Analytical Analysis of Italy's COVID-19 Stats through Exponential and Logarithmic Model

MS Project Report

Bachelor of Technology in

Department of Applied Mathematics by

SUNAKSHI (2K17/MC/106) and **VATSAL AGARWAL** (2K17/MC/119) *of branch* Mathematics and Computing *under the guidance of*

Mrs. Nilam

Professor

Department of Applied Mathematics

DEPARTMENT OF APPLIED MATHEMATICS

DELHI TECHNOLOGICAL UNIVERSITY

Nov 2020

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to our guide and mentor Mrs. Nilam for her guidance at every stage of this project work. We are grateful to her for guiding us and supporting us to complete the project successfully.

We express our gratitude to the university for providing us this oppurtunity to explore this topic and learn several method and facts.

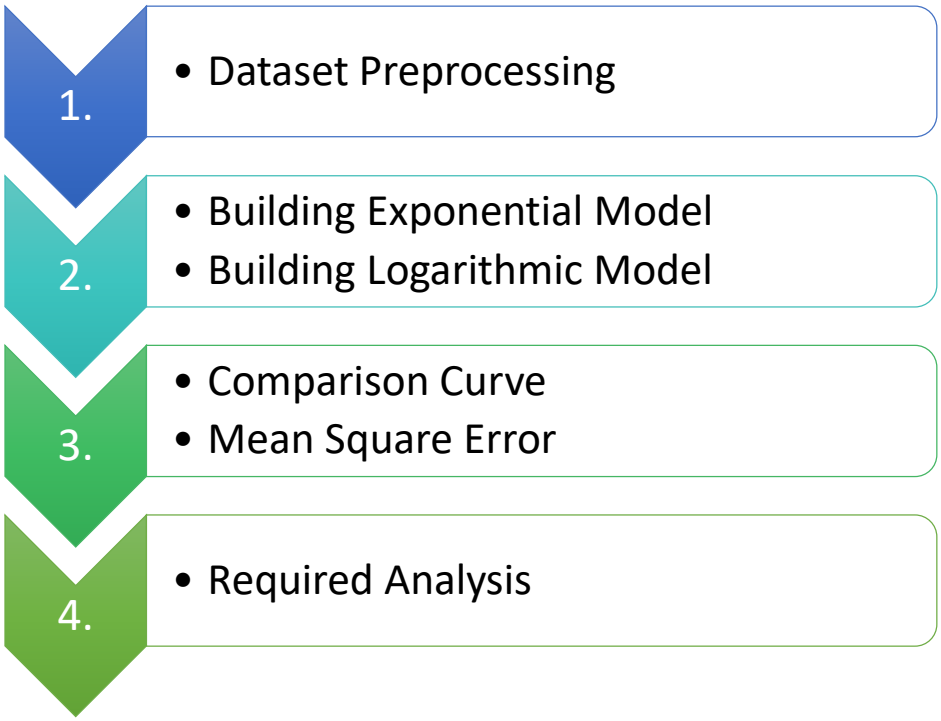
Sunakshi
Vatsal Agarwal

LIST OF CONTENTS

LIST OF FIGURES

1. OBJECTIVE

The aim of this project is to do an analytical case study. This case study is based on the daily reporting of the COVID-19 cases in Italy. The basic aim is to do a comparative case study between exponential model and logarithmic model for Italy's COVID-19 cases. We will be building models for both the case study topics based on data presented and certain parameters whose optimal values are calculated using some defined functions. Both the models will be used to make predictions independently and then they will be plotted against the actual COVID-19 numbers in separate graphs. Then we will calculate their errors using lease square fit and finally the model which provides the least error will then be accepted as suitable for Italy's COVID-19 stats case study.



2. DATASET

The dataset used by us is a public dataset and the link to it is –

https://raw.githubusercontent.com/pcm-dpc/COVID-19/master/dati-andamento-nazionale/dpc-covid19-ita-andamento-nazionale.csv .

This dataset is used as it provides the live data on the number of positive cases in Italy to model and predict the growth rate of the virus. The dataset contains cases starting from date 24th February 2020, IST 18:00:00 and the dataset is continuously updated with the recorded number of cases on each day. The dataset is in Italian Language.

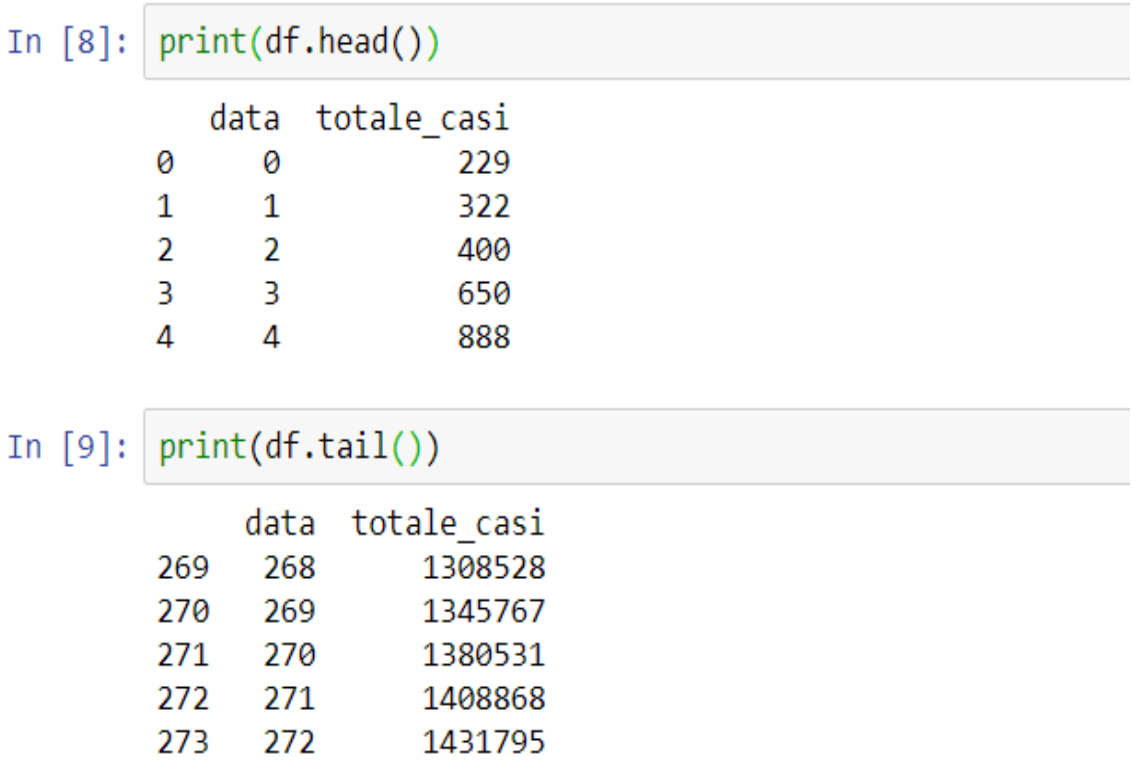


Fig no: 1

3. DATA PREPROCESSING

As with every project related to working on some predefined the dataset, we need to perform some data preprocessing in order to extract required features from the dataset as well as to convert them into a format so that using them in our mathematical model becomes a convenient task.

We started the preprocessing part by taking out the required columns from the dataset and making a dataframe (named as 'df' here) for future use.

In the next step we pick out the 'data' column from the dataset which means 'date' in English and mapped it into an integer value. This integer value represents the number of days of that date from 1st January, 2020. After this mapping the integer values are again stored into 'data' column.

The major process that needs to be done here is to perform scaling of data. Since the data involves too large values, distributed non-uniformly, the effect of values during the later phase of data collection can be considerably very high as compared to the initial phase of disease spread. This may lead to inconsistency in result. So, in order to solve this issue, we used 'MinMaxScaler' function. This is an inbuilt python function that comes with the library 'sklearn'. We need to scale down the data of the attribute 'totale_casi', so we passed this data through the MinMaxScaler and finally save the data back to the dataframe 'df'.

```
In [21]: from sklearn import preprocessing
         a = df.iloc[:, 1:2].values
         min_max_scaler = preprocessing.MinMaxScaler(feature_range =(0, 1))
         x_after_min_max_scaler = min_max_scaler.fit_transform(a)
         df['totale_casi'] = x_after_min_max_scaler
```

Fig no: 2

After performing min max scaling our data became:

```
In [32]: print(df.head())
```

```
   data  totale_casi
0     0     0.000000
1     1     0.000065
2     2     0.000119
3     3     0.000294
4     4     0.000460
```

```
In [33]: print(df.tail())
```

```
     data  totale_casi
269   268     0.913894
270   269     0.939906
271   270     0.964190
272   271     0.983985
273   272     1.000000
```

Fig no: 3

4. EXPONENTIAL MODEL

Exponential Model is associated with the concept that any species can potentially increase in numbers according to a geometric series.

For example, if a species has non-overlapping populations (e.g., annual plants), and each organism produces R offspring, then, population numbers N in generations t=0,1,2,... is equal to :

$$N_1 = N_0 \cdot R$$

$$N_t = N_0 \cdot R^t$$

When t is large, then this equation can be approximated by an exponential function:

$$N_t = N_0 \cdot \exp(r \cdot t) = N_0 \cdot e^{rt}$$

There are 3 possible model outcomes:

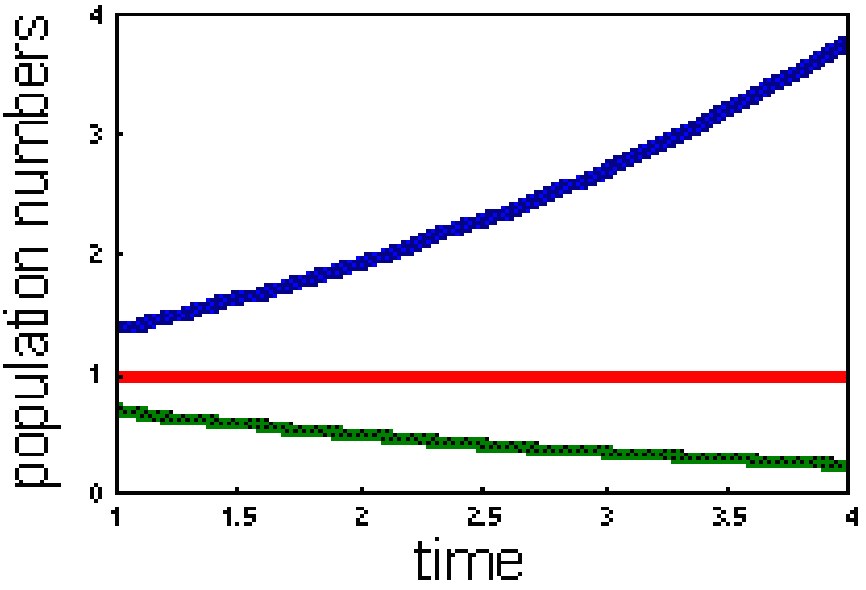


Fig no: 4

1. Population exponentially declines ($r < 0$)
2. Population exponentially increases ($r > 0$)
3. Population does not change ($r = 0$)

Parameter r is called:

- Malthusian parameter
- Intrinsic rate of increase
- Instantaneous rate of natural increase
- Population growth rate

"Instantaneous rate of natural increase" and "Population growth rate" are generic terms because they do not imply any relationship to population density. It is better to use the term "Intrinsic rate of increase" for parameter r in the logistic model rather than in the exponential model because in the logistic model, r equals to the population growth rate at very low density (no environmental resistance).

Assumptions of Exponential Model:

1. Continuous reproduction (e.g., no seasonality)
2. All organisms are identical (e.g., no age structure)
3. Environment is constant in space and time (e.g., resources are unlimited)

However, exponential model is robust; it gives reasonable precision even if these conditions do not met. Organisms may differ in their age, survival, and mortality. But the population consists of a large number of organisms, and thus their birth and death rates are averaged.

Parameter r in the exponential model can be interpreted as a difference between the birth (reproduction) rate and the death rate:

$$\frac{dN}{dt} = (b - m)N = rN$$

where *b* is the birth rate and *m* is the death rate. Birth rate is the number of offspring organisms produced per one existing organism in the population per unit time. Death rate is the probability of dying per one organism. The rate of population growth (*r*) is equal to birth rate (*b*) minus death rate *(m*).

Applications of the exponential model

- microbiology (growth of bacteria),
- conservation biology (restoration of disturbed populations),
- insect rearing (prediction of yield),

- plant or insect quarantine (population growth of introduced species),
- fishery (prediction of fish dynamics).

Working for model

The model starts by defining a function for calculating values for exponential model. In our function we will be passing four parameters: the first parameter is the date or number of day from 1st January, 2020, while the next 3 parameters are calculated using certain calculations which are explained later.

Our function definition for exponential function is:

```
In [13]: #Defining Exponential Model
         # Exponential function that we will use: f(a, k, l, m): k*(e^(l*(a-m)))
         #a => time
         #k => refers to cases on some day m
         #l => rate of growth
         #m => an optimal value of day m
         def f(a,k,l,m):
             return k*np.exp(l*(a-m))
```

Fig no: 5

The parameters that are used to calculate values in exponential function are obtained by using the curve_fit function present in the python. curve_fit function is a part of scipy library of the python. It uses non-linear least squares to fit a function, f, to data. curve_fit function uses multiple parameter that can be used to obtain the required values.

```
scipy.optimize.curve_fit(f, xdata, ydata, p0=None, sigma=None, absolute_sigma=False, check_finite=True, bounds=-inf, inf,
method=None, jac=None, **kwargs)
```

Fig no: 6

Out of all the parameters that can be used, we used certain essential parameters that can't be avoided while calling the function. These are:

1. 'f' – this is the function to which curve_fit fits the data. This function must take the independent variable as the first argument and the parameters to fit as separate remaining arguments,
2. 'xdata' – the array_like structure of the independent attribute in the dataset which in our case is number of days. It is usually a M-length sequence,
3. 'ydata' – the array_like structure of the dependent attribute in the dataset which in our case is the number of corona virus cases reported every day and. It is again a M-length sequence,
4. 'maxfev' – It represents the maximum number of calls to the function and it is an optional parameter. But due to calculation constraints, we need to specify it to equal to 5000.

```
covariance, fit = curve_fit(f,x,y,p0=[100,1,20], maxfev=5000)
```

Fig no: 7

This curve_fit function returns 2 values –

1. array1 – it is a 1D array representing the optimal values for the parameters so that the sum of the squared residuals of f(xdata, *popt) – ydata is minimized. We stored this data in 'covariance' variable,
2. array2 – it is a 2D array representing the estimated covariance of the parameters. The diagonals of this 2D array provides the variance of the parameter estimate. In our case, we stored this data in the 'fit' variable.

```
In [26]: print(fit)
         print('covariances diagonal elements')
         print('x = ', fit[0])
         print('y = ', fit[1])
         print('z = ', fit[2])
         print(covariance)
```

```
[[7.01361809e+07 1.44698055e-01 2.29628099e+11]
 [1.44698117e-01 3.35764614e-07 4.73751511e+02]
 [2.29628099e+11 4.73751315e+02 7.51809736e+14]]
covariances diagonal elements
x =  [7.01361809e+07 1.44698055e-01 2.29628099e+11]
y =  [1.44698117e-01 3.35764614e-07 4.73751511e+02]
z =  [2.29628099e+11 4.73751315e+02 7.51809736e+14]
[2.20915389e-02 1.38512306e-02 1.98939535e+01]
```

Fig no: 8

The above matrix can also be used to calculate the one standard deviation errors on the parameters. This can be calculated by –

```
np.sqrt(np.diag(array2))
```

```
In [17]: errors = [np.sqrt(fit[i][i]) for i in [0,1,2]]
         print(errors)
         print('[    +/- x,              +/- y,           +/- z      ]')
```

```
[8374.73467688218, 0.0005794519942049622, 27419149.077146575]
[    +/- x,              +/- y,           +/- z      ]
```

Fig no: 9

If the Jacobian matrix at the solution doesn't have a full rank, then 'lm' method returns a matrix filled with np.inf, on the other hand 'trf' and 'dogbox' methods use Moore-Penrose pseudoinverse to compute the covariance matrix.

After, the calculation of our required parameters. We switch to making prediction regarding the infection_end time. This is calculated using fsolve function of python. This function again comes under scipy.optimize library of python. This function returns the roots of a non-linear equation/function which is defined by f(x) = 0. This function also allows a variety of parameters that can be used to obtain the favourable solution and also aids in solving complex equations.

```
scipy.optimize.fsolve(func, x0, args=(), fprime=None, full_output=0, col_deriv=0, xtol=1.49012e-08, maxfev=0, band=None,
epsfcn=None, factor=100, diag=None)
```

Fig no: 10

Out of all the parameters that can be used, we used certain essential parameters that can't be avoided while calling the function. These are:

1. 'func' – The function that needs to be solved and takes at least one vector input argument. It will also return a value of same length as input. Here, we take it as f(a1, x1, y1, z1) – int(z1),
2. 'x0' – The initial value estimate for solving the 'func'. Here, we take it as y1.
3. 'maxfev'- It represents the maximum number of calls to the function and it is an optional parameter. But due to calculation constraints, we need to specify it to equal to 5000.

```
x1 = covariance[0]
y1 = covariance[1]
z1 = covariance[2]
a1 = list(df.iloc[:,0])
infection_end = int(fsolve(lambda a1 : f(a1,x1,y1,z1) - int(z1),y1, maxfev = 5000))
```

Fig no: 11

Now, using the exponential function, we will calculate the y_predicted values, i.e., number of cases reported on each date determined by exponential model. These will be calculated upto the date the infection period ends.

```
x_param = list(range(max(x),infection_end))
y_predicted = [f(i,covariance[0],covariance[1],covariance[2]) for i in x+x_param]
```

Fig no: 12

Finally, the we are now ready to plot the curves. Our plot will have two curves:

1. The first curve will be between the actual numbers vs number of days as provided by the dataset.
2. The second curve will be between the values predicted by the exponential function and number of days which here is a list of days from start to the time where infection ends.

In [21]:
```
#Plotting curve for exponential model and actual numbers
x = list(df.iloc[:,0])
y = list(df.iloc[:,1])
x_param = list(range(max(x),infection_end))
plt.rcParams['figure.figsize'] = [8, 5]
plt.rc('font', size= 15)
plt.scatter(x,y,label = "Real data",color = "m")
# Predicted logistic curve
plt.plot(x+x_param, y_predicted, label="Exponential model" )
plt.legend()
plt.xlabel("Number of days since 24 February 2020   ---->")
plt.ylabel("<----       Number of infected people       ---->")
#plt.ylim((min(y)*0.9,1))
plt.show()
```

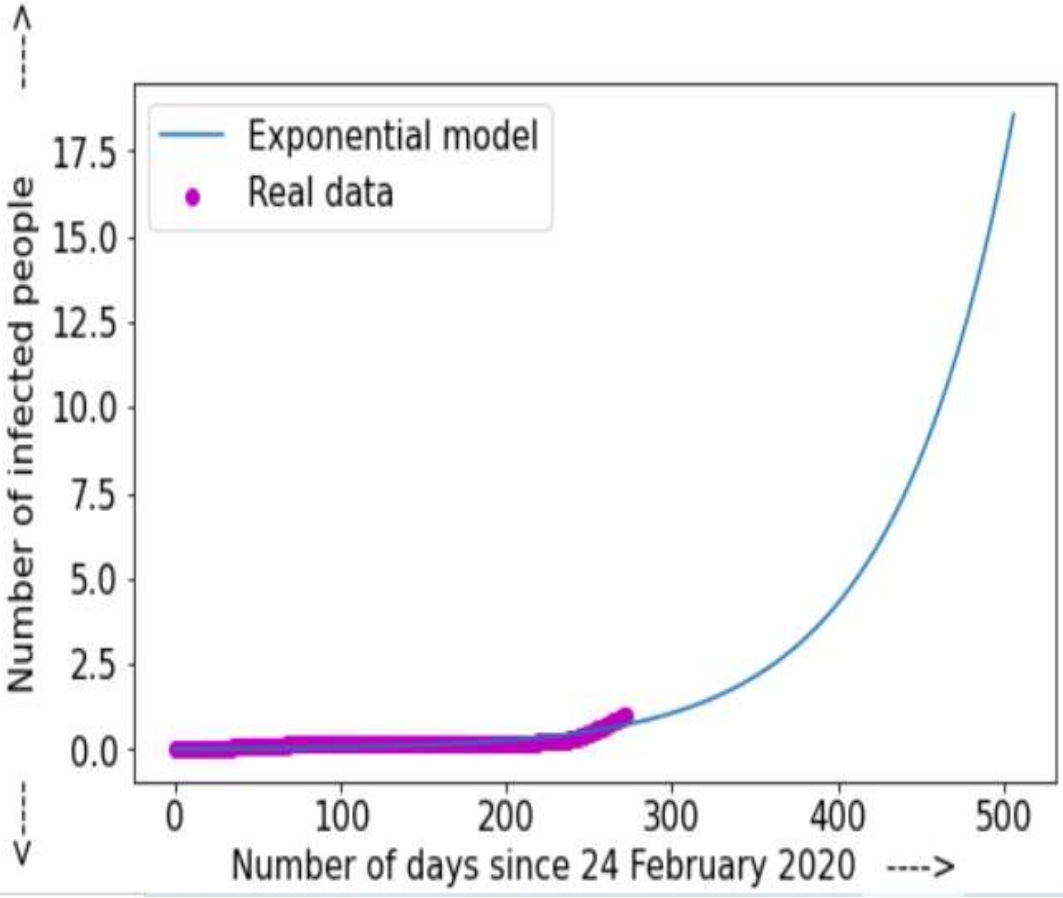


Fig no: 13

In above image, purple circles represent the actual data and the blue line represents the values of the exponential function.

And finally calculating the error of our model. Since, we don't have the actual values till the time the infection ends, we only calculate error using the values, whose actual data is present in the dataset, i.e., until the current date. We will be using mean_squared_error technique to find out the error.

```
#calculating mean-squared error
y_predicted = [f(i,fit[0][0],fit[0][1],fit[0][2]) for i in x]
expo_model = mean_squared_error(y, y_predicted)
print(expo_model)
```

```
0.07840510615806352
```

Fig no: 14

5. LOGARITHMIC MODEL

Logistic model was developed by Belgian mathematician Pierre Verhulst (1838) who suggested that the rate of population increase may be limited, i.e., it may depend on population density:

$$r = r_0\left(1 - \frac{N}{K}\right)$$

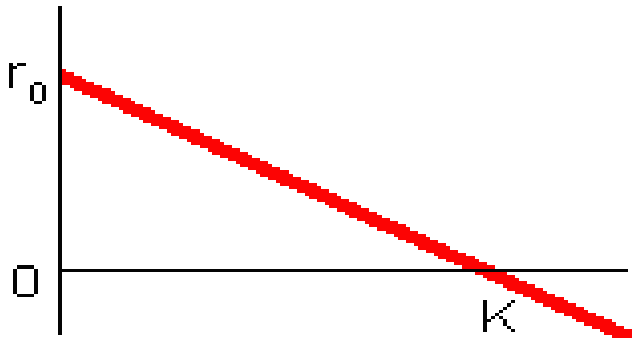


Fig no: 15

At low densities (N < < 0), the population growth rate is maximal and equals to ro. Parameter ro can be interpreted as population growth rate in the absence of intra-specific competition.

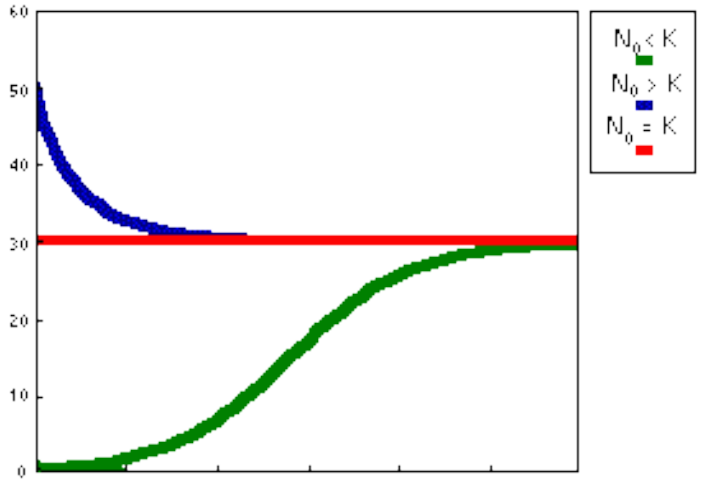
Population growth rate declines with population numbers, N, and reaches 0 when N = K. Parameter K is the upper limit of population growth and it is called carrying capacity. It is usually interpreted as the amount of resources expressed in the number of organisms that can be supported by these resources. If population numbers exceed K, then population growth rate becomes negative and population numbers decline. The dynamics of the population is described by the differential equation:

$$\frac{dN}{dt} = rN = r_0N\left(1 - \frac{N}{K}\right)$$

which has the following solution:

$$N_t = \frac{N_0 \cdot K}{N_0 + (K - N_0) \cdot \exp(-r_0 \cdot t)}$$

Three possible model outcomes



1. Population increases and reaches a plateau (No < K). This is the logistic curve.
2. Population decreases and reaches a plateau (No > K)
3. Population does not change
4. (No = K or No = 0)

Fig no: 16

Logistic model has two equilibria: N = 0 and N = K. The first equilibrium is unstable because any small deviation from this equilibrium will lead to population growth. The second equilibrium is stable because after small disturbance the population returns to this equilibrium state.

Logistic model combines two ecological processes: reproduction and competition. Both processes depend on population numbers (or density). The rate of both processes corresponds to the mass-action law with coefficients: ro for reproduction and ro/K for competition.

Interpretation of parameters of the logistic model

Parameter ro is relatively easy to interpret: this is the maximum possible rate of population growth which is the net effect of reproduction and mortality (excluding density-dependent mortality). Slowly reproducing organisms (elephants) have low ro and rapidly reproducing organisms (majority of pest insects) have high ro. The problem with the logistic model is that parameter ro controls not only population growth rate, but population decline rate (at N > K) as well. Here

biological sense becomes not clear. It is not obvious that organisms with a low reproduction rate should die at the same slow rate. If reproduction is slow and mortality is fast, then the logistic model will not work.

Parameter K has biological meaning for populations with a strong interaction among individuals that controls their reproduction. For example, rodents have social structure that controls reproduction, birds have territoriality, plants compete for space and light. However, parameter K has no clear meaning for organisms whose population dynamics is determined by the balance of reproduction and mortality processes (e.g., most insect populations). In this case the equilibrium population density does not necessary correspond to the amount of resources; thus, the term "carrying capacity" becomes confusing. For example, equilibrium density may depend on mortality caused by natural enemies.

Working for model

To start with logarithmic model, we again have to start with first defining a function to calculate values for our logarithmic model, which can further be plotted to get curves. This function will again generate values on the basis of four parameters: one is the attribute from the dataset and rest three are parameters generated from curve_fit function of the scipy.optimze. Our function definition is:

In [42]:
```
#Defining Logarithmic Model
# Logistic function that we will use: L(a, k, l, m): m/(1 + e^-((a - l)/k))
#a => time
#1/k => refers to the infection speed
#l => possible inflection point. dL/dx starts to decrease
#m => total number of recorded infected people at the end of this infection


def L(a,k,l,m):
    return (m/(1+np.exp(-(a-l)/k)))
```

Fig no: 17

The above formula makes correspondence with the logistic regression function that is used in machine learning:

$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}}$$

Standard Logistic Regression Formula

Now, we will start by using the curve_fit function again to fit to the logarithmic function using non-linear least squares method. Apart from all the parameters used previously to fit to the exponential functions, we will be using one additional parameter in the function definition here. This parameter is:

'p0' - a set of initial values for the parameters whose value is to be determined. The length of this array or object is to be equal to the number of parameters we want to estimate. In our case, since we wish to estimate 3 parameters, we passed an array of size 1*3 having initial values as (100, 1, 20) after trying and testing on certain different set of values. If we do not specify any initial values, all parameters will be initialized as 1.

Also, now in place of exponential function we will be using the logarithmic function, and value for the parameter maxfev here will be reduced to 5000 as that seems sufficient to solve these equations, therefore:

```
In [44]: covariance, fit = curve_fit(L,x,y,p0=[2,100,20000], maxfev=5000)
```

Fig no: 18

Optimal values of the parameter and the variances and covariances are again stored in a 2d array format:

```
In [45]: print(fit)
         print('covariances diagonal elements')
         print('x = ', fit[0])
         print('y = ', fit[1])
         print('z = ', fit[2])
         print(covariance)
```

```
[[9.04275139e+01 2.99816845e+07 5.06393495e+10]
 [2.99816845e+07 1.10546708e+13 1.86715149e+16]
 [5.06393495e+10 1.86715149e+16 3.15364859e+19]]
covariances diagonal elements
x =  [9.04275139e+01 2.99816845e+07 5.06393495e+10]
y =  [2.99816845e+07 1.10546708e+13 1.86715149e+16]
z =  [5.06393495e+10 1.86715149e+16 3.15364859e+19]
[7.21992386e+01 1.14096099e+03 1.22392847e+05]
```

Fig no: 19

Errors can be calculated as:

```
In [46]: errors = [np.sqrt(fit[i][i]) for i in [0,1,2]]
         print(errors)
         print('[    +/- x,              +/- y,          +/- z      ]')
```

```
[9.509338245271753, 3324856.50547023, 5615735561.0862665]
[    +/- x,              +/- y,          +/- z      ]
```

Fig no: 20

Now, computing the time to end infection according to logarithmic model is:

```
In [47]: x1 = covariance[0]
         y1 = covariance[1]
         z1 = covariance[2]
         a1 = list(df.iloc[:,0])
         infection_end = int(fsolve(lambda a1 : L(a1,x1,y1,z1) - int(z1),y1, maxfev = 5000))
```

Fig no: 21

Finally, we will plot the curves. Our plot will have two curves:

1. The first curve will be between the actual numbers vs number of days as provided by the dataset.
2. The second curve will be between the values predicted by the exponential function and number of days which here is a list of days from start to the time where infection ends.

```
In [50]: #Plotting curve for logarithmic model and actual numbers
x = list(df.iloc[:,0])
y = list(df.iloc[:,1])
x_param = list(range(max(x),infection_end))
plt.rcParams['figure.figsize'] = [8, 5]
plt.rc('font', size= 15)
plt.scatter(x,y,label = "Real data",color = "m")
# Predicted logistic curve
plt.plot(x+x_param, y_predicted, label="Logistic model" )
plt.legend()
plt.xlabel("Number of days since 24 February 2020   ---->")
plt.ylabel("<----       Number of infected people       ---->")
#plt.ylim((min(y)*0.9,1))
plt.show()
```

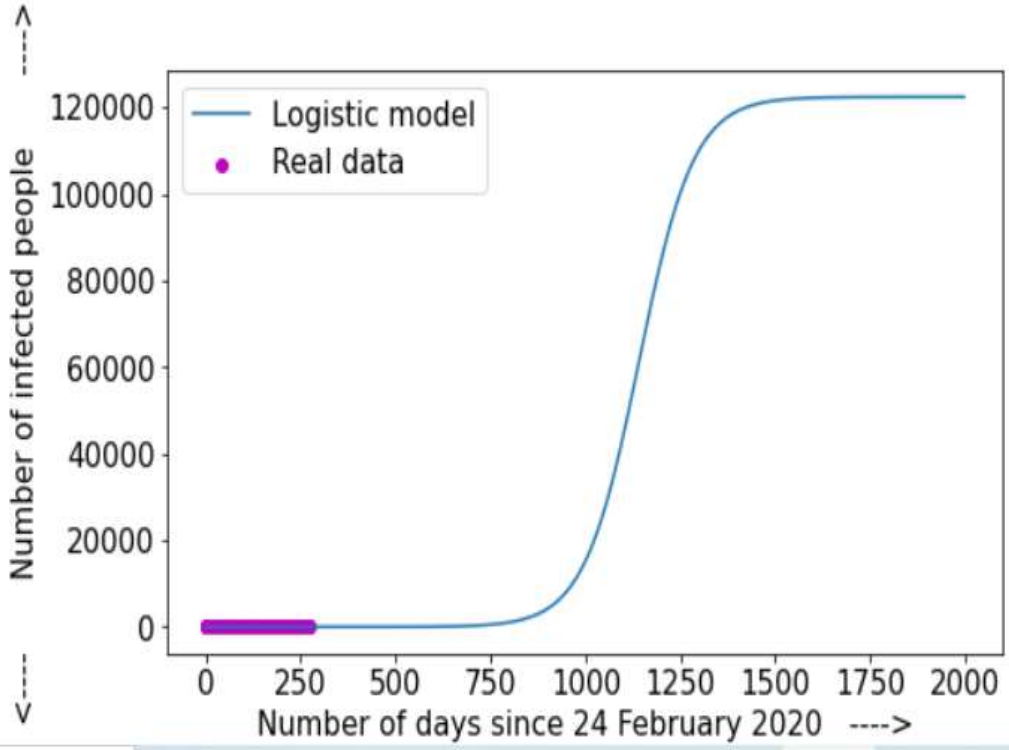


Fig no: 22

Finally, calculating the mean square error:

```
#Calculating mean-squared error
y_predicted =  [f(i,exp_fit[0][0], exp_fit[0][1], exp_fit[0][2]) for i in x]
mean_squared_error(y,y_predicted)
```

```
0.023608610093890075
```

Fig no: 23

6. SQM

S: System will include the population of Italy.

Q: Predict which model more closely resembles the corona virus cases curve of Italy and hence what is more likely to be possible scenario of curve in future.

M: It will include equations that will be used to build the logarithmic and exponential models.

Parameters, Assumption, Ristriction

Parameters -

Date, Number of cases on each date

Assumptions -

- Data provided is correct.
- No data related to vaccine or any medical research is taken into account.
- No data related to immune period of patients who have once faced corona is taken into account.

Restrictions –

- Number of cases on each date >= 0.
- Data present is from the date on which first case arrived.

7. CLASSIFICATION OF MODELS

Instationary Model – Because it does have involvement of time‘t’. We are looking for cases on each day.

Lumped Model – None of the parameter here depends on spatial coordinates or space.

Phenomenological/Statistical model – The model foregoes any attempt to explain why the variables interact the way they do, and simply attempts to describe the relationship.

8. CONCLUSION

Through above study and experiments we can clearly see the mean-squared errors f or both the models, i.e., exponential model and logistic model. Exponential model shows an error of 0.07840510615806352 and the logistic model shows an error of 0.023608610093890075.

Now, since the error given by logistic model is lesser than the error given by the exponential model, we can conclude that for the case study of Italy's COVID-19 Numbers, the logistic model is a better fit.