex problem that is marginally convex in the lattice parameters θ for fixed α, but not necessarily convex with respect to α even if θ is fixed. Despite the non-convexity of the objective, in our experiments we found sensible and effective solutions by using projected SGD, updating θ and α with the appropriate stochastic subgradient for each x_i. Calculate the subgradient w.r.t. θ holding α constant, essentially the same as before. Calculate the subgradient w.r.t α by holding θ constant and using the chain rule:

$$\frac{\partial \theta^T \phi(c(x_i, \alpha))}{\partial \alpha^{(d)}} = \frac{\partial \theta^T \phi(c(x_i, \alpha))}{\partial c(x_i, \alpha)} \frac{\partial c(x_i, \alpha)}{\partial \alpha^{(d)}}. \tag{16}$$

If the dth feature is categorical, the partial derivative is 1 for the calibration mapping parameter corresponding to the category of $x_i[d]$ and zero otherwise:

$$\frac{\partial c(x_i, \alpha)}{\partial \alpha^{(d)}[k]} = 1 \text{ if } x_i[d] \text{ is the } k\text{th category and 0 otherwise.} \tag{17}$$

If the dth feature is continuous, then the parameters $\alpha^{(j)}[d]$ are the values of the calibration function at the knots of the piecewise linear function. If $x_i[d]$ lies between the kth and $(k+1)$th knots at (fixed) positions β_k and β_{k+1}, then

$$\frac{\partial c(x_i, \alpha)}{\partial \alpha^{(d)}[k]} = \frac{(\beta_{k+1} - x_i[d])}{(\beta_{k+1} - \beta_k)}$$
$$\frac{\partial c(x_i, \alpha)}{\partial \alpha^{(d)}[k+1]} = \frac{(x_i[d] - \beta_k)}{(\beta_{k+1} - \beta_k)},$$

and the partial derviative is zero for all other components of $\alpha^{(d)}$. After taking an SGD step that updates $\alpha^{(d)}[k]$ and $\alpha^{(d)}[k+1]$, the $\alpha^{(d)}$ may violate the monotonicity constraints that ensure a monotonic calibration function, which can be fixed with a projection onto the constraints (see Section 9.4 for details).

A standard strategy with nonconvex gradient descent is to try multiple random initializations of the parameters. We did not explore this avenue; instead we simply try to initialize sensibly. Each lattice parameter is initialized to be the sum of its monotonically increasing components (multiply by -1 for any monotonically decreasing components) so that the lattice initialization respects the monotonicity constraints and is a linear function. The piecewise linear calibration functions are initialized to scale linearly to $[0, M_d - 1]$. The categorical calibration parameters are ordered by their mean label, then spaced uniformly on $[0, M_d - 1]$ in that order.

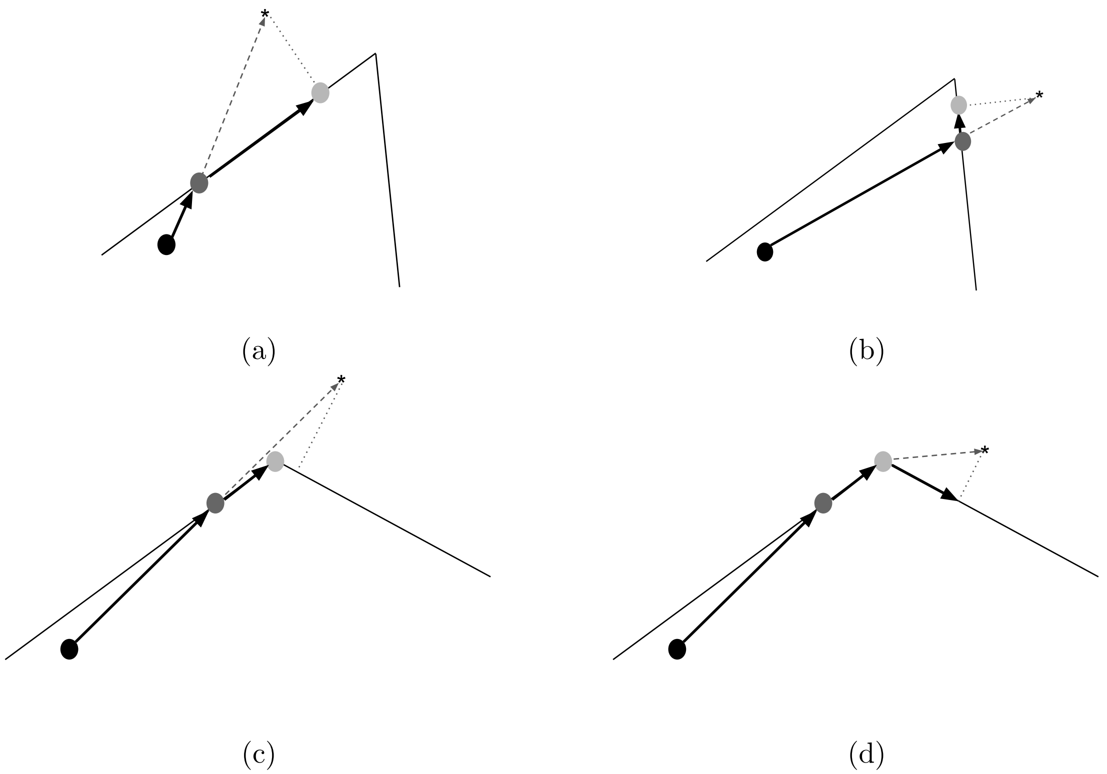
9.4 Large-Scale Projection Handling

Standard projected stochastic gradient descent projects the parameters onto the constraints after each stochastic gradient update. Given the extremely large number of linear inequality constraints needed to enforce monotonicity for even small D, we found a full projection each iteration impractical and un-necessary. We avoid the full projection at each iterate by using one of two strategies.

9.4.1 Suboptimal Projections

We found that modifying the SGD update to approximate the projection worked well. Specifically, for each new stochastic subgradient $\eta\Delta$, we create a set of active constraints initialized to $\emptyset$, and, starting from the last parameter values, move along the portion of $\eta\Delta$ that is orthogonal to the current active set until we encounter a constraint, add this constraint to the active set, and then continue until the update $\eta\Delta$ is exhausted or it is not possible to move orthogonal to the current active set. At all times, the parameters satisfy the constraints. It can be particularly fast because it is possible to exploit the sparsity of the monotonicity constraints (each of which depends on only two parameters) and the sparsity of Δ (when using simplex interpolation) to optimize the implementation.

But, this strategy is sub-optimal because we do not remove any constraints from the active set during each iteration, and thus parameters can "get stuck" at a corner of the feasible set, as illustrated in Figure 11. In practice, we found such problems resolve themselves because the stochasticity of the subsequent stochastic gradients eventually jiggles the parameters free. Experimentally, we found this suboptimal strategy to be very effective and to produce statistically similar objective function values and test accuracies more optimal approaches. All of the experimental results reported in this paper used this strategy. See Section 10.7 for example runtimes.



(a)

(b)

(c)

(d)

Figure 11: Four examples of the suboptimal projection stochastic gradient descent step described in Section 9.4. In each case, the constraints are marked by thin solid lines, the black dot represents the parameters at the end of the last SGD iteration, and the new full stochastic gradient descent update is marked by the dashed line, ending in a star. The optimal projection of the star onto the constraints is marked by the dotted line. The stochastic gradient is followed until it hits a constraint, and then the component of the remaining gradient orthogonal to the active constraint is applied. The update ends at the light gray dot. In cases (a) and (b), the resulting light dot is the optimal projection of the star onto the constraints. But in case (c), first one constraint is hit, and then another constraint is hit, and the update gets stuck at the corner of the feasible set without being able to apply all of the stochastic gradient. The resulting light gray dot is *not* the projection of the star onto the constraints, hence the projection for this iteration is suboptimal. However, it is likely that a future stochastic gradient will jiggle the optimization loose, as pictured in (d), producing an update that is again an optimal projection of the latest stochastic gradient.

9.4.2 Stochastic Constraints with LightTouch

An optimal approach we compared with for handling large-scale constraints is called *LightTouch* (Cotter et al., 2015). At each iteration, LightTouch does not project onto any constraints, but rather moves the constraints into the objective, and applies a random subset of constraints each iteration as stochastic gradient updates to the parameters, where the distribution over the constraints is learned as the optimization proceeds to focus on constraints that are more likely to be active. This replaces the per-iteration projections with cheap gradient updates. Intermediate solutions may not satisfy all the constraints, but one full projection is performed at the very end to ensure final satisfaction of the constraints. Experimentally, we found LightTouch generally converged faster (see Cotter et al. (2015) for its theoretical convergence rate), while producing similar experimental results to the above approximate projected SGD. LightTouch does require a more complicated implementation to effectively learn the distribution over the constraints.

9.4.3 Adapting Stepsizes with Adagrad

One can generally improve the speed of SGD with adagrad (Duchi et al., 2011), even for nonconvex problems (Gupta et al., 2014). Adagrad decays the step-size adaptively for each parameter, so that parameters updated more often or with larger magnitude gradients have a smaller step size. We found adagrad did speed up convergence slightly, but required a complicated implementation to correctly handle the constraints because the projections must be with respect to the adagrad norm rather than the Euclidean norm. We experimented with approximating the adagrad norm projection with the Euclidean projection, but found this approximation resulted in poor convergence. The experimental results did not make use of adagrad.

10. Case Studies

We present a series of experimental case studies on real world problems to demonstrate different aspects of the proposed methods, followed by some example runtimes for interpolation and training in Section 10.7, and some observations about the practical value of imposing monotonicity in Section 10.8.

Previous datasets used to evaluate monotonic algorithms have been small, both in the number of samples and the number of dimensions, as detailed in Table 1. In order to produce statistically significant experimental results, and to better demonstrate the practical need for monotonicity constraints, we use real-world case studies with relatively large datasets, and for which the application engineers have confirmed that they expect or want the learned function to be monotonic with respect to some subset of features. The datasets used are detailed in Table 3, and include datasets with eight thousand to 400 million samples, and nine to sixteen features, most of which are constrained to be monotonic.

The case studies demonstrate that for problems where the monotonicity assumption is warranted, the proposed calibrated monotonic lattice regression produces similar accuracy to random forests. Random forests is an unconstrained method that consistently provides competitive results on benchmark datasets, compared to many other types of machine learning methods (Fernandez-Delgado et al., 2014)).

Because any bounded function can be expressed using a sufficiently fine-grained interpolation look-up table, we expect that with appropriate use of regularizers, monotonic lattice regression will perform similarly to other guaranteed monotonic methods that use a flexible function class and are appropriately regularized, such as monotonic neural nets (see 2.2.5). However, of guaranteed monotonic methods, the only monotonic strategy that has

Dataset	# Training Samples	# Test Samples	# Features	# Lattice Parameters
Business Matching	8,000	4,000	9	1728
Ad–Query Matching	235,996	58,224	5	32
Rendering Classifier	20,000	2,500	16	65,536
Fusing Pipelines	1.6 million	390k	12	24,576
Video Ranking	400 million	25 million	12	531,441

Table 3: Summary of datasets used in the case studies.

been demonstrated to scale to the number of training samples and the number of features treated in our case studies is linear regression with non-negative coefficients (see Table 1).

10.1 General Experimental Details

We used ten-fold cross-validation on each training set to choose hyperparameters, including: whether to use graph Laplacian regularization or torsion regularization, how much regularization (in powers of ten), whether to calibrate missing data or use a missing data vertex, the number of change-points if feature calibration was used from the choices: $\{2, 3, 5, 10, 20, 50\}$, and the number of vertices for each feature was started at 2 and increased by 1 as long as cross-validation accuracy increased. The step size was tuned using ten-fold cross-validation and choices were powers of 10; it was usually chosen to be one of $\{.01, .1, 1\}$. If calibration functions were used, a hyperparameter was used to scale the step size for the calibration function gradients compared to the lattice function gradients; this calibration step size scale was also chosen using ten-fold cross-validation and powers of 10, and was usually chosen to be one of $\{.01, .1, 1, 10\}$. Multilinear interpolation was used unless it is noted that simplex interpolation was used. The loss function was squared error, unless noted that logistic loss was used.

Comparisons were made to random forests (Breiman, 2001), and to linear models, with either the logistic loss (logistic regression) or squared error loss (linear regression), and a ridge regularizer on the linear coefficients, with any categorical or missing features converted to Boolean features. All comparisons were trained on the same training set, hyperparameters were tuned using cross-validation, and tested on the same test set. Statistical significance was measured using a binomial statistical significance test with a p-value of .05 on the test samples rated differently by two models.

10.2 Case Study: Business Entity Resolution

In this case study, we compare the relative impact of several of our proposed extensions to lattice regression. The business entity resolution problem is to determine if two business descriptions refer to the same real-world business. This problem is also treated by Dalvi et al. (2014), where they focus on defining a good title similarity. Here, we consider only the problem of fusing different similarities (such as a title similarity and phone similarity) into one score that predicts whether a pair of businesses are the same business. The learned function is required to be monotonically increasing in seven attribute similarities, such as the similarity between the two business titles and the similarity between the street names. There are two other features with no monotonicity constraints, such as the geographic region, which takes on one of 14 categorical values. Each sample is derived from a pair of business descriptions, and a label provided by an expert human rater indicating whether that pair of business descriptions describe the same real-world business. We measure accuracy in terms of whether a predicted label matches the ground truth label, but in actual usage, the learned function is also used to rank multiple matches that pass the decision threshold, and thus a strictly monotonic function is preferred to a piecewise constant function. The training and test sets, detailed in Table 3, were randomly split from the complete labeled set. Most of the samples were drawn using active sampling, so most of the samples are difficult to classify correctly.

Table 4 reports results. The linear model performed poorly, because there are many important high-order interactions between the features. For example, the pair of businesses might describe two pizza places at the same location, one of which recently closed, and the other recently opened. In this case, location-based features will be strongly positive, but the classifier must be sensitive to low title similarity to determine the businesses are different. On the other hand, high title similarity is not sufficient to classify the pair as the same, for example, two Starbucks cafes across the street from each other in downtown London.

The lattice regression model was first optimized using cross-validation, and then we made the series of minor changes (with all else held constant) listed in Table 4 to illustrate the impact of these changes on accuracy. First, removing the monotonicity constraints resulted in a statistically significant drop in accuracy of half a percent. Thus it appears the monotonicity constraints are successfully regularizing given the small amount of training data and the known high Bayes error in some parts of the feature space. Lattice regression without the monotonicity constraints performed similarly to random forests (and not statistically significantly better), as expected due to the similar modeling abilities of the methods.

The cross-validated lattice was $3 \times 3 \times 3 \times 2^6$, where the first three features used a missing data vertex (so the non-missing data is interpolated from a 2^9 lattice). Calibrating the missing values for those three features instead of using missing data vertices statistically significantly dropped the accuracy from 81.9% to 80.7%. (However, if one subsamples the training set down to 3000 samples, then the less flexible option of calibrating the missing values works better than using missing data vertices.)

The cross-validated calibration used five changepoints for two of the four continuous features, and no calibration for the two other continuous features. Figure 8 shows the calibrations learned in the optimized lattice regression. Removing the continuous signal calibration resulted in a statistically significant drop in accuracy.

Another important proposal of this paper is calibrating categorical features to real-valued features. For this problem, this is applied to a feature specifying which of 14 possible geographical categories the businesses are in. Removing this geographic feature statistically significantly reduced the accuracy by half a percent.

The amount of torsion regularization was cross-validated to be 10^{-4}. Changing to graph Laplacian and re-optimizing the amount of regularization decreased accuracy slightly, but not statistically significantly so. This is consistent with what we often find: torsion is

	Test Set Accuracy	Monotonic Guarantee?
Linear Model	66.6%	yes
Random Forest	81.2%	no
Lattice Regression, Optimized	81.9%	yes
... Remove Monotonicity Constraints	81.4%	no
... Calibrate All Missing Data	80.7%	yes
... Remove Calibration	81.1%	yes
... Remove the Geographic Feature	81.4%	yes
... Change to Graph Laplacian	81.7%	yes
... Change to Simplex Interpolation	81.6%	yes

Table 4: Comparison on a business entity resolution problem.

often slightly better, but often not statistically significantly so, than the graph Laplacian regularizer.

Changing the multilinear interpolation to simplex interpolation (see Section 5.2) dropped the accuracy slightly, but not statistically significantly. For some problems we even see simplex interpolation provide slightly better results, but generally the accuracy difference between simplex and multilinear interpolation is negligible.

10.3 Case Study: Scoring Ad–Query Pairs

In this case study, we demonstrate the potential of the calibration functions. The goal is to score how well an ad matches a web search query, based on five different features that each measure a different notion of a good match. The score is required to be monotonic with respect to all five features. The labels are binary, so this is trained and tested as a classification problem. The train and test sets were independently and identically distributed, and are detailed in Table 3.

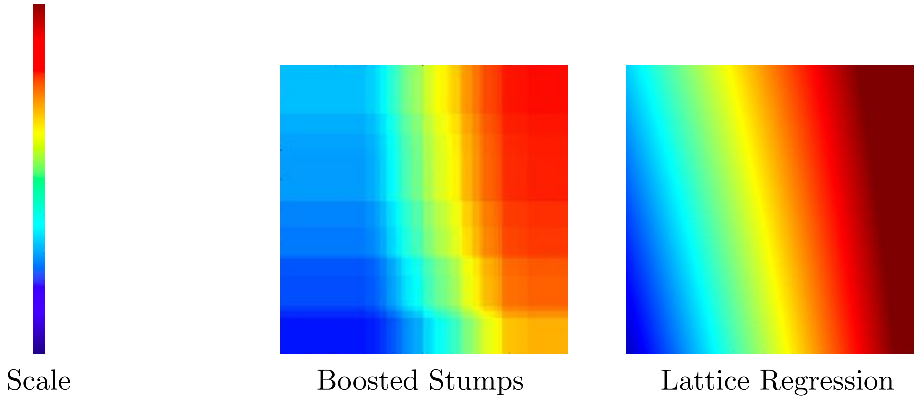
Results are shown in Table 5. The cross-validated lattice size was $2 \times 2 \times 2 \times 2 \times 2$, and the calibration functions each used 5 changepoints. Removing the calibration functions and re-cross-validating the lattice size resulted in a larger lattice sized $4 \times 4 \times 4 \times 4 \times 4$, and slightly worse (but not statistically significantly worse) accuracy. In total, the uncalibrated lattice model used 1024 parameters, whereas the calibrated lattice model used only 57 parameters. We hypothesize that the smaller calibrated lattice will be more robust to feature noise and drift in the test sample distribution than the larger uncalibrated lattice model. In general, we find that the one-dimensional calibration functions are a very efficient way to capture the flexibility needed, and that in conjunction with good one-dimensional calibrations, only coarse-grained (e.g. 2^D) lattices are needed.

Both with and without calibration functions, the lattice regression models were statistically significantly better than the linear model. The random forest performed well, but was not statistically significantly better than the lattice regression.

A boosted stumps model was also trained for this problem. See Fig. 12 for a comparison of two-dimensional slices of the boosted stumps and lattice functions. The boosted stumps' test set accuracy was relatively low at 75.4%. In practice, the goal of this problem is to have

	Test Set Accuracy	Monotonic Guarantee?
Linear Model	77.2%	yes
Random Forests	78.8%	no
Lattice Regression	78.7%	yes
... Remove Continuous Signal Calibration	78.4%	yes

Table 5: Comparison on an ad-query scoring problem.



Scale

Boosted Stumps

Lattice Regression

Figure 12: Slices of the learned ad-query matching functions for boosted stumps and a $2 \times 2 \times 2 \times 2 \times 2$ lattice regression, plotted as a function of two of the five features, with median values chosen for the other three features. The boosted stumps required hundreds of stumps to approximate the function, and the resulting function is piecewise constant, creating frequent ties when ranking a large number of ads for a given query, despite a priori knowledge that the output should be strictly monotonic in each of the features.

a score useful for ranking candidates as well as determining if they are a sufficiently good match. Even with many trees, this model produces many ties due its piecewise-constant surface. In addition, the live experiments with the boosted stumps showed that the output was problematically sensitive to feature noise, which would cause samples near the boundary of two piecewise constant surfaces to experience fluctuating scores.

10.4 Case Study: Rendering Classifier

This case study demonstrates training a flexible function (using a lattice) that is monotonic with respect to fifteen features. The goal is to score whether a particular display element should be rendered on a webpage. The score is required to be monotonic in fifteen of the features, and there is a sixteenth Boolean feature that is not constrained. The training and test sets (detailed in Table 3) consisted almost entirely of samples known to be difficult to correctly classify (hence the rather low accuracies).

We used a fixed 2^{16} lattice size, a fixed 5 changepoints per feature for the six continuous signals (the other ten signals were Boolean), and no graph regularization, so no hyperpa-

	Test Set Accuracy	Monotonic Guarantee?
Linear Model	54.6%	yes
Random Forest	61.3%	no
Lattice Regression	63.0%	yes

Table 6: Comparison on a rendering classifier.

rameters were optimized for this case study. Simplex interpolation was used for speed. A single training loop through the 20,000 training samples took around five minutes on a Xeon-type Intel desktop using a single-threaded C++ implementation with sparse vectors, with the training time dominated by the constraint handling. Training in total took around five hours.

Results in Table 6 show substantial gains over the linear model, while still producing a monotonic, smooth function. The lattice regression was also statistically significantly better than random forests, we hypothesize due to the regularization provided by the monotonicity constraints which is important in this case due to the difficulty of the problem on the given examples and the relatively small number of training samples.

10.5 Case Study: Fusing Pipelines

While this paper focuses on learning monotonic functions, we believe it is also the first paper to propose applying lattice regression to classification problems, rather than only regression problems. With that in mind, we include this case study demonstrating that lattice regression *without constraints* also performs similarly to random forests on a real-world large-scale multi-class problem.

The goal in this case study is to fuse the predictions from two pipelines, each of which makes a prediction about the likelihood of seven user categories based on a different set of high-dimensional features. Because each pipeline's probability estimates sum to one, only the first six probability estimates from each pipeline are needed as features to the fusion, for a total of twelve features. The training and test set were split by time, with the older 1.6 million samples used for training, and the newest 390,000 samples used as a test set.

The lattice was trained with a multi-class logistic loss, and used simplex interpolation for speed. The cross-validated model was a 2^{12} lattice for six of the output classes (with the probability of the seventh class being subtracted from one) and no calibration functions, resulting in a total of $2^{12} \times 6 = 24,576$ parameters.

The results are reported in Table 7. Even though Pipeline 2 alone is 6.5% more accurate than Pipeline 1 alone, the test set accuracy can be increased by fusing the estimates from both pipelines, with a small improvement in accuracy by lattice regression over the random forest classifier, logistic regression, or simply averaging the two pipeline estimates.

10.6 Case Study: Video Ranking and Large-Scale Learning

This case study demonstrates large-scale training of a large monotonic lattice and learning from ranked pairs. The goal is to learn a function to rank videos a user might like to watch,

	Test Set Accuracy Gain on top of Pipeline 1 Accuracy
Pipeline 2 Only	6.5%
Average the Two Pipeline Estimates	7.4%
Fuse with Linear Model	8.5%
Fuse with Random Forest	9.3%
Fuse with Lattice Regression	9.7%

Table 7: Comparison on fusing user category prediction pipelines.

based on the video they have just watched. Experiments were performed on anonymized data from YouTube.

Each feature vector x_i is a vector of features about a pair of videos, $x_i = h(v_j, v_k)$, where v_j is the watched video, v_k is a candidate video to watch next, and h is a function that takes a pair of videos and outputs a twelve-dimensional feature vector x_i. For example, a feature might be the number of times that video v_j and video v_k were watched in the same session.

Each of the twelve features was specified to be positively correlated with users viewing preference, and thus we constrained the model to be monotonically increasing with respect to each. Of course, human preference is complicated and these monotonicity constraints cannot fully model human judgement. For example, knowing that a video that has been watched many times is generally a very good indicator that it is good to suggest, and yet a very popular video at some point will flare out and become less popular.

Monotonicity constraints can also be useful to enforce secondary objectives. For example, all other features equal, one might prefer to serve fresher videos. While users in the long-run want to see fresh videos, they may preferentially click on familiar videos, thus click data may not capture this desire. This secondary goal can be enforced by constraining the learned function to be monotonic in a feature that measures video freshness. This achieves a multi-objective function without overly-complicating or distorting the training label definition.

There are billions of videos in YouTube, and thus many many pairs of watched-and-candidate videos to score and re-score as the underlying feature values change over time. Thus it is important the learned ranking functions to be cheap to evaluate, and so we use simplex interpolation for its evaluation speed; see Section 10.7 for comparison of evaluation speeds.

We trained to minimize the ranked pairs objective from (9), such that the learned function f is trained for the goal of minimizing pairwise ranking errors,

$$f(h(v_j, v_k^+)) > f(h(v_j, v_k^-)),$$

for each training event consisting of a watched video v_j, and a pair of candidate videos v_k^+ and v_k^- where there is information that a user who has just watched video v_j prefers to watch v_k^+ next over v_k^-.

10.6.1 Which Pairs of Candidate Videos?

A key question is which sample pairs of candidate videos v_k^+ and v_k^- should be used as the preferred and unpreferred videos for a given watched video v_j. We used anonymized click data from YouTube's current video-suggestion system. For each watched video v_j, if a user clicked a suggested video in the second position or below, then we took the clicked video as the preferred video v_k^+, and the video suggested right above the clicked video as the unpreferred video v_j^-. We call this choice of v_k^+ and v_k^- a *bottom-clicked pair.* This choice is consistent with the findings of Joachims et al. (2005), whose eye-tracking experiments on webpage search results showed that users on average look at least at one result above the clicked result, and that these pairs of preferred/unpreferred samples correlated strongly with explicit relevance judgements. Also, using bottom-clicked pairs removes the *trust bias* that users know they are being presented with a ranked list and prefer samples that are ranked-higher (Joachims et al., 2005). In a set of preliminary experiments, we also tried training using either a randomly sampled video as v_k^-, or the video just after the clicked video, and then tested on bottom-clicked pairs. Those results showed test accuracy on bottom-clicked pairs was up to 1% more accurate if the training set only included the bottom-clicked pairs, even though that meant fewer training pairs.

An additional goal (and one that is common in commercial large-scale machine learning systems for various practical reasons) is for the learned ranking function to be as similar to the current ranking function as possible. That is, we wish to minimize changes to the current scoring if they do not improve accuracy; such accuracy-neutral changes are referred to as *churn.* To reduce churn, we added in additional pairs that reflect the decisions of the current ranking function. Each of these pairs also takes the clicked video as the preferred v_k^+, but sets the unpreferred video v_k^- to be the video that the current system ranked ten candidates lower than the clicked video. The dataset is a 50-50 mix of these churn-reducing pairs and bottom-clicked pairs.

10.6.2 More Experimental Details

The dataset was randomly split into mutually exclusive training, test, and validation sets of size 400 million, 25 million, and 25 million pairs, respectively. To ensure privacy, the dataset only contained the feature vector, and no information identifying the video or user. The disadvantage of that is the train, test and validation sets are likely to have some samples from the same videos and same users. However, in total the datasets capture millions of unique users and unique watched videos.

We used a fixed 3^{12} lattice, for a total of 531,441 parameters. The pre-processing functions were fixed in this case, so no calibration functions were learned. We compared training on increasingly-larger randomly-sampled subsets of the 400 million training set (see Figure 13 for training set sizes). We compared training on a single worker to the parallelize-and-average strategy explained in Section 9.2. Parallel results were parallelized over 100 workers. The stepsize was chosen independently for each training set based on accuracy on the validation set.

We report results with and without monotonicity constraints. For the unconstrained results, each training (single or parallel) touched each sample in the training set once. For the monotonic results (single or parallelized), each sample was touched ten times, and minibatching was used with a minibatch size of 32 stochastic gradients. Logistic loss was used.

10.6.3 Results

Figure 13 compares test set accuracy for single and parallelized training for different amounts of training data, with and without monotonicity constraints. For each dataset, the single and parallel training saw the same total number of training samples and were allowed the same total number of stochastic gradient updates.

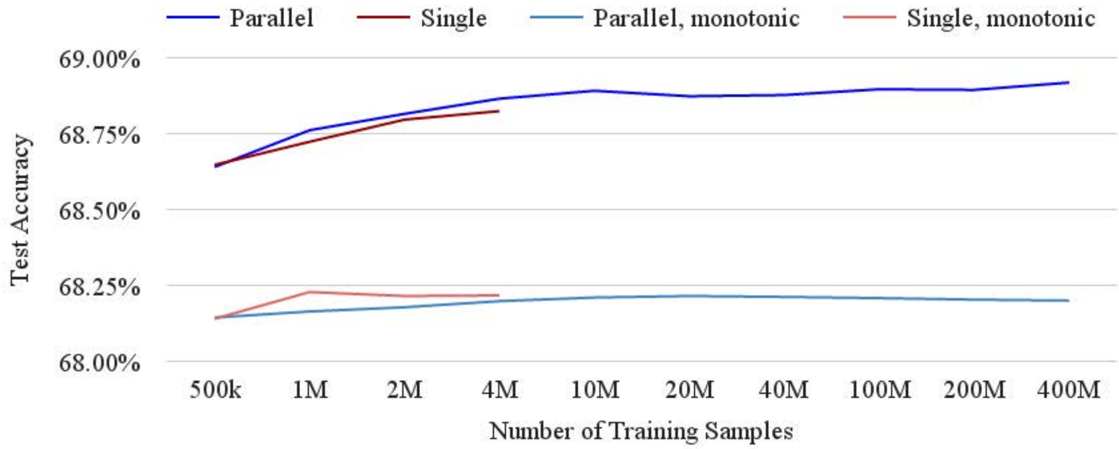


Figure 13: Comparison of training with a single worker versus 100 workers in parallel, as a function of training set size.

On the click data test set, not using monotonicity constraints (the dark lines) is about .5% better at pairwise accuracy than if we constrain the function to be monotonic. However, in live experiments that required ranking all videos (not only ones that had been top-ranked in the past, and hence included in the click data sets), models trained with monotonicity constraints showed better performance on the actual measures of user-engagement (as opposed to the training metric of pairwise accuracy). This discrepancy appears to be due to the biased sampling of the click data, as the click-data has a biased distribution over the feature space compared to the distribution of all videos which must get ranked in practice. The biased distribution of the click data appears to cause parameters in sparser regions of the feature space to be non-monotonic in an effort to increase the flexibility (and accuracy) of the function in the denser regions, thus increasing the accuracy on the click data. Enforcing monotonicity helps address this sampling bias problem by not allowing the training to ignore the accuracy in sparser regions that are important in practice to accurately rank all videos.

Even though there are 500k parameters to train, the click-data accuracy is already very good with only 500k training samples, and test accuracy increases only slightly when trained on 400 million samples compared to 10 million samples. This is largely because the click-data samples are densely clustered in the feature space, and with simplex interpolation, only a small fraction of the 500k parameters control the function over the dense part of the feature space.

The darker lines of Figure 13 show the parallelization versus single-machine results *without* monotonicity constraints. Unconstrained, the parallelized runs appear to perform slightly better to the single-machine training given the same number of training samples (and the same total number of gradient updates). We hypothesize this slight improvement is due to some noise-averaging across the 100 parallelized trained lattices. The lighter lines of Figure 13 show the parallelization versus single-machine results *with* monotonicity constraints. Trained on 500k pairs, the parallelized training and single-machine monotonic training produce the same test accuracy. However, as the training set size increases, the parallelized training takes more data to achieve the same accuracy as the single-machine training. We believe this is because averaging the 100 monotonic lattices is a convex combination of lattices likely on the edge of the monotonicity constraint set, producing an average lattice in the interior of the constraint set, that is, the averaged lattice is over-constrained.

10.7 Run Times

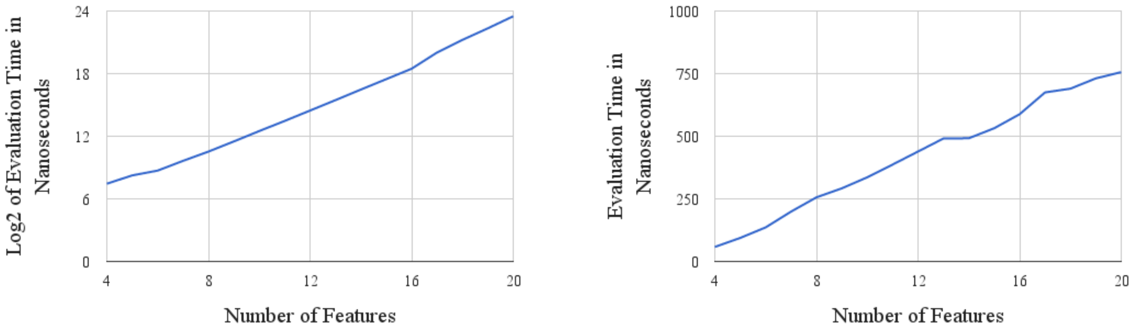
We give some timing examples for the different interpolations and for training.

Figure 14 shows average evaluation times for multilinear and simplex interpolation of one sample from a 2^D lattice for $D = 4$ to $D = 20$ using a single-threaded 3.5GHz Intel Ivy Bridge processor. Note the multilinear evaluation times are reported on a log-scale, and on a log scale the evaluation time increases roughly linearly in D, matching the theoretical $O(2^D)$ complexity given in Section 5.1. The simplex evaluation times scale roughly linearly with D, consistent with the theoretical $O(D \log D)$ complexity. For $D = 6$ features, simplex interpolation is already three times faster than multilinear. With $D = 20$ features, the simplex interpolation is still only 750 nanoseconds, but the multilinear interpolation is about $15,000$ times slower, at around 12 milliseconds.

Training times are difficult to report in an accurate or meaningful way due to the high-variance of running on a large, shared, distributed cluster. Here is one example: with every feature constrained to be monotonic, a single worker training one loop of a 2^{12} lattice on 4 million samples usually takes around 15 minutes, whereas with 100 parallelized workers one loop through 400 million samples (4 million samples for each worker) usually takes around 20 minutes. Large step-sizes can take much longer than smaller stepsizes, because larger updates tend to violate more monotonicity constraints and thus require more expensive projections. Minibatching is particularly effective at speeding up training because the averaged batch of stochastic gradients reduces the number of monotonicity violations and the need for projections. Without monotonicity constraints, training is generally 10× to 1000× faster, depending on how non-monotonic the data is.

10.8 Interpretability in Practice

It is difficult to quantify interpretability, but we summarize our observations from working with around 50 different users on around a dozen different real-world machine learning problems where there are a relatively small number of semantically meaningful features.



(a) Multilinear Interpolation

(b) Simplex Interpolation

Figure 14: Average evaluation time to interpolate a sample from a 2^D lattice. Figure (a) shows the multilinear interpolation time on a $\log_2$ scale in nanoseconds. Figure (b) shows the much faster simplex interpolation time in nanoseconds. Simplex interpolation is 10× faster than multilinear for $D = 9$ features, about 100× faster for $D = 13$ features, and over $1,000\times$ faster for $D = 17$ features.

First, we do find that being able to summarize a model as being a positive or negative function with respect to each input feature does help users feel that they understand and can predict the model's behavior better than a comparable unconstrained model. In particular, while aggregate measures like accuracy, precision, or recall over a test set provide summary statistics over that particular test set, we find that users in some cases do worry about the *unknown unknowns* of using a machine learning model, and that adding monotonicity constraints gives these users more confidence that the model can be trusted not to behave unreasonably for any examples. And this confidence is well-founded: as discussed in the video ranking case study in Section 10.6, monotonicity constraints do in practice guard against potentially strange behavior of highly nonlinear functions in rarer parts of the feature space.

We have also found that monotonicity constraints make debugging highly nonlinear models easier. We find that one particularly useful debugging tool is sensitivity plots like the one shown in Figure 15, which show how $f(x)$ relates to each feature value of x, for a particular sample x. Monotonicity constraints make these sensitivity plots monotonic, which we find makes it easier to identify problems with signals and training data.

Apart from the issue of monotonicity, we expected that using one-dimensional calibration functions and interpolated look-up tables would produce parameters that were interpretable. These expectations were half-right. We do find it helpful and common for users to check and analyze the signals' calibration functions, and that the calibrations provide useful information to users about what the model has learned, and helps identify unexpected behavior or problems with features. But, we do not find that users examine the *lattice* parameters directly very often, and that the readability of the lattice parameters becomes generally less useful as D increases. Users are more likely to utilize other analytics, such as how correlated the output is with each calibrated feature. While this information does not control for between-feature correlations (so a feature might be highly correlated with the output but not needed if it correlates with another feature), users can conclude the model behaves a lot like highly-correlated features, and this aids model understanding. Low correlation between a calibrated feature and the output either indicates an unimportant feature, or, if the feature is known to be important (because dropping it hurts accuracy), that it plays an important role interacting or conditioning other features.

To understand feature interactions, again we find that rather than analyze the lattice parameters explicitly, users generally prefer to analyze two-dimensional visualizations of the model over pairs of features (averaged or sliced over the other features), or to examine examples to check their hypotheses about expected higher-order interactions.

Another common problem is to understand how two similar models are different (for example, one might have been trained with more training data, or one might use an additional feature). For this purpose, we find it is again rare that users want to directly analyze differences in model parameters except for very tiny models, preferring instead to look at examples that are scored differently by the two models, and analyzing these example samples in their raw form (for example, looking at the videos that have been promoted or demoted by a new model, in conjunction with the features used by the model).

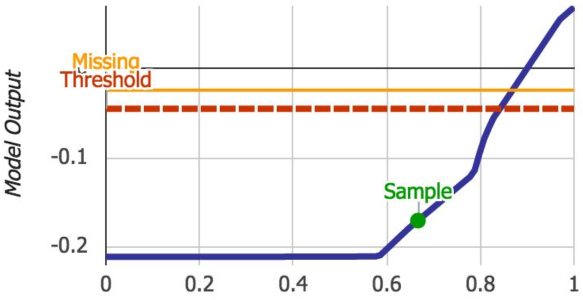


Figure 15: Illustration of a sensitivity plot for a calibrated monotonic lattice with respect to one of the D features. The blue line shows how the output $f(x)$ (y-axis) of a calibrated monotonic lattice changes if only this one feature value of x (x-axis) is changed, but all other components of x are kept fixed. The green dot marks the current input and output. The yellow line shows the model output if the first feature is missing. The red dotted line shows the binary classification threshold. This plot is piecewise linear because the calibrator function is piecewise linear and the simplex interpolation is also piecewise linear, and monotonically increasing because the function was constrained to be monotonic with respect to this feature. In this example, one sees that to change the classification decision without changing any other features, this feature would have to be increased from its current value of 0.67 to at least 0.84 at which point it would cross the red line marking the decision threshold.

11. Discussion and Some Open Questions

We have proposed using constrained interpolated look-up tables to effectively learn flexible, monotonic functions for low-dimensional machine learning problems of classification, ranking, and regression. We addressed a number of practical issues, including interpretability, evaluation speed, automated pre-processing of features, missing data, and categorical features. Experimental results show state-of-the-art performance on the largest training sets and largest number of features published for monotonic methods.

Practical experience has shown us that being able to check and ensure monotonicity helps users trust the model, and leads to models that generalize better. For us, the monotonicity constraints have come from engineers who believe the output should be monotonic in the feature. In the absence of clear prior information about monotonicity, it may be tempting to use the direction of a linear fit to specify a monotonic direction and then use monotonicity as a regularizer. Magdon-Ismail and Sill (2008) point out that using the linear regression coefficients for this purpose can be misleading if features are correlated and not jointly Gaussian.

For classifiers, requiring the entire function to be monotonic is a stronger requirement than needed to simply guarantee that the decision boundary (and hence classifier) is monotonic. It is an open question how to enforce only the thresholded function to be monotonic, and whether that would be more useful in practice.

One surprise was that for practical machine learning problems like those of Section 10, we found a simple 2^D lattice is often sufficient to capture the interactions of D features, especially if we jointly optimized D one-dimensional feature calibration functions. When we began this work, we expected to have to use much more fine-grained lattices with many vertices in each feature, or perhaps irregular lattices to achieve state-of-the-art accuracy. In fact, calibration functions help approximately linearize each feature with respect to the label, making a 2^D lattice sufficiently flexible for most of the real-world problems we have encountered.

For some cases, a 2^D lattice is too flexible. We reduced lattice flexibility with new regularizers: monotonicity, and the torsion regularizer that encourages a more linear model. While good for interpretability and accuracy, these regularization strategies do not reduce the model size.

For a large number of features D, the exponential model size of a 2^D lattice is a memory issue. On a single machine, training and evaluating with a few million parameters is viable, but this still limits this approach to not much more than $D = 20$ features. An open question is how such large models could be sparsified, and if useful sparsification approaches could also provide additional useful regularization.

A second surprise was that simplex interpolation provides similar accuracy to multilinear interpolation. The rotational dependence of simplex interpolation seemed at first troubling, but the proposed approach of aligning the shared axis of the simplices with the main increasing axis of the function appears to solve this problem in practice. The geometry of the simplices at first seemed odd in that it produces a locally linear surface over elongated simplices. However, this partitioning turns out to work well because it provides a very flexible piecewise linear decision boundary. Lastly, we found that the theoretical

$O(D \log D)$ computational complexity does result in practice in orders of magnitude faster interpolation than multilinear interpolation as D increases.

A common practical issue in machine learning is handling categorical data. We proposed to learn a mapping from mutually exclusive categories to feature values, jointly with the other model parameters. We found categorical-mapping to be interpretable, flexible, and accurate. The proposed categorical mapping can be viewed as learning a one-dimensional embedding of the categories. Though we generally only needed two vertices in the lattice for continuous features, for categorical features we often find it helpful to use more vertices (a finer-grained lattice) for more flexibility. Some preliminary experiments learning two-dimensional embeddings of categories (that is, mapping one category to $[0,1]^2$) showed promise, but we found this required more careful initialization and handling of the increased non-convexity.

Learning the monotonic lattice is a convex problem, but composing the lattice and the one-dimensional calibration functions creates a non-convex objective. We used only one initialization of the lattice and calibrators for all our experiments, but tuned the stepsize of the stochastic gradient descent separately for the set of lattice parameters and the set of calibration parameters. In some cases we saw a substantial sensitivity of the accuracy to the initial SGD stepsizes. We hypothesize that this is caused by some interplay of the relative stepsizes and the relative size of the local optima.

We employed a number of strategies to speed up training. One of the biggest speed-ups comes from randomly sampling the additive terms of the graph regularizers, analogous to the random sampling of the additive terms of the empirical loss that SGD uses. We showed that a parallelize-and-average strategy works for training the lattices. The largest computational bottleneck remains the projections onto the monotonicity constraints. Mini-batching the samples reduces the number of projections and provides speed-ups, but a faster approach to optimization given possibly hundreds of thousands of constraints would be valuable.

Lastly, this work leaves open a number of theoretical questions for the function class of interpolated look-up tables, for example how monotonicity constraints theoretically affect convergence speed.

12. Acknowledgments

We thank Sugato Basu, David Cardoze, James Chen, Emmanuel Christophe, Brendan Collins, Mahdi Milani Fard, James Muller, Biswanath Panda, and Alex Vodomerov for help with experiments and helpful discussions.

References

Y. S. Abu-Mostafa. A method for learning from hints. In *Advances in Neural Information Processing Systems*, pages 73–80, 1993.

N. P. Archer and S. Wang. Application of the back propagation neural network algorithm with monotonicity constraints for two-group classification problems. *Decision Sciences*, 24(1):60–75, 1993.

F. Bach. Learning with submodular functions: A convex optimization perspective. *Foundations and Trends in Machine Learning*, 6(2), 2013.

R. E. Barlow, D. J. Bartholomew, J. M. Bremner, and H. D. Brunk. *Statistical inference under order restrictions; the theory and application of isotonic regression*. Wiley, New York, USA, 1972.

A. Ben-David. Automatic generation of symbolic multiattribute ordinal knowledge based DSS: methodology and applications. *Decision Sciences*, pages 1357–1372, 1992.

A. Ben-David. Monotonicity maintenance in information-theoretic machine learning algorithms. *Machine Learning*, 21:35–50, 1995.

A. Ben-David, L. Sterling, and Y. H. Pao. Learning and classification of monotonic ordinal concepts. *Computational Intelligence*, 5(1):45–49, 1989.

S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1), 2010.

L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

J. Casillas, O. Cordon, F. Herrera, and L. Magdalena (Eds.). Trade-off between accuracy and interpretability in fuzzy rule-based modelling. *Physica-Verlag*, 2002.

R. Chandrasekaran, Y. U. Ryu, V. S. Jacob, and S. Hong. Isotonic separation. *INFORMS Journal on Computing*, 17(4):462–474, 2005.

A. Cotter, M. R. Gupta, and J. Pfeifer. A Light Touch for Heavily Constrained SGD. *arXiv preprint*, 2015. URL `http://arxiv.org/abs/1512.04960`.

N. Dalvi, M. Olteanu, M. Raghavan, and P. Bohannon. Deduplicating a places database. *Proc. ACM WWW Conf.*, 2014.

H. Daniels and M. Velikova. Monotone and partially monotone neural networks. *IEEE Trans. Neural Networks*, 21(6):906–917, 2010.

O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao. Optimal distributed online prediction using mini-batches. *Journal Machine Learning Research*, 13(1):165–202, January 2012.

J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal Machine Learning Research*, 12:2121–2159, 2011.

C. Dugas, Y. Bengio, F. Bélisle, C. Nadeau, and R. Garcia. Incorporating second-order functional knowledge for better option pricing. In *Advances in Neural Information Processing Systems (NIPS)*, 2000.

C. Dugas, Y. Bengio, F. Bélisle, C. Nadeau, and R. Garcia. Incorporating functional knowledge in neural networks. *Journal Machine Learning Research*, 2009.

W. Duivesteijn and A. Feelders. Nearest neighbour classification with monotonicity constraints. *Proc. European Conf. Machine Learning*, pages 301–316, 2008.

A. Feelders. Monotone relabeling in ordinal classification. *Proc. IEEE Conf. Data Mining*, pages 803–808, 2010.

M. Fernandez-Delgado, E. Cernadas, S. Barro, and D. Amorim. Do we need hundreds of classifiers to solve real world classification problems? *Journal Machine Learning Research*, 2014.

E. K. Garcia and M. R. Gupta. Lattice regression. In *Advances in Neural Information Processing Systems (NIPS)*, 2009.

E. K. Garcia, S. Feldman, M. R. Gupta, and S. Srivastava. Completely lazy learning. *IEEE Trans. Knowledge and Data Engineering*, 22(9):1274–1285, Sept. 2010.

E. K. Garcia, R. Arora, and M. R. Gupta. Optimized regression for efficient function evaluation. *IEEE Trans. Image Processing*, 21(9):4128–4140, Sept. 2012.

S. Garcia, A. Fernandez, J. Luengo, and F. Herrera. A study of statistical techniques and performance measures for genetics-based machine learning: accuracy and interpretability. *Soft Computing*, 13:959–977, 2009.

I. J. Good. *The Estimation of Probabilities: An Essay on Modern Bayesian Methods.* MIT Press, 1965.

H. Gruber, M. Holzer, and O. Ruepp. Sorting the slow way: an analysis of perversely awful randomized sorting algorithms. In *Fun with Algorithms*, pages 183–197. Springer, 2007.

M. Gupta, S. Bengio, and J. Weston. Training highly multiclass classifiers. *Journal Machine Learning Research*, 2014.

M. R. Gupta, R. M. Gray, and R. A. Olshen. Nonparametric supervised learning by linear interpolation with maximum entropy. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 28(5):766–781, 2006.

T. Hastie and R. Tibshirani. *Generalized Additive Models.* Chapman Hall, New York, 1990.

T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning.* Springer-Verlag, New York, 2001.

C. C. Holmes and N. A. Heard. Generalized monotonic regression using random change points. *Statistics in Medicine*, 22:623–638, 2003.

A. Howard and T. Jebara. Learning monotonic transformations for classification. In *Advances in Neural Information Processing Systems*, 2007.

H. Ishibuchi and Y. Nojima. Analysis of interpretability-accuracy tradeoff of fuzzy systems by multiobjective fuzzy genetics-based machine learning. *International Journal of Approximate Reasoning*, 44:4–31, 2007.

T. Joachims, L. Granka, B. Pan, H. Hembrooke, and G. Gay. Accurately interpreting clickthrough data as implicit feedback. *Proc. SIGIR*, 2005.

H. R. Kang. Comparison of three-dimensional interpolation techniques by simulations. *SPIE Vol. 2414*, 1995.

H. R. Kang. *Color Technology for Electronic Imaging Devices.* SPIE Press, USA, 1997.

J. Kasson, W. Plouffe, and S. Nin. A tetrahedral interpolation technique for color space conversion. *SPIE Vol. 1909*, 1993.

H. Kay and L. H. Ungar. Estimating monotonic functions and their bounds. *AIChE Journal*, 46(12):2426–2434, 2000.

R. E. Knop. A note on hypercube partitions. *Journal of Combinatorial Theory, Ser. A*, 15 (3):338–342, 1973.

W. Kotlowski and R. Slowinski. Rule learning with monotonicity constraints. In *Proceedings International Conference on Machine Learning*, 2009.

F. Lauer and G. Bloch. Incorporating prior knowledge in support vector regression. *Machine Learning*, 70(1):89–118, 2008.

X. Liao, H. Li, and L. Carin. Quadratically gated mixture of experts for incomplete data classification. *Proc. ICML*, 2007.

T.-Y. Liu. *Learning to Rank for Information Retrieval.* Springer, 2011.

Malik Magdon-Ismail and J. Sill. A linear fit gets the correct monotonicity directions. *Machine Learning*, pages 21–43, 2008.

G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. D. Walker. Efficient large-scale distributed training of conditional maximum entropy models. *Advances in Neural Information Processing Systems (NIPS)*, 2009.

D. G. Mead. Dissection of the hypercube into simplexes. *Proc. Amer. Math. Soc.*, 76: 302–304, 1979.

A. Minin, M. Velikova, B. Lang, and H. Daniels. Comparison of universal approximators incorporating partial monotonicity by structure. *Neural Networks*, 23(4):471–475, 2010.

H. Mukarjee and S. Stern. Feasible nonparametric estimation of multiargument monotone functions. *Journal of the American Statistical Association*, 89(425):77–80, 1994.

B. Neelon and D. B. Dunson. Bayesian isotonic regression and trend analysis. *Biometrics*, 60:398–406, 2004.

A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on Optimization*, 19(4):1574–1609, January 2009.

K. Neumann, M. Rolf, and J. J. Steil. Reliable integration of continuous constraints into extreme learning machines. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 21(supp02):35–50, 2013.

R. Nock. Inducing interpretable voting classifiers without trading accuracy for simplicity: Theoretical results, approximation algorithms, and experiments. *Journal Artificial Intelligence Research*, 17:137–170, 2002.

K.-M. Osei-Bryson. Post-pruning in decision tree induction using multiple performance measures. *Computers and Operations Research*, 34:3331–3345, 2007.

R. Potharst and A. J. Feelders. Classification trees for problems with monotonicity constraints. *ACM SIGKDD Explorations*, pages 1–10, 2002a.

R. Potharst and A. J. Feelders. Pruning for monotone classification trees. *Springer Lecture Notes on Computer Science*, 2810:1–12, 2002b.

Y.-J. Qu and B.-G. Hu. Generalized constraint neural network regression model subject to linear priors. *IEEE Trans. on Neural Networks*, 22(11):2447–2459, 2011.

J. O. Ramsay. Estimating smooth monotone functions. *Journal of the Royal Statistical Society, Series B*, 60:365–375, 1998.

G. Rätsch, S. Sonnenburg, and C. Schäfer. Learning interpretable SVMs for biological sequence classification. *BMC Bioinformatics*, 7, 2006.

J. Riihimäki and A. Vehtari. Gaussian processes with monotonicity information. In *International Conference on Artificial Intelligence and Statistics*, pages 645–652, 2010.

R. Rovatti, M. Borgatti, and R. Guerrieri. A geometric approach to maximum-speed n-dimensional continuous linear interpolation in rectangular grids. *IEEE Trans. on Computers*, 47(8):894–899, 1998.

F. Schimdt and R. Simon. Some geometric probability problems involving the Eulerian numbers. *Electronic Journal of Combinatorics*, 4(2), 2007.

G. Sharma and R. Bala. *Digital Color Imaging Handbook.* CRC Press, New York, 2002.

T. S. Shively, T. W. Sager, and S. G. Walker. A Bayesian approach to non-parametric monotone function estimation. *Journal of the Royal Statistical Society, Series B*, 71(1): 159–175, 2009.

P. K. Shukla and S. P. Tripathi. A review on the interpretability-accuracy trade-off in evolutionary multi-objective fuzzy systems (EMOFS). *Information*, 2012.

J. Sill. Monotonic networks. *Advances in Neural Information Processing Systems (NIPS)*, 1998.

J. Sill and Y. S. Abu-Mostafa. Monotonicity hints. *Advances in Neural Information Processing Systems (NIPS)*, pages 634–640, 1997.

D. J. Spiegelhalter and R. P. Knill-Jones. Statistical and knowledge-based approaches to clinical decision support systems, with an application in gastroenterology. *Journal of the Royal Statistical Society A*, 147:35–77, 1984.

J. Spouge, H. Wan, and W. J. Wilbur. Least squares isotonic regression in two dimensions. *Journal of Optimization Theory and Applications*, 117(3):585–605, 2003.

C. Strannegaard. Transparent neural networks. *Proc. Artificial General Intelligence*, 2012.

B. Sun and S. Zhou. Study on the 3D interpolation models used in color conversion. *IACSIT Intl. Journal Engineering and Technology*, 4(1), 2012.

A. van Esbroeck, S. Singh, I. Rubinfeld, and Z. Syed. Evaluating trauma patients: Addressing missing covariates with joint optimization. *Proc. AAAI*, 2014.

M. Villalobos and G. Wahba. Inequality-constrained multivariate smoothing splines with application to the estimation of posterior probabilities. *Journal of the American Statistical Association*, 82(397):239–248, 1987.

S. Wang. A neural network method of density estimation for univariate unimodal data. *Neural Computing & Applications*, 2(3):160–167, 1994.

L. Yu and J. Xiao. Trade-off between accuracy and interpretability: experience-oriented fuzzy modeling via reduced-set vectors. *Computers and Mathematics with Applications*, 57:885–895, 2012.