Experiments and codes

May 15, 2019

```
In [1]: import pandas as pd
        import numpy as np
        import sys
        sys.path.append('../imo/')
```

```
In [23]: melanoma = pd.read_table('../../../maleki-coombes/data/\
                                  raw/SKCM/edited/TCGA.SKCM.mutect.\
                                  4b7a5729-b83e-4837-9b61-a6002dce1c0a.DR-10.0.somatic.maf')
```

1 Driver Mutation

Using Mutsigcv to choose driver mutation this mutation are the first 29 mutation in mutcigcv. ### you can find the csv file of all the mutation and the score of them in result folder

```
In [24]: driver_genes = ["BRAF","PEA15","PPP6C","TP53","PTEN",
                         "NRAS","CDKN2A","NF1","RAC1","ARID2","RB1",
                         "C1QTNF9","DDX3X","AGXT2","ZFX","B2M","CTNNB1",
                         "GALNTL5","COL4A3","RPL5","CYP3A5","RARRES2","TNFRSF8",
                         "LCE1B","DDX4","HIST1H2AA","HNF4G","EREG","LCK"]
```

```
In [26]: melanoma = melanoma[melanoma.Hugo_Symbol.isin(driver_genes)]
```

2 OncoTree

Algorithm Note that we primarily want to learn the structure of the tree not the conditional probabilities $\alpha(e)$s.

To simplify the notation, let $p_i = P(X_i = 1)$, $p_{ij} = P(X_i = 1, X_j = 1)$, and $p_{j|i} = P(X_j = 1|X_i = 1)$. We define the weight of edges between mutation i and j as: $w_{ij} = \log \frac{p_i}{p_i+p_j} \frac{p_{j|i}}{p_j}$.

The weights captures the intuition behind "i being the parent of j": If i is more frequent and when it happens it increases the probability of j happening. Authors show that the maximum branching or the graph constructed with edge weights w_{ij} is exactly the tree T. To compute the weights from the data we replace the probabilities with the frequencies: $w_{ij} = \log \frac{f_{ij}}{f_j(f_i+f_j)}$.

Therefore, we are after a maximum directed tree in a weighted graph. To filter out false positives and reduce the size of the problem, we focus on a subset of mutations, i.e., maximum-weight clique of the graph. Then we use the Edmond's algorithm which runs in $O(n^2)$ and finds the minimum branching of the graph with negated edge weights $-w_{ij}$.

Algorithm 1 OncoTree

input: Frequencies of mutations f_i, f_{ij}, a threshold t, and a clique size s
output: Branching $T = (\mathcal{V}, \mathcal{E}, r, p)$
$w_{ij} \leftarrow \mathbf{1}(f_{ij} > t) \log \frac{f_{ij}}{f_j(f_i+f_j)}$
$G = (\mathcal{V}, \mathcal{W})$, where $\mathcal{V}$ and $\mathcal{W}$ are sets of all mutations and edge weights.
Optional: Focus on maximum-weight clique of size s in G
$w_{ij} \leftarrow -w_{ij}$
Compute the minimum branching of G: $T(\mathcal{V}, \mathcal{E}, r) \leftarrow$ Edmond's(G)
Compute edge probabilities of branching as: $p_{j|i} \leftarrow \frac{f_{ij}}{f_i}$.

2.1 Implementation

```
class OncoTree(object):
    """
    The tree model for oncogenesis by Desper, Richard, et al.

    This implementation is based on:

        Desper, R., Jiang, F., Kallioniemi, O. P., Moch, H., Papadimitriou, C. H., & Schäffer, A. A. (1999)
        Inferring tree models for oncogenesis from comparative genome hybridization data.
        Journal of computational biology, 6(1), 37-51. URL: https://www.ncbi.nlm.nih.gov/pubmed/10223663

    Parameters
    ----------
    min_mutations : integer or None, optional (default=None)
        Minimum number of mutations to consider. The genes which has mutated below this number
        will be dropped. If ``None``, no mutation is skipped.
    """

    def __init__(self, min_mutations=None):
        self.G = None
        self.edge_list = None
        self.optimum_branching = None
        self.min_mutations = min_mutations

    def create_init_graph(self, df):
        # for each gene, find the set of patients whom this gene has mutated in them and
        # its size of this set
        cases = pd.DataFrame(df.groupby('Hugo_Symbol')['case_id'].apply(set))
        cases = cases.append(pd.DataFrame(set(df.case_id.unique()), columns=['case_id'], index=['zero']))
        cases['count'] = cases['case_id'].apply(len)
```

```
    # drop low mutated genes if desired
    if self.min_mutations is not None:
        cases = cases[cases['count'] >= self.min_mutations]

    # for each pair of genes, find the size of the set of patients whom these two genes
    # has both mutated in them
    cases['key'] = 0
    cases['symbol'] = cases.index
    self.edge_list = cases[['case_id', 'symbol', 'key']].merge(cases[['case_id', 'symbol', 'key']],
                                                               how='outer', on='key')
    self.edge_list['count'] = \
        self.edge_list.apply(lambda row: len(row['case_id_x'] & row['case_id_y'])
                                                    , axis=1)
    self.edge_list.drop(['case_id_x', 'case_id_y', 'key'], axis=1, inplace=True)

    # compute the number of cases
    n_cases = len(df['case_id'].unique())

    # compute the edge weights of the graph using the formula:
    # w_ij = log(p_ij) - log(p_i + p_j) - log(p_j)
    self.edge_list['weight'] = self.edge_list.apply(
        lambda row: np.log(row['count'] / n_cases) -
                    np.log(cases.loc[row['symbol_x']]['count'] / n_cases +
                           cases.loc[row['symbol_y']]['count'] / n_cases) -
                    np.log(cases.loc[row['symbol_y']]['count'] / n_cases)
        , axis=1)

    # create a networkx graph from the edgelist we just created
    self.G = nx.from_pandas_edgelist(self.edge_list,
                                     source='symbol_x',
                                     target='symbol_y',
                                     edge_attr='weight',
                                     create_using=nx.DiGraph())
    return self.G

def find_branching(self):
    """
    Find the oncogenetic graph using Edmond's branching algorithm. Needs ``create_init_graph``
    function to be called before using this function.

    Returns
    -------
    branching: networkx.DiGraph
        the oncogenetic tree

    """
    self.optimum_branching = nx.algorithms.tree.branchings.maximum_spanning_arborescence(self.G)
    return self.optimum_branching
```

```
def fit(self, df):
    """
    Runs the Desper's algorithm on the provided data frame.

    Parameters
    ----------
    df: pandas.DataFrame
        The data matrix. Should have ``Hugo_Symbol`` and ``case_id`` columns.
    """
    self.create_init_graph(df)
    self.find_branching()

def draw(self,
         figsize=(20, 15),
         with_edges=True,
         node_color= 'white',
         node_size= 10000,
         font_color='black',
         font_size = 20,
         width_ = 3,
         arrowsize_ = 30,
         edge_label_size = 20,
         edge_node_color = 'black'
         ):  # TODO move this function to utils
    """
    Draws the oncogenetic tree. Needs ``find_branching`` to be called before drawing.
    """
    # set the figure size
    plt.figure(figsize=figsize)

    # save the edge weights for later use
    edge_weights = {(u, v): round(self.optimum_branching[u][v]['weight'], 2) for u, v in
                    self.optimum_branching.edges}
    # change all edge weights of the branching to 1 to get a nice hierarchical tree in drawing
    for u, v in self.optimum_branching.edges:
        self.optimum_branching[u][v]['weight'] = 1
    pos = nx.drawing.nx_pydot.pydot_layout(self.optimum_branching, prog='dot')
    nx.draw(self.optimum_branching,
            pos,
            with_labels = True,
            node_color = node_color,
            node_size = node_size,
            font_color = font_color,
            font_size = font_size,
            width = width_,
            arrowsize = arrowsize_,
            edgecolors = edge_node_color)
```

```
        if with_edges:
            nx.draw_networkx_edge_labels(self.optimum_branching,
                                         pos,
                                         edge_labels=edge_weights,
                                         font_size = edge_label_size)

        # reset the edge weights to their original weights
        for u, v in edge_weights:
            self.optimum_branching[u][v]['weight'] = edge_weights[(u, v)]
```

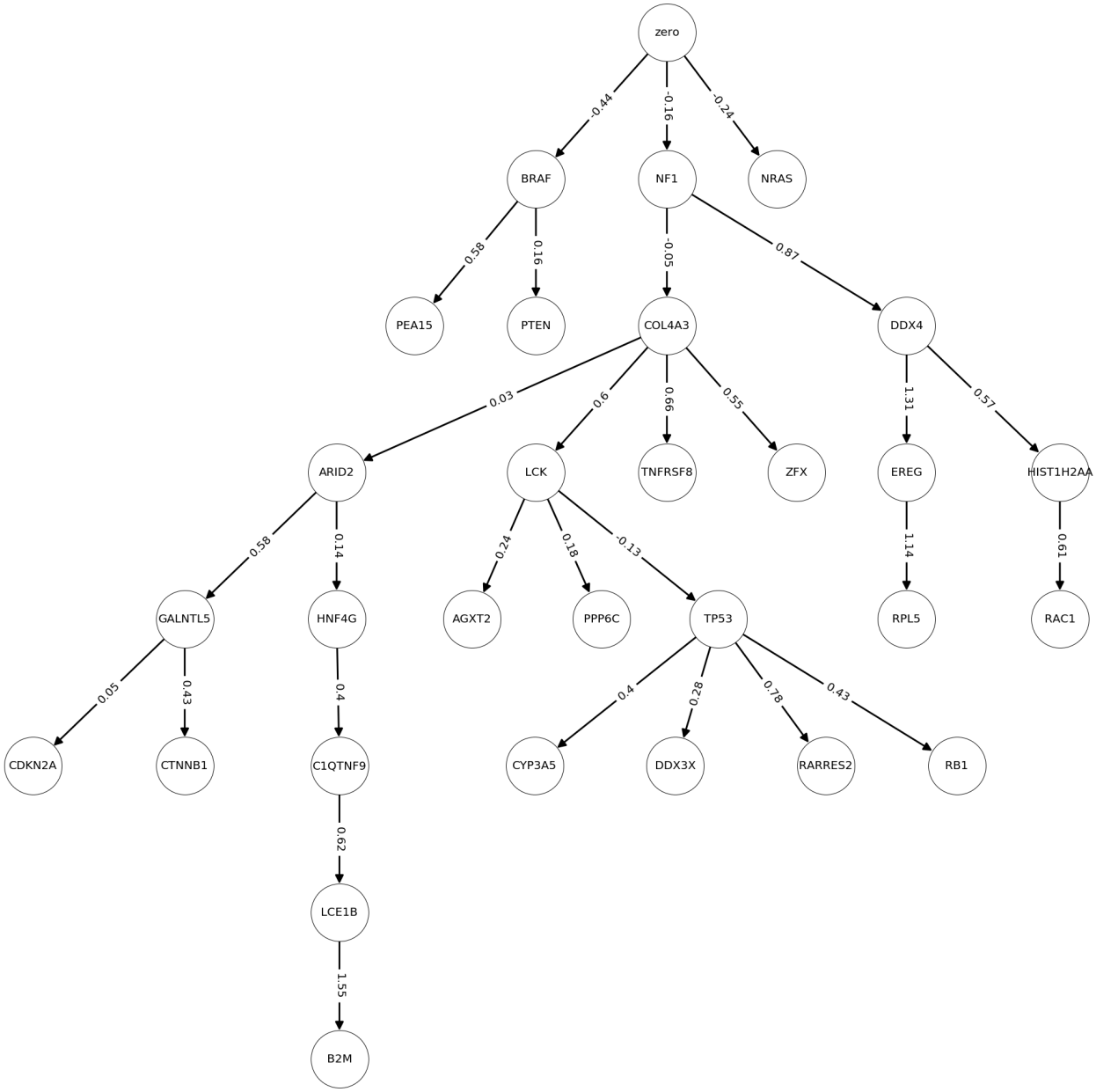
2.2 Test and Results

```
In [29]: from oncotree import OncoTree

         model = OncoTree()

         model.fit(melanoma)

In [30]: model.draw(figsize=(30, 30))
```


zero
-0.44
-0.16
-0.24
BRAF
NF1
NRAS
0.58
0.16
-0.05
0.87
PEA15
PTEN
COL4A3
DDX4
0.03
0.6
0.66
0.55
1.31
0.57
ARID2
LCK
TNFRSF8
ZFX
EREG
HIST1H2AA
0.58
0.14
0.24
0.18
-0.13
1.14
0.61
GALNTL5
HNF4G
AGXT2
PPP6C
TP53
RPL5
RAC1
0.05
0.43
0.4
0.4
0.28
0.78
0.43
CDKN2A
CTNNB1
C1QTNF9
CYP3A5
DDX3X
RARRES2
RB1
0.62
LCE1B
1.55
B2M

3 OncoTree with false positive and negetive

Algorithm 2 OncoTree with False Positive and Negative

input: Frequencies of mutations f_i, f_{ij}, a threshold t
output: Branching $T = (\mathcal{V}, \mathcal{E}, r, p)$
Compute edge weights: $w_{ij} \leftarrow \mathbf{1}(f_{ij} > t) \log \frac{f_{ij}}{f_j(f_i+f_j)}$
Form $G = (\mathcal{V}, \mathcal{W})$, where $\mathcal{V}$ and $\mathcal{W}$ are sets of all mutations and edge weights.
$\mathcal{S}_0 \leftarrow \emptyset, \mathcal{E}_0 \leftarrow \emptyset$
for $v = 1$ to $|\mathcal{V}|$ **do**
    $i \leftarrow \text{argmin}_{k \in \mathcal{V} \setminus \mathcal{S}_t} f_k, \quad j \leftarrow \text{argmax}_{k \in \mathcal{V} \setminus \mathcal{S}_t} w_{ki}$
    $\mathcal{S}_{t+1} \leftarrow \mathcal{S}_t \cup j, \quad \mathcal{E}_{t+1} \leftarrow \mathcal{E}_{t+1} \cup (j, i)$
end for
Set root r to the node without parent.
Form the branching $T = (\mathcal{V}, \mathcal{E}_{|\mathcal{V}|}, r)$
Compute edge probabilities of branching as: $p_{j|i} \leftarrow \frac{f_{ij}}{f_i}$.

3.1 Implementation

```
class OncoTreeWithFpFn(object):
    """
    The tree model for oncogenesis by Szabo, Aniko, et al.

    This implementation is based on:

        Szabo, A., & Boucher, K. (2002). Estimating an oncogenetic treewhen false negatives
        and positivesare present. Mathematical biosciences, 176(2), 219-236.

    Parameters
    ----------
    min_mutations : integer or None, optional (default=None)
        Minimum number of mutations to consider. The genes which has mutated below this number
        will be dropped. If ``None``, no mutation is skipped.
    """

    def __init__(self, min_mutations=None):
        self.edge_list = None
        self.opontimum_branching = None
        self.min_mutations = min_mutations
        self.mutation_freq = None
        self.branching_edge = None
```

```
def create_init_graph(self, df):
    # for each gene, find the set of patients whom this gene has mutated
    # in them and its size of this set
    cases = pd.DataFrame(df.groupby('Hugo_Symbol')['case_id'].apply(set))
    cases = cases.append(pd.DataFrame(set(df.case_id.unique()),
                                      columns=['case_id'],
                                      index=['zero']))
    cases['count'] = cases['case_id'].apply(len)

    # drop low mutated genes if desired
    if self.min_mutations is not None:
        cases = cases[cases['count'] >= self.min_mutations]

    # for each pair of genes, find the size of the set of patients whom these
    # two genes has both mutated in them
    cases['key'] = 0
    cases['symbol'] = cases.index
    self.edge_list = cases[['case_id', 'symbol', 'key']].merge(cases[['case_id',
                                                                      'symbol',
                                                                      'key']],
                                                               how='outer', on='key')
    self.edge_list['count'] = \
        self.edge_list.apply(lambda row: len(row['case_id_x'] & row['case_id_y'])
                                                 , axis=1)
    self.edge_list.drop(['case_id_x', 'case_id_y', 'key'], axis=1, inplace=True)

    # compute the number of cases
    n_cases = len(df['case_id'].unique())
    self.edge_list = self.edge_list[self.edge_list['symbol_x'] != self.edge_list['symbol_y']]

    # compute the edge weights of the graph using the formula:
    # w_ij = log(p_ij) - log(p_i + p_j) - log(p_j)
    self.edge_list['weight'] = self.edge_list.apply(
        lambda row: np.log(row['count'] / n_cases) -
                    np.log(cases.loc[row['symbol_x']]['count'] / n_cases +
                           cases.loc[row['symbol_y']]['count'] / n_cases) -
                    np.log(cases.loc[row['symbol_y']]['count'] / n_cases)
        , axis=1)
    # a list of mutation sorted by frequency
    self.mutation_freq = list(cases.sort_values('count')['symbol'])

def find_branching(self):

    """

    find the minimum branching tree by finding the parent of every nodes
```

```
    :return: optimum branching
    """
    tree = list()
    all_edge = self.edge_list.copy()
    all_edge.sort_values('weight', ascending=False, inplace=True)
    for i in self.mutation_freq:
        try:
            tree.append(all_edge[all_edge.symbol_y == i].iloc[0])
            all_edge = all_edge[all_edge.symbol_x != i]
        except:
            print("Warning: there is a mutation that has no parent.")
    self.branching_edge = pd.DataFrame(tree)
    self.optimum_branching = nx.from_pandas_edgelist(self.branching_edge,
                                                     source='symbol_x',
                                                     target='symbol_y',
                                                     edge_attr='weight',
                                                     create_using=nx.DiGraph())

    return self.optimum_branching

def fit(self, df):
    """
    Runs algorithm on the provided data frame.

    :param df: pandas.DataFrame
        The data matrix. Should have ``Hugo_Symbol`` and ``case_id`` columns.
    """
    self.create_init_graph(df)
    self.find_branching()

def draw(self,
         figsize=(20, 15),
         with_edges=True,
         node_color= 'white',
         node_size= 10000,
         font_color='black',
         font_size = 20,
         width_ = 3,
         arrowsize_ = 30,
         edge_label_size = 20,
         edge_node_color = 'black'
         ):  # TODO move this function to utils
    """
    Plot the graph

    :param figsize:
    :param weight: show the weight of edge
    """
```

```
        # set the figure size
        plt.figure(figsize=figsize)

        # save the edge weights for later use
        edge_weights = {(u, v): round(self.optimum_branching[u][v]['weight'], 2) for u, v in
                        self.optimum_branching.edges}
        # change all edge weights of the branching to 1 to get a nice hierarchical tree in drawing
        for u, v in self.optimum_branching.edges:
            self.optimum_branching[u][v]['weight'] = 1

        pos = nx.drawing.nx_pydot.pydot_layout(self.optimum_branching, prog='dot')
        nx.draw(self.optimum_branching,
                pos,
                with_labels=True,
                node_color=node_color,
                node_size=node_size,
                font_color=font_color,
                font_size=font_size,
                width=width_,
                arrowsize=arrowsize_,
                edgecolors=edge_node_color)

        if with_edges:
            nx.draw_networkx_edge_labels(self.optimum_branching,
                                         pos,
                                         edge_labels=edge_weights,
                                         font_size=edge_label_size)
        # reset the edge weights to their original weights
        for u, v in edge_weights:
            self.optimum_branching[u][v]['weight'] = edge_weights[(u, v)]
```

3.2 Test and Results

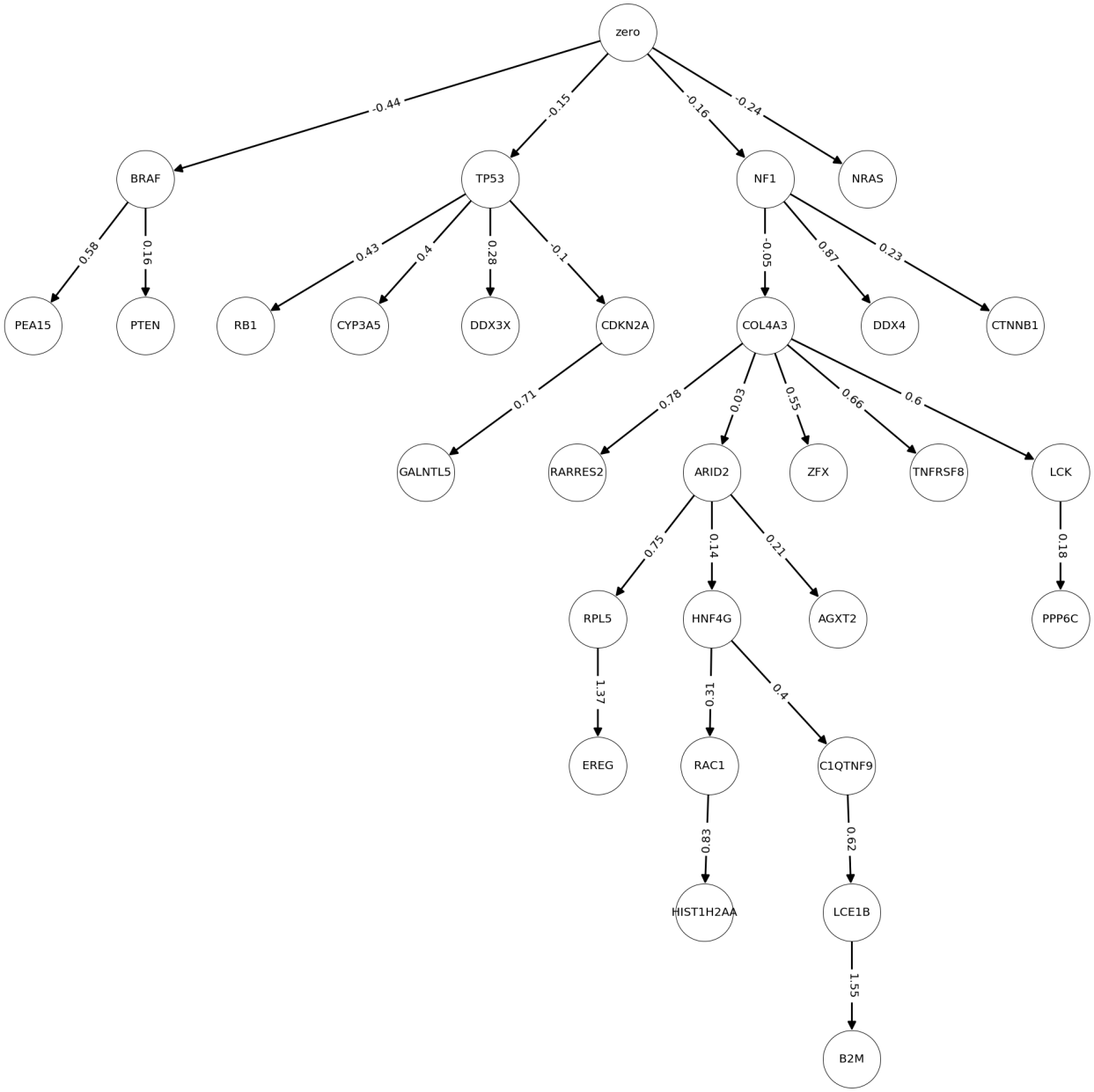
```
In [31]: from oncotreewithfpfn import OncoTreeWithFpFn

         model = OncoTreeWithFpFn()

         model.fit(melanoma)
```

```
../imo/oncotreewithfpfn.py:58: RuntimeWarning: divide by zero encountered in log
  cases.loc[row['symbol_y']]['count'] / n_cases) -
```

```
In [32]: model.draw(figsize=(30, 30))
```


zero
-0.44
-0.15
-0.16
-0.24
BRAF
TP53
NF1
NRAS
0.58
0.16
0.43
0.4
0.28
-0.1
-0.05
0.87
0.23
PEA15
PTEN
RB1
CYP3A5
DDX3X
CDKN2A
COL4A3
DDX4
CTNNB1
0.71
0.78
0.03
0.55
0.66
0.6
GALNTL5
RARRES2
ARID2
ZFX
TNFRSF8
LCK
0.75
0.14
0.21
0.18
RPL5
HNF4G
AGXT2
PPP6C
1.37
0.31
0.4
EREG
RAC1
C1QTNF9
0.83
0.62
HIST1H2AA
LCE1B
1.55
B2M

4 CBN

Algorithm 4 CBN: Conjunctive Bayesian Network

input: Input binary data matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$, a threshold t
output: CBN $= (\mathcal{V}, \hat{\mathcal{E}}, \hat{\Theta})$ and $\hat{\lambda}$
{Merging indistinguishable events}
for $u = 1$ to p **do**
  for $v = u + 1$ to p **do**
    if Events u and v always co-occur **then**
      Lump them together as a new event uv and decrease p by one.
    end if
  end for
end for
{Learning the structure of the DAG}
for $u = 1$ to p **do**
  for $v = u + 1$ to p **do**
    $\mathbf{1}_{uv} = \mathbf{1}_u + \mathbf{1}_v$
    if $\sum_{i=1}^{n} \left[(\mathbf{x}_i \cap \mathbf{1}_{uv}) == \mathbf{1}_u\right] \leq t \times n$ **then**
      $\hat{\mathcal{E}} \leftarrow \hat{\mathcal{E}} \cup \{(v, u)\}$
    end if
  end for
end for
{Learning the model's parameters}
Form a new input matrix $\mathbf{X}^{'} \in \mathbb{R}^{n^{'} \times p}$ by removing the incompatible observations $\mathbf{x}_i$ where $(\mathbf{x}_i \cap \mathbf{1}_{uv}) == \mathbf{1}_u$ and $(u, v) \in \hat{\mathcal{E}}$.
for $v = 1$ to $|\mathcal{V}|$ **do**
  $\hat{\theta}_v = \frac{\sum_{i=1}^{n^{'}} (\mathbf{x}_i \cap \mathbf{1}_v)}{\sum_{i=1}^{n^{'}} (\mathbf{x}_i \cap \mathbf{1}_{\text{pa}(v)})}$
end for
$\hat{\lambda} = \frac{n^{'}}{n}$

4.1 Implementation

```
class CBN(object):

    def __init__(self, epsilon=None, genes=None):
        self.epsilon = epsilon
        if epsilon is None:
            self.epsilon = 0
        self.events = genes
        self.G = None

    @classmethod
    def __df_to_prob_dist(cls, df):
        return df.groupby('case_id')['Hugo_Symbol'].apply(set)

    def _reduce_events(self, u):
        pass

    def _find_structure(self, u):
        self.G = nx.DiGraph()
        for f in self.events:
            for e in self.events:
                n_errors = u.apply(lambda g: g & {e, f} == {f}).sum()
                if n_errors <= self.epsilon:
                    self.G.add_edge(e, f)
        return self.G

    def _find_probabilities(self, u):
        theta = {}
        nodes = np.copy(self.G.nodes)
        for e in nodes:
            below = u.apply(lambda g: set(self.G.predecessors(e)).issubset(g)).sum()
            if below == 0:
                self.G.remove_node(e)
            else:
                theta[e] = round(u.apply(lambda g: e in g).sum() / below, 3)
        nx.set_node_attributes(self.G, theta, 'theta')
        return theta

    def fit(self, X):
        all_genes = X['Hugo_Symbol'].unique()
        u = CBN.__df_to_prob_dist(X)

        if self.events is None:
            self.events = all_genes
        self.events = [e for e in self.events if e in all_genes]
        if 0 < self.epsilon < 1:
            self.epsilon = int(u.shape[0] * self.epsilon)
```

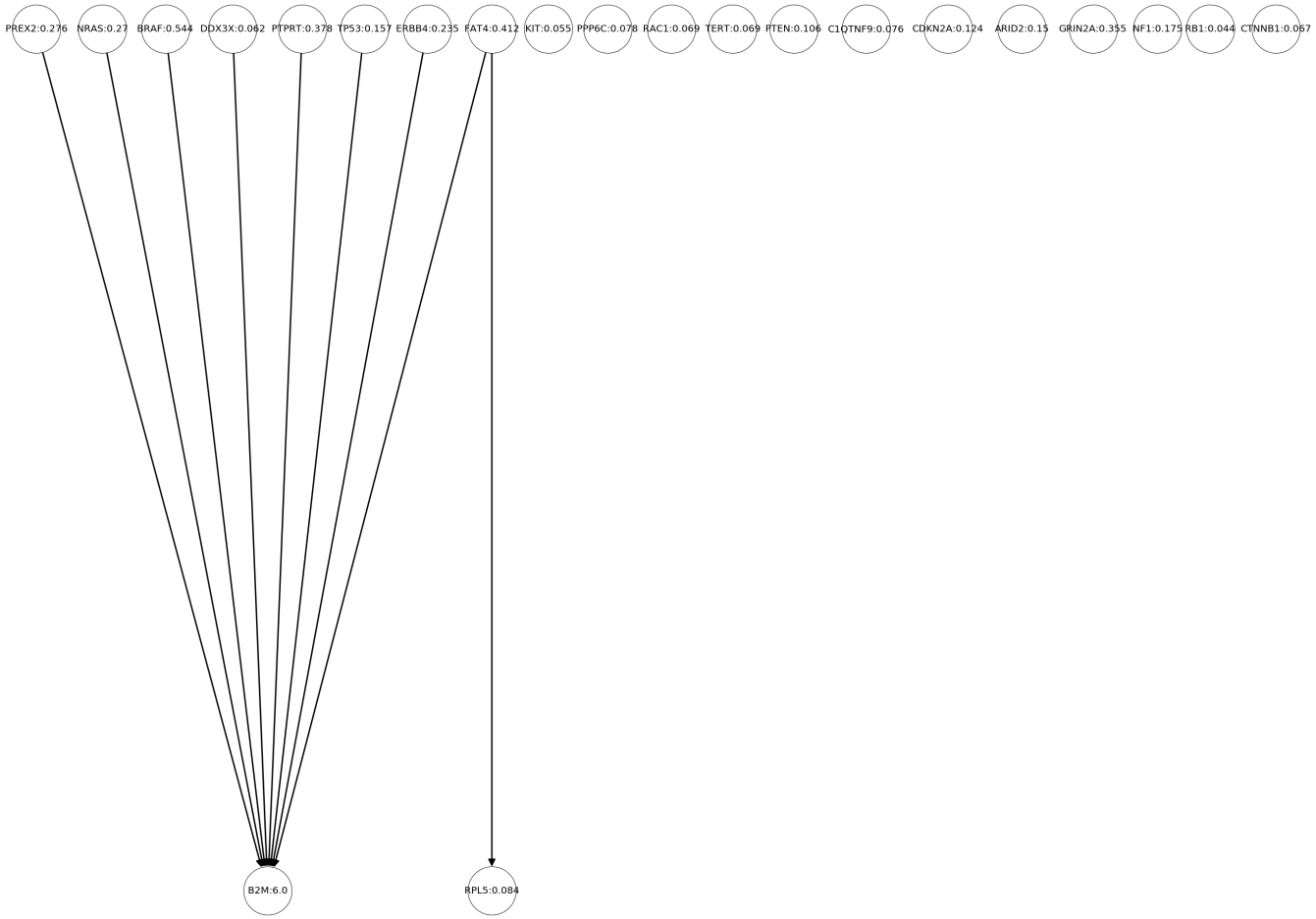
```
        self._reduce_events(u)
        self._find_structure(u)
        self._find_probabilities(u)

    def draw(self,
             figsize=(20, 15),
             with_node_probs=True,
             node_color='white',
             node_size=10000,
             font_color='black',
             font_size=20,
             width_=3,
             arrowsize_=30,
             edge_label_size=20,
             edge_node_color='black'):
        plt.figure(figsize=figsize)
        pos = nx.drawing.nx_pydot.pydot_layout(self.G, prog='dot')
        labels = {}
        if with_node_probs:
            for e in self.G.nodes:
                labels[e] = e + ':' + str(nx.get_node_attributes(self.G, 'theta')[e])
        nx.draw(self.G,
                pos,
                labels=labels,
                node_color=node_color,
                node_size=node_size,
                font_color=font_color,
                font_size=font_size,
                width=width_,
                arrowsize=arrowsize_,
                edgecolors=edge_node_color)
```

4.2 Test and Results

```
In [9]: from cbn import CBN
        model = CBN(epsilon= 0.01, genes=driver_genes)
        model.fit(melanoma)

In [10]: model.draw(font_size= 20, figsize=(40,30))
```


PREX2:0.276
NRAS:0.27
BRAF:0.544
DDX3X:0.062
PTPRT:0.378
TP53:0.157
ERBB4:0.235
FAT4:0.412
KIT:0.055
PPP6C:0.078
RAC1:0.069
TERT:0.069
PTEN:0.106
C1QTNF9:0.076
CDKN2A:0.124
ARID2:0.15
GRIN2A:0.355
NF1:0.175
RB1:0.044
CTNNB1:0.067
B2M:6.0
RPL5:0.084

5 CAPRI

Algorithm 5 CAPRI: CAncerPRogression Inference

input: Input binary data matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$.
output: DAGs $\mathcal{D} = (\mathcal{V}, \mathcal{E}, \alpha)$.
for i in $\mathcal{V}$ **do**
  for $j = 0$ in $\mathcal{V}$ **do**
    if $\mathbb{P}(i) > \mathbb{P}(j)$ & $\mathbb{P}(i|j) > \mathbb{P}(i|\bar{i})$ **then**
      Add arc (i, j) to edge list. $\mathcal{E} \leftarrow \mathcal{E} \cup (i, j)$
    end if
  end for
end for
$\forall i \in \mathcal{V}$ Define $\alpha(i)$ as follow:

$$\alpha(i) = \begin{cases} \mathbb{P}(i) & if \pi(i) = \emptyset \\ \mathbb{P}(i|j_1, j_2, ..., J_n) & if \pi(i) = \{j_1, j_2, ..., j_n\} \end{cases}$$

Fillter out all false positive relation by likelihood fit with regularization BIC or AIC score and set $\alpha(j) = 0$ for each removed edge.

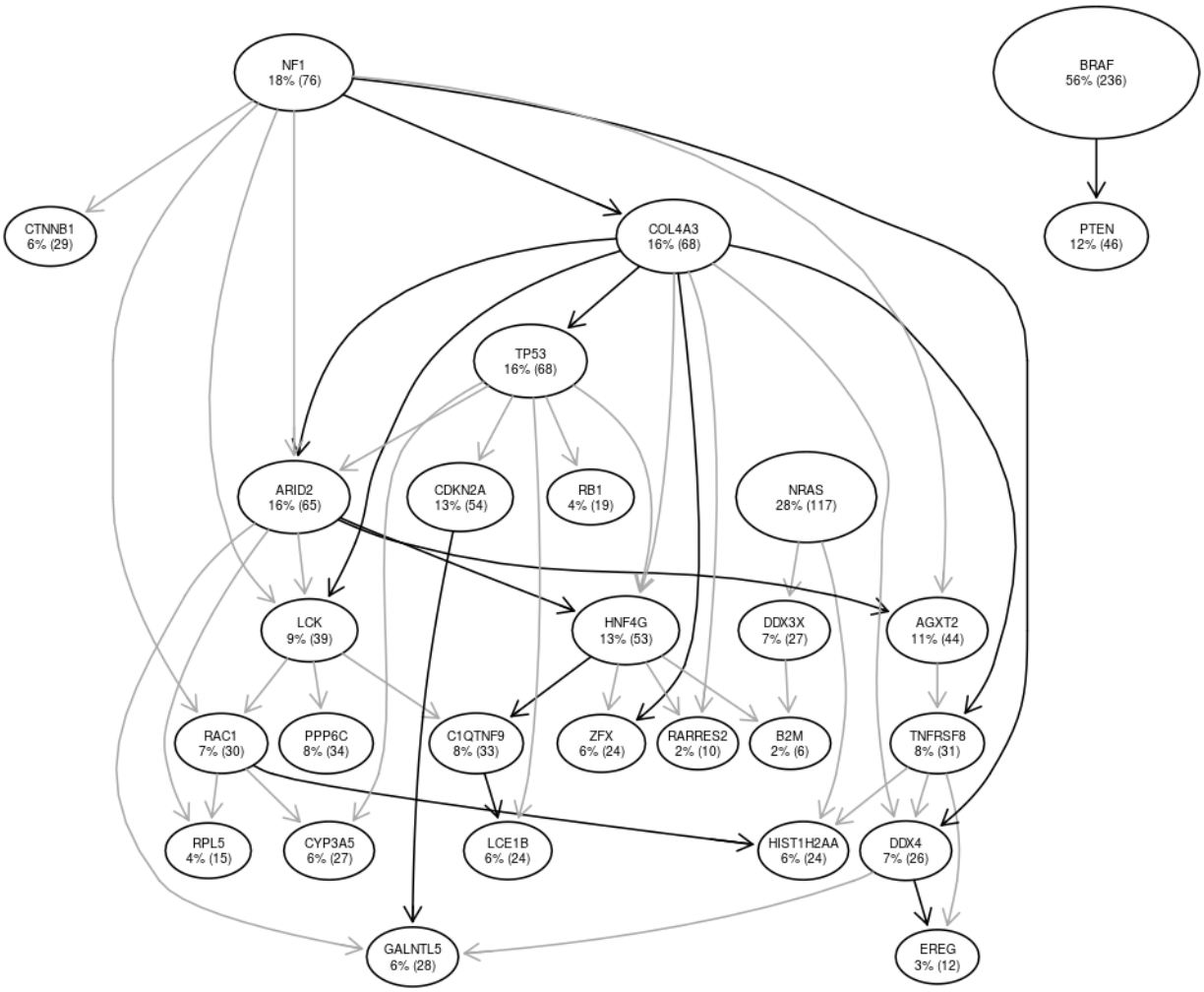
5.0.1 for this method we are using the implementation of the original papers. we add only a wrapper for python.

```
In [33]: from capri import CAPRI
```

5.1 Test and Results

```
In [34]: model = CAPRI()
         model.fit(melanoma, driver_gens=driver_genes)
         print('CAPRI')

In [35]: model.draw()
```



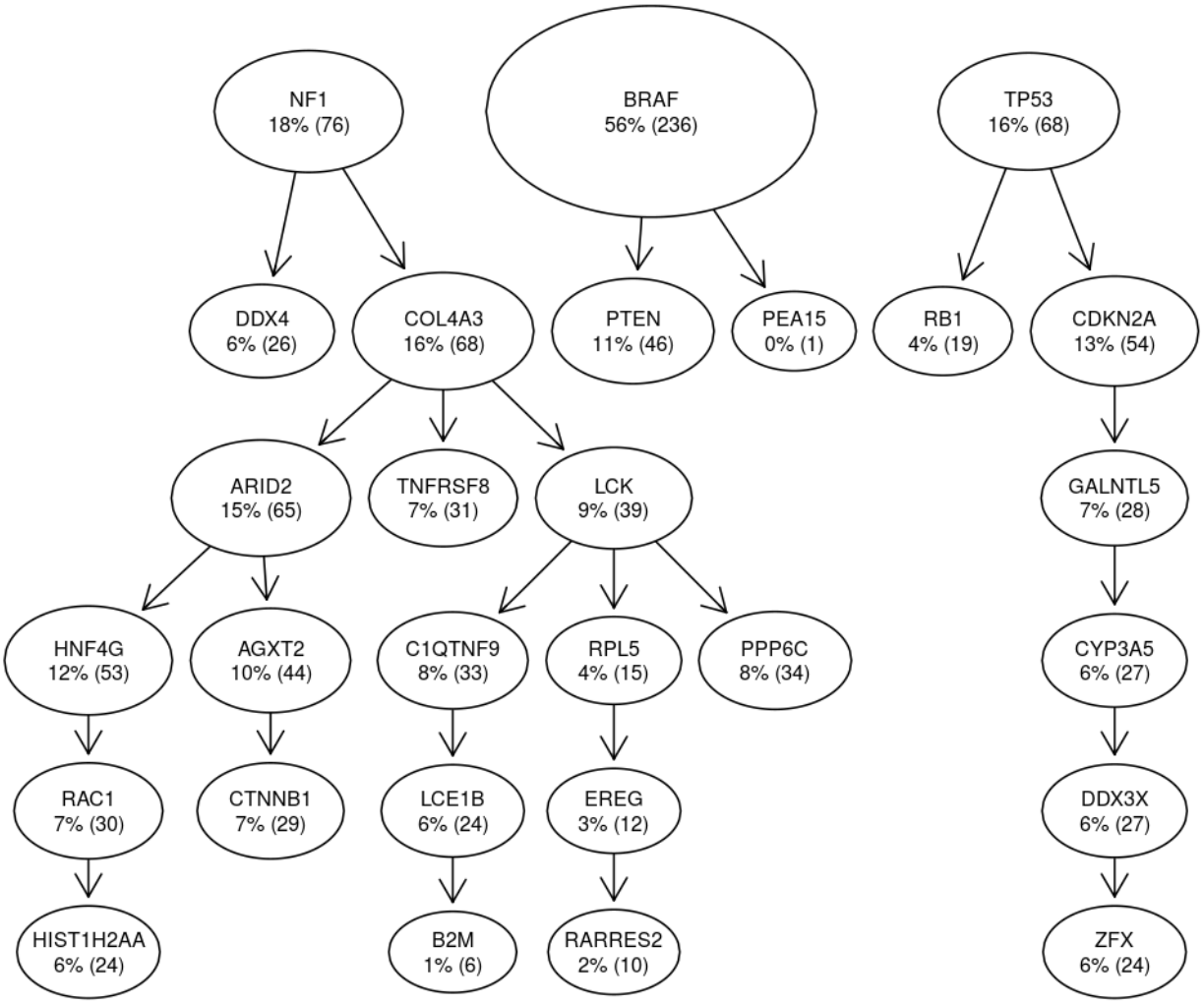
5.1.1 for this method we are using the implementation of the original papers. we add only a wrapper for python.

6 Caprese

6.1 Test and Results

```
In [36]: from caprese import CAPRESE

In [37]: model = CAPRESE()
         model.fit(melanoma, driver_gens=driver_genes)
         print('CAPRESE')
```



7 Mixtree

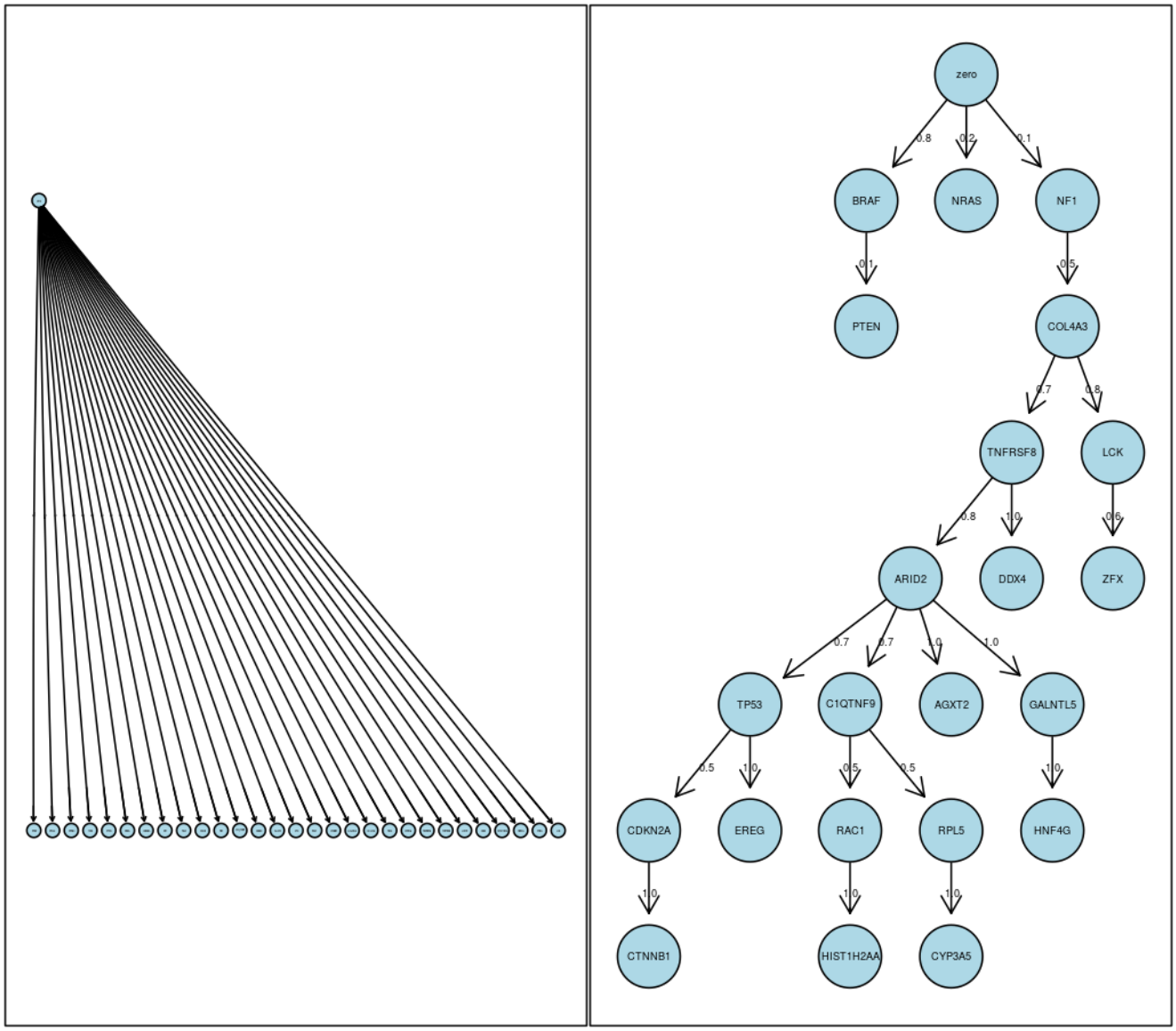
7.0.1 for this method we are using the implementation of the original papers. we add only a wrapper for python.

7.1 Test and Results

```
In [39]: from treemix import Treemix

In [40]: model = Treemix()
         model.fit(melanoma, driver_gens= driver_genes)

In [41]: model.draw()
```


zero
0.8
0.2
0.1
BRAF
NRAS
NF1
0.1
0.5
PTEN
COL4A3
0.7
0.8
TNFRSF8
LCK
0.8
1.0
0.6
ARID2
DDX4
ZFX
0.7
0.7
1.0
1.0
TP53
C1QTNF9
AGXT2
GALNTL5
0.5
1.0
0.5
0.5
1.0
CDKN2A
EREG
RAC1
RPL5
HNF4G
1.0
1.0
1.0
CTNNB1
HIST1H2AA
CYP3A5