Stochastic Optimization Applied to Epidemiology

Giorgia Rosalia Buccelli
Rachid El Amrani

November 8, 2022

Abstract

The Reed-Frost model is one of the earliest models used to study the behaviour of infectious diseases. It was created by Lowell Reed and Wade Hampton Frost during the 1920s. Nowadays, the model is used more as a template for more complex models. However, it is still worth looking at because it gives a good introduction to stochastic epidemiology theory. Epidemiological models are used in modern life to provide predictions and data for possible outbreaks of infections. This information can be useful for governments and public health organisations to implement measures to combat current epidemics and to make plans for possible future outbreaks. Whilst epidemiology covers a wide range of infectious diseases, the Reed-Frost model is used primarily for acute infectious diseases. An acute infectious disease is an infection that appears suddenly and may be of brief or prolonged duration such as colds, flus and STIs. We will explore the Reed-Frost model and a couple of example simulations, before analysing how applicable the model is to real life.

1 Introduction of Concepts

Before looking at the Reed-Frost model a few introductory concepts need to be explained, namely the different types of epidemiological and statistical models. Firstly, from an epidemiology viewpoint, the Reed-Frost model is an S-I-R epidemic model.

Definition 1 (S-I-R Model). An S-I-R (Susceptible-Infection-Removal) Model is an epidemiological model that calculates the number of people infected with a contagious illness in a closed population over time. The S-I-R format provides the basis of many epidemic models and is used mainly to model diseases transmitted by humans, where individuals pass through the following progression of states:

Susceptible to the disease $\rightarrow$ Infected $\rightarrow$ Removed (e.g. immunity or death)

The infection can only be passed between two individuals through "adequate contact". The concept of "adequate contact" is relative as different infections are passed in different ways. For example, close proximity to an infected individual may be sufficient for a common cold to spread but isn't for an infection such as an STI. This makes the term "adequate contact" difficult to define universally and it can be hard to define precisely for a specific infection (e.g requirements to catch a cold). For the Reed-Frost model, we consider an acute, infectious disease that is spread only by "adequate contact". The model itself comes in two forms: deterministic form and stochastic form.

Definition 2 (Deterministic Model). A deterministic model is one in which a given input into the model always produces the same output.

Equivalently, a deterministic model is one where the output is fully determined by the parameters and initial conditions. For example, in population dynamics a model that takes some initial population size as its input and outputs a predicted population size after some interval is a deterministic model. The lack of randomness within these models makes them relatively simple to work with, but limits its application to real life. For example, real populations have random

influences affecting growth such as varying birth and death rates. Therefore, we need to use stochastic processes in order to create a more realistic model.

Definition 3 (Stochastic Model). A stochastic model is one in which there is a random element such that for a given input to the model, the outcome takes a range of possible values.

This means that the outcome is not uniquely determined by the given input. Therefore, a stochastic model needs to be run multiple times to generate trends in behaviour. The stochastic version of the Reed-Frost model that we will be considering is called an epidemic chain binomial.

Definition 4 (Epidemic Chain Binomial Model). An epidemic chain binomial model is one in which the number of infected individuals to appear in the next unit interval of time follows a binomial distribution, with the probability of infection dependent on the number of infected individuals in the current time unit.

It is important to note that an epidemic chain binomial model is a discrete time model and are used to describe the spread of infection within closed populations, such as households. Therefore, we need to make some basic assumptions to base the model on.

2 Set-Up for the Model

2.1 Assumptions

There are many factors that can influence the spread of an infectious disease. Factors can range from simple influences like the movement of individuals and changes in living standards, to changes as extreme as a natural disaster or the outbreak of war."Since there are numerous influences on the spread of an infection, it is essential that some assumptions are made in order to create a coherent model. The Reed-Frost model is based on 5 assumptions:

1. The infection can only be spread via "adequate contact" with infected individuals.
2. Susceptible individuals become infected after "adequate contact" with an infected individual in a given time period. They will then only be infectious for the subsequent time period, before becoming fully immune.
3. The infection is introduced to a closed population (i.e. the total population is fixed with no one entering or leaving the pool of individuals).
4. . Individuals have a fixed probability of "adequate contact" with any other individual in a time period
5. The above conditions remain constant for the duration of the epidemic.

It is important to note that the final assumption also includes the requirement of external factors remaining constant. For example, a war or a famine could still impact a closed population and thus influence the spread of disease. Furthermore, the discrete time periods are defined to correspond to the time between an individual becoming infected and the point at which the individual is most infectious.

2.2 Notation

In a given time interval t, the number of susceptible individuals in the population is denoted by S_t and the number of infected individuals are denoted by I_t. For simplicity, immune individuals are often ignored as they stay immune for the remainder of the epidemic. The (fixed) probability that any two individuals come into "adequate contact" in one time frame is p, which is usually expressed as $p = (1 - q)$.

3 Stochastic Approach

The stochastic approach to the Reed-Frost model uses the same set-up defined previously and takes a series of binomial trials to model the epidemic. As before, the probability of a susceptible individual becoming infected is:

$$1 - q^{I_t}$$

However, now the number of infected individuals in the time period $t+1$ is modelled using a binomial distribution with parameters S_t and $1 - q^{I_t}$. We denote this as:

$$I_{t+1} \sim Binomial(S_t, 1 - q^{I_t})$$

Therefore, in a given time period $t+1$, the probability that there are r infected individuals is:

$$\mathbb{P}(I_{t+1} = r) = \binom{S_t}{r}(1 - q^{I_t})^r (q^{I_t})^{S_t - r}$$

The number of susceptible individuals in time period $t+1$ is simply the difference between the number of susceptible individuals in time t and the number of infected individuals in time $t+1$. So we have that:

$$S_{t+1} = S_t - I_{t+1}$$

These two equations form the stochastic Reed-Frost model for initial conditions (S_t, I_t, p). The sequence $\{I_0, I_1, ..., I_\tau\}$ is called the epidemic chain, where we define $\tau = min\{t : I_{t+1} = 0\}$ to be the time when the infection dies out (i.e. when there are no more infected individuals). Using this set up, can also calculate the (expected) final size of the epidemic.

Definition 5 (Final Size of an Epidemic). The final size of an epidemic, T, is the total number of individuals who became infected during the epidemic.

Therefore, we can see that:

$$T = S_0 - S_\infty = \sum_{k=1}^{\tau} I_k$$

where $\tau = min\{t : I_{t+1} = 0\}$.

It should be noted that the final size of the epidemic excludes the number of individuals who were initially infected.
Whereas I_t gives the value of the number of infected individuals in one time period, T provides a clearer indication of the total number of individuals who are affected by the infection. Using the above formula, the probability mass function and expectation for the variable T can be calculated in order to obtain estimations for the size of an epidemic.

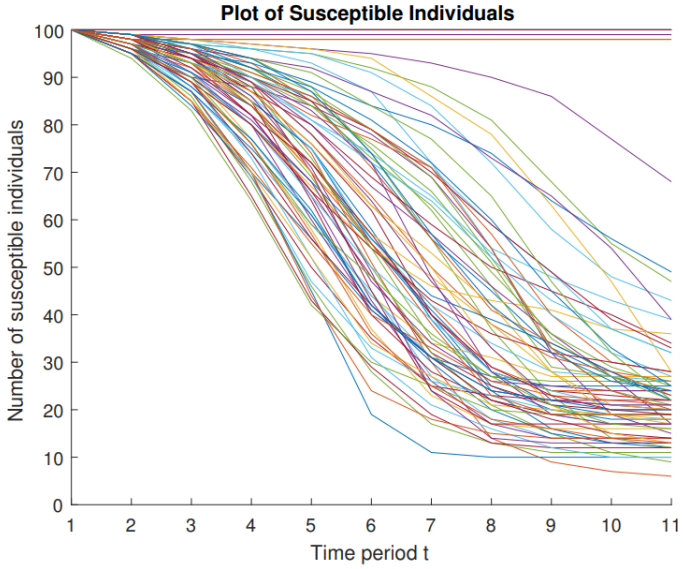


Figure 3: the number of susceptible individuals

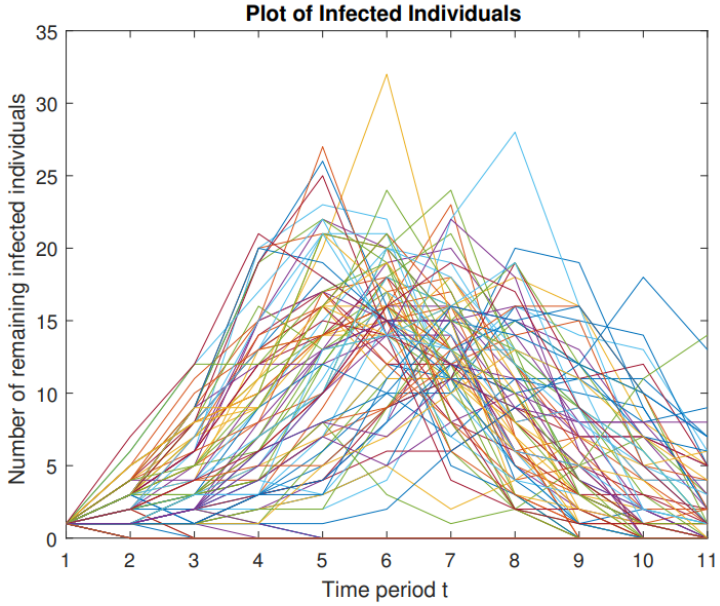


Figure 4: the number of infected individuals

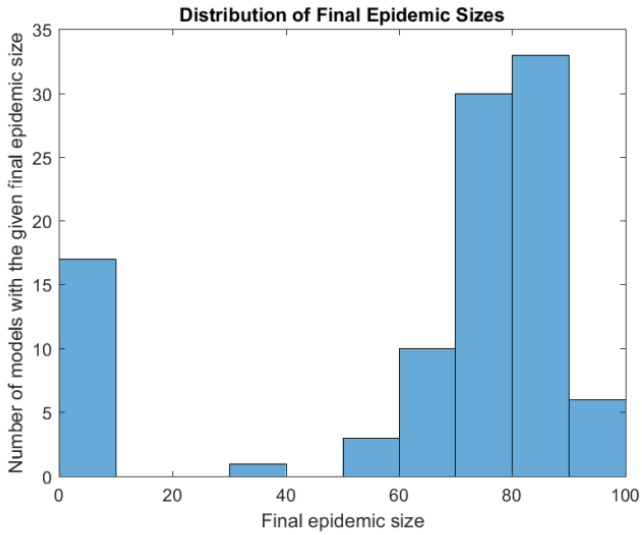


Figure 5: Histogram of the final epidemic sizes generated

4 Data Generation

For the purpose of generating the data, we used the functions listed below:

$$cost(prob) = (0.003/prob)^9 - 1$$

```
def cost(prob):
  return (0.003/prob)**9-1
```

this function computes the cost associated with the measurements taken.

$$Reed\_Frost(prob)$$

```
def Reed_Frost(prob):
  N = 1000
  S = 999
  I = 1
  R = 0
  for i in range(60):
    In = binomial(n=S, p=1-(1-prob)**I)
    Sn = S - In
    Rn = N - In - Sn
    # update the values corresponding to the (i+1)th day
    I = In
    S = Sn
    R = Rn
  return S, I, R
```

This functions generates the tuple (S, I, R) consisting of

- S: number of Susceptibles
- I: number of Infected
- R: number of Removed

after 60 consecutive days.

$$associated\_cost(prob)$$

```
def associated_cost(prob):
  (S, I, R) = Reed_Frost(prob)
  return R
```

This function retrieves the values from the Reed_Frost function and returns the number of removed.

After running the program, we end up with the following dataframe:

p	c(p)
0.00300	877.394000
0.00275	861.043751
0.00250	848.412780
0.00225	829.680045
0.00200	817.963559
0.00175	860.372667
0.00150	1186.435000
0.00125	3249.916790

5 Hyperparameter Tuning For Deep Learning

Hyperparameter optimization is a big part of deep learning. The reason is that neural networks are notoriously difficult to configure and there are a lot of parameters that need to be set. On top of that, individual models can be very slow to train. In this section, we illustrate the different hyperparams we focused on assessing to optimize our network setting:

Batch Size and Number of Epochs
As we know, the batch size in iterative gradient descent is the number of patterns shown to the network before the weights are updated. It is also an optimization in the training of the network, defining how many patterns to read at a time and keep in memory.The number of epochs is the number of times that the entire training dataset is shown to the network during training. Here we will evaluate only a suite of different values for both the number epochs (500,1000,2000,3000) and the number of batches (2, 4, 6) since the dataset has only 8 observations which is small by far.

Dropout Regularization
To get good results, dropout is best combined with a weight constraint such as the max norm constraint. This involves fitting both the dropout percentage and the weight constraint. We will try dropout percentages between 0.0 and 0.9 (1.0 does not make sense) and maxnorm weight constraint values between 0 and 5. 2 nodes, a bias term and ReLU as an activation function

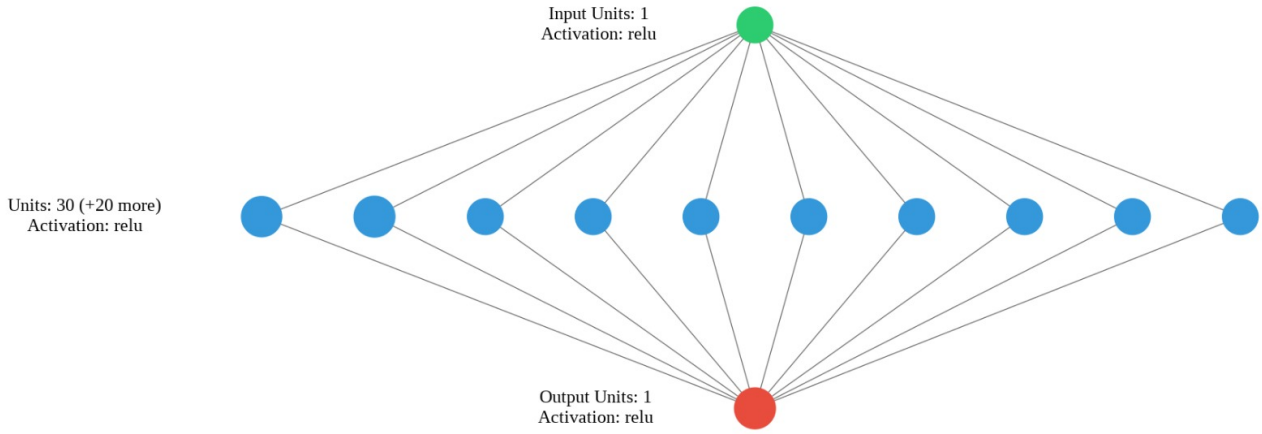
The Number of Neurons in the Hidden Layer
The number of neurons in a layer is an important parameter to tune. Generally the number of neurons in a layer controls the representational capacity of the network, at least at that point in the topology. Also, generally, a large enough single layer network can approximate any other neural network, at least in theory. In this assignment, we looked at tuning the number of neurons in a single hidden layer. We tried the following values (5, 10, 15,30) given the small dimension of the underlying problem. We definitely didn't try all the techniques listed above individually since parameters can interact. To aggregate the best parameters, we used the so-called GridSearchCV class from the notorious machine learning library known as **Scikit Learn** which, has the required functions to try all the different combinations, among the provided values, and outputs the one that gives the best accuracy. In our case, this was the best one

```
{'activation_function': 'relu', 'batch_size': 2, 'epochs': 3000, 'neurons': 30}
```

6 The structure of the Neural Network

6.1 Neural Network Design

According to the previous section, here is how our network will look like:



6.2 Performance Evaluation

In the table below, we can see how the model did draw enough patterns from the training data

Actual average Cost	Predicted average Cost
888.467000	869.436768
862.979251	861.618835
842.447114	853.800903
825.651045	845.982971
819.030359	838.165100
863.476334	830.347168
1190.064714	1187.700439
3251.714165	3249.615234

but, it might be way far from being able to perform the same way on eventual unseen data. In that case, we would automatically suggest that our model would have over-fitted the training data. In fact, generally, to overcome over-fitting in Deep Learning, we most of the time think of using Dropout although this doesn't always guarantee the expected good behavior.

7 Optimization

Since we decided to use Stochastic Gradient Descent, we thought it would make sense providing a recall: Gradient descent is the grandfather of first order methods. It simply starts at an initial point and then repeatedly takes a step opposite to the gradient direction of the function at the current point. The gradient descent algorithm to minimize a function $f(x)$ is as follows:

for $k = 0, 1, 2, \ldots$ **do**
$g_k \leftarrow \nabla f(x_k)$
$x_{k+1} \leftarrow x_k - t_k g_k$
end for

8 The best combination

In this last part, we used Nelder-Mead optimization algorithm. In fact, As a general principle, gradient-based methods tend to converge significantly faster on smooth functions than gradient-free optimization methods, but, the advantage of this algorithm is that it doesn't require the user to supply the cost function gradients with respect to the weights. after a couple of evaluations, the algorithm seems to find the minimum at approximately $p = 0.00274619$ with the corresponding optimal value ≈ 3248.64. We report, down below, the output of the algorithm based on a random starting point:

```
Status : Optimization terminated successfully.
Total Evaluations: 5
Solution: f([0.00274619]) = 3248.63647
```

```
% Importing all the packages/modules for the required functionalities
import numpy as np
from numpy.random import binomial
from statistics import mean
import pandas as pd

% compute the cost related to the measurements taken, as it has been explicitly
    defined in the assignment
def cost(prob):
  return (0.003/prob)**9-1


% the recurrent relation that provides the number of individuals in each category [
    Susceptible, Infected, Removed] in the (i+1)th day
% which will be used to generate the required data 60 days in a row
def Reed_Frost(prob):
  N = 1000
  S = 999
  I = 1
  R = 0
  for i in range(60):
    In = binomial(n=S, p=1-(1-prob)**I)
    Sn = S - In
    Rn = N - In - Sn
    % update the values corresponding to the (i+1)th day
    I = In
    S = Sn
    R = Rn
  return S, I, R

% this function will take the output of the Reed_Frost function and retrieves only the
    value of R
def associated_cost(prob):
  (S, I, R) = Reed_Frost(prob)
  return R

 probs=[0.003, 0.00275, 0.0025, 0.00225, 0.002, 0.00175,0.0015, 0.00125, 0.001,
    0.00075, 0.0005]
costs=[]
expec_cost=[]
for prob in probs:
  for i in range(1000):
    costs.append(associated_cost(prob))
  expec_cost.append(mean(costs) + cost(prob))
% lets form our two series of simulated data
X =pd.Series(probs, name='p')
Y =pd.Series(expec_cost,name='c(p)')
```

```
% our dataframe where p=probability and c(p)= the corresponding simulated average cost
df = pd.concat([X, Y], axis=1)
% Since we don't want the tiny values of p, we cut off the values of p below 0.00125
df=df.iloc[:8,:]

% our simulated data in the form of a dataframe to make it suitable for further
    analysis
df

% required packages/modules
import tensorflow as tf
import keras
from tensorflow.keras.optimizers import SGD
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import EarlyStopping
from sklearn import preprocessing
from sklearn.preprocessing import scale
from keras.layers import Dropout
% we extract the vectors for numbers as expected by the neural network
data= df.values
x = data[:,0]
y = data[:,1]
% reshape arrays into into rows and cols
x = x.reshape((len(x), 1))
y = y.reshape((len(y), 1))
% we rescale the data to avoid biases within the computations
scale_x = MinMaxScaler()
x = scale_x.fit_transform(x)
scale_y = MinMaxScaler()
y = scale_y.fit_transform(y)
% inverse transforms
% next, we structure our neural network in a sequential way using Keras API that has
    been concised
% on purpose for DeepLearning and Machine Learning
% here, an instance of the class Sequential
model = Sequential()
% one hiddine layer made of 30 neurons, a relu activation function and weights
    normally distributed
% expecting an input of dim = 1
model.add(Dense(30, input_dim=1,kernel_initializer = 'normal', activation = 'relu'))
% a drop out of 30% to avoid eventual overfitting
Dropout(0.3)
% output layer expected to output one value which is the predicted average cost
model.add(Dense(1))
% the stochastic gradient descent required for updating the weights
sgd=SGD(learning_rate=0.11, momentum=0.0, nesterov=False, name="SGD")
% compile the model specifying the metrics of interest, the loss function and the
    stochastic gradient algorithm
model.compile(
    loss = 'mse',
    optimizer = sgd,
    metrics = ['mean_squared_error']
)

% here, we fit our model to the training data. We perform the operation for 3000
    epochs and a bach of only 2 samples since
% the dataset is very small. The other arguments help extracting further informations
    from the training like displaying
% of the values we're getting + a summary of the changes undergone by the loss
    function
history = model.fit(
    x,y,
    batch_size=2,
    epochs = 3000,
    verbose = True,
    callbacks = [EarlyStopping(monitor = 'val_loss', patience = 1)]
)

% gather the prediction made by the model, rescale back to get the data in the
```

```
    required form
% for both inputs and outputs. The same should also be applied to the predictions
prediction = model.predict(x)
x = scale_x.inverse_transform(x)
y = scale_y.inverse_transform(y)
prediction = scale_y.inverse_transform(prediction)

% store the actual and the prediction in a flat list for an easy reading
actual = y.flatten()
pred = prediction.flatten()

% a dataframe that shows explicitly the differences between the predicted and the
    actual values
t = pd.DataFrame({'Actual average Cost':actual,'Predicted average Cost':pred})
t

% Down below, the Grid search cv class that helps find the best combination among many
     hyperparams
from keras.wrappers.scikit_learn import KerasRegressor
% Use scikit-learn to grid search the batch size and epochs
import numpy as np
import tensorflow as tf
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense

% Use scikit-learn to grid search the batch size and epochs
import numpy as np
import tensorflow as tf
from sklearn.model_selection import GridSearchCV
% Function to create model, required for KerasClassifier
def create_model(activation_function,neurons):
  % create model
  model = Sequential()
  model.add(Dense(neurons, input_shape=(1,),activation=activation_function))
  model.add(Dense(1))
  % Compile model
  model.compile(loss='mean_squared_error', optimizer='adam', metrics=['accuracy'])
  return model
% fix random seed for reproducibility
seed = 7
tf.random.set_seed(seed)

% split into input (X) and output (Y) variables
% define the dataset
X = df['p'].values
Y = df['c(p)'].values
% reshape arrays into into rows and cols
X = x.reshape((len(x), 1))
Y = y.reshape((len(y), 1))
% create model
estimator = KerasRegressor(build_fn=create_model, nb_epoch=100, batch_size=5, verbose
    =0)
% define the grid search parameters
neurons = [5, 10, 15,30]
activation_function = ['relu','tanh','linear','sigmoid']
batch_size = [2, 4, 6]
epochs = [500,1000,2000,3000]
param_grid = dict(batch_size=batch_size, epochs=epochs,activation_function =
    activation_function,neurons=neurons)
grid = GridSearchCV(estimator=estimator, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X, Y)
% summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))

```

```
% nelder-mead optimization of noisy one-dimensional function
from scipy.optimize import minimize
from numpy.random import rand
from numpy.random import randn

% objective function
def objective(x):
  xn = scale_x.fit_transform([x])
  z = model.predict(xn)
  v = scale_y.inverse_transform(z)
  return v

% define range for input
r_min, r_max = 0.00125, 0.003
% define the starting point as a random sample from the domain
pt = r_min + rand(1) * (r_max - r_min)
% perform the search
result = minimize(objective, pt, method='nelder-mead')
% summarize the result
print('Status : %s' % result['message'])
print('Total Evaluations: %d' % result['nfev'])
% evaluate solution
solution = result['x']
evaluation = objective(solution)
print('Solution: f(%s) = %.5f' % (solution, evaluation))
```