THE POON-DOMINGOS PARAMETER LEARNING ALGORITHM FOR IMAGE COMPLETION AND CLASSIFICATION ON SUM-PRODUCT NETWORKS

Renato Lui Geh
Computer Science
Institute of Mathematics and Statistics
University of São Paulo
renatolg@ime.usp.br

Abstract. In this document we describe the Poon-Domingos [PD11] parameter learning algorithm for image classification and completion.

1. Structure

The Poon-Domingos algorithm uses a fixed structure and then learns the weights through generative learning. We first give an overview on how to build the structure given an image and then provide a pseudo-code algorithm for building such structure.

1.1. Overview

The Poon architecture models a probability distribution over a set of images. It is constructed by taking all possible rectangular axis-aligned regions in the image and assigning product nodes to each of these regions. Two sum nodes are then added as children for each of these regions, representing all the possible pairings of subregions in each region determined by the axis-aligned division set in the previous step. We then add the product nodes to a single sum node that represents the undivided original area. We then recursively apply the same steps on each sum node we constructed this way, taking that sum node as the new root of the sub-SPN. The Poon structure accepts different multiple resolution levels. At every region splitting, we consider a step r that indicates how fine the granularity is for the architecture.

1.2. Definitions and properties

Let us organize in a clearer way what we have extracted from [DV12].

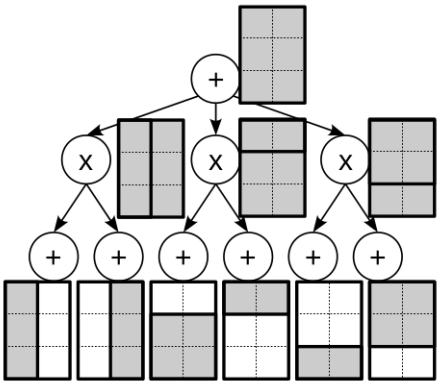

FIGURE 1. The Poon architecture with $r = 1$ resolution. At each product node division, we consider an r length division on each axis. Since $r = 1$ in this case, we have 3 product nodes, with each of their two possible subregions represented by sum nodes. Note how product nodes are always decomposable and sum nodes are always complete.

Definition 1.1 (Region)**.** *A region π is a product node. Graphically, it represents an axis-aligned rectangular region of the image. We shall denote an SPN $S(\cdot)$ rooted at a region as $\pi(\cdot)$.*

Definition 1.2 (Subregion)**.** *A subregion σ is a sum node. Given a region R, the children of R are the two possible rectangles that compose R. We shall denote an SPN rooted at a subregion as $\sigma(\cdot)$.*

The idea behind the Poon structure algorithm is to recursively take a rectangular subarea of an image and divide it into all k possible regions given a granularity r, with each region having two children (subregions) that represent the possible pairings of each region. We then recurse through each of these subregions.

Let $I(x_0, y_0, x_1, y_1)$ be the area of the image we are to create the structure for, with (x_0, y_0) and (x_1, y_1) being the top left and bottom right positions in the image. The entire image is given by $I(0, 0, w, h)$, where w, h are the width and height of the image respectively.

A subregion σ implicitly represents an area $I(x_0, y_0, x_1, y_1)$, whilst each region of σ is a possible subdivision of I by drawing a line, either horizontally or vertically, through one of the axis of I, partitioning it into two subregions.

We can clearly see that, for a subregion σ, there are $x_1 - x_0$ possible vertically divided regions and $y_1 - y_0$ horizontally divided regions, bringing the total to $(x_1 + y_1) - (x_0 + y_0)$ possible product nodes as children of σ for $r = 1$. For the general case, we can clearly see that we have $\lceil (x_1 - x_0)/r \rceil + \lceil (y_1 - y_0)/r \rceil$ possible regions.

1.3. Algorithm

The structure algorithm is recursive. It takes as parameters a sum node S as root of the SPN, the (x_0, y_0) top-left position of the subregion S relative to the original complete image, the (x_1, y_1) bottom-right position of the subregion S, the resolution granularity step k and a dataset $\mathcal{D}$ where $\mathcal{D}[X]$ is the set of instances of variable X.

Let's define a few helper functions before we see the algorithm. Since we are dealing with an image divided into $k \times k$ regions, we must assume that a rectangular $k \times k$ collection of pixels is the atomic unit for our leaves. From the definition of SPNs, a leaf of an SPN is a univariate distribution. Therefore, we must consider our leaves as a univariate distribution over a set of pixels of an image region. Before we go into how we're going to deal with leaves, let's define what the surface area of a rectangular region is given a k resolution.

Definition 1.3 (Surface area)**.** *The surface area of an image given a resolution k is the surface area divided by k. Let S be the regular surface area. We'll use the following notation $S^k = S/k$.*

Since we're mainly dealing with rectangular regions, we can use the following function:

Algorithm 1 `Area`(p, q, k)

Input Top-left position $p = (x, y)$.
Input Bottom-right position $q = (x, y)$.
Input Resolution k.
Output S^k
1: **return** $\lceil \frac{(p.x - q.x) \cdot (p.y - q.y)}{k} \rceil$

We are now going to make a strong assumption about our images. Let w and h be the width and height of all images respectively. We're going to assume, to simplify our algorithm, that w and h are both multiples of k. Having said that, we have w/k and h/k region partitions on the x-axis and y-axis respectively. Had we considered the general case in which we have any $w, h \in \mathbb{Z}$, we would then have $w \mod k$ and $h \mod k$ left-over corner case regions we would have to take into account.

Now that we have a function `Area` to compute the surface area of a region given a resolution k, we can start thinking of how we're going to build the algorithm. From the overview section, we saw that we need to keep decomposing our image until we hit a base case, in which case we would then create a leaf and build a univariate distribution over the pixels in that base case region. But how do we decide when we have reached that point? Well, since a region can be no bigger than a $k \times k$ region, then it is reasonable to assume that our atomic unit is the $k \times k$ square.

From this assumption, we can consider the following function that takes a rectangular region's top-left and bottom-right positions and dataset $\mathcal{D}$:

Algorithm 2 `NewLeaf`$(p, q, \mathcal{D})$

Input Top-left position $p = (x, y)$
Input Bottom-right position $q = (x, y)$
Input Resolution k
Input Dataset $\mathcal{D}$ containing instances of variable set $\mathbf{X}$
Output A univariate distribution over pixels in the rectangular region (p, q)

```
1: 𝒳 = {X ∈ X | X is inside the rectangular region (p, q)}
2: Let p be an array containing the frequencies of each possible variable instantiation where p[i] is equivalent to the frequency of all X ∈ 𝒳 such that X = i
3: for each instantiation with X = x ∈ 𝒟 such that X is also in 𝒳 do
4:     Increment p[x]
5: end for
6: return Leaf(p, |𝒳|)
```

Note that we are compressing a rectangular region $k \times k$ into a single k-sized pixel. When $k = 1$, `NewLeaf` will create a univariate distribution over each pixel, taking into account all instantiations in the dataset $\mathcal{D}$.

We now have the base case figured. Generating the rest of the structure is straightforward. For each axis, we recurse over the k separated regions, creating two sum nodes for the subregion pairings. If we were to implement this method without much thinking, we would end up creating redundancies that could impact performance. Consider the following cases:

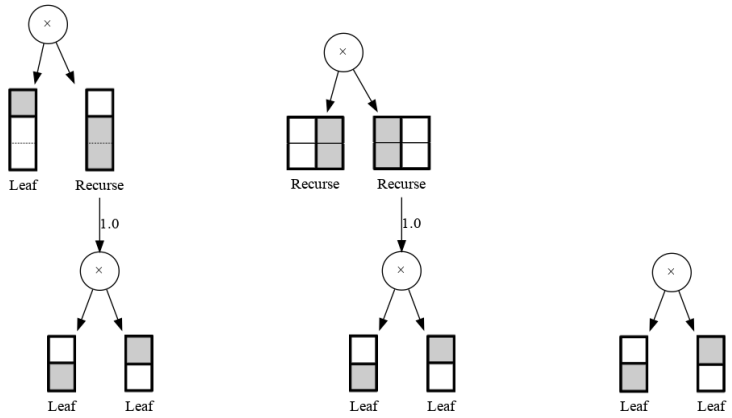

FIGURE 2. Special cases for the y-axis. Note how in both first cases the newly created sum node on the right has only one edge with weight 1. This sum node is redundant and the SPN can be simplified by removing the sum node and adding its children as children of the parent product node. The far right case occurs when there are only two k regions on the y-axis.

These sub-SPNs showcase the three possible situations in which we may create a leaf node. Although they refer to the y-axis, we can easily conclude the same for the x-axis, as Figure 3 illustrates. Note how in Figure 2a and Figure 2c, after we recurse through the 1×2 k-sized rectangles, we end up with a sum node that has only one child. This creates a redundancy, as the value of the sum node would be the same as replacing the sum node in question with its only child. We take this case into account in our algorithm, watching for 2×1 rectangles for the x-axis and 1×2 rectangles for the y-axis.

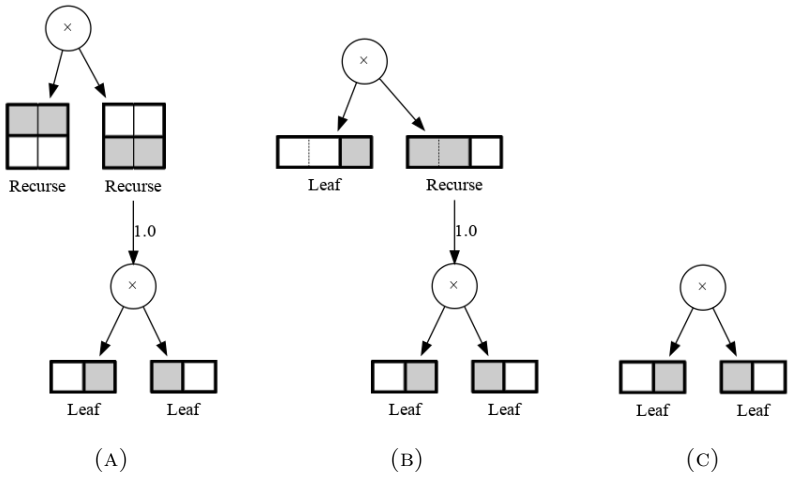

FIGURE 3. Special cases for the x-axis. Just like the y-axis, there is a redundant sum node with weight 1 that can be removed and the SPN simplified in the two first cases. The third case illustrates the base case when there are only two k-sized rectangles placed horizontally.

Now that we have our special base cases figured out, we can start building our algorithm. We will divide our algorithm into two parts. In one we iterate through all possible k-resolution regions that are x-aligned. For each of these regions, we check for the special cases mentioned above. If none of those occur in our current region, then we recurse. In the other we do the same, but with the y-axis. Each of these two parts should return two product nodes that can be interpreted as a partitioning of the image into each axis regions. We then create a sum node and add the two resulting product nodes as children. This sum node is the SPN of the image.

Note how this algorithm always creates two children for sum nodes and a decreasing number of children for product nodes at each layer. Furthermore, the number of layers of the SPN is fixed according to the size of the image. We also pre-initialized the weights of sum nodes to a uniform distribution.

Algorithm 3 `GenerateDenseSPN`$(k, p_0, p_1, \mathcal{D})$

Input Region resolution k
Input Top-left position $p_0 = (x, y)$ of the underlying image area
Input Bottom-right position $p_1 = (x, y)$ of the underlying image area
Input Dataset $\mathcal{D}$ containing instances of variable set $\mathbf{X}$
Output A dense SPN structure created from dataset $\mathcal{D}$

```
n ← ⌈(p1.x − p0.x)/k⌉
m ← ⌈(p1.y − p0.y)/k⌉
S ← NewSum()
q ← (p0.x, p1.y)
if n > 1 then
    for i ← 1 to n do
        q ← (min{p1.x, q.x + k}, p1.y)
        r ← (q.x, p0.y)
        Π ← NewProduct()
        Let c1 and c2 be children of Π
        a1 ← Area(p0, q, k)
        a2 ← Area(r, p1, k)
        if a1 = 1 then
            c1 ← NewLeaf(p0, q, D)
        else if a1 = 2 then
            c1 ← NewProduct()
            ĉ1 ← NewLeaf(p0, p0 + (k, k), D)
            ĉ2 ← NewLeaf(p0 + (k, 0), q, D)
            c1.AddChildren(ĉ1, ĉ2)
        else
            c1 ← GenerateDenseSPN(k, p0, q, D)
        end if
        Π.AddChild(c1)
        if a2 = 1 then
            c2 ← NewLeaf(r, p1, D)
        else if a2 = 2 then
            c2 ← NewProduct()
            ĉ1 ← NewLeaf(r, r + (k, k), D)
            ĉ2 ← NewLeaf(r + (k, 0), p1, D)
            c2.AddChildren(ĉ1, ĉ2)
        else
            c2 ← GenerateDenseSPN(k, r, p1, D)
        end if
        Π.AddChild(c2)
        S.AddChild(Π, 1/n)
    end for
end if
```

`GenerateDenseSPN` (continued)

$q \leftarrow (p_1.x, p_0.y)$
if $m > 1$ **then**
  for $j \leftarrow 1$ to m **do**
    $q \leftarrow (p_1.x, \min\{p_1.y, q.y + k\})$
    $r \leftarrow (p_0.x, q.y)$
    $\Pi \leftarrow$ `NewProduct`()
    Let c_1 and c_2 be children of Π
    $a_1 \leftarrow$ `Area`(p_0, q, k)
    $a_2 \leftarrow$ `Area`(r, p_1, k)
    if $a_1 = 1$ **then**
      $c_1 \leftarrow$ `NewLeaf`$(p_0, q, \mathcal{D})$
    else if $a_1 = 2$ **then**
      $c_1 \leftarrow$ `NewProduct`()
      $\hat{c}_1 \leftarrow$ `NewLeaf`$(p_0, p_0 + (k, k), \mathcal{D})$
      $\hat{c}_2 \leftarrow$ `NewLeaf`$(p_0 + (0, k), q, \mathcal{D})$
      c_1`.AddChildren`$(\hat{c}_1, \hat{c}_2)$
    else
      $c_1 \leftarrow$ `GenerateDenseSPN`$(k, p_0, q, \mathcal{D})$
    end if
    Π`.AddChild`(c_1)
    if $a_2 = 1$ **then**
      $c_2 \leftarrow$ `NewLeaf`$(r, p_1, \mathcal{D})$
    else if $a_2 = 2$ **then**
      $c_2 \leftarrow$ `NewProduct`()
      $\hat{c}_1 \leftarrow$ `NewLeaf`$(r, r + (k, k), \mathcal{D})$
      $\hat{c}_2 \leftarrow$ `NewLeaf`$(r + (0, k), p_1, \mathcal{D})$
      c_2`.AddChildren`$(\hat{c}_1, \hat{c}_2)$
    else
      $c_2 \leftarrow$ `GenerateDenseSPN`$(k, r, p_1, \mathcal{D})$
    end if
    Π`.AddChild`(c_2)
    S`.AddChild`$(\Pi, 1/m)$
  end for
end if
return S

The actual code is available at `https://github.com/RenatoGeh/gospn`.

References

[DV12] Aaron Dennis and Dan Ventura. "Learning the Architecture of Sum-Product Networks Using Clustering on Variables". In: *Advances in Neural Information Processing Systems* 25 (2012).

[PD11] Hoifung Poon and Pedro Domingos. "Sum-Product Networks: A New Deep Architecture". In: *Uncertainty in Artificial Intelligence* 27 (2011).