Copula dependence and risk sensitivity of asset portfolio by Qinqin Huang, Yongyi Tang, Xiaohan Shen, and Pai Peng

1 Introduction

In this report, we refer to the paper by Bruneau et al.2 to estimate the risk sensitivity of financial assets through multivariate copula. The structure of the report is as follows. First, we analyze the data and build models to realize transforming data from ppf to cdf and the inverse process. Second, we introduce the canonical vine and simulate data from the canonical vine which have the same dependences with our input data as the parameters of the canonical vine are fitted based on the input data. Third, we calculate the Cross Conditional Value at Risk (CVaR).

2 Modeling

2.1 Copulae

In this section, we basically infer the models from the paper by Aas et al. 1.

2.1.1 C-Vine Considering a multivariate cumulated distribution function F of n random variables $\mathbf{X} = X_1, ..., X_n$ with marginals $F_1(x_1), ..., F_n(x_n)$, Skalar's Theorem states that there exists a unique n-dimensional copula C to describbte the joint distribution of these these variables. If F is absolutely continuous with strictly increasing and continuous marginal cdf F_i, the joint density function f can be written as:

$$f(x_1, x_2, ..., x_n) = c_{1:n}(F_1(x_1), F_2(x_2), ..., F_n(x_n)) \cdot \prod_{i=1}^{n} f_i(x_i).$$

which is the product of the n-dimensional copula density $c_{1:n}(\cdot)$ of C and the marginal densities $f_i(\cdot)$.

Building high-dimensional copulae is generally recognized as a challenging task. One of the most popular methods is the pair-copula construction (PCC) proposed by Aas et al. [1]1. The idea is to construct a high-dimensional copula by combining bivariate copulae. The basic principle behind PCC is that the density can be factorized as:

$$f(x_1, x_2, ..., x_n) = f_n(x_n) \cdot f(x_{n-1}|x_n) \cdot f(x_{n-2}|x_{n-1}, x_n) \cdot ... \cdot f(x_1|x_2, ..., x_n). \quad (1)$$

Each term in (1) can be decomposed into the appropriate pair-copula times a conditional marginal density, using the general formula:

$$f(x|\mathbf{v}) = c_{xv_j|\mathbf{v}_{-j}}\{F(x|\mathbf{v}_{-j}), F(v_j|\mathbf{v}_{-j})\} \cdot f_x(\mathbf{v}_{-j}).$$

Here $\mathbf{v}$ is a vector of variables, $\mathbf{v}_{-j}$ is the vector $\mathbf{v}$ with the j-th element removed.

The pair-copula construction involves marginal conditional distribution of the form $F(x|\mathbf{v})$. For every j :

$$F(x|\mathbf{v}) = \frac{\partial C_{xv_j|\mathbf{v}_{-j}}\{F(x|\mathbf{v}_{-j}), F(v_j|\mathbf{v}_{-j})\}}{\partial F(v_j|\mathbf{v}_{-j})}.$$

For the special case where v is a univariate, we have:

$$F(x|v) = \frac{\partial C_{xv}(F(x), F(v))}{\partial F(v)}.$$

We will use the function $h(x, v, \Theta)$ to represent this conditional distribution function when x and v are uniform:

$$h(x, v, \Theta) = F(x|v) = \frac{\partial C_{xv}(F(x), F(v))}{\partial F(v)}, \Theta\text{−the set of parameters of the joint distribution function.}$$

Furture, let $h^{-1}(u, v, \Theta)$ be the inverse of the h-function with respect to u, or the equivalently the inverse of the conditional distribution function.

For high-dimension distribution, there are significant number of possible pair-copular. To help organising them, Bedford and Cooke have introduced a graphical model denoted as the regular vine. Here, we concentrate on the special case of regular vines - the canonical vine (C-vine), which gives a specific way of decomposing the density. The figure below shows a C-vine with 5 variables.

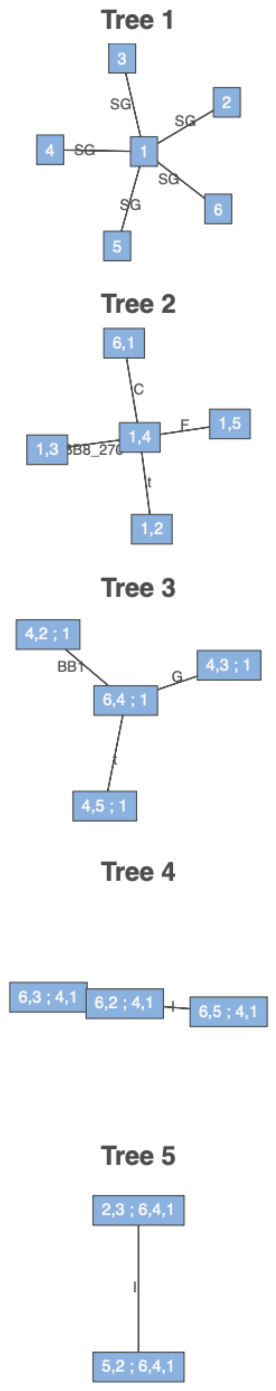


Figure 1: C-vine

The n-dimensional density corresponding to a C-vine is given by:

$$\prod_{k=1}^{n} f(x_k) \prod_{j=1}^{n-1} \prod_{i=1}^{n-j} c_{i,i+j|1,...,j-1}\{F(x_j|x_1,...,x_{j-1}), F(x_{i+j}|x_1,...,x_{i+j-1})\}.$$

Fitting a canonical vine might be advantageous when a particular variable is known to be a key variable that governs interaction in the data set. In such a situation one may decide to locate this variable at the root of the canonical vine, as we have done with variable in the figure.

2.1.2 Conditional independence Assuming conditional independence may reduce the number of levels of the pair-copula decomposition, and hence simplify the construction.

In general, for any vector of variables $\mathbf{v}$, and two variables X and Y, the latter are conditionally independent given $\mathbf{V}$ if and only if:

$$c_{xy|\mathbf{v}}\{F(x|\mathbf{v}), F(y|\mathbf{v})\} = 1.$$

2.1.3 Simulation from a pair-copula decomposed model In this section we show the simulation algorithm for canonical vines which follows the method discussed in Bruneau (2019) We assume for simplicity that the margins of the distribution are uniform. [^1]: For variables with other marginal distributions, we transform the data to uniform marginals before fitting the vine copula.

To sample n dependent uniform[0, 1] variables, we first sample $w_1, ..., w_n$ independent uniform on [0, 1]. Then set

$$x_1 = w_1,$$

$$x2 = F^{-1}(w_2|x_1),$$

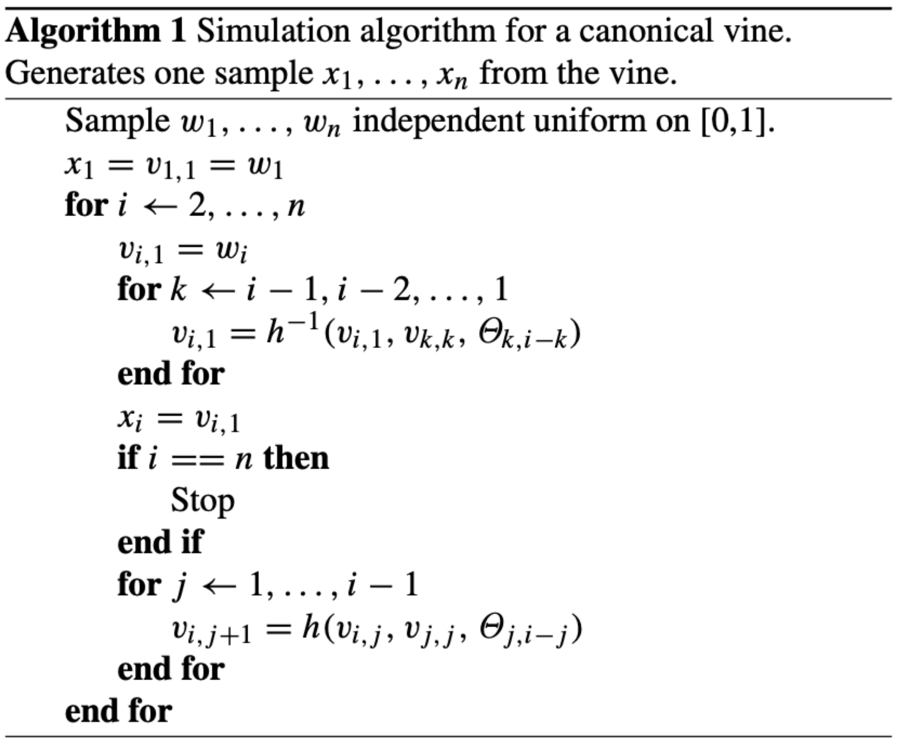
$$x3 = F^{-1}(w_3|x_1, x_2), ...$$

$$x_n = F^{-1}(w_n|x_1, ..., x_{n-1}).$$

To determine the conditional distribution $F^{-1}(w_j|x_1, ..., x_{j-1})$, we use the h-function.

The algorithm gives the procesdure for sampling from a canonical vine. The outer loop runs over the variables to be sampled. This loop consists of two other for-loops. In the first, the ith varaible is sampled, while in the other, the conditional distribution functions needed for sampling the $(i+1)$th variable are updated. To compute these conditional distribution functions, we repeatedly use the h-function, with previously computed conditional distribution functions, $v_{i,j} = F(x_i|x_1,...,x_{j-1})$, as the first two arguments. The last argument of the h-function is the parameter $\Theta_{j,i}$ of the corresponding copula density $c_{j,j+i}|1,...,j-1(\cdot,\cdot)$

The algorithm is as follows:



Algorithm 1 Simulation algorithm for a canonical vine.
Generates one sample $x_1, \ldots, x_n$ from the vine.

```
Sample $w_1, \ldots, w_n$ independent uniform on [0,1].
$x_1 = v_{1,1} = w_1$
for $i \leftarrow 2, \ldots, n$
    $v_{i,1} = w_i$
    for $k \leftarrow i-1, i-2, \ldots, 1$
        $v_{i,1} = h^{-1}(v_{i,1}, v_{k,k}, \Theta_{k,i-k})$
    end for
    $x_i = v_{i,1}$
    if $i == n$ then
        Stop
    end if
    for $j \leftarrow 1, \ldots, i-1$
        $v_{i,j+1} = h(v_{i,j}, v_{j,j}, \Theta_{j,i-j})$
    end for
end for
```

Figure 2: simulation algorithm

2.1.4 Estimation of the parameters In this section we describe how the parameters of the canonical vine density are estimated. To simplify the process as mentioned before, we assumme that the marginals are uniform and the the time series is stationary and independent over time. This assumption is not restrictive, as we can always transform the data to uniform marginals before fitting the vine copula.

We use the maximum likelihood method to estimate the parameters of the canonical vine. Since the actual margins are normally unknown in practice, what is being maximised is a pseudo-likelihood.

The log-likelihood is given by:

$$\sum_{j=1}^{n-1}\sum_{i=1}^{n-j}\sum_{t=1}^{T}\log c_{j,j+i|1,\dots,j-1}\{F(x_{j,t}|x_{1,t},...,x_{j-1,t}),F(x_{i+j,t}|x_{1,t},...,x_{j-1,t})\}.$$

For each copula in the above formula, there is at least one parameter to be determined. The algorithm for estimating the parameters is listed below in the figure. The ourter for-loop corresponds to the outer sum in the pseudo-likelihood. The inner for-loop corresponds to the sum over i. The innermost for-loop corresponds to the sum over the time series. Here, the element t of $textbfv_Pj,i$ is $v_{j,i,t}=F(x_{i+j,t}|x_{1,t},...,x_{j,t})$. $L(\mathbf{x},\mathbf{v},\Theta)$ is the log-likelihood of the chosen bivariate copula with parameters Θ and the data $\mathbf{x}$ and $\mathbf{v}$. That is,

$$L(\mathbf{x},\mathbf{v},\Theta)=\sum_{t=1}^{T}\log c(x_t,v_t,\Theta), c(u,v,\Theta)\text{is the density of the bivariate copula with parameters }\Theta.$$

```
log-likelihood = 0
for $i \leftarrow 1, \dots, n$
    $\boldsymbol{v}_{0,i} = \boldsymbol{x}_i$.
end for
for $j \leftarrow 1, \dots, n-1$
    for $i \leftarrow 1, \dots, n-j$
        log-likelihood = log-likelihood
                         $+ L(\boldsymbol{v}_{j-1,1}, \boldsymbol{v}_{j-1,i+1}, \Theta_{j,i})$
    end for
    if $j == n-1$ then
        Stop
    end if
    for $i \leftarrow 1, \dots, n-j$
        $\boldsymbol{v}_{j,i} = h(\boldsymbol{v}_{j-1,i+1}, \boldsymbol{v}_{j-1,1}, \Theta_{j,i})$
    end for
end for
```

Figure 3: estimation algorithm

Starting values of the parameters needed in the numerical maximization of the log-likelihood are determined as follows:

1. Estimate the parameters of the copulae in the first level of the vine tree from the original data.
2. Compute observations for tree 2 using the copula parameters from tree 1 and the h-function.
3. Estimate the parameters of the copulae in the second level of the vine tree from the observations computed in step 2.
4. Repeat steps 2 and 3 until the parameters of all copulae in the vine tree have been estimated.

2.1.5 Copula selection In the above content, we introduce the canonical vine copula, the calibration of the parameters, and the simulation of the data. However, we didn't specify which copula to use in the pair-copula decomposition. The choice of copula is crucial for the performance of the model. In this section, we discuss the selection of the copulae.

2.1.5.1 Gaussian copula The density of the bivariate Gaussian copula is given by:

$$c(u,v,\theta) = \frac{1}{\sqrt{(1-\theta^2)}} exp\{-\frac{\theta^2(x_1^2+x_2^2)-2\theta x_1 x_2}{2(1-\theta^2)}\}, -1 < \theta < 1.$$

Here, θ is the correlation parameter, which is normally denoted as ρ. $x_1 = \Phi^{-1}(u)$, $x_2 = \Phi^{-1}(v)$, and Φ is the standard normal distribution function.

The h-function is given by:

$$h(u,v,\theta) = \Phi(\frac{\Phi^{-1}(u)-\theta\Phi^{-1}(v)}{\sqrt{1-\theta^2}}).$$

Suppose the h-function is equal to w, then the inverse h-function is given by:

$$h^{-1}(w,v,\theta) = \Phi\{\Phi^{-1}(w)\sqrt{1-\theta^2}+\theta\Phi^{-1}(v)\}$$

2.1.5.3 Clayton copula The density of Clayton copula is given by:

$$c(u,v,\theta) = (1+\theta)(u\cdot v)^{-\theta}-1) \times (u^{-\theta}+v^{-\theta}-1)^{-1/\theta-2}, 0 < \theta < \infty.$$

Perfect dependence is obtained when $\theta \to \infty$, while $\theta \to 0$ implies independence.

For this copula the h-function is given by:

$$h(u,v,\theta) = v^{-\theta-1}(u^{-\theta}+v^{-\theta}-1)-1-\theta.$$

Suppose the h-function is equal to w, then the inverse h-function is given by:

$$h^{-1}(w,v,\theta) = \{(w\cdot v^{\theta+1})^{\frac{\theta}{\theta+1}}+1-v^{-\theta}\}^{-1/\theta}.$$

2.2 Distribution of the data

2.3 CCVaR

The Cross Conditional Value at Risk (CCVaR) quantifies the expected return of an asset under the extreme conditions of a given risk factor. For an asset R_i and a risk factor X, the CCVaR at confidence level α is defined as:

$$CCVaR_\alpha(R_i \mid X; F_X) = \mathbb{E}[R_i \mid F_X(X) \leq \alpha],$$

where:

$$F_X(X): \text{ the cumulative distribution function (CDF) of the risk factor } X,$$

α : the confidence level defining the extreme quantile (e.g., $\alpha = 0.05$ for the worst 5 %).

3 Implementation

3.1 Data

3.2 Code

3.2.1 Distribution

3.2.2 CVine In this code, we use the class `CVine` to realize the canonical vine copula. Basically, the class involves the following methods:

1. `build_tree()` - to build the tree structure of the canonical vine copula. This method will fill the class attribute `tree` with the tree structure.
2. `fit()` - to fit the canonical vine copula to the data. This method will estimate the parameters of the copulae in the vine tree, which will call the method `get_likelihood()` to calculate the log-likelihood of the tree. Here, we use `scipy.optimize.minimize` to maximize the log-likelihood.
3. `simulate()` - to simulate data from the canonical vine copula. This method will simulate data from the fitted vine copula. In this algorithm, we generate independent uniform random variables and then use the algorithm mentioned in Section 2 to generate dependent uniform random variables.

3.2.3 CCVar In this section, we describe the code implementation of CCVaR using Python. The implementation is encapsulated in the `CCVaR` class, which contains methods to calculate CCVaR for single asset-factor pairs and generate a CCVaR matrix for all assets and factors.

1. **Initialization**: The `__init__` method initializes the CCVaR model by taking the following inputs:

- `data`: Asset return matrix ($T \times N$).
- `factors`: Risk factor matrix ($T \times F$).
- `alpha`: Confidence level for defining extreme conditions.

2. **Data Transformation**: The `_transform_to_uniform` method transforms raw data to the uniform space $[0, 1]$ using the empirical cumulative distribution function (CDF).
3. **Extreme Event Identification**: The `_get_extreme_indices` method identifies indices corresponding to extreme events, where the risk factor falls below the α-quantile.
4. **Single CCVaR Calculation**: The `calculate_ccvar` method computes CCVaR for a single asset with respect to a specific risk factor.
5. **CCVaR Matrix Calculation**: The `calculate_all_ccvar` method generates a matrix of CCVaR values for all assets and risk factors.

6. **Result Summarization**: The `summarize_results` method outputs the CCVaR matrix with labels for assets and factors.

3.3 Results

3.3.1 Distribution

3.3.2 CVine We test the `CVine` based on the return data of the assets. According to our definition of copula, the parameters of the copulae will reflect the level of dependence between the assets.

We first test the `CVine` with the Gaussian copula. The results are shown below:

Then we test the `CVine` with the Clayton copula. The results are shown below:

3.3.3 CCVaR

4 Conclusion

The canonical vine copula is a powerful tool for modeling the dependence structure of multivariate data. In this report, we have introduced the canonical vine copula and its application in estimating the risk sensitivity of asset portfolios. We have implemented the canonical vine copula in Python and demonstrated its use in simulating data and estimating the parameters of the copulae. We have also implemented the Cross Conditional Value at Risk (CCVaR) to quantify the expected return of an asset under extreme conditions of a given risk factor. The CCVaR provides a useful measure of the risk sensitivity of asset portfolios to different risk factors. Further research can explore the application of the canonical vine copula in asset portfolio optimization and risk management. The drawbacks of our implementation include that we have not compare the results of different copulae and simply assume returns follow Gaussian Copula.

References

[1] Aas, K., Czado, C., Frigessi, A., and Bakken, H. (2009). Pair-copula constructions of multiple dependence. Insurance: Mathematics and Economics, 44(2), 182-198.

[2] Catherine Bruneau, Alexis Flageollet, and Zhun Peng. (2019). Vine Copula Based Modeling.

[3] Claudia Czado and Thomas Naglar. (2021). Vine copula based modeling.

Appendix

A Code

A.1 CVine

```
class CVine(object):
    layer = {"root": [],  # list of root nodes. ex. in F(u1, u2|v), v is the root node
             "parentnode": {},
             # index of nodes in last level. ex. {1: (1,2)} means the node 1 in this tree le
             "node": [],  # index of the nodes. from 0 to l. this is not the actual number o
             "pair": [],
             # list of node pairs in the tree, ex. in F(u1, u2|v), (u1, u2) is a node pair.
             "level": 0,  # level of the tree (k). 0-root, 1-1st level, 2-2nd level, ...
             "nodenum": 0,  # number of the nodes in this tree (l). equal to n - k
             "edgenum": 0,
             # number of the edges in this tree. equal to l as our node number is the actual
             "V": None,  # h functions in this level. V[:, j] is the h function of node j.
             "dependence": []  # match the pair of nodes
             }

    tree = {"thetaMatrix": None,
            # copula parameter matrix in this level. it is a upper matrix. thetaMatrix[i, j]
            "structure": {},  # the tree structure in this level. the key is the node index,
            "depth": 0,  # the depth of the tree, 0 means only has root.
            }
    max_depth = None  # if the max_depth is given, the only use the first max_depth layers t

    def __init__(self, U, copulaType="Clayton", dependence=None, max_depth=None):

        """
        U: np.array, data matrix. follows uniform distribution
        dependence: np.array, dependence matrix constructed by 0 and 1. dependence[i, j] = 1

        """
        self.U = U
        self.T = U.shape[0]
        self.variable_num = U.shape[1] - 1  # to make the structure more clear, all the vari
        self.dependence = dependence if dependence is not None else np.ones((self.variable_n
        if copulaType == "Clayton":
            self.copula = Clayton()
        elif copulaType == "Gaussian":
            self.copula = Gaussian()
        else:
            raise ValueError("The copula type is not supported.")

        self.max_depth = max_depth


    def build_tree(self):
        """
        build the tree structure.
        """
        self.build_root()
        while self.tree["depth"] < self.variable_num:
            self.build_kth_tree()

    def build_root(self):
        """
        build the root of the tree. the root is basically
        """

        layer = self.layer.copy()
        layer["level"] = 0
        layer["V"] = self.U.copy()  # the F(x|v) in the first layer is the empirical cdf of
        layer["nodenum"] = self.variable_num
        layer["edgenum"] = self.variable_num
        layer["node"] = list(range(0, layer["nodenum"] + 1))
        layer["dependence"] = self.dependence[0, :]
        self.tree["structure"][0] = layer

    def build_kth_tree(self):
        """
        build the kth tree.
        """

        if self.tree["depth"] >= self.variable_num:
            print("The tree depth is already the maximum.")

        last_layer = self.tree["structure"][self.tree["depth"]]

        layer = self.layer.copy()
        layer["level"] = last_layer["level"] + 1
        layer["nodenum"] = last_layer["nodenum"] - 1
        layer["edgenum"] = layer["nodenum"]
        layer["node"] = list(range(0, layer["nodenum"]+ 1))
        (layer["pair"], layer["node"], layer["parentnode"], layer["root"], layer["dependence

        self.tree["structure"][layer["level"]] = layer
        self.tree["depth"] = self.tree["depth"] + 1

    def pair_nodes(self, last_layer):
        """
        pair the nodes in this layer.
        here we use the first node in each level as the new central node and combine it with
        """

        nodes = range(0, last_layer["nodenum"] + 1)
        if last_layer[
            "level"] == 0:  # the second layer is not conditional copula, so we just combine
            pair_left = last_layer["node"][0]
            pairs = tuple(zip(last_layer["edgenum"] * [pair_left], last_layer["node"][1:]))
            parentnodes = dict(zip(nodes, pairs))
            dependent = np.empty(last_layer["nodenum"] + 1)
            dependent[0] = 1
            for i in range(1, last_layer["nodenum"] + 1):
                if i > 1: # todo
                    dependent[i] = 0 if self.dependence[pairs[-1][1], pairs[0][1]] == 0 else

            return (pairs,
                   nodes,
                   parentnodes,
                   [],
                   dependent)

        else:
            pairs = []
            parentnodes = {}
            last_pairs = last_layer["pair"]

            common_node = last_pairs[0][
                0]  # always set the first node as the center node in each layer. and the co

            new_root = last_layer["root"] + [common_node]
            pair_left = last_pairs[0][1] # the right element in the center pair will be the
            dependent = np.empty(last_layer["nodenum"] + 1)
            dependent[0] = 1
            for i in range(1, last_layer["nodenum"] + 1):
                pairs.append(tuple((pair_left, last_pairs[i][1])))
                parentnodes[i-1] = (0, i)  # the i-1th node in this layer is from the pair
                if i > 1: # todo
                    dependent[i] = 0 if self.dependence[pairs[-1][1], pairs[0][1]] == 0 else


            return (pairs,
                nodes,
                parentnodes,
                new_root,
                dependent)

    def fit(self):
        """
        fit the vine tree model by maximizing the likelihood of the whole tree.
        """
        paramNum = sum([self.tree["structure"][layer]["edgenum"] for layer in range(0, self.

        thetaParams = np.ones(paramNum)
        bounds = [self.copula.bound] * paramNum

        result = minimize(self.get_likelihood, thetaParams, bounds=bounds)
        thetaMatrix = np.zeros((self.tree["depth"], self.tree["structure"][0]["edgenum"]))
        n = 0
        for i in range(0, self.tree["depth"]):
            for j in range(0, self.tree["structure"][i]["edgenum"]):
                thetaMatrix[i, j] = result.x[n]
                n += 1

        self.tree["thetaMatrix"] = thetaMatrix

    def fit2(self):

        """
        fit the parameters through maximizing the likelihood in each layer.
        """
        self.tree["thetaMatrix"] = np.zeros((self.tree["depth"], self.tree["structure"][0]['

        for i in range(1, self.tree["depth"] + 1):
            last_layer = self.tree["structure"][i - 1]
            layertheta = np.ones(last_layer["edgenum"])
            bounds = [self.copula.bound] * last_layer["edgenum"]
            result = minimize(self.get_layer_likelihood, layertheta, args=(last_layer,), bou

            self.tree["thetaMatrix"][i - 1, :last_layer["edgenum"]] = result.x

            self.tree["structure"][i]["V"] = self.get_layer_h(result.x, last_layer)

    def simulate(self, n):
        """
        simulate the data from the vine tree model
        param n: int, the number of the data to be simulated for each variable.

        """
        if self.tree["thetaMatrix"] is None:
            print("Please fit the model first.")
            return None

        else:
            W = np.random.uniform(0, 1, n * (self.variable_num + 1))
            V = np.empty((n, self.variable_num + 1, self.tree["depth"] + 1))
            W = W.reshape((n, self.variable_num + 1))
            U = np.empty((n, self.variable_num + 1))
            U[:, 0] = W[:, 0]
            V[:, 0, 0] = W[:, 0]
            for i in range(1, self.variable_num + 1):
                V[:, 0, i] = W[:, i]
                for k in range(0, i):
                    self.copula.theta = self.tree["thetaMatrix"][k, i - k - 1]
                    V[:, 0, i] = self.copula.inverse_h(V[:, 0, i], V[:, k, k])

                U[:, i] = V[:, 0, i]

                for j in range(0, i):
                    self.copula.theta = self.tree["thetaMatrix"][j, i - j - 1]
                    V[:, j + 1, i] = self.copula.h(V[:, j, i], V[:, j, j])
            return U

    def get_likelihood(self, thetaParams):
        """get the likelihood of the vine tree model"""

        total_likelihood = 0
        left = 0
        right = 0
        for k in range(1, self.tree["depth"] + 1):  # ignore the root layer
            # each layer' c function is determined by the last layer's and this layer's the
            last_layer = self.tree["structure"][k - 1]

            left = right
            right = right + last_layer["edgenum"]
            layertheta = thetaParams[left:right]
            total_likelihood += self.get_layer_likelihood(layertheta, last_layer)

            self.tree["structure"][k]["V"] = self.get_layer_h(layertheta, last_layer)

        return total_likelihood

    def get_layer_likelihood(self, thetaParams, last_layer):
        """get the likelihood of the layer"""
        likelihood = 0

        for i in range(1, last_layer["nodenum"]+1): # totally l copula functions

            self.copula.theta = thetaParams[i-1]
            likelihood += (np.nansum(np.log(self.copula.c(last_layer["V"][:, 0], last_layer
        for i in range(1, last_layer["nodenum"] + 1):  # totally l copula functions

            self.copula.theta = thetaParams[i - 1]
            likelihood += np.nansum(np.log(self.copula.c(last_layer["V"][:, 0], last_layer['

        return -likelihood

    def get_layer_h(self, thetaParams, last_layer):
        """get the h function of the layer"""
        V = np.empty((self.T, last_layer[
            "nodenum"]))  # the total nodes of this layer is the number of nodes in last la

        for i in range(1, last_layer["nodenum"] + 1):
            self.copula.theta = thetaParams[i - 1]
            V[:, i - 1] = self.copula.h(last_layer["V"][:, 0], last_layer["V"][:, i])

        return V
```

A.2 Bivariate Copula

```
def mypower(x, y):
    """
    use different method to calculate the power of x and y to avoid overflow.
    return: np.array, the power of x and y.
    """
    x = np.clip(x, 1e-10, 1e10)
    log_x = np.log(x)
    power = np.exp(y * log_x)

    return power


class Clayton:
    def __init__(self):
        self.theta = 0
        self.bound = (1e-5, 1)

    def c(self, u: np.ndarray, v: np.ndarray):
        """
        return: np.array, the density of Clayton copula
        """
        return (1 + self.theta) * mypower(u * v, -1 - self.theta) \
            * mypower(mypower(u, -self.theta) + mypower(v, -self.theta) - 1, -2 - 1 / self.t

    def h(self, u: np.ndarray, v: np.ndarray):
        """

        return: np.array, the h function/partial derivative F(u|v)  of Clayton copula
        since h function is basically a kind of conditional CDF, it should be between 0 and

        """
        a = mypower(v, -self.theta - 1)
        b = mypower(u, -self.theta) + mypower(v, -self.theta) - 1
        c = mypower(b, -1 - 1 / self.theta)
        result = a * c

        # todo check which theta value will lead to nan value.
        if self.theta > 1000:
            result[np.isnan(result)] = 1
        else:
            result[np.isnan(result)] = 0

        return result

    def inverse_h(self, w: np.ndarray, v: np.ndarray):

        """
        return: np.array, the inverse of h function, which is the conditional CDF of u give
        since the inverse of h function will lead to the x, which is uniform distributed, t
        """

        a = w * mypower(v, self.theta + 1)
        b = mypower(a, -self.theta / (1 + self.theta))
        c = mypower(v, -self.theta)
        d = mypower(b + 1 - c, -1 / self.theta)

        # todo: to avoid the nan value we add this adjustment here. when the correlation is
        if self.theta > 1000:
            d[np.isnan(d)] = v[np.isnan(d)]
        else:
            d[np.isnan(d)] = w[np.isnan(d)]

        return d


class Gaussian:
    def __init__(self):
        self.theta = 0
        self.bound = (1e-5, 1)

    def c(self, u, v):
        """
        return the density of Clayton copula
        """
        return (1 + np.sqrt(1 - mypower(self.theta, 2))) * np.exp(-(mypower(self.theta, 2) *
                                                               2 * (1 - mypower(self.thet

    def h(self, u, v):
        """
        return the h function/partial derivative F(u|v)  of Clayton copula

        """

        a = (norm.ppf(u) - self.theta * norm.ppf(v)) / np.sqrt(1 - self.theta ** 2)

        return norm.cdf(a)

    def inverse_h(self, w, v):
        """
        return the inverse of h function, which is the conditional CDF of u given v.
        """

        a = norm.ppf(w) * np.sqrt(1 - self.theta ** 2) + self.theta * norm.ppf(v)

        return norm.cdf(a)
```