# Inference on the Champagne Model using a Gaussian Process

## TODO

- Change kernel to ARD kernel in scratchpad.py
- Change from MLE to cross validation

## Setting up the Champagne Model

### Imports

```python
import pandas as pd
import numpy as np
from typing import Any
import matplotlib.pyplot as plt

import tensorflow as tf
import tensorflow_probability as tfp
tfb = tfp.bijectors
tfd = tfp.distributions
tfk = tfp.math.psd_kernels
```

### Model itself

```python
np.random.seed(590154)

population = 1000
initial_infecteds = 10
epidemic_length = 1000

pv_champ_alpha = 0.4  # prop of effective care
pv_champ_beta = 0.4  # prop of radical cure
pv_champ_gamma_L = 1 / 223  # liver stage clearance rate
pv_champ_delta = 0.05  # prop of imported cases
pv_champ_lambda = 0.04  # transmission rate
pv_champ_f = 1 / 72  # relapse frequency
pv_champ_r = 1 / 60  # blood stage clearance rate


def champagne_stochastic(
    alpha_,
    beta_,
    gamma_L,
    lambda_,
    f,
    r,
    N=population,
    I_L=initial_infecteds,
    I_0=0,
    S_L=0,
    delta_=0,
    end_time=epidemic_length,
):
    t = 0
    S_0 = N - I_L - I_0 - S_L
    list_of_outcomes = [{"t": 0, "S_0": S_0, "S_L": S_L, "I_0": I_0, "I_L": I_L}]

    while t < end_time:
        if S_0 == N:
            break

        S_0_to_I_L = (1 - alpha_) * lambda_ * (I_L + I_0) / N * S_0
        S_0_to_S_L = alpha_ * (1 - beta_) * lambda_ * (I_0 + I_L) / N * S_0
        I_0_to_S_0 = r * I_0 / N
        I_0_to_I_L = lambda_ * (I_L + I_0) / N * I_0
        I_L_to_I_0 = gamma_L * I_L
```

```
        I_L_to_S_L = r * I_L
        S_L_to_S_0 = (gamma_L + (f + lambda_ * (I_0 + I_L) / N) * alpha_ * beta_) * S_L
        S_L_to_I_L = (f + lambda_ * (I_0 + I_L) / N) * (1 - alpha_) * S_L

        total_rate = (
            S_0_to_I_L
            + S_0_to_S_L
            + I_0_to_S_0
            + I_0_to_I_L
            + I_L_to_I_0
            + I_L_to_S_L
            + S_L_to_S_0
            + S_L_to_I_L
        )

        t += np.random.exponential(1 / total_rate)
        new_stages_prob = [
            S_0_to_I_L / total_rate,
            S_0_to_S_L / total_rate,
            I_0_to_S_0 / total_rate,
            I_0_to_I_L / total_rate,
            I_L_to_I_0 / total_rate,
            I_L_to_S_L / total_rate,
            S_L_to_S_0 / total_rate,
            S_L_to_I_L / total_rate,
        ]
        new_stages = np.random.choice(
            [
                {"t": t, "S_0": S_0 - 1, "S_L": S_L, "I_0": I_0, "I_L": I_L + 1},
                {"t": t, "S_0": S_0 - 1, "S_L": S_L + 1, "I_0": I_0, "I_L": I_L},
                {"t": t, "S_0": S_0 + 1, "S_L": S_L, "I_0": I_0 - 1, "I_L": I_L},
                {"t": t, "S_0": S_0, "S_L": S_L, "I_0": I_0 - 1, "I_L": I_L + 1},
                {"t": t, "S_0": S_0, "S_L": S_L, "I_0": I_0 + 1, "I_L": I_L - 1},
                {"t": t, "S_0": S_0, "S_L": S_L + 1, "I_0": I_0, "I_L": I_L - 1},
                {"t": t, "S_0": S_0 + 1, "S_L": S_L - 1, "I_0": I_0, "I_L": I_L},
                {"t": t, "S_0": S_0, "S_L": S_L - 1, "I_0": I_0, "I_L": I_L + 1},
            ],
            p=new_stages_prob,
        )

        list_of_outcomes.append(new_stages)
```

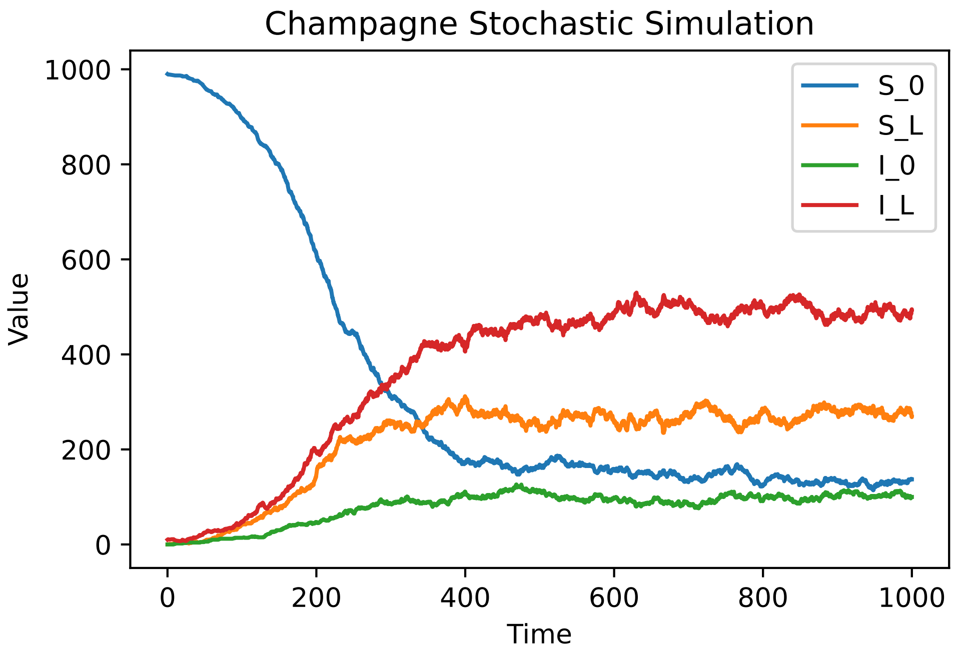
```python
        S_0 = new_stages["S_0"]
        I_0 = new_stages["I_0"]
        I_L = new_stages["I_L"]
        S_L = new_stages["S_L"]

    outcome_df = pd.DataFrame(list_of_outcomes)
    return outcome_df


champ_samp = champagne_stochastic(
    pv_champ_alpha,
    pv_champ_beta,
    pv_champ_gamma_L,
    pv_champ_lambda,
    pv_champ_f,
    pv_champ_r,
)  # .melt(id_vars='t')
```

### Plotting outcome

```python
champ_samp.plot(x = 't',legend=True)
plt.xlabel('Time')
plt.ylabel('Value')
plt.title('Champagne Stochastic Simulation')
plt.show()
```




#### Function that Outputs Final Prevalence

```
def champ_prevalence(alpha_, beta_, gamma_L, lambda_, f, r):
    champ_df_ = champagne_stochastic(alpha_, beta_, gamma_L, lambda_, f, r)

    return(champ_df_.iloc[-1]["I_0"] + champ_df_.iloc[-1]["I_L"])

observed_final_prevalence = champ_prevalence(pv_champ_alpha, pv_champ_beta,
pv_champ_gamma_L, pv_champ_lambda, pv_champ_f, pv_champ_r)
```

### Gaussian Process Regression on Final Prevalence Discrepency

```
my_seed = np.random.default_rng(seed=1795)  # For replicability

num_samples = 1000

variables_names = ["alpha", "beta", "gamma_L", "lambda", "f", "r"]
```

```python
pv_champ_alpha = 0.4  # prop of effective care
pv_champ_beta = 0.4  # prop of radical cure
pv_champ_gamma_L = 1 / 223  # liver stage clearance rate
pv_champ_lambda = 0.04  # transmission rate
pv_champ_f = 1 / 72  # relapse frequency
pv_champ_r = 1 / 60  # blood stage clearance rate

samples = np.concatenate(
    (
        my_seed.uniform(low=0, high=1, size=(num_samples, 1)),  # alpha
        my_seed.uniform(low=0, high=1, size=(num_samples, 1)),  # beta
        my_seed.exponential(scale=pv_champ_gamma_L, size=(num_samples, 1)),  # gamma_L
        my_seed.exponential(scale=pv_champ_lambda, size=(num_samples, 1)),  # lambda
        my_seed.exponential(scale=pv_champ_f, size=(num_samples, 1)),  # f
        my_seed.exponential(scale=pv_champ_r, size=(num_samples, 1)),  # r
    ),
    axis=1,
)

random_indices_df = pd.DataFrame(samples, columns=variables_names)

print(random_indices_df.head())

random_prevalences = random_indices_df.apply(
    lambda x: champ_prevalence(
        x["alpha"], x["beta"], x["gamma_L"], x["lambda"], x["f"], x["r"]
    ),
    axis=1,
)

random_discrepencies = np.abs(random_prevalences - observed_final_prevalence)

print(random_discrepencies.head())
```

```
      alpha      beta   gamma_L    lambda         f         r
0  0.201552  0.678250  0.004617  0.044661  0.034909  0.029033
1  0.332324  0.374357  0.003530  0.042614  0.003884  0.028896
2  0.836050  0.345550  0.008055  0.012402  0.001777  0.016656
3  0.566773  0.442576  0.004172  0.021682  0.006649  0.005813
4  0.880603  0.527607  0.003333  0.005784  0.031122  0.002323
0     49.0
1    237.0
```

```
2    563.0
3     12.0
4    545.0
dtype: float64
```

### Differing Methods to Iterate Function

```
# import timeit

# def function1():
#     np.vectorize(champ_prevalence)(random_indices_df['alpha'],
#     random_indices_df['beta'], random_indices_df['gamma_L'],
#     random_indices_df['lambda'], random_indices_df['f'], random_indices_df['r'])
#     pass

# def function2():
#     random_indices_df.apply(
#         lambda x: champ_prevalence(
#             x['alpha'], x['beta'], x['gamma_L'], x['lambda'], x['f'], x['r']),
#             axis = 1)
#     pass

# # Time function1
# time_taken_function1 = timeit.timeit(
#     "function1()", globals=globals(), number=100)

# # Time function2
# time_taken_function2 = timeit.timeit(
#     "function2()", globals=globals(), number=100)

# print("Time taken for function1:", time_taken_function1)
# print("Time taken for function2:", time_taken_function2)
```

Time taken for function1: 187.48960775700016 Time taken for function2: 204.06618941299985

### Custom Quadratic Mean Function

```
class quad_mean_fn(tf.Module):
    def __init__(self):
        super(quad_mean_fn, self).__init__()
        self.amp_alpha_mean = tf.Variable(
            586.0, dtype=np.float64, name="amp_alpha_mean"
        )
        self.alpha_tp = tf.Variable(
            pv_champ_alpha, dtype=np.float64, name="alpha_tp"
        )
        self.amp_beta_mean = tf.Variable(
            457.0, dtype=np.float64, name="amp_beta_mean"
        )
        self.beta_tp = tf.Variable(
            pv_champ_beta, dtype=np.float64, name="beta_tp"
        )
        self.amp_gamma_L_mean = tf.Variable(
            57.0, dtype=np.float64, name="amp_gamma_L_mean"
        )
        self.gamma_L_tp = tf.Variable(
            pv_champ_gamma_L, dtype=np.float64, name="gamma_L_tp"
        )
        self.amp_lambda_mean = tf.Variable(
            858.0, dtype=np.float64, name="amp_lambda_mean"
        )
        self.lambda_tp = tf.Variable(
            pv_champ_lambda, dtype=np.float64, name="lambda_tp"
        )
        self.amp_f_mean = tf.Variable(559.0, dtype=np.float64, name="amp_f_mean")
        self.f_tp = tf.Variable(
            pv_champ_f, dtype=np.float64, name="f_tp"
        )
        self.amp_r_mean = tf.Variable(1138.0, dtype=np.float64, name="amp_r_mean")
        self.r_tp = tf.Variable(
            pv_champ_r, dtype=np.float64, name="r_tp"
        )
        self.bias_mean = tf.Variable(264., dtype=np.float64, name="bias_mean")

    def __call__(self, x):
        return (
            self.amp_alpha_mean * (x[..., 0] - self.alpha_tp) ** 2
            + self.amp_beta_mean * (x[..., 1] - self.beta_tp) ** 2
            + self.amp_gamma_L_mean * (x[..., 2] - self.gamma_L_tp) ** 2
```

```
            + self.amp_lambda_mean * (x[..., 3] - self.lambda_tp) ** 2
            + self.amp_f_mean * (x[..., 4] - self.f_tp) ** 2
            + self.amp_r_mean * (x[..., 5] - self.r_tp) ** 2
            + self.bias_mean
        )
```

### Making the ARD Kernel

```
index_vals = random_indices_df.values
obs_vals = random_discrepencies.values

amplitude_champ = tf.Variable(200.0, dtype=np.float64, name="amplitude_champ")
# length_scale_champ = tf.Variable(0.02, dtype=np.float64, name="length_scale_champ")
observation_noise_variance_champ = tf.Variable(
    981.0, dtype=np.float64, name="observation_noise_variance_champ"
)
```

```
len_alpha = tf.Variable(0.2, dtype=np.float64, name="amp_alpha_mean")
len_beta = tf.Variable(550., dtype=np.float64, name="amp_beta_mean")
len_gamma_L = tf.Variable(400., dtype=np.float64, name="amp_gamma_L_mean")
len_lambda = tf.Variable(600., dtype=np.float64, name="amp_lambda_mean")
len_f = tf.Variable(100., dtype=np.float64, name="amp_f_mean")
len_r = tf.Variable(900., dtype=np.float64, name="amp_r_mean")

length_scales_champ = tf.Variable(
    np.ones(
        6,
    ),
    dtype=np.float64,
    name="length_scales_champ",
)
```

```
kernel_champ = tfk.FeatureScaled(tfk.ExponentiatedQuadratic(
    amplitude=amplitude_champ), scale_diag=length_scales_champ)
```

```
# Define Gaussian Process with the custom kernel
champ_GP = tfd.GaussianProcess(
    kernel=kernel_champ,
    observation_noise_variance=observation_noise_variance_champ,
    index_points=index_vals,
```

```
    mean_fn=quad_mean_fn(),
)

print(champ_GP.trainable_variables)

Adam_optim = tf.optimizers.Adam(learning_rate=.01)
```

```
(<tf.Variable 'amplitude_champ:0' shape=() dtype=float64, numpy=200.0>, <tf.Variable 'length_
```

```
@tf.function()
def optimize():
    with tf.GradientTape() as tape:
        loss = -champ_GP.log_prob(obs_vals)
    grads = tape.gradient(loss, champ_GP.trainable_variables)
    Adam_optim.apply_gradients(zip(grads, champ_GP.trainable_variables))
    return loss


num_iters = 10000

lls_ = np.zeros(num_iters, np.float64)
for i in range(num_iters):
    loss = optimize()
    lls_[i] = loss

amp_fin = champ_GP.trainable_variables[0].numpy()
obs_fin = champ_GP.trainable_variables[1].numpy()
alpha_amp_fin = champ_GP.trainable_variables[2].numpy()
beta_amp_fin = champ_GP.trainable_variables[3].numpy()
gamma_L_amp_fin = champ_GP.trainable_variables[4].numpy()
lambda_amp_fin = champ_GP.trainable_variables[5].numpy()
f_amp_fin = champ_GP.trainable_variables[6].numpy()
r_amp_fin = champ_GP.trainable_variables[7].numpy()
bias_fin = champ_GP.trainable_variables[8].numpy()

print("Trained parameters:")
for var in champ_GP.trainable_variables:
    print("{} is {}".format(var.name, var.numpy().round(decimals = 2)))
```

```
Trained parameters:
amplitude_champ:0 is 162.22
```

```
length_scales_champ:0 is [-0.   -0.    0.35  0.02  0.27  0.2 ]
observation_noise_variance_champ:0 is 994.35
alpha_tp:0 is 0.29
amp_alpha_mean:0 is 510.27
amp_beta_mean:0 is 364.09
amp_f_mean:0 is 561.67
amp_gamma_L_mean:0 is 58.57
amp_lambda_mean:0 is 888.72
amp_r_mean:0 is 1171.1
beta_tp:0 is 0.49
bias_mean:0 is 191.44
f_tp:0 is 0.06
gamma_L_tp:0 is 0.02
lambda_tp:0 is 0.3
r_tp:0 is -0.11
```

```python
plt.figure(figsize=(7, 4))
plt.plot(lls_)
plt.xlabel("Training iteration")
plt.ylabel("Log likelihood")
plt.show()
```