Rapid-SPN: Sum-Product Network using Random Projection

Prajay Shetty
prajay.shetty2016@gmail.com
Department of Computer Science
University of Georgia
Athens,Georgia

September 2024

Abstract

We introduce a novel lightning-fast simple structure learning for Sum-product Networks (SPNs). Our approach also works in an area where number of columns are pretty high and no of samples are pretty low i.e.High Dimensional Dataset with Low Samples (HDLS). SPN are Probabilistic Graphical Models (PGM) that can perform marginalization,generalization and inference in linear time. SPNs are well known for their simplicity in structure as compared to deep learning networks. However, they need a robust layer approach to build a good structure. We investigate a new simple approach that optimizes the structure of SPN inside the product node itself for minimal time and memory by using CPU only. Our approach leads to better results in a standard dataset in terms of computation time and log-likelihood. In the end, we come up with [exciting] results for the CIFAR-10 and STL-10 dataset for HDLS.

Keywords— Sum Product Network, Random Projection, RP-Tree, High Dimension with Low Sample Datasets

INTRODUCTION

Prominent AI models, like Convolutional Neural Networks (CNNs) (Yann LeCun, Patrick Haffner, Leon Bottou Yoshua Bengio 1998), rely on vast amounts of data. However, such data is not always accessible for every domain. Despite this, Deep Neural Networks (DNNs) have substantially advanced AI capabilities. Notably, current CNN models lack the ability to perform marginalization, inference, and MPE concurrently in near real-time. However, this requires a huge amount of dataset and GPU time, making them unsuitable for CPU-based systems. In contrast, Probabilistic Graphical Models (PGMs) harness graph structures to deliver swift and efficient inference, MPE, and marginalization in near real-time

Fields such as Biology are often hindered by a scarcity of data points, leading to datasets that fall within the category of HDLS.Despite advances in computing power, which may facilitate computational solutions, understanding the underlying probability distributions in these fields remains a formidable challenge due to data scarcity.

Principal Component Analysis (PCA) are well known as dimensionality reduction technique. However, PCA cost a lot of CPU and memory usage. Additionally, dimensionality reduction technique removes the aspect of linear dimension. Since it reduces dimension, it makes it impossible for SPNs to perform inference in original dimension. Therefore, we resort to random projection. In order to introduce random projection principle and to preserve marginalization,generativity and inference, we use Random Projection Tree (RP-Trees) only in sum node as a clustering technique

[1] SPNs (Robert Gens, Pedro Domingos 2013) are one such model that reduces model complexity as compared to DNNs. Moreover, they also provide marginalization and inference in linear time. Besides, Random SPNs (Rat-SPNs) (Robert Peharz, Antonio Vergari and et al 2019) support not only architecture like DNNs but also have relative simplicity of SPNs. However, all SPN models performs well when there is huge amount of data points. To this end, we investigate a technique wherein a SPN model can learn on less amount of data points

Probabilistic Graphical Models (PGMs), a distinct class of models, leverage graphs for efficient inference and marginalization in linear time. Unlike traditional Neural Networks, PGMs exploit graph-based representations to encode dependencies and relationships. However, Bayesian Networks suffers from computation time, when the number of variables increase while calculating normalization function. Therefore, we resort to SPN. SPNs do have relationship with Bayesian Networks (Han Zhao, Mazen Melibari, Pascal Poupart 2015) in terms of probability distributions and structure.

Our main motivation is to address the curse of dimensionality by integrating Sum-Product Networks (SPNs) with Random Projection trees, effectively mitigating the challenges of high-dimensional data with limited samples(HDLS).

We contribute following to the field of the Sum Product Networks:

- We introduce the novel SPN model in the domain of HDLS.
- We introduce Random Projection to the Sum Product networks.
- We introduce a Lightning Fast structure learning approach.
- We provide computationally Low-Cost model for SPNs.
- We introduce a novel log-likelihood based product node approach for better independence assumption.
- We derive a asymptotic running time for the SPNs .
- We introduce therom proving that SPN aquires properties from tree algorithm.
- We prove the experimental validation of Rapid-SPN for HDLS by simulating CIFAR-10 and STL-10 dataset.

We start by introducing various existing SPNs. We then introduce Rapid-SPN and its related theorems. Then, we thoroughly test and evaluate Rapid-SPN especially CIFAR-10 (Alex Krizhevsky 2009) and STL-10 (Adam Coates, Honglak Lee, Andrew Y. Ng 2011) simulated for HDLS. Surprisingly, our result is better in time and accuracy. Finally, we conclude and discuss future work.

[1]Github: https://github.com/CodeMaster001/Rapid-SPN

BACKGROUND AND RELATED WORK

RANDOM PROJECTION TREES

Random Projection property is defined as follows: Choose a random unit direction $v \in D$ Pick any $x \in S$: Let $y \in S$ be the farthest point from it. Choose δ uniformly at random in [-1,-1].6 $\|x\text{-}y\|\sqrt{D}$. The entire process can be done in linear time.

The Random Projection Tree (RP-Tree) algorithm (Sanjoy DasGupta, Yoav Ferund 2008) implements the aforementioned property. Various RP-Tree versions exist, differing in branch splitting: RP-Tree-max, RP-Tree-mean and RP-Tree-min. For this paper, we opted for the RP-Tree-max version.

Another variation of RP-Tree algorithm exists known as Spill-Tree (Brian McFee 2011). The difference between the Spill-Tree and the RP-Tree is that it recursively splits branches for each dimension. Moreover, they add extra width at the center of the line of the projection. However, this process is slightly costlier than the original RP-Tree. Therefore, we did not opted for to Spill-Tree. RP-Tree and Spill-Tree are also called as spatial trees. For further discussion on various spatial trees, please refer [(Sanjoy DasGupta, Yoav Ferund 2008), (Dmitriy Fradkin, David Madigan 2002), (Brian McFee 2011)].

SUM-PRODUCT NETWORKS

The SPN $\boldsymbol{S}$ (Hoifung Poon, Pedro Domingos 2009) is comprises of directed acyclic graph (DAG) consisting of a sum node, product node and leaf node defined over a scope and variable X. Internal part of the DAG consists of a sum node or a product node. A sum node is defined as $\sum_0^n X_i = 1$ over $V \subset S$. On the contrary, a product node is defined as $\prod_{i=0}^n X_i$ over a disjoint scope $V \subset S$. Finally, leaf nodes represents the individual probability distributions of each variable $\boldsymbol{X}$.

Completeness and Decomposable property: The Sum-Product Network (SPN) is considered complete if every child of a sum node shares the same scope. Conversely, an SPN is deemed decomposable if all product nodes are defined over mutually independent subsets.

In a Sum-Product Network (SPN), completeness of all sum nodes ensures a proper mixture model, while decomposability of product nodes guarantees a factorized distribution among their children. This structural integrity enables efficient inference on random variables.

Consequently, operations like marginalization and conditioning can be performed in linear time, facilitating streamlined probabilistic reasoning. **Structure Learning and Weight Learning approaches**: Different types of structure learning approaches had been proposed to date [(Diarmaid Conaty, Jesús Martínez Del Rincon and et al 2018), (Robert Peharz, Antonio Vergari and et al 2019), (Aaron Dennis, Dan Ventura 2015), (Mattia Desana, Christoph Schnorr 2017), (Jinghua Wang, Gang Wang 2016), (Amirmohammad Rooshenas, Daniel Lowd 2014)]

Building Sum-Product Networks (SPNs) entails two crucial processes. Structure learning determines the optimal network architecture, while weight learning optimizes sum and leaf node weights using an optimizer and loss function (Robert Gens, Pedro Domingos 2012). Various weight learning approaches have been proposed (Robert Gens, Pedro Domingos 2012); however, RapidSPN uniquely relies solely on structure learning, bypassing weight learning.We have not opted for weight learning.

RANDOM PROJECTION INSIDE SUM PRODUCT NETWORK

A Sum-Product Network performs clustering under the Sum Node. To perform clustering, it is essential to split the dataset into at least two parts. We utilize Random Projection to perform this split in real-time, using the equation mentioned in the theorem. This process of splitting in a Random Projection Tree is referred to as the Random Projection Principle, which we integrate into the Sum Node. Once the split is performed, one side of the split becomes the left-hand side of the Sum Node, and the other side becomes the right-hand side. Then, a Product Node is formed using the Naive Factorization technique.

RANDOM SUM-PRODUCT NETWORKS

We extend the concept of Randomized Sum-Product Networks (Rat-SPNs). Rat-SPNs operate by constructing sum and product nodes from Gaussian Random Matrices. They generate Randomized Tensor Networks using region graphs, which define the structural arrangement of sum and product nodes. This graph-based structure facilitates the creation of tensor networks.

For further details, please refer to (Robert Peharz, Antonio Vergari and et al 2019).

CONDITIONAL SUM-PRODUCT NETWORK

Conditional Sum-Product Networks (CSPNs) integrate deep neural networks with SPNs, enhancing marginalization and inference capabilities in linear time. CSPNs introduce Gating nodes, functioning as experts within Gaussian Mixture Models (GMMs). Additionally, they propose leveraging Random Projection Trees as Gating nodes. For further details, please refer to (Xiaoting Shao, Alejandro Molina and et al 2019).

GREEDY BASED SUM-PRODUCT NETWORKS

They build the structure learning algorithm using greedy approach. Their key factor includes using MPE inference to select a product node. They called this algorithm Mix Clones. For additional detail please refer (Aaron Dennis, Dan Ventura 2015).

SUM-PRODUCT NETWORKS STRUCTURE LEARNING BY EFFICIENT PRODUCT NODES DISCOVERY

This literature introduces LearnSPN for high-dimensional datasets, leveraging random subspaces around product nodes. The authors develop a mechanism called Generalized Variance Splitting (GVS). GVS conducts G-tests on graph-connected components. Specifically, a graph forms when variables pass the G-test, applied to randomly selected variables.

For additional details, please refer to (Nicola Di Mauro, Floriana Esposito, and et al. 2018)

ON THE LATENT VARIABLE INTERPRETATION IN SUM PRODUCT NETWORKS

They address the latent variable issue in SPN literature by introducing product nodes that function as switch operators. These switches toggle on/off based on indicator variable status, ensuring both completeness and decomposability properties. Furthermore, they provide a sound proof for the Expectation-Maximization (EM) algorithm in SPNs.

For details, please refer to (Robert Peharz, Robert Gens and et al. 2016)

RANDOM PROBABILISTIC CIRCUTS

This study investigates the mixture mode of Probabilistic Circuits (PCs) utilizing V-trees. Notably, it employs Region Graphs, similar to Random Sum Product Networks (RSPNs). However, un-

like RSPNs, which train random nodes, this approach trains Probabilistic Circuits (PCs). For further details, please refer to (Nicola Di Mauro, Gennaro Gala, Marco Iannotta, Teresa M.A. Basile 2021).This model also runs on CPU.

Rapid-SPN

The Rapid-SPN consist of sum nodes, product nodes and leaf nodes.A comparison of RapidSPN and LearnSPN using the Iris dataset reveals similarities in their structure

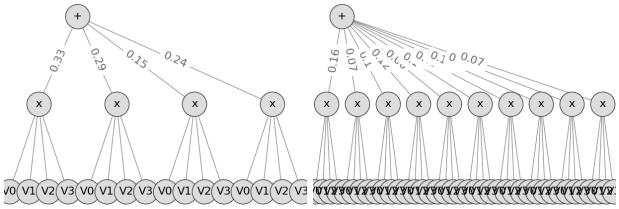

Figure 1: **LearnSPN vs Rapid-SPN for IRIS Dataset**

SUM NODE

We utilize sum nodes to incorporate the random projection property, exclusively for clustering purposes. To achieve this, we employ the RP-Tree algorithm. Following each RP-Tree split, we verify if the partition's mean is zero. If it is, we bypass that partition; otherwise, we create a sum node from it. This process prevents normal distributions with zero mean or standard deviation from forming at SPN leaf nodes. As illustrated in Algorithm 1, the **per-**

Algorithm 1 Sum Node

$left, right \leftarrow perform\_rp().$
$lweight \leftarrow determine\_weight(left))$
$rweight \leftarrow determine\_weight(right))$
$lsum, rsum \leftarrow build\_sum(left, lweight, right, rweight)$
return $Sum(lsum, rsum)$

form_rp function returns the dataset after applying random projection. Based on the resulting partition, we skip splits containing only one element (left or right). Furthermore, similar to Spill-Tree, if either split's size falls below a predetermined threshold, we create a Naive Factorized distribution for that partition.

PRODUCT NODE

Our product node consists of 3 parts. The Preference algorithm, the Elector algorithm, and the Validator.

PREFERENCE ALGORITHM (δ) Let the Preference algorithm be denoted as δ. This algorithm establishes relationships between feature scopes by leveraging distance metrics such as Cosine Similarity or Dice Coefficient. Specifically, we utilize Cosine Similarity to construct a (Feature, Feature) matrix, which is subsequently passed to the Elector algorithm for further processing.

ELECTOR ALGORITHM (β) Let the Elector Algorithm be called β. The elector algorithm's job is to decide who will be our candidates for the validator algorithm.

Algorithm 2 illustrates the Elector algorithm. It operates in two stages:

Algorithm 2 Elector Algorithm(δ, m)

$variance\_value, variance\_features \leftarrow variance\_threshold(\delta)$

if $variance\_value < threshold$ **then**
  return $scope$
end if
$selected\_feature \leftarrow variance\_features[0]$
$pref \leftarrow preference\_algorithm(variance)$
$sorted \leftarrow sort(pref)$
$chunks \leftarrow split\_by\_chunks(\hat{sorted}, m)$

- Variance Thresholding: We apply variance thresholding to filter features. If multiple features exceed the threshold, we return the full scope. Otherwise, we select the first feature surpassing the threshold.
- Feature Combination: We sort features based on the Preference algorithm's output. Then, we employ round-robin selection to divide the sorted features into chunks (m). To generate diverse feature combinations, we permute chunk sizes. These chunks are subsequently passed to the Validator algorithm.

VALIDATOR ALGORITHM (γ) The Validator algorithm's job is to evaluate various candidates produced by the Elector Algorithm. To do this, we create a sum-rooted SPNs. The childrens of the sum rooted SPN will be a product nodes from a Naive Factorized distribution of each chunk i.e. they will have indepen-

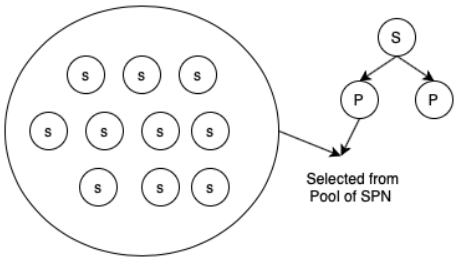

Figure 2: **SPN selection process: Validator algorithm and Preference Algorithm** The left side of the —derived by elector algorithm. The right side of the figure[2] shows an example of SPNs consisting of Factorized Product Distribution.

dent scopes. Each of this sum-rooted SPN will have linear weights assigned to every node. After this ll of each of the candidate SPNs LL is evaluated. The SPNs that scores the maximum mean LL is assigned as the children of the current product node. An example of this process is shown in Figure[2]. The Validator algorithm formula is given below:

$$\boxed{\gamma = \max_{1 \le j \le n} [\hat{S}_{\text{MLE}}(\beta(j))]} \quad (1)$$

LEAF NODE

Let us now examine leaf nodes. There are two ways to get leaf nodes depending on the scope. First, If the RP-Tree has not reached the designated depth and we have only one scope, then we create a leaf node out of univariate distribution. Second, if RP-Tree has

reached the selected depth(h), and we have more than one column present in scope, then we build a Naive Factorized leaves.

NAIVE BAYES ASSUMPTION In leaf nodes, we incorporate Laplace Smoothing from the Naive Bayes Algorithm. To address zero-valued data points, we avoid setting the mean to zero; instead, we append a standard value. This ensures consistency with the assumption that unseen data points within the current scope may still appear in the future. Notably, this approach contrasts with LearnSPN, which removes nodes corresponding to zero-valued data points. [(Hoifung Poon, Pedro Domingos 2009), (Alejandro Molina, Antonio Vergari and et al 2019)].

THEOREMS

In this paper, we present four theorems that advance the understanding of Sum-Product Networks (SPNs). The first theorem explains why integrating RP-Trees with SPNs reduces the error rate. The second theorem bridges the gap between theory and practice by using mathematical induction to show how RP-Tree variations can be used to develop new SPN algorithms. Additionally, it justifies the equivalence of inference and marginalization in Rapid-SPN and traditional SPNs. The third theorem introduces a novel method for analyzing the running time of any SPN algorithm. The fourth theorem demonstrates the power of SPNs in absorbing properties from tree algorithms and generating machine learning models, while also proving that Rapid-SPN's running time is comparable to that of the RP-Tree algorithm.

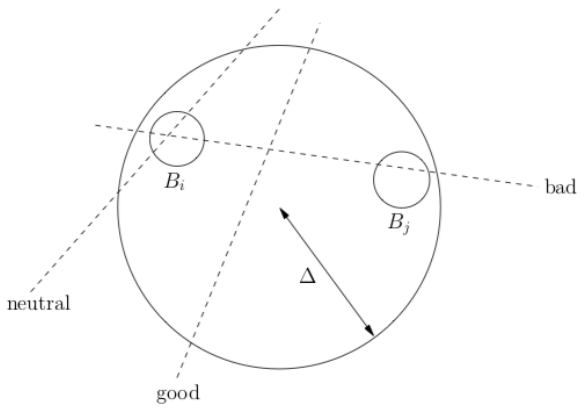

Figure 3: **Splits performed by RP-Tree.** The above figure shows 3 type of split performed by RP-Tree-good, neutral and bad, which is taken from (Sanjoy DasGupta, Yoav Ferund 2008). The Rapid-SPN helps RP-Tree to achieve more good split compared to bad and neutral. The theorem 4.1 proves that Rapid-SPN helps RP-Tree

Theorem 0.1 *A SPN helps RP-Tree to provide better log-likelihood compared to random projection.*

The learnSPN algorithm is a recursive algorithm that splits data up to leaf criteria. LearnSPN is defined in (Hoifung Poon, Pedro Domingos 2009).

Now, consider the Figure[3] wherein balls B_i and B_j can be separated either as a good split wherein they fall in the proper position. They might fall as a bad split wherein part of the ball is destroyed due to cut, as shown in the Figure[3]. Alternatively, they can have a neutral split wherein a small part of one ball is broken. This depends on the random projection performed by the RP-Tree. Further discussion about the split performed by the RP-Tree, please read (Sanjoy DasGupta, Yoav Ferund 2008).

The probability that the RP-Tree algorithm performs the best split is at least 1/192, as per Lemma 9 of 3.2 of (Sanjoy DasGupta, Yoav Ferund 2008). So there will be split that is bad or neutral. Suppose there is a split performed by RP-Tree. When there is bad or neutral split, total number of elements in the ball will be incorrect. Because of this, the weight of the sum node is incorrect. It can be seen from (Robert Gens, Pedro Domingos 2012), that weight learning algorithm improves weight of the nodes. Hence, the total number of elements in bad and neutral split will be less accurate.

On the side note, if the structure of SPN is perfect, we would get enhanced log-likelihood if the structure of the SPN is correct. The aim of the product node discussed in Section 3.2 is to provide better structure. The Rapid-SPN uses Cosine Similarity to determine the best possible column for a given dataset. By selecting the best possible nodes, we not only improve the probability of good split but also provide the best possible selection for the RP-Tree at the next level. Hence, we have shown that RP-Tree benefits from Rapid-SPN.

Proposition 1 *The RP-Tree algorithm is similar to that of binary tree (Sanjoy DasGupta, Yoav Ferund 2008).*

Theorem 0.2 *Every RP-Tree can be converted to a SPN.*

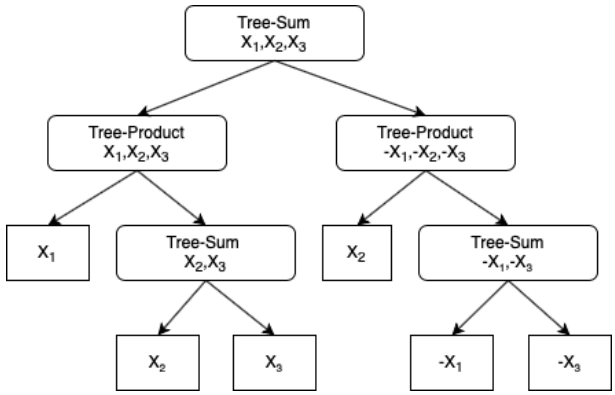

Figure 4: **A Decision made by Rapid-SPN.** The Tree-Sum, Tree-Product represent decision made by RP-Tree discussed in section 3.1 and product node discussed in section 3.2 respectively. The X represents decision made by the distribution leaf nodes.

For the base case, RP-Tree has a tree of level 2. If RP-Tree has a height of only level 1, then it would create a product-rooted SPN. However, RP-Tree with level 2 would create a sum-rooted SPN with 2 children. Please refer Figure[4] for decision made by RP-Tree algorithm and Figure[5] for level-wise comparison.

For induction hypothesis, we assume that if we have a SPN till level l_i, then we can form a SPN at level l_{i+1}.

Below are 3 cases which can be formed:

1. **Node l_{i-1} is a product node and node l_i is a sum node**: Consider the Figure[4] which shows a tree diagram till level l_i and a SPN generated till level l_{i-1}. As it can see in this Figure[4] that till level l_{i-1} a spn is already generated till level l_{i-1}. Further, For the next level l_i we see that node at level l_{i-1} is a sum node, so we will append a product node and generate children based on

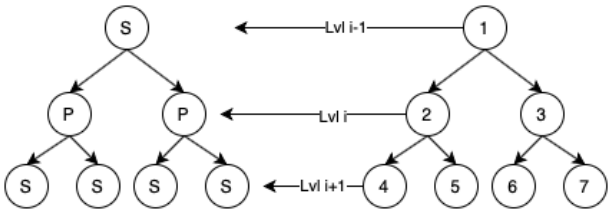

Figure 5: **A Decision made by LearnSPN with respect to levels**. At all even levels, sum node is formed with the help of RP-Tree and at all odd levels, product node is formed using our novel methodology. The last level represents the distribution of leaves.

the data available at node n_i as per to the paper by Poon and Domigos (Hoifung Poon, Pedro Domingos 2009).

2. **Node l_{i-1} is a sum node and node l_i is a product node.** This can be proved similarly like above case.
3. **Node l_{i-1} is a sum node or a product node but node l_i is a leaf node**: In this case we assume RP-Tree has already reached designated depth. Hence we create naive-factorized or univariate distribution depending on respective scopes.

Hence, using the above cases, we proved that any RP-Tree can be converted to a SPN

Theorem 0.3 *The asymptotic structure learning time of Rapid-SPN is as fast as RP-Tree.*

Let h represent the height of the Rapid-SPN. Let d represent direction of the RP-Tree. Now, let us consider a scenario where operations of sum node, product node, and leaf node viz. $S(D), P(D), L(D)$ take the same time asymptotically. Let m represent the total number of rows of a dataset and n represent total number of columns of a dataset. Let W represent the cost incurred asymptotically due to preference algorithm. Since, we are using cosine similarity, O(W) can be calculated as per below:

$$O(W) = O(m * n) \tag{2}$$

Let $F(D)$ represent final computation time asymptotically incurred due to dataset D.

In such cases, the structure produced by Rapid-SPN is similar to that of Binary Tree. Therefore, we conclude the below:

$$\therefore F(D) = O(logh) \tag{3}$$

Now, let us consider time taken for all the 3 nodes i.e $S(D), P(D), L(D)$ individually to derive $F(D)$ i.e final computation time of Rapid-SPN

- $P(D)$**:** As discussed in Section 3.2, it should be noted that the cost is primarily taken by variance threshold or Preference algorithm $O(m * n)$. After that, we perform a sorting operation which takes linear time because number of columns are pretty less.

$$\therefore P(D) = O(W) = Om * n) \tag{4}$$

Let the number of different lengths of chunks be k. Since, we have l size array, the maximum split can occur only till $m * m$.

$$\therefore P(D) = O(m^2) \tag{5}$$

However, $O(m * m) < O(mn)$.

$$\therefore P(D) = O(mn) \tag{6}$$

- $S(D)$**:** From (Brian McFee 2011), we know the spit-rule of RP-Tree takes about $O(mdn)$

$$S(D) = O(mdn) \tag{7}$$

- $L(D)$**:** The time taken to create a leaf node depends on total amount of dimensions.

$$\therefore L(D) = O(n) \tag{8}$$

In a Rapid-SPN, since the time taken for $P(D), S(D), L(D)$ are mutually exclusive. We infer $F(D)$ using $max(6, 7, 8)$ and (3) as per below. Hence,

$$\boxed{\therefore F(D) = O(loghmdn)} \tag{9}$$

Hence, the structure learning time to create product node and leaf node is smaller then sum node. The structure learning time of Rapid-SPN is similar to RP-Tree. Hence Rapid-SPN, is lightning fast.

Theorem 0.4 ***An SPN inherits the properties of the candidate algorithm***

Sum-Product Network (SPN) consists of Sum Nodes and Product Nodes, constructed through alternating additions of these nodes. The choice of Sum Node depends on the clustering algorithm used, such as K-Means, which generates Sum Nodes. Assuming a fixed Product Node, the resulting structure mirrors the Sum Node. However, when Product Nodes vary, they enhance the efficiency of Sum Nodes in a Sum-Rooted SPN. During Maximum Probability Explanation (MPE) inference, the SPN follows the path determined by K-Means clustering, due to the graph structure formed during SPN Structure Learning. Consequently, the SPN inherits properties from the candidate algorithm.

We have already proved this therotically and empirically for RP-Tree in the current paper

Corollary 0.4.1 *Rapid-SPN structure building time in practical terms is faster then RP-Tree for same height.*

The RP-Tree structure is similar to that of Rapid-SPN since both of them have the structure of the binary tree. As per Theorem 4.3, we have proved that Product operation $P(D)$ is cheaper than that of Sum Operation $S(D)$. Hence in practical terms, we should get a slightly higher speed than the RP-Tree algorithm.

EXPERIMENTAL EVALUATIONS

For building structure of Rapid-SPN, we use the SPFLOW(Alejandro Molina, Antonio Vergari and et al 2019) library. In order to do that we modify the code of (Brian McFee 2011) and use the below parameters.

- The cross-validation of **K=10** is used for our experiments.
- The **height** parameter specifies the maximum depth of RP-Tree.
- The **leaves_size** specifies the maximum number of parameters.
- The **selector** parameter specifies the amount of Scopes to scan.
- All calculations performed in the log domain.
- Classical **log-likelihood** is used for Generative Learning.
- Data points are normalized to 1.

We divide our results into multiple section but mainly we talk about 2 sections viz, CIFAR-10 and STL-10 representing HDLS, and secondly, representing other normal dataset and finally discussion about promotergene and dna datasets which are also HDLS.

Note:We could have explored CNN-based models, but those would not be feasible in the CPU space, so that option is out of consideration.

Note:For CIFAR-10 and STL-10 dataset it tooks us 2 years to get those results due to product node complexity present in the LearnSPN,

CIFAR-10 AND STL-10

In these experiments, Table[1-2] we show that our model works in the area of the HDLS. We select two dataset such as CIFAR-10 (Alex Krizhevsky 2009) and STL 10 (Adam Coates, Honglak Lee, Andrew Y. Ng 2011). The original dataset of CIFAR-10 consists of 60,000 samples, whereas STL-10 consists of 13,000 samples.

Now, we split our data points into 40 percent [2,000 samples] and 60 percent [3,000 samples], wherein 40 percent represent the training dataset and 60 percent represent the testing dataset. Then, we perform L2-Normalization and train LearnSPN and Rapid-SPN on these datasets.

Now, we modify the parameters. We increase the feature size as given here

$F = [10, 20, 40, 100, 200, 300, 400, 500, 800, 1000]$ respectively. For LearnSPN, we use instance-slice $I = [10, 20, 40, 50]$ and $T = [0.4]$ as threshold for every independent testing. For Rapid-SPN, we used $H = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]$ and $L = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]$.

Table 1: **Structure Learning Results: LEARNSPN (LSPN) VS Rapid-SPN LL and Time(sec) for different no of column width on CIFAR-10 dataset**

No of Columns	LSPN-LL	Rapid-SPN-LL	SPN-TIME(sec)	Rapid-SPN TIME(sec)
10	**27.28**	**25.52**	**52.22**	**5.4**
20	**52.13**	**49.07**	**158.94**	**13.49**
40	**134.86**	**105.08**	**981.92**	**33.21**
100	**286.53**	**301.17**	**2036.08**	**81.21**
200	**590.81**	**648.17**	**7650.71**	**173.24**
300	**891.92**	**990.03**	**14186.77**	**241.27**
400	**1195.01**	**1328.68**	**28358.15**	**344.34**
500	**1505.46**	**1687.37**	**34973.37**	**428.54**
800	**2477.60**	**2796.29**	**70897.30**	**748.74**
1000	**3152.80**	**3601.21**	**3152.80**	**883.03**

Table 2: **Structure Learning Results: LEARNSPN (LSPN) VS Rapid-SPN LL and Time(sec) for different no of column width on STL-10 dataset**

No of Columns	LSPN-LL	Rapid-SPN-LL	SPN-TIME(sec)	Rapid-SPN TIME(sec)
10	**50.27**	**44.80**	**40.15**	**2.66**
20	**103.28**	**94.36**	**127.01**	**9.36**
40	**207.36**	**196.25**	**502.16**	**17.73**
100	**533.73**	**513.73**	**3882.35**	**74.53**
200	**1090.30**	**993.36**	**14193.71**	**151.95**
300	**1646.30**	**1558.05**	**30275.55**	**243.35**
400	**2213.99**	**2195.39**	**58740.25**	**281.32**
500	**2737.40**	**2742.05**	**89643.52**	**336.20**
800	**4437.51**	**4453.10**	**259066.35**	**601.53**
1000	**5565.54**	**5775.69**	**199100.80**	**796.06**

As shown in Table [1-2], the performance of Rapid-SPN improves drastically as compared to LearnSPN. We observe the higher LL and Time improvement in Rapid-SPN in our results.

With regards to Time, LearnSPN implementation used in our experiment (Alejandro Molina, Antonio Vergari and et al 2019) uses a high amount of parallelization especially while creating product nodes. Moreover, as shown in Table[1-2], it should be noted that as the no of features increase, the time required for LearnSPN increase exponentially whereas the time required for Rapid-SPN increase linearly.

PCA Evaluation for CIFAR-10 and STL-10

Principal Component Analysis (PCA) has been a widely used method for handling HDLS. By reducing the dimensionality to a smaller set of components, PCA enables the effective use of models for prediction.

Table 3: **Structure Learning Results: LearnSPN when used with PCA for Cifar-10 dataset for 100 columns and 5000 samples**

No of PCA Components	LSPN-LL
10	**10.48**
20	**27.10**
40	**104.25**

Table 4: **Structure Learning Results: LearnSPN when used with PCA for STL-10 dataset for 100 columns and 5000 samples**

No of PCA Components	LSPN-LL
10	**42.64**
20	**96.72**
40	**212.90**

We evaluated the performance of Sum-Product Network (SPN) and RapidSPN models on STL-10 and CIFAR-10 datasets, with results presented in Table[3-4]. Furthermore, we compared log-likelihood values from RapidSPN Table[1-2]. RapidSPN yielded significantly higher log-likelihood values than PCA-generated components.

MSE value of CIFAR-10

We conducted an additional experiment, applying Principal Component Analysis (PCA) for dimensionality reduction, retaining 40 components. This setup mirrored the CIFAR-10 HDLS configuration (100 columns) except we don't use K-Fold here, instead we split our dataset by keeping test size at 25 percent. Notably,

Table 5: **Structure Learning Results: LearnSPN when used with PCA for CIFAR-10 dataset for 100 columns and 5000 samples**

No of PCA Components	PCA-LSPN MSE
10	**0.0017**
20	**0.0018**
40	**0.0019**

RapidSPN performed comparably to PCA-based SPN, achieving a Mean Squared Error (MSE) of 0.0026, whereas PCA yielded an MSE based on the table.

MSE value of STL-10

We conducted an additional experiment where PCA was applied for dimensionality reduction, retaining 40 components, within the same

Table 6: **Structure Learning Results: LearnSPN when used with PCA for STL-10 dataset for 100 columns and 5000 samples**

No of PCA Components	PCA-LSPN-MSE
10	**0.0058**
20	**0.0058**
40	**0.00626**

configuration as STL-10 HDLS (100 columns). Results showed RapidSPN achieved a mean score of 0.0057, whereas the PCA model yielded a Mean Square error based on the table

Normal Dataset

We evaluated Log Likelihood and computational Time on additional datasets (Table 5 and 6). These datasets, sourced from the UCI Machine Learning Repository and STATLib, were conveniently retrieved online using OPENML [1], eliminating the need for local storage and streamlining our codebase. Here, Our primary

Table 7: **Structure Learning Results for lower dimension results: LEARNSPN (LSPN) VS Rapid-SPN (Rapid-SPN) LL for K=10.** The value after + represent average standard deviation and the value before + represents average LL for 10 splits of cross validation.

DATASET	LSPN-LL	Rapid-SPN-LL
Iris[N](UCI 1987)	**8.25±0.41**	8.25±0.59
Balance-Scale[N](UCI 1987)	1.46±0.13	**1.68±0.2**
liver-disorder[N](UCI 1987)	5.72±0.38	**5.69±0.42**
blood-transfusion-service[N](UCI 1987)	26.89±16.83	**36.81±0.82**
Ecoli[N](Horton 1996)	**8.97±1.92**	8.86±0.87
Hayes-Roth[N](UCI 1987)	4.10±2.97	**25.98±10.63**
tic-tac-toe[C](UCI 1987)	**-9.6±0.08**	-9.70±0.08
car[C](ASA 1989)	**-7.48±0.02**	-7.55±0.01
molecular-biology_promoters[C](UCI 1987)	**-80.19±2.17**	-80.39±0.90
dna[C](UCI 1987)	**-95.43±0.33**	-100.55±0.19
kr-vs-k[C](KEEL 1993)	**-10.51±0.06**	-10.62±0.01
wine[N](UCI 1987)	**65.63±2.51**	29.59±2.55
diabetes[N]	14.86±0.34	**20.32±2.44**
phoneme[N](KEEL 1993)	0.75±0.17	**1.33±0.12**

Table 8: **Structure Learning Results: LEARN-SPN(LSPN) VS Rapid-SPN Time for K=10.** The value after + represent average standard deviation and the value before + represents average time for 10 splits of cross-validation.It can be seen that FSPN is lightning fast in terms of time

DATASET	LSPN-Time	Rapid-SPN Time
Iris[N]	0.34±0.03	**0.03±0.02**
Balance-Scale[N]	1.29±0,03	**0.11±0.01**
liver-disorders[N]	0.63±0.06	**0.05±0.004**
blood-transfusion-service[N]	3.80±0.25	**0.50±0.04**
Ecoli[N]	6.14±0.62	**0.14±0.02**
Hayes-Roth[N]	2.75±0.23	**0.63±0.12**
tic-tac-toe[C]	0.72±0.02	**0.02±0.001**
car[C]	1.11±0.22	**0.05±0.06**
molecular-biology_promoters[C]	29.89±8.00	**0.22±0.03**
dna[C]	208.32±20.72	**23.88±7.26**
kr-vs-k[C]	**3.06±0.69**	11.92±0.15
wine[N]	1.63±0.17	**0.09±0.009**
diabetes[N]	9.31±1.77	**1.21±0.16**
phoneme[N]	11.54±0.51	**3.61±0.05**

objective is to demonstrate that our model performs effectively on normal datasets. To achieve this, we selected 12 diverse datasets for evaluation, listed in Table [5-6]. Notably, Promoter-Gene and DNA datasets are HDLS, which will be discussed separately. For all 14 datasets, including Promoter-Gene and DNA, we employed 10-fold cross-validation.

For LearnSPN, we use instance-slice $I = [10, 20, 40, 50]$ and threshold $T = [0.4]$ as threshold for independent testing. For Rapid-SPN, we use

$H = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]$ and $L = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]$.

In Preprocessing steps, we employ 10-Fold Cross Validation for the dataset.

In these experiments, we find that in most of the datasets, the time required for Rapid-SPN is very less as compared to Learn-SPN.We perform average in terms of LL, when compared with LearnSPN.

DNA and Promoter Gene Dataset

For Promoter-Gene datasets, characterized as High-Dimensional Low-Sample (HDLS) datasets, the training set dimensions are 95x57. Additionally, we evaluated DNA performance with 800x180 dimensions using K-fold cross-validation (Table 7), showcasing HDLS effectiveness with -99.98 LL outperforming standard DNA version, which confirms that Random Projection Principle is under effect. Previously explored in DNA datasets performance

Table 9: **Structure Learning Results: LEARNSPN (LSPN) VS Rapid-SPN LL and Time(sec) for DNA dataset**

Fold No	LSPN-LL	Rapid-SPN-LL	LSPN-TIME(sec)	Rapid-SPN TIME(sec)
1	**-99.19**	**-99.59**	**120.53**	**6.752**
2	**-94.93**	**-99.82**	**118.30**	**7.49**
3	**-95.17**	**-100.30**	**119.50**	**5.96**
4	**-94.25**	**-101.11**	**119.58**	**5.42**
5	**-99.73**	**-100.55**	**124.87**	**5.55**
6	**-95.04**	**-99.25**	**121.49**	**5.36**
7	**-97.84**	**-100.16**	**117.73**	**7.41**
8	**-98.13**	**-99.52**	**118.67**	**6.22**
9	**-94.75**	**-99.11**	**121.76**	**6.23**
10	**-94.37**	**-100.35**	**120.63**	**5.49**

(Robert Peharz, Antonio Vergari and et al 2019), under HDLS settings remained uninvestigated. Notably, our results indicate Sum Product Networks (SPNs) outperform RapidSPNs in these datasets. However, LearnSPN's efficiency is hindered by product node generation, leading to prolonged computation times. Evaluating CIFAR-10 and STL-10 datasets revealed this limitation. Despite LearnSPN's HDLS suitability, RapidSPN is preferable when prioritizing computational efficiency and time efficiency. Fortunately, DNA and Promoter-Gene datasets' smaller sizes enabled 10-fold cross-validation. Conversely, CIFAR-10 and STL-10 datasets posed computational challenges due to larger sizes. Results for the generated DNA samples are presented in Appendix 1 Table 8.

PARALLELIZATION OF Rapid-SPN

The Rapid-SPN can include multiple level parallelization. In this literature and for our experiment, we did not perform any kind of parallelization. However, in this section we extend the idea of parallelization in Rapid-SPN, example Figure[6]. The structure of Rapid-SPN

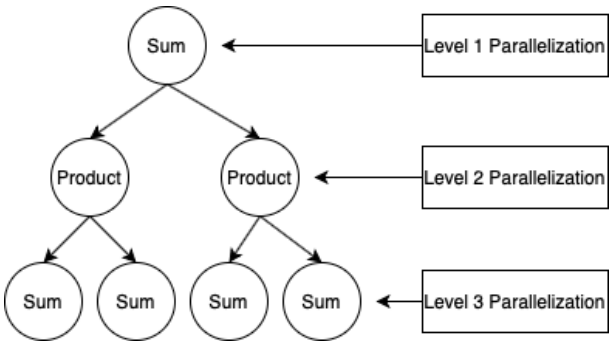

Figure 6: **Rapid-SPN PARALLELIZATION** The Rapid-SPN can be produced in parallel from level h_i and merged into level $_{i-1}$

is similar to Binary Tree. As the height increases, level of parallelization also increase. This is because L_{i-1} can be first performed linearly after which the remaining levels can be performed in parallel. Once the results are received from parallelization, the nodes can be attached to the parent tree structure of L_{i-1}. For example, if we perform parallelization after level 2, then we can create 2 separate threads or processes . Finally, we attach the nodes to level 1. However, this is only especially advantageous when the height is high. Because, the each unit cost (sum node, product node and leaf node) operation of Rapid-SPN is low. In this case the total cost operation is $O(loghmdn/P) + c$ where P is the amount of parallelization and c is a constant cost incurred due to building the final-rooted SPN by therom 4.1. Therefore, parallelization can greatly boost the performance of Rapid-SPN.

Thus, Rapid-SPN using RP-Tree builds a lighting fast algorithm with improved LogLikelihood.

CONCLUSION AND FUTURE WORK

By integrating Sum-Product Networks (SPNs) with Random Projection (RP)-Trees, we developed a robust model that effectively learns from limited data points, overcoming High-Dimensional Low-Sample (HDLS) dataset challenges. However, addressing the curse of dimensionality remains a significant challenge, particularly in bioinformatics and medicine, where datasets often have numerous features but scarce data points. Our contribution marks a crucial step forward.

Future research directions include:

1. Developing a discriminative Rapid-SPN version (Figure 7) to enhance classification performance. 2. Exploring alternative RP-tree algorithms for optimized Rapid-SPN implementation.

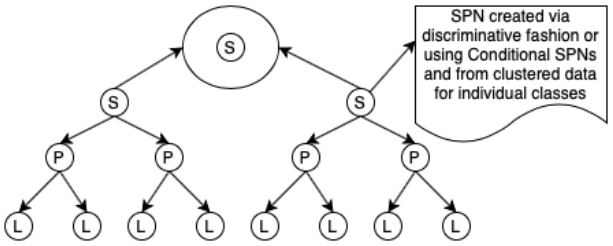

Figure 7: **Discriminative Rapid-SPN.** The Rapid-SPN created by discriminative fashion (DSPN) or may be by using Conditional SPN.

Acknowledgement

We would like to extend our sincere appreciation to the European Research community for granting us free access to their computational resources. We are also grateful to Dr. Kristian for suggesting the innovative approach of integrating Random Projection Trees (RP-Trees) with Sum-Product Networks (SPN). Furthermore, we would like to thank Dr. Doshi and Dr. Kristian for their teaching of SPN, as well as Fabrizio for his assistance in executing the program on the cluster. Additionally, we would like to thank the reviewers for their constructive feedback and suggested changes, which have improved the quality of our paper.

References

Aaron Dennis, Dan Ventura. 2015. Greedy Structure Search for Sum-Product Networks. In *Greedy Structure Search for Sum-Product Networks*. IJCAI.

Adam Coates, Honglak Lee, Andrew Y. Ng. 2011. An Analysis of Single Layer Networks in Unsupervised Feature.

Alejandro Molina, Antonio Vergari and et al. 2019. SPFLOW: AN EASY AND EXTENSIBLE LIBRARY FOR DEEP PROBABILISTIC LEARNING USING SUM-PRODUCT NETWORKS. In *SPFLOW: AN EASY AND EXTENSIBLE LIBRARY FOR DEEP PROBABILISTIC LEARNING USING SUM-PRODUCT NETWORKS*. arxiv.

Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images, Tech Report.

Amirmohammad Rooshenas, Daniel Lowd. 2014. Learning Sum-Product Networks with Direct and Indirect Variable Interactions. In *Learning Sum-Product Networks with Direct and Indirect Variable Interactions*. ICML.

ASA. 1989. American Stastical Association.

Brian McFee, G. L. 2011. Large-scale music similarity search with spatial trees. In *Large-scale music similarity search with spatial trees*. ISMLR.

Diarmaid Conaty, Jesús Martínez Del Rincon and et al. 2018. Cascading Sum-Product Networks using Robustness. In *Cascading Sum-Product Networks using Robustness*. PGM.

Dmitriy Fradkin, David Madigan. 2002. Experiments with Random Projections for Machine Learning. In *Experiments with Random Projections for Machine Learning*. ACM.

Han Zhao, Mazen Melibari, Pascal Poupart. 2015. On the Relationship between Sum-Product Networks and Bayesian Networks. In *On the Relationship between Sum-Product Networks and Bayesian Networks*. ICML.

Hoifung Poon, Pedro Domingos. 2009. Sum-Product Networks: A New Deep Architecture. In *Sum-Product Networks: A New Deep Architecture*. arxiv.

Horton, P. 1996. Protein Localization Sites.

Jinghua Wang, Gang Wang. 2016. Hierarchical Spatial Sum-Product Networks for Action Recognition in Still Images. In *Hierarchical Spatial Sum-Product Networks for Action Recognition in Still Images*. IEEE Transactions on Circuits and Systems for Video Technology.

KEEL. 1993. KEEL.

Mattia Desana, Christoph Schnorr. 2017. Sum-Product Graphical Models. In *Sum-Product Graphical Models*. arxiv.

Nicola Di Mauro, Floriana Esposito, and et al. 2018. Sum-Product Network structure learning by efficient product nodes discovery. In *Sum-Product Network structure learning by efficient product nodes discovery*. IOS Press.

Nicola Di Mauro, Gennaro Gala, Marco Iannotta, Teresa M.A. Basile. 2021. Random Probabilistic Circuts. In *Random Probabilistic Circuts*. UAI.

Robert Gens, Pedro Domingos. 2012. Discriminative Learning of Sum-Product Networks. In *Discriminative Learning of Sum-Product Networks*. NeurIPS.

Robert Gens, Pedro Domingos. 2013. Learning the Structure of Sum-Product Networks. In *Learning the Structure of Sum-Product Networks*. JMLR.

Robert Peharz, Antonio Vergari and et al. 2019. Random Sum-Product Networks:A Simple and Effective Approach to Probabilistic Deep Learning. In *Random Sum-Product Networks:A Simple and Effective Approach to Probabilistic Deep Learning*. UAI.

Robert Peharz, Robert Gens and et al. 2016. On the Latent Variable Interpretation in Sum-Product Networks. IEEE Transaction of Pattern Recognition.

Sanjoy DasGupta, Yoav Ferund. 2008. Random projection trees and low dimensional manifolds. In *Random projection trees and low dimensional manifolds*. STOC.

UCI. 1987. UCI Machine Learning Repository.

Xiaoting Shao, Alejandro Molina and et al. 2019. Conditional Sum-Product Networks: Imposing Structure on Deep Probabilistic Architectures. In *Conditional Sum-Product Networks: Imposing Structure on Deep Probabilistic Architectures*. arxiv.

Yann LeCun, Patrick Haffner, Leon Bottou Yoshua Bengio. 1998. Object Recognition with Gradient based Learning. In *Object Recognition with Gradient based Learning*. ATT.

Appendix

Table 10: **first 29 character of DNA-RapidSPN(HDLS)**

A	C	T	G	A	C	T	G	A	C	T	G	A	C	T
1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0	0.0	1.0	1.0	0.0	1.0	0.0	1.0	1.0	0.0	0.0	1.0
1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	1.0	0.0	0.0	0.0	1.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0
0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0
1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	1.0	0.0	1.0
0.0	1.0	0.0	0.0	1.0	0.0	1.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	1.0
0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	1.0	0.0	1.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0
0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0
0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
1.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	1.0
0.0	0.0	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
0.0	1.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0
0.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0
0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0
0.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0
0.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	1.0	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
1.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	1.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0
1.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
0.0	0.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0
0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0
0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0
1.0	0.0	0.0	1.0	0.0	0.0	1.0	1.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0
0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	1.0	1.0	0.0	0.0
0.0	0.0	1.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0
0.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0
1.0	0.0	0.0	0.0	0.0	1.0	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0
0.0	1.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0
0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0
0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0
0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	1.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0
0.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	1.0	0.0	0.0	1.0	1.0
0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0