functions

December 9, 2021

1 Functions for plotting and outputs

1.0.1 Comment

This file originally contained both functions and scripts.

To simplify we put all the code into functions. Some of then doesn't run since some code inbetween was remove.

[39]:
```
from covid.simulator import Population
from covid.auxilliary import symptom_names
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from covid.policy import Policy
from scipy.stats import beta, bernoulli, uniform
```

[40]:
```
class ThompsonSampling:
    """Thompson sampling"""
    def __init__(self, nvacc):
        # Priors for the beta-bernoulli model
        self.a = np.ones(nvacc) # uniform prior
        self.b = np.ones(nvacc) # uniform prior
        self.nvacc = nvacc

    def update(self, action, outcome):
        self.a[action] += outcome
        self.b[action] += 1 - outcome

    def update_with_laplace(self, action, outcome, epsilon=1):
        """Alternative update model with added Laplace noise"""
        outcome = outcome + np.random.laplace(0, 1/epsilon)
        self.a[action] += outcome
        self.b[action] += 1 - outcome

    def get_params(self):
        # Returns the parameters of all the beta distrobutions.
```

```
        return self.a, self.b

    def get_prob(self):
        for i in range(self.nvacc):
            if self.a[i] <= 1:
                self.a[i] = 1
            if self.b[i] <= 1:
                self.b[i] = 1
        return beta.rvs(self.a, self.b)
```

[41]:
```
class Naive(Policy):
    def set_model(self, model):
        self.model = model

    def get_action(self):
        probs = self.model.get_prob()

        #print(probs)
        ret_val = np.argmin(probs)
        #print(ret_val)
        #print(ret_val)
        return ret_val

    def observe(self, action, outcome):
        self.model.update(action, outcome)

    def observe_with_laplace(self, action, outcome, epsilon):
        self.model.update_with_laplace(action, outcome, epsilon)
```

[42]:
```
def compare_laplace():
    """Comparing different privacy guarantees."""

    action_space = np.array([-1,0,1,2])
    n_actions = action_space.shape[0]

    epsilons = [100, 10, 1, 0.1, 0.01, 0.001]

    alphas = np.zeros((len(epsilons),4))
    betas = np.zeros((len(epsilons),4))

    for i in range(len(epsilons)):
        n_genes = 128
        n_vaccines = 3 # DO NOT CHANGE, breaks the simulator.
        n_treatments = 4
        n_population = 10_000
        n_symptoms = 10
```

```
        #np.random.seed(2)
        population = Population(n_genes, n_vaccines, n_treatments)
        X = population.generate(n_population)
        n_features = X.shape[1]

        policy = Naive(n_actions, action_space)
        model = ThompsonSampling(n_actions)
        policy.set_model(model)
        action_arr = np.zeros(n_population)
        symptom_arr = np.zeros((n_population, n_symptoms+1))

        for t in range(n_population):
            a_t = policy.get_action()
            y_t = population.vaccinate([t], a_t.reshape((1, 1)))
            new_col = y_t[:,[5,7,8]].sum(axis=1)
            new_col0 = new_col > 0
            new_col01 = new_col0.astype(int)
            new_col01 = np.reshape(new_col01,(1,-1))
            y_t_new = np.hstack((y_t,new_col01))
            policy.observe_with_laplace(a_t, y_t_new[:,10], epsilon=epsilons[i])
            action_arr[t] = a_t
            symptom_arr[t] = y_t_new

        alpha, beta = policy.model.get_params()

        print(i, alpha, beta)
        alphas[i,:] = alpha
        betas[i,:] = beta
    return alphas, betas
```

[43]:
```
def privacy_output():
    alphas_comp, betas_comp = compare_laplace()
    epsilons = [100, 10, 1, 0.1, 0.01, 0.001]
    utility_comp = alphas_comp.sum(axis=1)
    out = f'========= Utility when privacy guarantee changes ======\n'
    out += f'Epsilon: {100:5}, Accumulated utility: {utility_comp[0]:11.1f}\n'
    out += f'Epsilon: {10:5}, Accumulated utility: {utility_comp[1]:11.1f}\n'
    out += f'Epsilon: {1:5}, Accumulated utility: {utility_comp[2]:11.1f}\n'
    out += f'Epsilon: {0.1:5}, Accumulated utility: {utility_comp[3]:11.1f}\n'
    out += f'Epsilon: {0.01:5}, Accumulated utility: {utility_comp[4]:11.1f}\n'
    out += f'Epsilon: {0.001:5}, Accumulated utility: {utility_comp[5]:11.1f}\n'
    out += f'======================================================\n'
    with open('outputs/utility_comp.txt', 'w') as outfile:
        outfile.write(out)
```

[44]:
```
def expected_utility(samples=10):
    action_space = np.array([-1,0,1,2])
```

```
    n_actions = action_space.shape[0]

    alphas = np.zeros((samples,4))
    betas = np.zeros((samples,4))

    for i in range(samples):
        n_genes = 128
        n_vaccines = 3 # DO NOT CHANGE, breaks the simulator.
        n_treatments = 4
        n_population = 10_000
        n_symptoms = 10
        #np.random.seed(2)
        population = Population(n_genes, n_vaccines, n_treatments)
        X = population.generate(n_population)
        n_features = X.shape[1]

        policy = Naive(n_actions, action_space)
        model = ThompsonSampling(n_actions)
        policy.set_model(model)
        action_arr = np.zeros(n_population)
        symptom_arr = np.zeros((n_population, n_symptoms+1))

        for t in range(n_population):
            a_t = policy.get_action()
            y_t = population.vaccinate([t], a_t.reshape((1, 1)))
            new_col = y_t[:,[5,7,8]].sum(axis=1)
            new_col0 = new_col > 0
            new_col01 = new_col0.astype(int)
            new_col01 = np.reshape(new_col01,(1,-1))
            y_t_new = np.hstack((y_t,new_col01))
            policy.observe(a_t, y_t_new[:,10])
            action_arr[t] = a_t
            symptom_arr[t] = y_t_new

        alpha, beta = policy.model.get_params()

        print(i, alpha, beta)
        alphas[i,:] = alpha
        betas[i,:] = beta
    return alphas, betas
```

[45]:
```
def expected_utility_laplace(samples=100):
    action_space = np.array([-1,0,1,2])
    n_actions = action_space.shape[0]

    alphas = np.zeros((samples,4))
    betas = np.zeros((samples,4))
```

```
    for i in range(samples):
        n_genes = 128
        n_vaccines = 3 # DO NOT CHANGE, breaks the simulator.
        n_treatments = 4
        n_population = 10_000
        n_symptoms = 10
        #np.random.seed(2)
        population = Population(n_genes, n_vaccines, n_treatments)
        X = population.generate(n_population)
        n_features = X.shape[1]

        policy = Naive(n_actions, action_space)
        model = ThompsonSampling(n_actions)
        policy.set_model(model)
        action_arr = np.zeros(n_population)
        symptom_arr = np.zeros((n_population, n_symptoms+1))

        for t in range(n_population):
            a_t = policy.get_action()
            y_t = population.vaccinate([t], a_t.reshape((1, 1)))
            new_col = y_t[:,[5,7,8]].sum(axis=1)
            new_col0 = new_col > 0
            new_col01 = new_col0.astype(int)
            new_col01 = np.reshape(new_col01,(1,-1))
            y_t_new = np.hstack((y_t,new_col01))
            policy.observe_with_laplace(a_t, y_t_new[:,10], epsilon=1)
            action_arr[t] = a_t
            symptom_arr[t] = y_t_new

        alpha, beta = policy.model.get_params()

        print(i, alpha, beta)
        alphas[i,:] = alpha
        betas[i,:] = beta
    return alphas, betas
```

[46]:
```
def laplace_histogram():
    alphas_laplace, betas_laplace = expected_utility_laplace(100)
    utility_laplace = alphas_laplace.sum(axis=1)
    plt.hist(-utility_laplace, bins=30)
    plt.title('Histogram utility policy with laplace noise (1/epsilon)')
    mean_laplace = np.mean(utility_laplace)
    std_laplace = np.std(utility_laplace)
    xlabel_str_laplace = f"Utility: Mean {-mean_laplace:.4f}, Std {std_laplace:.
 ↪4f}"
    plt.xlabel(xlabel_str_laplace)
```

```
    plt.ylabel('Frequency')
    plt.savefig('figures/histogram_utility_laplace.png')
    plt.show()
```

[47]:
```
def beta_comparision_short_scale_plot():
    from scipy.stats import beta
    x = np.linspace(0.034,0.038,500)
    y0 = beta.pdf(x, a[0], b[0])
    y1 = beta.pdf(x, a[1], b[1])
    y2 = beta.pdf(x, a[2], b[2])
    y3 = beta.pdf(x, a[3], b[3])
    plt.plot(x, y0, x, y1, x, y2, x, y3)
    plt.title('Beta distribution with average parameters over 100 runs')
    plt.xlabel('x')
    plt.ylabel('Density')
    plt.legend(['No vaccine', 'Vaccine 1', 'Vaccine 2', 'Vaccine 3'])
    plt.show()
```

[48]:
```
def beta_comparison_plot_long_scale():
    x = np.linspace(0,1,500)
    y0 = beta.pdf(x, a[0], b[0])
    y1 = beta.pdf(x, a[1], b[1])
    y2 = beta.pdf(x, a[2], b[2])
    y3 = beta.pdf(x, a[3], b[3])
    plt.plot(x, y0, x, y1, x, y2, x, y3)
    plt.title('Beta distribution with average parameters over 100 runs')
    plt.xlabel('x')
    plt.ylabel('Density')
    plt.legend(['No vaccine', 'Vaccine 1', 'Vaccine 2', 'Vaccine 3'])
    plt.show()
```

[49]:
```
def vaccination_comparison_plot():
    expectation = alphas/(alphas+betas)
    x0 = expectation[:,0]
    x1 = expectation[:,1]
    x2 = expectation[:,2]
    x3 = expectation[:,3]

    out = '======================================================\n'
    out += f'Expected utility per vaccine averaged over 100 runs:\n'
    out += f'No vaccine: {np.mean(x0):.4f}, std: {np.std(x0):.4f}\n'
    out += f'Vaccine 1:  {np.mean(x1):.4f}, std: {np.std(x1):.4f}\n'
    out += f'Vaccine 2:  {np.mean(x2):.4f}, std: {np.std(x2):.4f}\n'
    out += f'Vaccine 3:  {np.mean(x3):.4f}, std: {np.std(x3):.4f}\n'
    out += '======================================================'

    with open('outputs/expectation_per_vaccine.txt', 'w') as outfile:
```

```
        outfile.write(out)
```

[50]:
```
def utility_histogram():
    utility = alphas.sum(axis=1)
    plt.hist(-utility, bins=30)
    plt.title('Histogram utility improved policy')
    mean = np.mean(utility)
    std = np.std(utility)
    xlabel_str = f"Utility: Mean {-mean:.4f}, Std {std:.4f}"
    plt.xlabel(xlabel_str)
    plt.ylabel('Frequency')
    plt.savefig('figures/histogram_utility_improved_policy.png')
    plt.show()
```

[51]:
```
def plot_fair_balance(actions, data, symptom=0, save=False, name="figures/
↪Default.png"):
    mm_y = np.where(np.logical_and(data[:,11]==1, symptom_arr[:,10]==symptom))
    ff_y = np.where(np.logical_and(data[:,11]==0, symptom_arr[:,10]==symptom))

    m_a_y = np.take(actions, mm_y)
    m_a_y = m_a_y.flatten()

    f_a_y = np.take(actions, ff_y)
    f_a_y = f_a_y.flatten()

    title = "Vaccines per gender y=0" if not symptom else "Vaccines per gender␣
↪y=1"
    plt.hist([m_a_y, f_a_y],label=['male','female'])
    plt.xlabel('Vaccine')
    plt.ylabel('Frequency')
    plt.title(title)
    plt.legend(loc='upper right')
    if save:
        plt.savefig(name)
    plt.show()
```

[52]:
```
def fair_get_arrays(actions, data, symptom_arr):
    y0 = np.where(symptom_arr[:,10]==0)
    y1 = np.where(symptom_arr[:,10]==1)
    y0_m = np.where(np.logical_and(data[:,11]==1, symptom_arr[:,10]==0))
    y1_m = np.where(np.logical_and(data[:,11]==1, symptom_arr[:,10]==1))
    y0_f = np.where(np.logical_and(data[:,11]==0, symptom_arr[:,10]==0))
    y1_f = np.where(np.logical_and(data[:,11]==0, symptom_arr[:,10]==1))

    a_y0 = np.take(actions, y0).flatten()
    a_y1 = np.take(actions, y1).flatten()
    a_y0_m = np.take(actions, y0_m).flatten()
```

```
    a_y1_m = np.take(actions, y1_m).flatten()
    a_y0_f = np.take(actions, y0_f).flatten()
    a_y1_f = np.take(actions, y1_f).flatten()

    y0 = []
    y1 = []
    y0_m = []
    y1_m = []
    y0_f = []
    y1_f = []
    for i in range(4):
        y0.append(np.sum(a_y0==i))
        y1.append(np.sum(a_y1==i))
        y0_m.append(np.sum(a_y0_m==i))
        y1_m.append(np.sum(a_y1_m==i))
        y0_f.append(np.sum(a_y0_f==i))
        y1_f.append(np.sum(a_y1_f==i))
    return y0, y1, y0_m, y1_m, y0_f, y1_f
```

[53]:
```
def fair_get_F_metric(y0, y1, y0_m, y1_m, y0_f, y1_f):
    p_y0 = y0/sum(y0)
    p_y1 = y1/sum(y1)
    p_y0_m = y0_m/sum(y0_m)
    p_y1_m = y1_m/sum(y1_m)
    p_y0_f = y0_f/sum(y0_f)
    p_y1_f = y1_f/sum(y1_f)

    squared_diff_arr = []
    # y = 0
    for i in range(4):
        squared_diff_arr.append((p_y0[i]-p_y0_m[i])**2)
        squared_diff_arr.append((p_y0[i]-p_y0_f[i])**2)
    # y = 1
    for i in range(4):
        squared_diff_arr.append((p_y1[i]-p_y1_m[i])**2)
        squared_diff_arr.append((p_y1[i]-p_y1_f[i])**2)

    return sum(squared_diff_arr)
```

[54]:
```
def fair_get_output(to_file=False, name='outputs/default.txt'):
    y0, y1, y0_m, y1_m, y0_f, y1_f = fair_get_arrays(actions=action_arr,␣
 ↪data=X, symptom_arr=symptom_arr)
    F = fair_get_F_metric(y0, y1, y0_m, y1_m, y0_f, y1_f)
    output_str = ""
    output_str += f'============================== y=0␣
 ↪==================================\n'
```

```
    output_str += f'             |       P(a|y=0)  | P(a|y=0,z=male) | P(a|y=0,z=female) |\n'
    output_str += f'No vaccine: | {y0[0]:4}/{sum(y0):4}={y0[0]/sum(y0):.3f} | {y0_m[0]:4}/{sum(y0_m):4}={y0_m[0]/sum(y0_m):.3f} | {y0_f[0]:4}/{sum(y0_f):4}={y0_f[0]/sum(y0_f):.3f}   |\n'
    output_str += f'Vaccine 1:  | {y0[1]:4}/{sum(y0):4}={y0[1]/sum(y0):.3f} | {y0_m[1]:4}/{sum(y0_m):4}={y0_m[1]/sum(y0_m):.3f} | {y0_f[1]:4}/{sum(y0_f):4}={y0_f[1]/sum(y0_f):.3f}   |\n'
    output_str += f'Vaccine 2:  | {y0[2]:4}/{sum(y0):4}={y0[2]/sum(y0):.3f} | {y0_m[2]:4}/{sum(y0_m):4}={y0_m[2]/sum(y0_m):.3f} | {y0_f[2]:4}/{sum(y0_f):4}={y0_f[2]/sum(y0_f):.3f}   |\n'
    output_str += f'Vaccine 3:  | {y0[3]:4}/{sum(y0):4}={y0[3]/sum(y0):.3f} | {y0_m[3]:4}/{sum(y0_m):4}={y0_m[3]/sum(y0_m):.3f} | {y0_f[3]:4}/{sum(y0_f):4}={y0_f[3]/sum(y0_f):.3f}   |\n'
    output_str += f'============================== y=1 ==================================\n'
    output_str += f'             |       P(a|y=0)  | P(a|y=0,z=male) | P(a|y=0,z=female) |\n'
    output_str += f'No vaccine: | {y1[0]:4}/{sum(y1):4}={y1[0]/sum(y1):.3f} | {y1_m[0]:4}/{sum(y1_m):4}={y1_m[0]/sum(y1_m):.3f} | {y1_f[0]:4}/{sum(y1_f):4}={y1_f[0]/sum(y1_f):.3f}   |\n'
    output_str += f'Vaccine 1:  | {y1[1]:4}/{sum(y1):4}={y1[1]/sum(y1):.3f} | {y1_m[1]:4}/{sum(y1_m):4}={y1_m[1]/sum(y1_m):.3f} | {y1_f[1]:4}/{sum(y1_f):4}={y1_f[1]/sum(y1_f):.3f}   |\n'
    output_str += f'Vaccine 2:  | {y1[2]:4}/{sum(y1):4}={y1[2]/sum(y1):.3f} | {y1_m[2]:4}/{sum(y1_m):4}={y1_m[2]/sum(y1_m):.3f} | {y1_f[2]:4}/{sum(y1_f):4}={y1_f[2]/sum(y1_f):.3f}   |\n'
    output_str += f'Vaccine 3:  | {y1[3]:4}/{sum(y1):4}={y1[3]/sum(y1):.3f} | {y1_m[3]:4}/{sum(y1_m):4}={y1_m[3]/sum(y1_m):.3f} | {y1_f[3]:4}/{sum(y1_f):4}={y1_f[3]/sum(y1_f):.3f}   |\n'
    output_str += f'================== Fairness metric balance ==========================\n'
    output_str += f'Sum(|P(a_j|y_i) - P(a_j|y_i,z_k)|^2 for all i,j,k: {F:.5f}\n'
    output_str += f'======================================================================\n'
    if to_file:
        with open(name, 'w') as outfile:
            outfile.write(output_str)
    print(output_str)
```

[55]:
```
def fair_F_simulation(samples=10):
    F_values = []
    for i in range(samples):
        n_genes = 128
        n_vaccines = 3 # DO NOT CHANGE, breaks the simulator.
```

```
        n_treatments = 4
        n_population = 10_000
        n_symptoms = 10
        #np.random.seed(2)
        population = Population(n_genes, n_vaccines, n_treatments)
        X = population.generate(n_population)
        n_features = X.shape[1]

        policy = Naive(n_actions, action_space)
        model = ThompsonSampling(n_actions)
        policy.set_model(model)
        action_arr = np.zeros(n_population)
        symptom_arr = np.zeros((n_population, n_symptoms+1))

        for t in range(n_population):
            a_t = policy.get_action()
            y_t = population.vaccinate([t], a_t.reshape((1, 1)))
            new_col = y_t[:,[5,7,8]].sum(axis=1)
            new_col0 = new_col > 0
            new_col01 = new_col0.astype(int)
            new_col01 = np.reshape(new_col01,(1,-1))
            y_t_new = np.hstack((y_t,new_col01))
            policy.observe(a_t, y_t_new[:,10])
            action_arr[t] = a_t
            symptom_arr[t] = y_t_new

        y0, y1, y0_m, y1_m, y0_f, y1_f = fair_get_arrays(actions=action_arr,␣
↪data=X, symptom_arr=symptom_arr)
        F = fair_get_F_metric(y0, y1, y0_m, y1_m, y0_f, y1_f)
        print(i, F)
        F_values.append(F)
    return F_values
```

[56]:
```
def fair_F_simulation_histogram(samples=100, save=False, name='figures/
↪fair_F_simulation_histogram.png'):
    F_values = fair_F_simulation(samples=100)
    plt.hist(F_values)
    title = 'Histogram of unfairness in balance, ' + str(samples) + 'runs'
    plt.title(title)
    plt.xlabel('F_balance metric: unfairness')
    plt.ylabel('frequency')
    if save:
        plt.savefig(name)
    plt.show()
```

[57]:
```
def get_gender_fraction(to_file=False, name='outputs/gender_average_100_runs.
↪txt'):
    n_genes = 128
    n_vaccines = 3 # DO NOT CHANGE, breaks the simulator.
    n_treatments = 4
    n_population = 10_000
    n_symptoms = 10

    avg = []
    for i in range(100):
        population = Population(n_genes, n_vaccines, n_treatments)
        X = population.generate(n_population)
        m = np.mean(X[:,11])
        print(m)
        avg.append(np.mean(X[:,11]))
    np.mean(avg)

    if to_file:
        with open('outputs/gender_average_100_runs.txt', 'w') as outfile:
            out =␣
↪f'==========================================================\n'
            out += f'Percentage males, 100 runs with 10_000 individuals: {np.
↪mean(avg):.4f}\n'
            out += f'Standard deviation: {np.std(avg):.4f}\n'
            out += f'=========================================================='
            outfile.write(out)
```