Complexity-Regularized Tree-Structured Partition for Mutual Information Estimation

Maximiliano Rojas
Algorithm Review

1 Classes Overview

In the algorithm[1] two classes are used:

- TSPNode: represents a node (space partition) in the tree. From now on, every "node" mentioned will be a short-name to an object of this class.
- TSP: represents a tree (set of nodes) and the values associated with the MI estimation problem, such as empirical MI, regularized MI and tree size.

2 TSPNode

2.1 Class Instance Variables

Let T be a binary tree with one node of degree 2 (root node), and the remaining nodes of degree 3 (internal nodes) or degree 1 (leaf nodes). Let $v \in \mathcal{I}(T)$ denote an arbitrary internal node and $U_v = A_1 \times A_2$ the respective partition ($A_1 \in \mathbb{R}^p, A_2 \in \mathbb{R}^q$). Inside the TSPNode class, the following instance variables are defined:

- node left: left children of a given node ($l(v)$).
- node right: right children of a given node ($r(v)$).
- node parent: parent of a given node ($\hat{v} \in \mathcal{I}(T) : v \in \{l(\hat{v}), r(\hat{v})\}$).
- float condJointDist: empirical conditional joint distribution of a given node.

$$\tilde{P}_n(U_v) = \frac{P_n(U_v)}{P_n(U_{\hat{v}})}$$

- float condMargProd: empirical product of conditional marginal distributions of a given node.

$$\tilde{Q}_n(U_v) = \frac{Q_n(U_v)}{Q_n(U_{\hat{v}})}$$

  where

$$Q_n(U_v) = P_n(A_1 \times \mathbb{R}^q) \cdot P_n(\mathbb{R}^q \times A_2)$$

- float relativeCMIgain: conditional mutual information gained by partitioning a given node into both their childs, relative to the node. *See equation (44) of the paper.*

$$\Delta\rho(v \mid U_v) = \tilde{P}_n(U_{l(v)}) \log_2 \left(\frac{\tilde{P}_n(U_{l(v)})}{\tilde{Q}_n(U_{l(v)})} \right) + \tilde{P}_n(U_{r(v)}) \log_2 \left(\frac{\tilde{P}_n(U_{r(v)})}{\tilde{Q}_n(U_{r(v)})} \right)$$

[1] The Python packages used in this implementation are Numpy, Pandas, Matplotlib and Math.

- float absoluteCMIgain: conditional mutual information gained by partitioning a given node into both their childs, relative to the root. *See equation (45) of the paper.*

$$\Delta\rho(v \mid U_v)_{root} = \hat{P}_n(U_{l(v)}) \log_2 \left(\frac{\hat{P}_n(U_{l(v)})}{\hat{Q}_n(U_{l(v)})} \right) + \hat{P}_n(U_{r(v)}) \log_2 \left(\frac{\hat{P}_n(U_{r(v)})}{\hat{Q}_n(U_{r(v)})} \right) - \hat{P}_n(U_v) \log_2 \left(\frac{\hat{P}_n(U_v)}{\hat{Q}_n(U_v)} \right)$$

Corollary 1 shows that this expression is equivalent to

$$\Delta\rho(v \mid U_v)_{root} = \Delta\rho(v \mid U_v) \cdot \hat{P}_n(U_v)$$

where

$$\hat{P}_n(U_v) = \frac{P_n(U_v)}{P_n(U_{root})} \qquad \hat{Q}_n(U_v) = \frac{Q_n(U_v)}{Q_n(U_{root})}$$

- int n_samples: sample number of the current partition node.

$$|U_v| = \sum_{i=1}^{n} \mathbb{1}_{U_v}(Z_{(i)})$$

where n is the total sample number of the dataset and $Z_{(i)}$ is the $i-nth$ sample.

- int n_marginal_samples_X: sample number of the marginal in X in the given partition node.

$$|\mathbb{R}^p \times A_2|$$

- int n_marginal_samples_Y: sample number of the marginal in Y in the given partition node.

$$|A_1 \times \mathbb{R}^q|$$

- list idx_marginal_samples_X: indices of the samples present in the node partition over the marginal of $\mathcal{X}$.

$$\{i \in \{1, ..., n\} : Z_{(i)} \in (\mathbb{R}^p \times A_2)\}$$

- list idx_marginal_samples_Y: indices of the samples present in the node partition over the marginal of $\mathcal{Y}$.

$$\{i \in \{1, ..., n\} : Z_{(i)} \in (A_1 \times \mathbb{R}^q)\}$$

- list lowerBounds: list of the lower bounds for each dimension of the partition node $(l_1, l_2, ..., l_d)$.
- list upperBounds: list of the upper bounds for each dimension of the partition node $(u_1, u_2, ..., u_d)$.

where $U_v = \bigtimes_{i=1}^{d} [l_i, h_i]$

- list partitions: list of triples, where each triple represents the lower bound, upper bound, and dimension of a specific partition.

2.2 Class Methods

The methods that are used in the ***TSPNode*** class are the following:

- void **Constructor**: creates an empty node.

  (i) parent: parent node.

- node **grow**: recursively creates the children (subpartitions) of every node (partition) as long as the minimum critical mass per node is respected. Its parameters are the following:

  (i) node parent: parent node.

  (ii) list nodeIdx: indices of the samples present in the node.

$$\{i \in \{1, ..., n\} : Z_{(i)} \in U_v\}$$

  (iii) array data: matrix with the initial sample points.

$$\text{data} = \begin{bmatrix} Z_{(1)} \\ Z_{(2)} \\ \vdots \\ Z_{(n)} \end{bmatrix} = \begin{bmatrix} X_{(1)} & Y_{(1)} \\ X_{(2)} & Y_{(2)} \\ \vdots & \vdots \\ X_{(n)} & Y_{(n)} \end{bmatrix} = \begin{bmatrix} x_{1_{(1)}} & x_{2_{(1)}} & \cdots & x_{p_{(1)}} & y_{1_{(1)}} & \cdots & y_{q_{(1)}} \\ x_{1_{(2)}} & x_{2_{(2)}} & \cdots & x_{p_{(2)}} & y_{1_{(2)}} & \cdots & y_{q_{(2)}} \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ x_{1_{(n)}} & x_{2_{(n)}} & \cdots & x_{p_{(n)}} & y_{1_{(n)}} & \cdots & y_{q_{(n)}} \end{bmatrix}$$

  (iv) list lowerBounds: list of the lower bounds $(l_1, l_2, ..., l_d)$, for every dimension, of the partition node.

  (v) list upperBounds: list of the upper bounds, for every dimension, of the partition node $(u_1, u_2, ..., u_d)$.

  (vi) list dimX_indicator: boolean list whose first p elements are $True$ (dimensions of $\mathcal{X}$) and the remaining q elements are $False$ (dimensions of $\mathcal{Y}$). Its sole purpose is to indicate if a given dimension came from the vector $\mathcal{X}$ or the vector $\mathcal{Y}$.

  (vii) int dim: dataset dimensionality $(p + q = d)$

  (viii) float kn: stopping value of the recursion, which represents the minimum samples per node. *See equation (8) of the paper.*

  (ix) int projDim: determines the dimension which is going to be partitioned to split the node into both children in the current iteration.

  Firstly the algorithm creates a basic node which represents the whole sample space (let it be the node $\hat{v}$). Next, if the first partition of the TSP is going to take place, the marginal samples variables are defined by the parameters of the method **grow** (by the fact that the first node doesn't have a parent, it parent is himself).

```
def grow(self, parent, nodeIdx, data, lowerBounds, upperBounds, ...):

    node = TSPNode(parent)

    if projDim == 0:
        node.parent = node
        node.idx_marginal_samples_X = nodeIdx.copy()
        node.idx_marginal_samples_Y = nodeIdx.copy()
        node.n_marginal_samples_X = len(nodeIdx)
        node.n_marginal_samples_Y = len(nodeIdx)
```

The bounds are the ones provided by the method, the number of samples in the node are the amount of indices inside *nodeIdx*, the conditional (empirical) joint distribution is the ratio between the sample number of the current node and it parent, and the conditional (empirical) product of the marginals is obtained vía the method **conditionalMarginalProd**.

```
node.lowerBounds = lowerBounds
node.upperBounds = upperBounds
node.n_samples = len(nodeIdx)
node.condJointDist = node.n_samples / node.parent.n_samples
node.condMargProd = node.conditionalMarginalProd(data, lowerBounds, ...)
```

To initiate the space partitioning process, it is essential first to verify whether the node meets the critical mass criterion ($|U_{\hat{v}}| \geq 2 \cdot k_n$). Initially, the values of the dimension being partitioned for all sample points within the node are stored in a variable named *projectedData*. Subsequently, the indices of these projected points are sorted in ascending order. These indices are then mapped back to their original positions in the data array, enabling the identification of the median point's index in the original data array."

```
if node.n_samples // 2 >= kn:

    # Sample points projection into the target dimension
    projectedData = data[nodeIdx, projDim % dim]


    # Indices that sort the node samples
    idx = np.argsort(projectedData)


    # Map sorted indices back to the original array's indices
    nodeIdxSorted = np.array(nodeIdx)[idx]


    # Index of projectedData's median
    medianIdx = len(nodeIdxSorted) // 2
```

Using the median index, the node sample indices can be easily divided into two groups: those greater than the median and those less. Indices whose points are greater than the median are stored in a list representing the left child's indices, while indices whose points less than the median are saved in another list representing the right child's indices. With these two lists, the sample points for both the left and right children are stored in two new lists.

```
    # Space partition indices
    leftNodeIdx = nodeIdxSorted[:medianIdx]

    rightNodeIdx = nodeIdxSorted[medianIdx:]
```

```
# Space partition samples
leftNode = data[leftNodeIdx]

rightNode = data[rightNodeIdx]
```

To delineate the subspaces, the maximum value of the left node and the minimum value of the right node within the current dimension are averaged. This average determines the upper bound of the left node and the lower bound of the right node, setting the mean value as the boundary. These bounds, together with the partitioned dimension, are then recorded.

```
# Limits of each partition
left_max = max(leftNode[:, projDim % dim])
right_min = min(rightNode[:, projDim % dim])
mean = (left_max + right_min) / 2


# Partitions
leftUpperBounds = np.copy(upperBounds)
leftUpperBounds[projDim % dim] = mean

rightLowerBounds = np.copy(lowerBounds)
rightLowerBounds[projDim % dim] = mean


# Save partitions
node.partitions = [(projDim % dim, rightLowerBounds, ...)]
```

Recursively, the method **grow** is applied to both children, utilizing their respective node samples and bounds, and partitioning is performed on the subsequent dimension. Finally, both relative and absolute conditional mutual information gained from this partition is calculated using the method **getCMIGain** and stored in the corresponding instance variables.

```
# Node growth recursion
node.left = self.grow(node, leftNodeIdx, data,..., projDim + 1)

node.right = self.grow(node, rightNodeIdx, data,..., projDim + 1)


# Conditional mutual information gains
node.relativeCMIgain = node.getCMIgain()

node.absoluteCMIgain = node.relativeCMIgain * (node.n_samples...
```

- float **conditionalMarginalProduct**: computes the empirical product of the marginal distributions. *See equation (5) of the paper.*

  (i) array data: matrix with the initial sample points.

  (ii) list lowerBounds: list of the lower bounds $(l_1, l_2, ..., l_d)$, for every dimension, of the partition node.

  (iii) list upperBounds: list of the upper bounds, for every dimension, of the partition node $(u_1, u_2, ..., u_d)$.

  (iv) list dimX_indicator: boolean list whose first p elements are $True$ (dimensions of $\mathcal{X}$) and the remaining q elements are $False$ (dimensions of $\mathcal{Y}$). Its sole purpose is to indicate if a given dimension came from the vector $\mathcal{X}$ or the vector $\mathcal{Y}$.

  (v) int dim: dataset dimensionality $(p + q = d)$.

  (vi) float kn: stopping value of the recursion, which represents the minimum samples per node. *See equation (8) of the paper.*

  (vii) int projDim: determines the dimension which is going to be partitioned to split the node into both children in the current iteration.

  First, verify whether the initial partition of the TSP will occur; if so, the product of the marginals is 1. Subsequently, it is necessary to determine if the product of the marginals needs to be calculated in relation to the marginal of $\mathcal{X}$ or $\mathcal{Y}$. Assuming, without loss of generality, that the previous partition occurred along $\mathcal{X}$, this dimension will constitute the marginal. Consequently, the samples corresponding to $\mathcal{Y}$ will remain unchanged. Thus, the instance variable for the samples in the marginal on $\mathcal{X}$ is initialized as an empty vector, ready to be populated.

```
def conditionalMarginalProd(self, data, lower_bounds, upper_bounds, ...):
    if proj_dim == 0:
        return 1

    # Check if the previous partition was on X
    if Xdim_indicator[(proj_dim - 1) % dim]:

        # Use parent's marginal samples for Y
        self.idx_marginal_samples_Y = self.parent.idx_marginal_samples_Y
        self.n_marginal_samples_Y = self.parent.n_marginal_samples_Y


        # Initialize marginal samples for X
        self.idx_marginal_samples_X = np.zeros(...)
```

  The marginal samples for the node will be derived from those in the parent, requiring a thorough review of all the parent's marginal samples.

```
        # Iterate through the parents marginal samples
        for i in range(self.parent.n_marginal_samples_X):
```

  For each dimension, it must first be determined whether the dimension is originated from the random variable $\mathcal{X}$. Subsequently, it should be verified whether the marginal sample falls within the bounds of the marginal partition of $\mathcal{X}$. If there is a dimension in which the sample is not bounded within these limits, the sample is discarded.

```
        # Iterate through dimensions
            for d in range(dim):

                # Check if the dimension came from variable X
                if Xdim_indicator[d]:

                    # Check if the sample is not in the 'marginal partition'
                    if (data[self.parent.idx_marginal_samples_X[i], d] < ...)
                        # Sample is discarded
                        break
```

If the sample is within bounds for all dimensions, its index is saved in the list of marginal samples, and the count of the node's marginal samples increases.

```
                # Check if the dimension is the last one
                if d == dim - 1:
                    # Sample is within bounds
                    self.idx_marginal_samples_X[self.n_marginal_samples_X] = ...
                    self.n_marginal_samples_X += 1
```

Once all the marginal samples of the parent have been processed, the empirical product of the conditional marginals will be the ratio of the marginal samples of the current node to those of the parent.

```
        # Return proportion of samples
        return self.n_marginal_samples_X / self.parent.n_marginal_samples_X
```

- float **getCMIgain**: calculates the conditional mutual information gain of the current partition. *See equation (44) of the paper.*

```
def getCMIgain(self):
    left_CMIgain = self.left.condJointDist *
    math.log2(self.left.condJointDist / self.left.condMargProd)

    right_CMIgain = self.right.condJointDist *
    math.log2(self.right.condJointDist / self.right.condMargProd)

    return left_CMIgain + right_CMIgain
```

- float **getEMI**: calculates the estimated mutual information of the tree. *See equation (46) of the paper.*

```
def getEMI(self):
    if self.left is None and self.right is None:
        return 0
    else:
        return self.relativeCMIgain
               + (self.left.condJointDist * self.left.getEMI())
               + (self.right.condJointDist * self.right.getEMI())
```

- int **getSize**: calculates the number of leaves (tree zize) of the tree rooted in the current node.

```
def getSize(self):
    if self.left is None and self.right is None:
        return 1
    else:
        return self.left.getSize() + self.right.getSize()
```

- list **getPartitions**: obtains a list of triples, each of these consists of the lowerbound, upperbound and dimension of a especific partition made.

```
def getPartitions(self):
    if self.left is None and self.right is None:
        return []
    else:
        return self.partitions + self.left.getPartitions() + ...
```

3 TSP

3.1 Class Instance Variables

- float l_bn: exponent of the critical mass approximation ($b_n \approx n^{-l}, l \in (0, \frac{1}{3})$) *See equation (14) of the paper.*
- float w_bn: weighting factor of the critical mass approximation ($b_n \approx wn^{-l}, l \in (0, \frac{1}{3})$)
- float kn: stopping value of the recursion, which represents the minimum samples per node. *See equation (8) of the paper.*
- float _lambda: factor which controls the regularization process.
- node root: node representing the root of the binary tree.
- int dim: dataset dimensionality ($p + q = d$).
- int n_samples: total sample number of the dataset present in the tree.
- int tsp_size: number of leaves (size) of the tree.
- int tsp_reg_size: number of leaves (size) of the regularized tree.
- float tsp_emi: estimated mutual information.
- float tsp_reg_emi: regularized mutual information.
- list tsp_partitions: space partitions bounds.

3.2 Class Methods

- void **grow**: generates the data-driven tree-structured partitions for mutual information estimation between two random variables samples $\mathcal{X}$ and $\mathcal{Y}$ according to the growth stopping criterion.

  (i) list x: random variable $\mathcal{X}$ samples.

  (ii) list y: random variable $\mathcal{Y}$ samples.

  To initiate the TSP, the data matrix is created by concatenating the random variables $\mathcal{X}$ and $\mathcal{Y}$.

```
def grow(self, x, y):

    # Data array
    data = np.concatenate((pd.DataFrame(x).values, pd.DataFrame(y).values), ...
```

  In the data matrix, the number of rows corresponds to the total number of samples, while the number of columns indicates the dimensions in the problem.

```
    # Number of samples and dimensions of Data
    self.n_samples = data.shape[0]
    self.dim = data.shape[1]
```

  To initiate the space partitioning (growing) process, the root of the tree is established as a simple node, and the critical mass criterion for subsequent nodes is determined by the expression described in the paper.

```
    # Define a root node and determine the critical mass according to its formula
    self.root = TSPNode()
    self.kn = np.ceil(self.w_bn * pow(self.n_samples, 1 - self.l_bn))
```

  To precisely define the boundaries of each node, the bounds of the entire space are determined by identifying the lowest value sample across all dimensions for the lower bound, and similarly, the upper bound is defined by the highest value sample across all dimensions.

```
    # Initial lower and upper bounds of the whole sample space
    lowerBounds = data.min(axis=0) - 0.001
    upperBounds = data.max(axis=0) + 0.001
```

  To determine which random variable each dimension originated from, a list is defined where True indicates dimensions originating from variable $\mathcal{X}$ and False for those from $\mathcal{Y}$. Additionally, a list is created to hold the indices (which can be interpreted as identifiers) for every sample in the space. Lastly, the tree growth process begins at the root by invoking the grow method of a node object.

```
    # Two lists: one which indicates whose dimensiones came from the vector X...
    Xdim_indicator = [True] * pd.DataFrame(x).shape[1] + [False] * ...
    nodeIdx = list(range(self.n_samples))


    # Tree growth begins at the root node
    self.root = self.root.grow(None, nodeIdx, data, lowerBounds, upperBounds,...)
```

```
        # Once the tree is fully grown obtain the TSP results
        self.tsp_emi = self.root.getEMI()
        self.tsp_size = self.root.getSize()
        self.tsp_partitions = self.root.getPartitions()
```

Once the tree is fully grown, the estimated mutual information, tree size, and the partitions made are obtained through the methods of the node class and stored as instance variables.

- void **regularize**: complexity regularization, or prunning, of the tree structured partition.

  Initially, it is checked whether a tree was grown prior to applying the regularization method.

```
    def regularize(self):

        # A grown tree is required
        if self.root is None:
            raise ValueError("Observations not provided.")
```

The size of the full tree is obtained via the **getSize** method, and the parameters for the regularization term are declared according to the expressions described in the paper.

```
            # Size of the full grown tree
            full_tree_size = self.root.getSize()


            # Regularizer term that balances the cost and penalization associated ...
            bn = self.w_bn * np.power(self.n_samples, -self.l_bn)
            inv_deltan = np.exp(np.power(self.n_samples, 1 / 3.0))
```

Define an array to store the optimal EMI for each minimum cost tree, with the EMI at index k corresponding to the tree of size k. After defining the array, the method **minimum_cost_trees** is called to populate the EMI array.

```
            # Initialize array which will have the EMI for every minimum cost tree
            treesEMI = np.zeros(full_tree_size)
            self.minimum_cost_trees(treesEMI, full_tree_size)
```

Initialize the optimal cost and optimal size using the EMI from the minimum cost tree of size 1, which consists solely of the root node.

```
            # Initialize cost and tree size for the optimizacion problem
            optimal_cost = -treesEMI[0]
            optimal_size = 1
```

The optimization problem aims to minimize the cost (thereby maximizing the EMI) while maintaining an adequate tree size, which reflects the tree complexity in the penalization term. During the evaluation process, each pair of minimum cost tree EMI and size is examined. If a lower cost than the current optimal is found, this new cost becomes the new optimal, and the tree size is updated accordingly, until all trees have been evaluated.

```
        # Solve for every minimum cost tree
        for k in range(2, full_tree_size + 1):

            # Optimization problem
            epsilon = (12 / bn) * np.sqrt((8.0 / self.n_samples) * (np.log(8 * ...
            cost = -treesEMI[k-1] + self._lambda * epsilon


            # Check if the cost is less than the optimal cost
            if cost < optimal_cost:
                optimal_cost = cost
                optimal_size = k
```

Finally, once the optimization problem has been resolved, the regularized tree EMI and size are stored in the corresponding instance variables.

```
        # Once the tree is prunned obtain the TSP regularization results
        self.tsp_reg_emi = -optimal_cost
        self.tsp_reg_size = optimal_size
```

- void **minimum_cost_trees**: calculates the estimated mutual information from the minimum cost trees of sizes $k \in \{1, ..., |T_{full}|\}$, where each optimal tree of size k is obtained by the split of the leaf of maximum absolute conditional information gain of the tree of size $k-1$ according to the subbaditive penalty property.

  (i) list treesEMI: array filled with zeros that will store the EMI for every minimum cost tree.

  (ii) list full_tree_size: size of the full grown tree.

  At fist, an auxiliary array is declared, which will store the leaves (external nodes) of the minimum cost tree of size k during each iteration. The purpose of this array is to keep the leaves sorted in ascending order by their absolute CMI.

```
    def minimum_cost_trees(self, treesEMI, full_tree_size):

        # An auxiliary array that will store the leaves of each minimum cost tree ...
        # The goal is to sort these leaves in ascending order, according to their CMI
        leaves = [None] * full_tree_size
        leaves[0] = self.root
```

  To calculate the EMI for every minimum cost tree, each possible size is evaluated sequentially. Given that the leaves of the tree of size k are sorted, the leaf at index k possesses the highest CMI among all leaves. Consequently, the EMI of the tree of size $k+1$ is determined by adding the CMI of this maximum absolute CMI leaf to the EMI of the minimum cost tree of size k.

```
        # Obtain the EMI for every possible minimum cost tree
        for k in range(full_tree_size - 1):

            # Leaf of the tree of size k which has the maximum absolute CMI.
            leaf_maxCMI = leaves[k]


            # The EMI of the tree of size k+1 is calculated by adding the absolute ...
            # tree of size k to the EMI of the tree of size k
            treesEMI[k + 1] = treesEMI[k] + leaf_maxCMI.absoluteCMIgain
```

In order to get to the next minimum cost tree, the maximum absolute CMI leaf is splitted. Both child nodes are then inserted in the leaves array, with the left one replacing the parent and the right one just being appended.

```
            # Next tree is obtained by the split of the maximum absolute CMI leaf

            # The left child will sustitute the parent in the leaves array and the
            # right child will be simply appended to the leaves array
            self.subadditive_insert(leaf_maxCMI.left, leaves, k)

            self.subadditive_insert(leaf_maxCMI.right, leaves, k + 1)
```

- void **subadditive_insert**: inserts a node into the array according to an increasing order of the node's absolute conditional mutual information gain.

  (i) list new_leaf: new leaf of the minimum cost tree.

  (ii) list leaves: array with the leaves of the minimum cost tree.

  (iii) list j: parameter that specifies the position for inserting the new leaf in the leaves array.

```
    def subadditive_insert(self, new_leaf, leaves, j):

        # Storage the new leaf
        leaves[j] = new_leaf


        # Bubble sort algorithm to order the leaves in ascending order according to ...
        for k in range(j, 0, -1):

            # Check if the previous leaf has a higher absolute CMI; if so, they ...
            if  leaves[k-1].absoluteCMIgain > leaves[k].absoluteCMIgain:
                leaves[k], leaves[k-1] = leaves[k-1], leaves[k]
```

- void **visualize**: gives a two dimensional plot of the data samples with the partitions made by the TSP.

  (i) list x: random variable $\mathcal{X}$ samples.

  (ii) list y: random variable $\mathcal{Y}$ samples.

  First, the system verifies whether a tree has been grown previously and if the space is two-dimensional. Then, the TSP partitions are obtained and categorized into vertical partitions (along the $\mathcal{X}$ dimension) and horizontal partitions (along the $\mathcal{Y}$ dimension).

```
def visualize(self, x, y):

    # A grown tree is required
    if self.root is None:
        raise ValueError("Observations not provided.")


    # Check if the problem is two dimensional
    if self.dim != 2:
        raise ValueError("The tree can only be visualized for a two ...)


    # Partitions
    partitions = self.partitions()


    # Initialize lists for the partitions in every dimension
    horizontal_partition = [partition[1:] for partition in partitions if ...]
    vertical_partition = [partition[1:] for partition in partitions if ...]
```

  A basic plot is produced by drawing the data samples and the vertical and horizontal lines.

```
    # Create a figure and axis object
    fig, ax = plt.subplots()


    # Plot sample points
    ax.plot(x, y, 'bo', markersize=5, alpha=0.5)


    # Plot horizontal lines
    for bound in vertical_partition:
        ax.hlines(y=bound[0][1], xmin=bound[0][0], xmax=bound[1][0] ...)


    # Plot vertical lines
    for bound in horizontal_partition:
        ax.vlines(x=bound[0][0], ymin=bound[0][1], ymax=bound[1][1] ...)
```

- float **emi**: returns the estimated mutual information (simply calls the instance variable tsp_emi).

```
def emi(self):
    if self.root is void:
        raise ValueError("Observations not provided.")
    return self.tsp_emi
```

- float **reg_emi**: returns the regularized mutual information (simply calls the instance variable tsp_reg_emi).

```
def reg_emi(self):
    if self.root is None:
        raise ValueError("Observations not provided.")
    return self.tsp_reg_emi
```

- int **size**: returns the tree size (simply calls the instance variable tsp_size).

```
def size(self):
    if self.root is None:
        raise ValueError("Observations not provided.")
    return self.tsp_size
```

- int **reg_size**: returns the regularized tree size (simply calls the instance variable tsp_reg_size).

```
def reg_size(self):
    if self.root is None:
        raise ValueError("Observations not provided.")
    return self.tsp_reg_size
```

- list **partitions**: returns the space partitions bounds (simply calls the instance variable tsp_partitions).

```
def partitions(self):
    if self.root is None:
        raise ValueError("Observations not provided.")
    return self.tsp_partitions
```

4 Appendix

Corollary 1. *The absolute conditional mutual information gain of a given node can be calculated using its relative conditional mutual information gain and the ratio of samples between the current node and the root node.*

$$\Delta\rho(v \mid U_v)_{root} = \Delta\rho(v \mid U_v) \cdot \frac{P_n(U_v)}{P_n(U_{root})}$$

Proof. Using the expressions provided in equation (45) of the paper, the gain for both children is added, and the gain of their parent (which is no longer an external node) is subtracted

$$\Delta\rho(v \mid U_v)_{root} = \frac{P_n(U_{l(v)})}{P_n(U_{root})} \log_2 \left(\frac{P_n(U_{l(v)})/P_n(U_{root})}{Q_n(U_{l(v)})/Q_n(U_{root})} \right) + \frac{P_n(U_{r(v)})}{P_n(U_{root})} \log_2 \left(\frac{P_n(U_{r(v)})/P_n(U_{root})}{Q_n(U_{r(v)})/Q_n(U_{root})} \right)$$

$$- \frac{P_n(U_v)}{P_n(U_{root})} \log_2 \left(\frac{P_n(U_v)/P_n(U_{root})}{Q_n(U_v)/Q_n(U_{root})} \right)$$

$$= \frac{P_n(U_v)}{P_n(U_{root})} \left[\frac{P_n(U_{l(v)})}{P_n(U_v)} \log_2 \left(\frac{P_n(U_{l(v)})/P_n(U_{root})}{Q_n(U_{l(v)})/Q_n(U_{root})} \right) + \frac{P_n(U_{r(v)})}{P_n(U_v)} \log_2 \left(\frac{P_n(U_{r(v)})/P_n(U_{root})}{Q_n(U_{r(v)})/Q_n(U_{root})} \right) \right.$$

$$\left. -1 \cdot \log_2 \left(\frac{P_n(U_v)/P_n(U_{root})}{Q_n(U_v)/Q_n(U_{root})} \right) \right]$$

note that $1 = \frac{P_n(U_v)}{P_n(U_v)} = \frac{P_n(U_{l(v)}) + P_n(U_{r(v)})}{P_n(U_v)}$

$$= \frac{P_n(U_v)}{P_n(U_{root})} \left[\frac{P_n(U_{l(v)})}{P_n(U_v)} \log_2 \left(\frac{P_n(U_{l(v)})/P_n(U_{root})}{Q_n(U_{l(v)})/Q_n(U_{root})} \right) + \frac{P_n(U_{r(v)})}{P_n(U_v)} \log_2 \left(\frac{P_n(U_{r(v)})/P_n(U_{root})}{Q_n(U_{r(v)})/Q_n(U_{root})} \right) \right.$$

$$\left. - \frac{P_n(U_{l(v)}) + P_n(U_{r(v)})}{P_n(U_v)} \log_2 \left(\frac{P_n(U_v)/P_n(U_{root})}{Q_n(U_v)/Q_n(U_{root})} \right) \right]$$

$$= \frac{P_n(U_v)}{P_n(U_{root})} \left[\frac{P_n(U_{l(v)})}{P_n(U_v)} \left[\log_2 \left(\frac{P_n(U_{l(v)})/P_n(U_{root})}{Q_n(U_{l(v)})/Q_n(U_{root})} \right) - \log_2 \left(\frac{P_n(U_v)/P_n(U_{root})}{Q_n(U_v)/Q_n(U_{root})} \right) \right] \right.$$

$$\left. + \left[\log_2 \left(\frac{P_n(U_{r(v)})/P_n(U_{root})}{Q_n(U_{r(v)})/Q_n(U_{root})} \right) - \log_2 \left(\frac{P_n(U_v)/P_n(U_{root})}{Q_n(U_v)/Q_n(U_{root})} \right) \right] \right]$$

$$= \frac{P_n(U_v)}{P_n(U_{root})} \left[\frac{P_n(U_{l(v)})}{P_n(U_v)} \log_2 \left(\frac{P_n(U_{l(v)})/P_n(U_v)}{Q_n(U_{l(v)})/Q_n(U_v)} \right) + \frac{P_n(U_{r(v)})}{P_n(U_v)} \log_2 \left(\frac{P_n(U_{r(v)})/P_n(U_v)}{Q_n(U_{r(v)})/Q_n(U_v)} \right) \right]$$

$$= \Delta\rho(v \mid U_v) \cdot \frac{P_n(U_v)}{P_n(U_{root})}$$

□