On Training Neurons with Bounded Compilations

Lance Kennedy , **Issouf Kindo** and **Arthur Choi**
Department of Computer Science, Kennesaw State University
lkenne31@students.kennesaw.edu issouf_kindo@yahoo.fr, achoi13@kennesaw.edu

Abstract

Knowledge compilation offers a formal approach to explaining and verifying the behavior of machine learning systems, such as neural networks. Unfortunately, compiling even an individual neuron into a tractable representation such as an Ordered Binary Decision Diagram (OBDD), is an NP-hard problem. In this paper, we consider the problem of training a neuron from data, subject to the constraint that it has a compact representation as an OBDD. Our approach is based on the observation that a neuron can be compiled into an OBDD in polytime if (1) the neuron has integer weights, and (2) its aggregate weight is bounded. Unfortunately, we first show that it is also NP-hard to train a neuron, subject to these two constraints. On the other hand, we show that if we train a neuron generatively, rather than discriminatively, a neuron with bounded aggregate weight can be trained in pseudo-polynomial time. Hence, we propose the first efficient algorithm for training a neuron that is guaranteed to have a compact representation as an OBDD. Empirically, we show that our approach can train neurons with higher accuracy and more compact OBDDs.

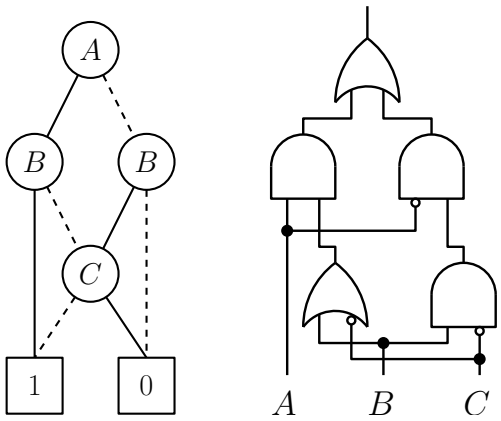
1 Introduction

Over the past decade, rapid advances in artificial intelligence (AI), coupled with the increasing pervasiveness of AI, has brought with it the need to better understand and explain the behavior of the resulting systems. As a result, a new sub-field of AI, called eXplainable Artificial Intelligence (XAI) has arisen (Baehrens et al. 2010; Ribeiro, Singh, and Guestrin 2016; Ribeiro, Singh, and Guestrin 2018; Lipton 2018). *Formal* approaches to XAI, in particular, seek to provide mathematical guarantees on the behavior of such systems, e.g., by providing bounds on the output of a neural network—say, a guarantee that a self-driving car does not exceed safe driving speeds (Katz et al. 2017; Leofante et al. 2018; Shih, Choi, and Darwiche 2018b; Shih, Choi, and Darwiche 2018a; Ignatiev, Narodytska, and Marques-Silva 2019; Audemard, Koriche, and Marquis 2020; Cooper and Marques-Silva 2021).

Our paper is motivated by the *knowledge compilation* approach to formal XAI (Shih, Choi, and Darwiche 2018b; Shi et al. 2020; Audemard, Koriche, and Marquis 2020). Knowledge compilation is another sub-field of AI that studies in part *tractable* representations of Boolean functions, and the trade-offs between their succinctness and tractability (Selman and Kautz 1996; Cadoli and Donini 1997; Darwiche and Marquis 2002). By enforcing different properties on the compiled representation, one can obtain greater tractability (the ability to perform certain queries and transformations in polytime) at the expense of succinctness (the size of the representation).

Consider the Ordered Binary Decision Diagram (OBDD), a popular target representation for knowledge compilation. If we can represent the decision function of a neural network into a representation such as an OBDD, then the OBDD would facilitate the explanation and formal verification of the neural network's behavior, due to the many polynomial time operations and transformations supported by OBDDs (Shih, Choi, and Darwiche 2018a; Shih, Choi, and Darwiche 2019; Audemard, Koriche, and Marquis 2020). This includes different types of formal analyses on a neural network, including the verification of its monotonicity or its robustness. Unfortunately, compiling a neural network to an OBDD is an NP-hard problem. In fact, compiling an *individual neuron* to an OBDD is an NP-hard problem (Shih, Choi, and Darwiche 2018b; Shi et al. 2020).

More generally, it is NP-hard to compile a *linear classifier* to an OBDD. In this paper, we propose a principled approach for training such a classifier directly from data, subject to the constraint that it has a *compact representation as an OBDD*. Our approach is based on the observation that a linear classifier will admit a pseudo-polynomial time compilation if its weights are integers. In particular, if the aggregate weight of such a classifier is bounded, then it admits a compact representation as an OBDD (Chan and Darwiche 2003; Shi et al. 2020). Hence, if we can train a linear classifier that (1) has integer weights, and (2) has bounded aggregate weight, then we can train a classifier that is guaranteed to have a compact OBDD. We first find that, unfortunately, learning such a classifier is also an NP-hard problem. Despite this, we find that a linear classifier will also admit a *pseudo-polynomial* time algorithm, if they are trained in a generative (as opposed to a discriminative) way. We further show that this same algorithm can also be used to approximate a discriminative classifier by one that uses integer weights, and with arbitrary fidelity. Empirically, we show that we are able to learn linear classifiers with more compact representations as OBDDs, and with higher accuracy.



(a) An OBDD (b) A circuit

Figure 1: An OBDD and circuit representation of a linear classifier $A + B - C \geq \frac{1}{2}$.

Several approaches to training neural networks with integer, and even binary, weights have been proposed in the literature. XNOR-Networks, for example, are trained with binary weights (Rastegari et al. 2016). The binarized neural networks (BNNs) of (Hubara et al. 2016) have binarized parameters and activations. (Narodytska et al. 2018) showed that BNNs have neurons that simplify to threshold gates. (Shih, Darwiche, and Choi 2019) compiled binarized neural networks to OBDD, but managed the complexity of compilation by reducing its scope. Our work follows (Shi et al. 2020) who compiled another class of binary neural networks to OBDD, by compiling its individual neurons to OBDDs, and then aggregating them. More specifically, the authors trained a neuron normally, and then truncated its weights until the neuron (as a linear classifier) became compilable—a much less principled approach compared to ours.

This paper is organized as follows. First, we provide technical background in Section 2. In Section 3, we introduce the problem of training a linear classifier with a compact OBDD, and show that this is an NP-hard problem. In Section 4, we identify conditions under which such a classifier can be trained in pseudo-polynomial time. We provide an empirical analysis of our algorithm in Section 5, and show how it facilitates the explanation of a linear classifier, via a case study in Section 6. Finally, we conclude in Section 7.

2 Technical Preliminaries

In this paper, we consider a family of linear classifiers that include neurons (with step activations) as a special case. Consider first a neuron of the form

$$f(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + b)$$

where $\mathbf{x}$ is the input vector, $\mathbf{w}$ is the weight vector, b is a bias, and σ is an activation function. Such a neuron can be viewed (locally) as a linear classifier, with features $\mathbf{x}$. For the purposes of training a neuron/classifier, we assume a sigmoid activation function $\sigma(z) = (1 + \exp\{-z\})^{-1}$, which is continuous and allows us to train a classifier using gradient descent. For the purposes of testing a neuron/classifier, we assume a step activation function $\sigma(z) = 1$ if $z \geq 0$ and $\sigma(z) = 0$ otherwise. A step activation is not continuous (and hence, not used for training), and it is also equivalent to rounding the output of a sigmoid activation. Consider the following more general notion of a threshold-based linear classifier, which we focus on in the remainder of the paper.

Definition 1 (Linear Classifier)**.** *Let $\mathbf{X}$ be a set of binary features where each feature X in $\mathbf{X}$ has a value $x \in \{-1, +1\}$. Let $\mathbf{x}$ denote an instantiation of variables $\mathbf{X}$. Consider functions f that map instantiations $\mathbf{x}$ to a value in $\{0, 1\}$. We call f a linear classifier if it has a decision function with the following form:*

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_{x \in \mathbf{x}} w_X \cdot x \geq T \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where $x \in \mathbf{x}$ is the value of variable X in instantiation $\mathbf{x}$, and where we have a threshold T (or a negative bias $-b$) and a weight w_X for each variable $X \in \mathbf{X}$.

Neurons (with step activations) and logistic regression are both types of threshold-based linear classifiers. Such a classifier has binary inputs and a binary output, and hence the classifier has a decision function that is a Boolean function. For the remainder of the paper, we focus on linear classifiers (and neurons) that induce Boolean decision functions.

If we can extract the Boolean function of a linear classifier, then to explain the behavior of the classifier, it suffices to explain the behavior of the Boolean function. In particular, we want a *tractable* representation of this Boolean function that will support polytime queries and transformations that facilitate such formal explanation and verification.

As an example, consider a linear classifier with 3 inputs A, B and C with weights $w_1 = 1, w_2 = 1$ and $w_3 = -1$ and a threshold of $\frac{1}{2}$. This classifier outputs 1 iff:

$$A + B - C \geq \frac{1}{2}$$

We can visualize the decision function of this classifier, by enumerating all possible inputs and the corresponding output, as we would in a truth table:

A	B	C	f
$+1$	$+1$	$+1$	1
$+1$	$+1$	-1	1
$+1$	-1	$+1$	0
$+1$	-1	-1	1

A	B	C	f
-1	$+1$	$+1$	0
-1	$+1$	-1	1
-1	-1	$+1$	0
-1	-1	-1	0

Figure 1 highlights two logically equivalent representations of this classifier's Boolean function. Figure 1a highlights an Ordered Binary Decision Diagram (OBDD) representation[1] and Figure 1b highlights a circuit representation. These functions are equivalent to the sentence:

$$[\neg C \wedge (A \vee B)] \vee [C \wedge A \wedge B],$$

[1]An Ordered Binary Decision Diagram (OBDD) is a rooted DAG with two sinks: a 1-sink and a 0-sink. An OBDD is a graphical representation of a Boolean function on variables $\mathbf{X} = \{X_1, \ldots, X_n\}$. Every OBDD node (but the sinks) is labeled with a variable X_i and has two labeled outgoing edges: a 1-edge and a 0-edge. The labeling of the OBDD nodes respects a global ordering of the variables $\mathbf{X}$: if there is an edge from a node labeled X_i to a node labeled X_j, then X_i must come before X_j in the ordering. To evaluate the OBDD on an instance $\mathbf{x}$, start at the root node of the

if we take $+1$ to be true and -1 to be false. From this sentence, we see that if C is -1 (false) then A or B must be $+1$ (true) to meet or surpass the threshold $\frac{1}{2}$, and if C is $+1$ then both A and B must be $+1$.

In this paper, we seek to *compile* a linear classifier into a tractable representation in the form of an OBDD, i.e., we seek to obtain an OBDD representation of a classifier's decision function. As OBDDs support many polytime operations and transformations, the ability to compile a classifier into an OBDD will allow us to explain and verify its behavior more efficiently. Note that while linear classifiers are generally considered *interpretable* (say, its weights and parameters have semantics or meaning), they may still not support *formal verification* of its behavior. For example, consider the following decision problem:

ϵ-DECISION-BOUNDARY (ϵ-DB): Given a linear classifier f and $\epsilon \geq 0$, does there exist an input $\mathbf{x}$ where

$$\left| T - \sum_{x \in \mathbf{x}} w_X \cdot x \right| \leq \epsilon?$$

Decision problem **ϵ-DB** asks whether there is an input setting that is ϵ-close to the decision boundary. The ability to answer this question provides insight on the robustness of a classifier to changes in its parameters, for example. However, this question is still hard to answer for linear classifiers.

Proposition 1. *ϵ-DB is NP-complete.*

A proof of Proposition 1 appears in the Appendix. Unfortunately, the problem of compiling a linear classifier into an OBDD, is also a problem that appears to be intractable.

Theorem 1. *Given a linear classifier, compiling an OBDD representing its decision function is an NP-hard problem.*

The proof follows (Shih, Choi, and Darwiche 2018b). Despite this apparent intractability, there is a broad class of linear classifiers that can be compiled to OBDD in pseudo-polynomial time.

Definition 2 (Integer Linear Classifier)**.** *We call a linear classifier an integer one iff all the weights w_X and the threshold T are also integers. We refer to the sum $W = |T| + \sum_{X \in \mathbf{X}} |w_X|$ as the aggregate weight of the classifier.*

We can compile an integer linear classifier to an OBDD in polynomial time, if its aggregate weight is bounded. In Section 6, we will also revisit the **ϵ-DB** problem, and see how it can also be solved efficiently in this case.

Theorem 2. *Consider an integer linear classifier having an aggregate weight of W over n binary features. Such a classifier can be represented by an OBDD of size $O(nW)$ nodes, and compiled in $O(nW)$ time.*

A proof of this theorem is provided in (Shi et al. 2020). Variations of this theorem have appeared in the literature in varying forms (Chan and Darwiche 2003; Shih, Choi, and Darwiche 2018b; Chubarian and Turan 2020). Note that a classifier with fixed-precision floating-point weights is also an integer classifier, after we multiply its weights by a sufficiently large constant. However, a classifier with high-precision weights may become difficult to compile.

Theorem 2 further implies that if we can train an integer classifier with bounded aggregate weight, then we can train a classifier that has an OBDD whose size is bounded by the budget and number of variables, which we consider next.

3 On Training a Compilable Classifier

We next consider the problem of learning an integer classifier with bounded aggregate weight. We refer to such a classifier as *weight-budgeted* if we assert a budget B on its aggregate weight. By Theorem 2, a weight-budgeted classifier can be compiled to an OBDD in time linear in the budget.

Definition 3 (Weight-Budgeted Linear Classifier)**.** *We call an integer linear classifier a weight-budgeted one iff its aggregate weight is less than or equal to a given budget B.*

Consider the problem of training a linear classifier from data. Say we have a dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$ consisting of examples $\mathbf{x}_i$ with labels y_i. For the purposes of training, we can assume a more neuron-like representation of the form $g(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + b)$ where $\mathbf{x}$ is the input vector, $\mathbf{w}$ is the weight vector and b is a bias (or negative threshold). As discussed in Section 2, σ can be a sigmoid activation during training, and a step activation during testing. We can then learn the weights $\mathbf{w}$ and bias b by minimizing the mean-squared-error (MSE):

$$\frac{1}{N} \sum_{i=1}^{N} \left(y_i - g(\mathbf{x}_i)\right)^2.$$

Typically, when a linear classifier is trained in this manner, it is also referred to as a logistic regression classifier.

This learning problem is a convex optimization problem, and is typically solved using gradient descent. However, in an integer linear classifier, the weights are constrained to be integers, and hence, minimizing the squared-error becomes a combinatorial optimization problem. Further, when we assert an upper-bound on the aggregate weight, the problem appears to become intractable in general.

Theorem 3. *Given a dataset $\mathcal{D}$, training a weight-budgeted linear classifier with minimum MSE is an NP-hard problem.*

A proof appears in the Appendix. Note that neither (1) constraining the weights to be integers, nor (2) constraining their aggregate weight to be bounded, makes training a linear classifier NP-hard by themselves; it is their combination.

4 A Pseudo-Polynomial Time Algorithm

The computational hardness of Theorem 3 depends on the linear classifier being trained *discriminatively*, i.e., by minimizing mean-squared-error. That is, the model is being trained so that it optimizes its accuracy when classifying the training set. Alternatively, we may train a classifier *generatively*, where the model is being trained to fit the underlying probability distribution from where the dataset came from.

OBDD and let x_i be the value of variable X_i that labels the current node. Repeatedly follow the x_i-edge of the current node, until a sink node is reached. Reaching the 1-sink means $\mathbf{x}$ is evaluated to 1 and reaching the 0-sink means $\mathbf{x}$ is evaluated to 0 by the OBDD.

In this section, we show that if we train a linear classifier generatively, then the optimal classifier can be found in pseudo-polynomial time. That is, the optimal classifier may be found in time that is polynomial in the size of the input (the number of features n) and on the magnitude of the budget B.[2] We proceed as follows. First, we explain how we formulate a linear classifier in generative terms (as a naive Bayes classifier), and explicate the assumptions implied by such a classifier. We next show that the maximum-likelihood parameters for this classifier can be found in closed-form. We then present our pseudo-polynomial time algorithm for the maximum-likelihood parameters, subject to the constraint that the weights are integer and bounded.

Definition 4 (Generative Linear Classifier)**.** *We call a linear classifier a generative one iff its weight and threshold have been trained via maximum likelihood (ML).*

Again, a linear classifier that is trained discriminatively is typically referred to as logistic regression. A linear classifier that is trained generatively is typically referred to as naive Bayes. That is, naive Bayes and logistic regression classifiers can be viewed as having the same functional form, but whose parameters are obtained using different methods, e.g., by optimizing the log likelihood in the generative case, and by optimizing the conditional log likelihood (or, alternatively, the mean-squared-error) in the discriminative case; see, e.g., (Ng and Jordan 2001; Elkan 1997).

A naive Bayes classifier, in contrast to logistic regression, represents a joint distribution $Pr(\mathbf{X}, Y)$, where the variables $\mathbf{X}$ represent the features of the classifier, and variable Y represents the class label. The induced joint distribution is:

$$Pr(\mathbf{X}, Y) = Pr(Y) \prod_{X \in \mathbf{X}} Pr(X \mid Y).$$

where $Pr(Y)$ is the prior class distribution, and $Pr(X \mid Y)$ are the conditional feature-given-class distributions. A linear classifier, as defined in Definition 1, can be represented as a naive Bayes classifier, as follows.

Proposition 2. *If f is a linear classifier with weights w_X for each $X \in \mathbf{X}$ and a threshold T, then the corresponding naive Bayes classifier has the parameters:*

$$Pr(Y) = \sigma(-T)$$

$$Pr(X{=}{+1} \mid Y{=}1) = Pr(X{=}{-1} \mid Y{=}0) = \sigma(w_X)$$

where $\sigma(z) = (1 + \exp\{-z\})^{-1}$ and where

$$\log \frac{Pr(Y{=}1 \mid \mathbf{x})}{Pr(Y{=}0 \mid \mathbf{x})} = -T + \sum_{x \in \mathbf{x}} w_X \cdot x.$$

A proof appears in the Appendix. First, observe that the naive Bayes classifier's parameters are functions of the original linear classifier's parameters. The class probabilities

$$Pr(Y) = \sigma(-T) = \sigma(b),$$

depend on the negated threshold $-T$, and the corresponding bias b. Each conditional probability table $Pr(X \mid Y)$ depends on the corresponding weight w_X for feature X from the linear classifier, where the true-positive rate is equal to the true-negative rate:[3]

$$Pr(X{=}{+1} \mid Y{=}1) = Pr(X{=}{-1} \mid Y{=}0) = \sigma(w_X).$$

We also have the false-positive and false-negative rates:

$$Pr(X{=}{+1} \mid Y{=}0) = Pr(X{=}{-1} \mid Y{=}1) = \sigma(-w_X)$$

since $\sigma(-x) = 1 - \sigma(x)$. The output of the linear classifier f is obtained by thresholding the log-odds $\log \frac{Pr(Y=1|\mathbf{x})}{Pr(Y=0 \mid \mathbf{x})}$. This corresponds to the decision rule of a naive Bayes classifier, where we label a feature vector $\mathbf{x}$ positively if $Pr(Y{=}1 \mid \mathbf{x}) \geq Pr(Y{=}0 \mid \mathbf{x})$ and 0 otherwise. That is, the naive Bayes classifier of Proposition 2 labels a feature vector positively iff the corresponding linear classifier of Definition 1 labels it positively.

The ML parameters of a linear classifier can be obtained in closed form, by viewing it as a naive Bayes classifier. As in Definition 4, we call the resulting classifier a generative linear classifier. Let $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$ denote a dataset of N examples, where the i-th example has feature vector $\mathbf{x}_i$ and label y_i. Let $\mathcal{D}\#(Y{=}1)$ and $\mathcal{D}\#(Y{=}0)$ denote the number of times that class Y appears positively and negatively, respectively. Let $\mathcal{D}\#(Y{=}X)$ denote the number of times that class Y and feature X have the same sign in dataset $\mathcal{D}$. Let $\mathcal{D}\#(Y \neq X)$ denote the number of times where their signs are not equal. Note that

$$\mathcal{D}\#(Y{=}X) + \mathcal{D}\#(Y \neq X) = N.$$

We can express the log-likelihood as:

$$LL(\mathcal{D}) = \sum_{i=1}^{N} \log Pr(\mathbf{x}_i, y_i)$$
$$= \mathcal{D}\#(Y{=}1) \log \sigma(b) + \mathcal{D}\#(Y{=}0) \log \sigma(-b) +$$
$$\sum_{X \in \mathbf{X}} \mathcal{D}\#(Y{=}X) \log \sigma(w_X) + \mathcal{D}\#(Y \neq X) \log \sigma(-w_X)$$

yielding the following closed-form for the ML parameters.

Proposition 3. *Given a dataset $\mathcal{D}$, a generative linear classifier has the maximum likelihood parameters:*

$$b = \log \frac{\mathcal{D}\#(Y{=}1)}{\mathcal{D}\#(Y{=}0)} \quad \textit{and} \quad w_X = \log \frac{\mathcal{D}\#(Y{=}X)}{\mathcal{D}\#(Y \neq X)}$$

for each feature variable X.

These closed-forms can be derived by setting to zero the partial derivatives of the log-likelihood with respect to the parameters, and then solving for the parameters.

Suppose now that we want to constrain the weights to be (1) integers and (2) bounded, as in a weight-budgeted linear classifier. Maximizing the likelihood is now a combinatorial

[2] It is only pseudo-polynomial because it is polynomial in the budget, but not on the number of bits needed to represent the budget. Equivalently, it is polynomial in the size of a unary (but not binary) representation of the budget.

[3] Although we assume $Y \in \{0, 1\}$ and $X \in \{-1, +1\}$, we interpret both -1 and 0 as being "false" or "negative," and +1 and 1 are both "true" or "positive." We say $Y{=}X$ if Y and X are both positive, or both negative. We say $Y \neq X$ if their signs differ.

search problem. Say we want to learn integer weights w_X and an integer bias b where the aggregate weight must be *exactly* a given budget B.[4] That is, we think of B as a set of (integer) units that we must allocate across all weights, while maximizing the log likelihood. We can solve this problem optimally using dynamic programming (DP), as follows.

Suppose that the features $\mathbf{X}$ are ordered (i.e., $X_1, X_2, \ldots$) and let $LL[n, B]$ denote the log-likelihood of the data with respect to the first n features $\mathbf{X}_n$ where we use exactly the budget B. Note that if we allocate b units of the budget B to the last feature X_n, then the remaining $B - b$ units must be allocated to the remaining features $\mathbf{X}_{n-1} = \mathbf{X}_n \setminus X_n$. Since the log-likelihood decomposes according to the features, we can independently find the optimal weights for (1) the last feature X_n which is either $w = b$ or $w = -b$, and for (2) the remaining $n-1$ features $\mathbf{X}_{n-1}$, which we can find recursively. If we consider all possible ways of allocating B units between X_n and $\mathbf{X}_{n-1}$, we obtain the recurrence:

$$LL[n, B] = \max_{b \in \{0,\ldots,B\}} LL_{X_n}[b] + LL[n-1, B-b]$$
$$LL[0, B] = LL_Y[B]$$

where $LL_X[b]$ optimizes the weight for feature X:

$$\max_{w \in \{b,-b\}} \mathcal{D}\#(Y{=}X) \log \sigma(w) + \mathcal{D}\#(Y{\neq}X) \log \sigma(-w)$$

and where $LL_Y[b]$ optimizes the weight for class Y:

$$\max_{w \in \{b,-b\}} \mathcal{D}\#(Y{=}1) \log \sigma(w) + \mathcal{D}\#(Y{=}0) \log \sigma(-w).$$

While evaluating the recurrence, we may encounter identical sub-problems $LL[n, B]$ multiple times. By caching the results, we obtain a dynamic programming algorithm that runs in time polynomial in n and B, and hence in overall pseudo-polynomial time (because we are polynomial in B, not in the number of bits needed to represent B).

Theorem 4. *The maximum likelihood estimates of a weight-budgeted linear classifier over n features X using a budget of exactly B can be computed in $O(nB^2)$ time.*

This complexity is obtained by observing that the table $LL[.,.]$ has size $O(nB)$, and each of its cells can be filled in $O(B)$ time, by scanning the row above it. Again, to find the optimal weights within (and not just equal to) a given budget B, after populating the table, we scan the row of values $LL[n, .]$ and select the solution with maximum likelihood.

4.1 An Example

Consider a dataset $\mathcal{D}$ over two features X_1 and X_2 and class Y, with the following sufficient statistics:

$$\mathcal{D}\#(Y{=}1) = 25 \qquad \mathcal{D}\#(Y{=}0) = 75$$
$$\mathcal{D}\#(Y{=}X_1) = 66 \qquad \mathcal{D}\#(Y{\neq}X_1) = 34$$
$$\mathcal{D}\#(Y{=}X_2) = 90 \qquad \mathcal{D}\#(Y{\neq}X_2) = 10$$

Our DP algorithm populates the table $LL[n, B]$ of optimal likelihoods where the full budget B was used:

$n \backslash B$	0	1	2	3	4	5
0 (b)	-69	-56	-62	-79	-101	-125
1 (w_1)	-138	-125	-121	-128	-143	-160
2 (w_2)	-207	-179	-166	-158	$\mathbf{-154}$	-156

The last row $LL[n, .]$ corresponds to solutions over all n features. Hence, the optimal solution is found by considering all possible budgets B, which is cell $LL[2, 4]$, given in bold. Below, is a corresponding table giving the optimal parameter vector (b, w_1, w_2) for each cell $LL[n, B]$:

$n \backslash B$	0	1	2	3	4	5
0 (b)	0, 0, 0	-1, 0, 0	-2, 0, 0	-3, 0, 0	-4, 0, 0	-5, 0, 0
1 (w_1)	0, 0, 0	-1, 0, 0	-1, 1, 0	-2, 1, 0	-2, 2, 0	-3, 2, 0
2 (w_2)	0, 0, 0	0, 0, 1	-1, 0, 1	-1, 0, 2	-1, 1, 2	-1, 1, 3

Consider, for example, how to populate cell $LL[1, 2]$, where we want to find the ML parameters for b and w_1 given a budget $B = 2$, where we take the maximum out of 3 choices:

1. $LL_{X_1}[0] + LL[0, 2] = -69 - 62 = -131$
2. $LL_{X_1}[1] + LL[0, 1] = \max\{-65, -97\} - 56 = \mathbf{-121}$
3. $LL_{X_1}[2] + LL[0, 0] = \max\{-80, -144\} - 69 = -149$

The maximum (in bold) allocates one unit to feature X_1 and combines it with the solution from $LL[0, 1]$. In this example, we can calculate $LL_{X_1}[1]$ as the log likelihood of the feature X_1 given that $w_{X_1} = 1$, via:

$$\mathcal{D}\#(Y{=}X_1) \log \sigma(w_{X_1}) + \mathcal{D}\#(Y{\neq}X_1) \log \sigma(-w_{X_1})$$
$$= 66 \log(.731) + 34 \log(.269) = -65$$

and similarly compute -97 if $w_{X_1} = -1$.

4.2 Improvements

Up to this point, we have considered integer weights w_X; however, this leads to a limited domain of realizable probabilities $Pr(Y{=}{+}1 \mid X{=}{+}1) = \sigma(w_X)$. For example, for a budget of $B = 3$, we have the possibilities:

w_X	-3	-2	-1	0	1	2	3
$\sigma(w_X)$	4.7%	11.9%	26.9%	50.0%	73.1%	88.1%	95.3%

In general, if we have a budget B, the set of weights is confined to $\{-B, \ldots, 0, \ldots, B\} = \{-i, i\}_{i=0}^{B}$. We can expand the set of weights, and the corresponding set of realizable probabilities, by assuming fractional weights. In particular, we assume an (inverse) step-size k and assume instead that we have weights from the set $\{-\frac{i}{k}, \frac{i}{k}\}_{i=0}^{kB}$. For example, a weight $\frac{3}{2}$ can now represent the value $\sigma(\frac{3}{2}) = 62.2\%$. With a large enough budget B and step-size k, we can represent any (rational) probability value. Further, the time complexity of our DP algorithm in Theorem 4 becomes $O(n(kB)^2)$.

Note that the relative scale of the parameters b and w_X determine the probabilities $\sigma(b)$ and $\sigma(w_X)$ that are realizable, and hence our ability to fit a given distribution. The scale of the parameters, however, do not effect the decision function of the resulting classifier (we can always multiply both sides of a classifier by a constant, in Definition 1). Further, the scale does not effect the resulting OBDD. Hence, for compiling a linear classifier into an OBDD, as in Theorem 2, using a step size k amounts to multiplying all weights (and the budget B) by k (to restore the integrality of the weights).

[4] This simplifies the discussion. To find the best weights within a given budget B, we simply examine all solutions from 0 to B.

Finally, the conventional wisdom is that discriminative classifiers tend to outperform generative classifiers;[5] c.f., (Ng and Jordan 2001). The DP algorithm that we just proposed learns a generative classifier that maximizes the log-likelihood of the data. We can apply this same algorithm to learn a discriminative classifier based on the following. First, finding a maximum likelihood weight-vector $\mathbf{w}$ can be viewed as minimizing the KL-divergence between the data distribution $Pr_{\mathcal{D}}$ and the distribution $Pr_{\mathbf{w}}$ induced by $\mathbf{w}$:[6]

$$\max_{\mathbf{w}} LL(\mathcal{D}; \mathbf{w}) = \min_{\mathbf{w}} KL(Pr_{\mathcal{D}}, Pr_{\mathbf{w}}).$$

Logistic regression is typically viewed as learning the conditional distribution $Pr(Y \mid \mathbf{X})$. We can also view it as inducing a joint distribution $Pr(\mathbf{X}, Y)$, when we treat its weights $\hat{\mathbf{w}}$ as the parameters of a generative linear classifier, as in Definition 4. Hence, instead of using our DP algorithm to fit a dataset $\mathcal{D}$, we use it to fit the (sufficient statistics) of the distribution $Pr_{\hat{\mathbf{w}}}$ induced by a discriminative classifier's weights $\hat{\mathbf{w}}$. Hence, to learn a discriminative weight-budgeted classifier using our DP algorithm, we can first learn a discriminative linear classifier from data, and then fit a weight-budgeted classifier to its induced distribution, using our DP algorithm. This approach can also be viewed as a more principled way of approximating a linear classifier by an integer one, based on minimizing the KL-divergence between the two classifiers. This approach can also be applied to compiling a neural network to an OBDD, as in (Shi et al. 2020), based on the following steps: (1) train a neural network from data, (2) approximate its neurons with weight-budged classifiers, (3) compile the resulting neurons to OBDD, and (4) aggregate the OBDDs of the neurons into a single OBDD for the neural network. In the next section, we show empirically how our approach can obtain more accurate neurons with more compact OBDDs.

5 Experiments

We next empirically evaluate the pseudo-polynomial time training algorithm that we just proposed in Section 4 for learning a weight-budgeted classifier. We first evaluate the impact that the budget has on the quality of the resulting weight-budgeted classifier learned from data. We next evaluate the ability of a weight-budgeted classifier in fitting to a discriminative model, such as a logistic regression. Then, we explore the trade-off that our approach provides in trading off classifier fidelity and the compactness of the classifier's representation as an OBDD, which ultimately provides the ability to formally explain and verify its own behavior.

In our experiments, we use the MNIST dataset of handwritten digits, consisting of 28×28 pixel images, which we binarized to black-and-white. We consider one-versus-one classification of digits, so that we restrict ourselves to binary classification. Hence, the classifiers that we learn correspond to Boolean functions with 784 binary inputs and a single binary output. There are in total $\binom{10}{2} = 45$ different pairs (i, j) of digits. Each plot that we present in this section is an average over all 45 pairs of digits.

[5] A discriminative classifier typically estimates the posterior $Pr(Y \mid \mathbf{X})$ whereas a generative classifier typically estimates the joint distribution $Pr(\mathbf{X}, Y)$. The latter includes the former as a special case; hence the discriminative classifier can be viewed as more specialized to the classification task.

[6] A dataset $\mathcal{D}$ induces a data distribution $Pr_{\mathcal{D}}(\mathbf{x}) = \frac{1}{N}\mathcal{D}\#(\mathbf{x})$.

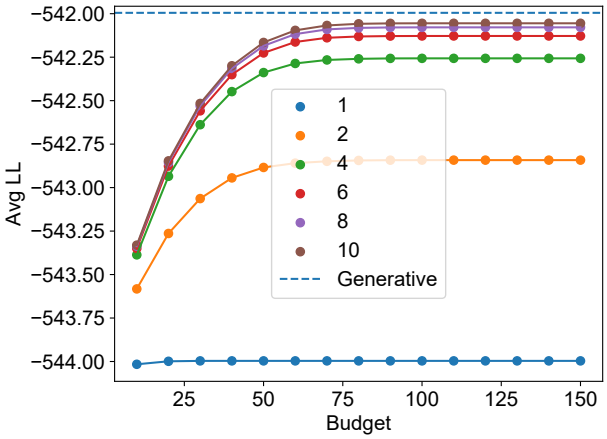


Figure 2: Budget versus Log Likelihood, Generative Classifier

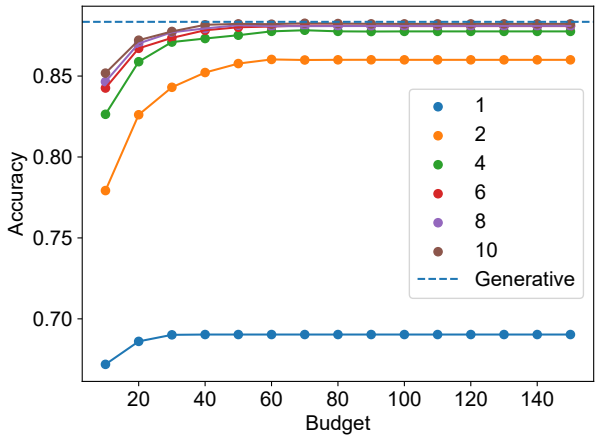


Figure 3: Budget versus Accuracy, Generative Classifier

5.1 Budget versus Quality

In a weight-budgeted classifier, an increasing budget admits a higher-performance model. Given enough of a budget, we can approach the performance of a generative linear classifier that was not constrained to have integer weights (from Theorem 1, such a classifier may not admit a compact representation as an OBDD). In particular, given a budget B and step-size k, the weights of the classifiers that we learn are drawn from the set $\{-\frac{i}{k}, \frac{i}{k}\}_{i=0}^{kB}$ of possible weights. A larger budget B and a larger step-size k provide a finer granularity to the model, allowing for greater preciseness.

A generative linear classifier assumes a true-positive rate equal to the true-negative rate, as in Section 4. Hence, the data was re-scaled prior to binarization to better fit this

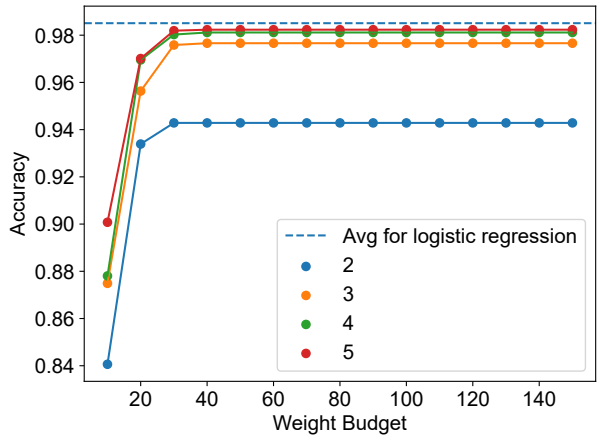


Figure 4: Budget versus Accuracy, Discriminative Classifier

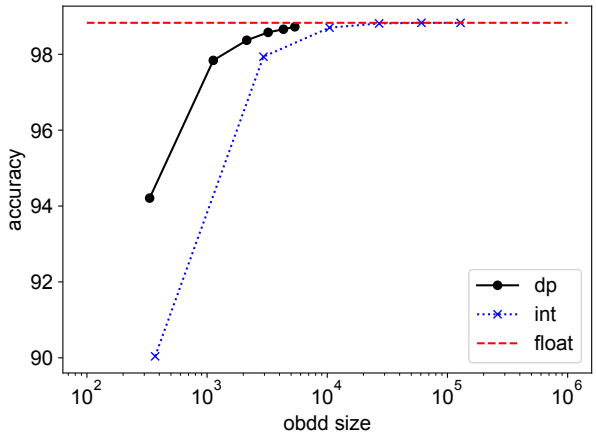


Figure 5: OBDD Size vs Accuracy

assumption.[7] In our experiments, we learned weighted-budgeted classifiers using the dynamic programming (DP) algorithm from Section 4. We compared these classifiers with the (unconstrained) generative linear classifier, whose parameters were learned in closed form using Proposition 3.

Consider Figure 2, where we increase on the x-axis the budget B available to the learning algorithm, and evaluate on the y-axis the resulting (test set) log likelihood (bigger is better). Each solid line also shows the effect of increasing the step-size k, from 1 to 10. The dashed line shows the performance of the generative linear classifier that was not constrained to using integer weights. First, as we increase the available budget B the log likelihood also increases. Second, we see that as we increase the step-size k the log likelihood also increases, and approaches the log likelihood of the (unconstrained) generative classifier, as expected. In fact, given sufficient step size k and budget B, we come arbitrarily close to the unconstrained classifier. Figure 3 illustrates a similar story, where we evaluate the quality of a classifier by (test set) accuracy rather than by log likelihood.

5.2 Fitting to a Discriminative Classifier

By viewing a (discriminative) linear classifier as inducing a joint (rather than a conditional) distribution, we can first train a discriminative classifier, and then fit a weight-budgeted classifier to it, as in Section 4.2. We may prefer this when the modeling assumptions of a generative linear classifier do not hold in the data, and allows us to approximate what is often a higher performing model. This approach can be also be used to learn a classifier (with a step-activation) with a compact OBDD: we first train a discriminative linear classifier from data, and then fit a weight-budgeted classifier to it.

Here, we trained a discriminative linear classifier for each pair (i, j) of digits, using a `LogisticRegression` model in the `scikit-learn` library.[8] We subsequently fit a weight-budgeted classifier to each, using our dynamic programming algorithm, as described in Section 4.2. Consider Figure 4, where we again increase on the x-axis the budget B available, and evaluate on the y-axis the resulting test-set accuracy. Each solid line increases the step-size k, from 2 to 5. The dashed line shows the accuracy of the original logistic regression models that we trained. Again, we see that the accuracy improves as we increase the budget, and that the performance approaches the original logistic regression model as the step-size k is increased, as expected. As shown, neither the step-size or budget need to be significantly high to have a close approximation to the original discriminative model, but we can continue to push them arbitrarily close.

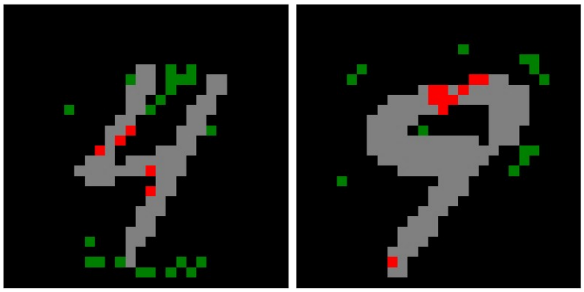
5.3 Compiling Linear Classifiers to OBDDs

In our last set of experiments, we took the weight-budgeted classifiers fit to logistic regression models in the previous experiments, and compiled them to OBDDs. We evaluate the ability of our dynamic programming algorithm in learning higher quality linear classifiers with more compact OBDDs.

Figure 5 highlights the results, where we compare the resulting OBDD size (x-axis) and accuracy of the classifier (y-axis). The black solid line represents the results of our dynamic programming (dp) algorithm. Each point represents a different level of step-size of k from 2 to 7. As we increase the level of step-size k, we find that OBDD size increases along with accuracy, as expected. A larger step-size corresponds to a higher budget used: a higher budget increases the quality (accuracy) of the classifier, but also yields a looser upper-bound on the size of the result-

[7] Each pixel was originally a grayscale value from 0 to 1. For each pixel, we found the mean value of the pixel for the positive instances, and its mean value for its negative instances. We took the middle-point of these means as the threshold for binarizing the pixel value. Some pixels were always 0 in the dataset, which implies a true-negative rate of 1 but a true-positive rate of 0. By adding random noise to each pixel, the true-negative/positive rate of such pixels become closer to $\frac{1}{2}$, hence we added random noise to all pixels so that the dataset better fits the modeling assumptions.

[8] We assumed an L_1 penalty, inverse regularization strength $C = 0.1$, tolerance 10^{-6}, and used the `liblinear` solver.



(a) Classified as 4 (b) Classified as 9

Figure 6: The red and green pixels denote the PI-explanation. Red (and green) pixels were originally white (and black); all other pixels are irrelevant to the classifier's decision.

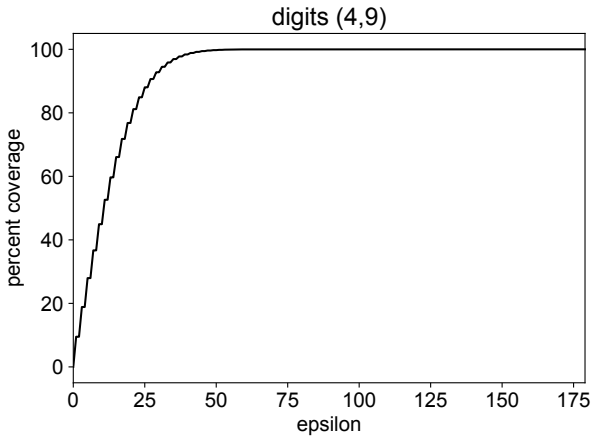


Figure 7: Percentage of all input settings which are within some ϵ of the decision boundary

ing OBDD. The red dashed-line plots the accuracy of the original classifier learned with floating-point weights, and we see that as the step-size increases, our DP algorithm approaches the accuracy of the original classifier. Finally, consider the blue dotted line. Here, we have taken the original classifier, truncated its weights to integers, and compiled the result to an OBDD; this is the approach taken by (Shi et al. 2020) to compile neurons to OBDD.[9] Here, we have multiplied all weights by increasingly larger factors $\{4, 8, 16, 32, 64, 128\}$. A larger scaling factor used preserves more precision when we truncate the weights. Again, we see that larger scaling factors yield classifiers with accuracies approaching the original classifier, along with larger OBDDs. Finally, when we compare the solid black line (dp) with the dotted blue line (casting to integer), we find that our dynamic programming algorithm can train linear classifiers that obtain better accuracies than by casting to integer, while also yielding much smaller OBDDs.

6 Explaining the Behavior of a Classifier

We next provide a case study on explaining and verifying the behavior of a linear classifier. A `LogisticRegression` classifier was trained, then fit to a weight-budgeted classifier (as in Section 5.2) to classify images of 4's and 9's. We learned a classifier with a step-size $k=2$ and a sufficient weight budget to achieve its maximum likelihood estimates. Finally, we compiled the linear classifier to an OBDD, which we used to provide PI-explanations of its predictions (Shih, Choi, and Darwiche 2018b; Darwiche and Hirth 2020).

Figure 6 depicts two PI-explanations for correctly classified images of a 4 and a 9. A PI-explanation is a sub-instantiation of a feature vector (a sub-image in this case), which is sufficient to determine the output of the classifier. For instance, in Figure 6a, the PI-explanation corresponds to the red and green pixels. The red (originally white) pixels may be used to find the sharper angles typically present in a 4. The green (originally black) pixels are mostly concentrated at the top of the number, where a 9 would usually connect, but where a 4 would not. These pixels being set to these values is sufficient for classifying this image as a 4: the other pixels in the image become irrelevant for the classifier to make its determination. Figure 6b depicts a PI-explanation for why the classifier labeled a given image as a 9. In contrast to the PI-explanation for the 4, we look for white pixels at the top of the image, where a 4 ordinarily would not connect. We also look for white pixels low in the number and at an angle, where sometimes the bottom of a 9 may hook from right to left. The PI-explanations further provide insight into a linear classifier's behavior: it did not learn *conceptually* what a digit is. Instead, it appears to be finding simple patterns in the training set that allows it to achieve a high accuracy (Shi et al. 2020).

We remark that similar observations can be made about PI-explanations for other pairs of digits. We highlighted an example from 4-vs.-9, as it represented one of the more challenging digit pairs (by test set accuracy). Further, we note that for linear classifiers, the shortest PI-explanation can be computed in polynomial time (Marques-Silva et al. 2020).

As discussed in Section 2, it is NP-complete to determine if any input will fall within a given ϵ of a linear classifier's decision boundary. However, if we have a weight-budgeted linear classifier, these types of analyses can be performed in polynomial time, via compilation to OBDD.[10] Such analyses may be useful when evaluating the robustness of a classifier to adversarial examples, which are often close to the decision boundary. Consider Figure 7, where we have taken the same 4-vs.-9 classifier, where on the x-axis we range values of ϵ from 0 to the aggregate weight $W = 179$ of the classifier. On the y-axis we plot the percentage of all of the 2^{784} possible input settings that fall within this value of ϵ. First, we observe that no input setting fall exactly on the decision

[9] Available at https://github.com/art-ai/nnf2sdd.

[10] We can modify the algorithm from (Shi et al. 2020, Appendix) for compiling a neuron into an OBDD. In the algorithm, each terminal node corresponds to a threshold value, and is set to true if it is ≥ 0 and false otherwise. Instead, we set a terminal node to true if its absolute threshold value is $\leq \epsilon$.

boundary.[11] As we increase ϵ, a larger percentage of the input space falls within ϵ of the decision boundary, until we capture the entire input space when $\epsilon = W$. Over all possible input settings, we compute the average distance from the decision boundary for this classifier to be 13.35. Over all pairs of digits, the lowest average distance was 8.58 for digit pair $(6, 7)$ while the highest was 17.35 for digit pair $(1, 8)$.

7 Conclusion

We proposed an approach for training a linear classifer from data, which guarantees a compact representation when compiled to an OBDD. Our approach is based on observing that (1) a linear classifier can be compiled to an OBDD efficiently, if its weights are bounded integers, and (2) we can learn a linear classifier with bounded and integer weights. In the latter case, we show that it is NP-hard to learn such a classifier discriminatively, but it is possible to learn such a classifier generatively, in pseudo-polynomial time via dynamic programming. We provide a more principled approach, based on maximum likelihood estimation, to prior approaches that would first train a linear classifier (or a neuron) from data, and then truncate its weights until it can be compiled to an OBDD. Empirically, our approach produces more accurate classifiers with more compact OBDD representations. Our results have implications in eXplainble Artificial Intelligence (XAI): they represent a step towards training explainable neural networks directly from data.

A Proofs

Number partitioning is a well-known NP-complete problem.

PARTITION: Given a set of n positive integers $A = \{a_1, \ldots, a_n\}$, does there exist a subset $A^\star \subseteq A$ s.t.

$$\sum_{a \in A^\star} a = \sum_{a \in A \setminus A^\star} a?$$

Proof of Proposition 1. To show that ϵ-**DB** is NP-complete, we show that it is in NP and it is NP-hard. Membership in NP is trivial. We show how to reduce **PARTITION** to ϵ-**DB**.

Our linear classifier has n weights, where $w_i = a_i$. Further, the threshold is $T = 0$. Suppose we find an input setting where $\sum_i w_i \cdot x_i = 0$, where each $x_i \in \{-1, +1\}$. The inputs set to +1 form a subset $A^\star$, and the inputs set to -1 form the complement $A \setminus A^\star$. To sum to 0, each subset must each sum to the same value. □

X3C is an NP-complete problem (Garey and Johnson 1979).

EXACT COVER BY 3-SETS (X3C): Given a set $\mathbf{X}$ containing $3n$ elements, and a collection $\mathbf{C}$ of 3-element subsets of $\mathbf{X}$, does there exist an exact cover of $\mathbf{X}$, i.e., is there a subset $\mathbf{C}' \subseteq \mathbf{C}$ where every element of $\mathbf{X}$ appears in exactly one element of $\mathbf{C}'$?

[11] Since each input $X \in \{-1, 1\}$, the parity of $\mathbf{w}^T\mathbf{x}$ does not change. In this case, this parity of $\mathbf{w}^T\mathbf{x}$ differed from the parity of the threshold, and hence they can never match.

Proof of Theorem 3. Consider an X3C problem with $|\mathbf{X}| = 3n$ and $n \geq 2$, and with $|\mathbf{C}| = m$. We reduce X3C to minimizing the MSE of a dataset for a weight-budgeted linear classifier. We assume non-positive weights. We construct a dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^{6n}$ where each data example $\mathbf{x}_i$ is a vector $(x_{i1}, \ldots, x_{im})$ over m features. The dataset has two halves:

- The first $3n$ data examples have features $x_{ij} = 1$ if the i-th element of $\mathbf{X}$ is present in the j-th element of $\mathbf{C}$, and $x_{ij} = -1$ otherwise. Further, we have labels $y_i = 1$.
- The last $3n$ data examples have features $x_{ij} = 1$. Further, we have labels $y_i = 0$.

Assume we have a budget $B = n$ that we must allocate across the weights $w_1, \ldots, w_m$ and bias b. We want to show that an allocation minimizes the MSE iff it is an exact cover of the original X3C problem. In particular, $w_j = -1$ if the j-th element of $\mathbf{C}$ is part of the cover, and $w_j = 0$ otherwise. Further, $b = 0$.

Each term of the MSE has the form $[y_i - \sigma(\mathbf{w}^T\mathbf{x}_i + b)]^2$. In the first $3n$ terms of the MSE, we have the more specific form $[1 - \sigma(W_i^+ - W_i^- + b)]^2$ where

$$W_i^+ = \sum_{j : x_{ij} = 1} w_j \qquad W_i^- = \sum_{j : x_{ij} = -1} w_j$$

The last $3n$ terms of the MSE have the common form $[0 - \sigma(W + b)]^2$ where $W = \sum_j w_j$. Note that decreasing a feature's weight w_j will increase the error in 3 terms and decrease the error in $6n - 3$ terms. Further, increasing the bias b decreases the error in the first $3n$ terms, and increases the error in the last $3n$ terms. Analogously, if we decrease the bias. Hence, for $n \geq 2$, if we allocate a unit of weight, we prefer to allocate it to a feature (negatively), rather than allocate it to the bias. Hence, $b = 0$ when the MSE is minimized.

Next, we show that to minimize the MSE, exactly one (negative) unit of weight is allocated to each feature of an exact cover. In this case, each of the first $3n$ terms has $W^+ = -1$ and $W^- = -(n-1)$ and error $(1 - \sigma(n-2))^2$. Further, each of the last $3n$ terms has the error $(0 - \sigma(-n))^2$. We make use of the following Lemma.

Lemma 1. *Suppose we have a partial allocation that covers an element of* $\mathbf{X}$ *more than once by elements of* $\mathbf{C}$*. That is,* $|W_i^+| > 1$ *for some* $\mathbf{x}_i$*. There exists another partial allocation using the same total weight, with lower MSE, and where* $|W_i^+| \leq 1$ *for all* $\mathbf{x}_i$*.*

Proof. Since our budget is $B = n$, if one element i of $\mathbf{X}$ is covered more than once, then there is another element j of $\mathbf{X}$ that is uncovered. Suppose that k units have been allocated so far, and that $m > 1$ units were allocated to i and no units were allocated to j, i.e., $|W_i^+| = m$ and $W_j^+ = 0$. The i-th and j-th term of the MSE is thus:

$$[1 - \sigma(k - 2m)]^2 + [1 - \sigma(k)]^2.$$

We show that re-allocating one unit of weight from i to j will result in a lower error:

$$[1 - \sigma(k - 2m + 2)]^2 + [1 - \sigma(k - 2)]^2.$$

Once we show this, we can iteratively re-allocate weights from covered elements to uncovered elements, until we obtain a partial allocation where elements are covered at most once, and further with lower MSE. (A subset of k elements from an exact cover would be such a partial allocation).

Since $1-\sigma(x)=\sigma(-x)$ it suffices to show that

$$\sigma(-k+2m)^2+\sigma(-k)^2 > \sigma(-k+2m-2)^2+\sigma(-k+2)^2$$

or equivalently, we show that

$$\sigma(-k+2m)^2-\sigma(-k+2m-2)^2 > \sigma(-k+2)^2-\sigma(-k)^2.$$

First, note that if $a<b<c<d$ and $d-c>b-a$ then $d^2-c^2>b^2-a^2$. This follows from the fact that d^2-c^2 is the difference in area between two squares of widths c and d, which is greater than the difference in area b^2-a^2 of two smaller squares with a smaller difference between widths a and b. Noting that

$$\sigma(-k)<\sigma(-k+2)<\sigma(-k+2m-2)<\sigma(-k+2m)$$

it now suffices to show that

$$\sigma(-k+2m)-\sigma(-k+2m-2)>\sigma(-k+2)-\sigma(-k).$$

For $m\in[2,k)$, the left-hand side ranges from

$$\sigma(-k+4)-\sigma(-k+2)$$

when $m=2$ to

$$\sigma(k-2)-\sigma(k-4)$$

when $m=k-1$. In any range $[-z,z]$, the sigmoid function σ increases slowest at the endpoints. Thus, since the gradient is always steeper between $\sigma(-k+2)$ and $\sigma(k-2)$, the inequality holds (the differences are equal when $m=k$). □

Assume, for contradiction, that the MSE solution does not form a perfect cover. We consider three cases.

Case 1: the solution does not use the full budget. Say the solution has allocated $1\le k<n$ units of weight (if $k=0$, then adding one negative unit of weight to any feature decreases the error). By Lemma 1, we can assume that the solution is a partial allocation where each element of $\mathbf{X}$ is covered at most once. Further, we can assume that it is a subset of an exact cover (which would have the same MSE). In this case, there exists a feature j in $\mathbf{C}$ whose elements of $\mathbf{X}$ are not yet covered. Suppose that we allocate one (negative) unit of weight to this feature. 3 terms of the MSE, those covered by feature j, would increase from $[1-\sigma(k)]^2$ to $[1-\sigma(k-1)]^2$. The other $6n-3$ terms of the MSE would decrease, either from $[1-\sigma(k-2)]^2$ to $[1-\sigma(k-1)]^2$ if it was already covered, or from $[1-\sigma(k)]^2$ to $[1-\sigma(k+1)]^2$ if it was not yet covered. Since the gradient of $\sigma(x)$ becomes smaller as x is further from 0, we have

$$[1-\sigma(k-1)]^2-[1-\sigma(k)]^2 \le [1-\sigma(k-2)]^2-[1-\sigma(k-1)]^2$$

and the increase in error from the 3 newly covered features (left-hand-side) is outweighted by the decrease in error by at least 3 of the previously covered features (right-hand-side). Hence, we could improve the solution by allocating an additional unit of weight.

Case 2: the solution uses the full budget, but allocates more than one unit of weight to a single feature. If a feature is allocated more than one unit of weight, then each element of $\mathbf{X}$ that the feature covers, is covered multiple times. By Lemma 1, an allocation that assigns at most one unit of weight to each feature (i.e., an exact cover) would result in a lower MSE.

Case 3: the solution distributes one unit weight to each of n features, but it does not correspond to a perfect cover. In this case, there is some uncovered element of $\mathbf{X}$, and also, there is some element of $\mathbf{X}$ that is covered more than once. By Lemma 1, an allocation that covers each element of $\mathbf{X}$ once (i.e., an exact cover) would result in a lower MSE. □

Proof of Proposition 2. To simplify the proof, we assume $Y\in\{-1,+1\}$ rather than $Y\in\{0,1\}$, where $Y=0$ and $Y=-1$ are both equivalent to Y being "false" or "negative."

First, assuming $Pr(Y=+1)=\sigma(b)$ we have

$$Pr(Y=-1)=1-\frac{1}{1+\exp\{-b\}}=\frac{1}{1+\exp\{b\}}$$

Similarly, assuming $Pr(X_a=\pm1 \mid Y=\pm1)=\sigma(w_X)$:

$$Pr(X_a=\pm1 \mid Y=\mp1)=\frac{\exp\{-w_a x_a\}}{1+\exp\{-w_a x_a\}}=\frac{1}{1+\exp\{w_a x_a\}}.$$

Thus,

$$Pr(X_a \mid Y=+1)=\frac{1}{1+\exp\{-w_a x_a\}}$$
$$Pr(X_a \mid Y=-1)=\frac{1}{1+\exp\{w_a x_a\}}$$

since in the first case we want the exponent to differ from X_a and in the second case we want them to match. Thus

$$\log\frac{Pr(Y=+1\mid\mathbf{x})}{Pr(Y=-1\mid\mathbf{x})}=\log\frac{Pr(Y=+1)Pr(\mathbf{x}\mid Y=+1)}{Pr(Y=-1)Pr(\mathbf{x}\mid Y=-1)}$$
$$=\log\Big(\frac{1}{1+\exp\{-b\}}\div\frac{\exp\{-b\}}{1+\exp\{-b\}}\Big)$$
$$\cdot\prod_{i=1}^{n}\Big(\frac{1}{1+\exp\{-w_i x_i\}}\div\frac{\exp\{-w_i x_i\}}{1+\exp\{-w_i x_i\}}\Big)$$
$$=\log\exp\{b\}\cdot\prod_{i=1}^{n}\exp\{w_i x_i\}=b+\sum_{i=1}^{n}w_i x_i$$

as desired. □

References

Audemard, G.; Koriche, F.; and Marquis, P. 2020. On tractable XAI queries based on compiled representations. In *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning (KR)*.

Baehrens, D.; Schroeter, T.; Harmeling, S.; Kawanabe, M.; Hansen, K.; and Müller, K. 2010. How to explain individual classification decisions. *Journal of Machine Learning Research (JMLR)* 11:1803–1831.

Cadoli, M., and Donini, F. M. 1997. A survey on knowledge compilation. *AI Commun.* 10(3-4):137–150.

Chan, H., and Darwiche, A. 2003. Reasoning about Bayesian network classifiers. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, 107–115.

Chubarian, K., and Turan, G. 2020. Interpretability of bayesian network classifiers: Obdd approximation and polynomial threshold functions. In *International Symposium on Artificial Intelligence and Mathematics (ISAIM)*.

Cooper, M. C., and Marques-Silva, J. 2021. On the tractability of explaining decisions of classifiers. In *27th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 210, 21:1–21:18.

Darwiche, A., and Hirth, A. 2020. On the reasons behind decisions. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI)*.

Darwiche, A., and Marquis, P. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research (JAIR)* 17:229–264.

Elkan, C. 1997. Boosting and naive Bayesian learning. Technical Report CS97-557, Department of Computer Science and Engineering, University of California, San Diego.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman.

Hubara, I.; Courbariaux, M.; Soudry, D.; El-Yaniv, R.; and Bengio, Y. 2016. Binarized neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, 4107–4115.

Ignatiev, A.; Narodytska, N.; and Marques-Silva, J. 2019. On relating explanations and adversarial examples. In *Advances in Neural Information Processing Systems 32 (NeurIPS)*, 15857–15867.

Katz, G.; Barrett, C. W.; Dill, D. L.; Julian, K.; and Kochenderfer, M. J. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Computer Aided Verification (CAV)*, 97–117.

Leofante, F.; Narodytska, N.; Pulina, L.; and Tacchella, A. 2018. Automated verification of neural networks: Advances, challenges and perspectives. *CoRR* abs/1805.09938.

Lipton, Z. C. 2018. The mythos of model interpretability. *Commun. ACM* 61(10):36–43.

Marques-Silva, J.; Gerspacher, T.; Cooper, M. C.; Ignatiev, A.; and Narodytska, N. 2020. Explaining naive Bayes and other linear classifiers with polynomial time and delay. In *Advances in Neural Information Processing Systems 33 (NeurIPS)*.

Narodytska, N.; Kasiviswanathan, S. P.; Ryzhyk, L.; Sagiv, M.; and Walsh, T. 2018. Verifying properties of binarized deep neural networks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI)*.

Ng, A. Y., and Jordan, M. I. 2001. On discriminative vs. generative classifiers: A comparison of logistic regression and naive Bayes. In *Advances in Neural Information Processing Systems 14 (NIPS)*, 841–848.

Rastegari, M.; Ordonez, V.; Redmon, J.; and Farhadi, A. 2016. XNOR-Net: Imagenet classification using binary convolutional neural networks. In *Proceedings of the 14th European Conference on Computer Vision (ECCV)*, 525–542.

Ribeiro, M. T.; Singh, S.; and Guestrin, C. 2016. "why should i trust you?": Explaining the predictions of any classifier. In *Knowledge Discovery and Data Mining (KDD)*.

Ribeiro, M. T.; Singh, S.; and Guestrin, C. 2018. Anchors: High-precision model-agnostic explanations. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI)*.

Selman, B., and Kautz, H. A. 1996. Knowledge compilation and theory approximation. *J. ACM* 43(2):193–224.

Shi, W.; Shih, A.; Darwiche, A.; and Choi, A. 2020. On tractable representations of binary neural networks. In *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning (KR)*.

Shih, A.; Choi, A.; and Darwiche, A. 2018a. Formal verification of Bayesian network classifiers. In *Proceedings of the 9th International Conference on Probabilistic Graphical Models (PGM)*.

Shih, A.; Choi, A.; and Darwiche, A. 2018b. A symbolic approach to explaining Bayesian network classifiers. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*.

Shih, A.; Choi, A.; and Darwiche, A. 2019. Compiling Bayesian networks into decision graphs. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI)*.

Shih, A.; Darwiche, A.; and Choi, A. 2019. Verifying binarized neural networks by Angluin-style learning. In *The 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT)*.