In [8]:
```
from covid.simulator import Population
from covid.auxilliary import symptom_names
import numpy as np
import pandas as pd

from covid.policy import Policy
```

In [65]:
```
symptom_names = ['covid_recovered','covid_positive', 'no_taste_smell',
'fever','headache', 'pneumonia','stomach','myocarditis', 'blood_clots','death']
```

In [9]:
```
w = np.array([0.2, 0.1, 0.1, 0.1, 0.5, 0.2, 0.5, 1.0, 100])
assert w.shape[0] == 9, 'Shape of weights does not fit number of symptoms'
```

In [10]:
```
class Model:
    def __init__(self, nsymptom, nvacc):
        # Priors for the beta-bernoulli model
        self.a = np.ones(shape=[nvacc, nsymptom])/2 # using jeffreys prior
        self.b = np.ones(shape=[nvacc, nsymptom])/2 # using jeffreys prior
        self.nvacc = nvacc
        self.nsymptom = nsymptom

    def update(self, features, actions, outcomes):
        """
        for index in range(self.nvacc):
            print(outcomes[np.where(actions == (index-1))])
            print(actions)
            print(outcomes)
            if (np.sum(outcomes[np.where(actions == index - 1)], axis=1).size != 0):
                self.a[index] += np.sum(outcomes[np.where(actions == index - 1)], axis=1)
                self.b[index] += np.sum(outcomes[np.where(actions == index - 1)]==0, axis=1)\
                    - np.sum(outcomes[np.where(actions == index - 1)], axis=1)
            else:
                self.b[index] += np.sum((outcomes==0)[np.where(actions == index - 1)], axis=1)
        """
        for index, outcome in enumerate(outcomes):
            self.a[int(actions[index])] += outcome[1:]
            self.b[int(actions[index])] += 1 - outcome[1:]

    def get_params(self):
        return self.a, self.b


    def get_prob(self, features, action):
        return self.a[action] / (self.a[action] + self.b[action])


    def retrain(self, features, actions, outcomes):
        # Use aggregated database of outcomes etc...

        self.a = np.ones(shape=[self.nvacc, self.nsymptom])/2 # using jeffreys prior
        self.b = np.ones(shape=[self.nvacc, self.nsymptom])/2 # using jeffreys prior

        self.update(features, actions, outcomes)
```

In [41]:
```
class Naive(Policy):
    def get_utility(self, features, action, outcome):
        utility = 0
        for t, o in enumerate(outcome):
            utility -= np.dot(w, o[1:])*(1+int(action[t] != -1))

        return utility

    def set_model(self, model):
        self.model = model

    def get_action(self, features):
        """Get a completely random set of actions, but only one for each individual.

        If there is more than one individual, feature has dimensions t*x matrix, otherwise it is an x-s
ize array.

        It assumes a finite set of actions.

        Returns:
        A t*|A| array of actions
        """
        n_obs = features.shape[0]
        actions = np.zeros(n_obs)
        #u_list = np.zeros((n_obs, self.n_actions))
        for index, t in enumerate(features):
            u_list = []
            for a in self.action_set:
                u_list.append(self.get_expected_utility(a, t))
            actions[index] = np.argmax(np.array(u_list))
        return actions

    def get_expected_utility(self, action, features):
        p = self.model.get_prob(features, action)
        return -np.dot(p, w)*(1+int(action != -1))

    def observe(self, features, action, outcomes):
        self.model.update(features, action, outcomes)
```

In [52]:
```
## Baseline simulator parameters
n_genes = 128
n_vaccines = 3
n_treatments = 4
n_population = 100000
n_symptoms = 9

# symptom names for easy reference
from covid.auxilliary import symptom_names
```

In [53]:
```
population = Population(n_genes, n_vaccines, n_treatments)
X = population.generate(n_population)
n_features = X.shape[1]
```

```
<__array_function__ internals>:5: VisibleDeprecationWarning: Creating an ndarray from ragged nested s
equences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) i
s deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray.
```

In [54]:
```
print("With a for loop")
vaccine_policy = Naive(2, np.array([-1, 0]))
vaccine_policy.set_model(Model(n_symptoms, 2))
# The simplest way to work is to go through every individual in the population
Y = np.zeros((n_population, n_symptoms+1))
A = np.zeros(n_population)
for t in range(n_population):
    #print("Person nr: ", t)
    a_t = vaccine_policy.get_action(X[t].reshape((1, n_features)))
    A[t] = a_t
    # Then you can obtain results for everybody
    y_t = population.vaccinate([t], a_t.reshape((1, 1)))
    Y[t] = y_t
    # Feed the results back in your policy. This allows you to fit the
    # statistical model you have.
    vaccine_policy.observe(X[t], a_t, y_t)
```

```
With a for loop
Initialising policy with  2 actions
A = { [-1  0] }
```

In [61]:
```
print(len(symptom_names))
```

```
10
```

In [68]:
```
def print_pre_statistics(X):
    print(f'Statistic (N={X.shape[0]})')
    for i in range(len(symptom_names)-1):
        print(f'{symptom_names[i].ljust(15)} {X[:, i].sum()}')
```

In [75]:
```
print_pre_statistics(Y)
print(A)
```

```
Statistic (N=100000)
covid_recovered 0.0
covid_positive  0.0
no_taste_smell  0.0
fever           100000.0
headache        0.0
pneumonia       0.0
stomach         0.0
myocarditis     0.0
blood_clots     0.0
[0. 0. 1. ... 0. 0. 0.]
```

In [76]:
```
a, b = vaccine_policy.model.get_params()
print(a[0]/(a[0] + b[0]))
print(a[1]/(a[1] + b[1]))
```

```
[5.02542867e-06 5.02542867e-06 9.99994975e-01 5.02542867e-06
 5.02542867e-06 5.02542867e-06 5.02542867e-06 5.02542867e-06
 5.02542867e-06]
[9.84251969e-04 9.84251969e-04 9.99015748e-01 9.84251969e-04
 9.84251969e-04 9.84251969e-04 9.84251969e-04 9.84251969e-04
 9.84251969e-04]
```