P1-MS960

Bryan Alves do Prado - 195171
Hugo Ricardo Ribeiro Matarozzi - 155743
Isabella Mi Hyun Kim - 170093

Agosto 2021

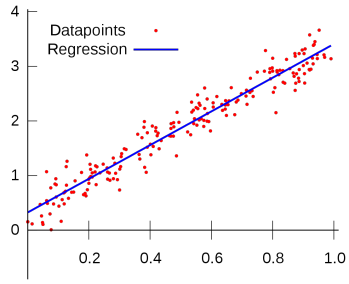
Conteúdo

1 Introdução da Primeira Parte:

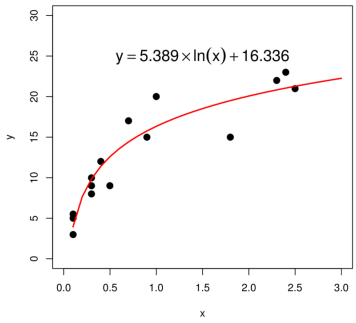
Tal projeto se trata de uma modelagem e análise de dados com a utilização de regressão linear e não linear acerca do início da pandemia de COVID-19 no Brasil. Para tal, utilizamos dados de entrada compostos por 134 dias e o respectivo acumulado de casos para cada um, e em seguida, implementamos modelos de forma que representassem o problema abordado da melhor forma possível.

2 Regressão Linear/Não-linear e o Gradiente de Descida:

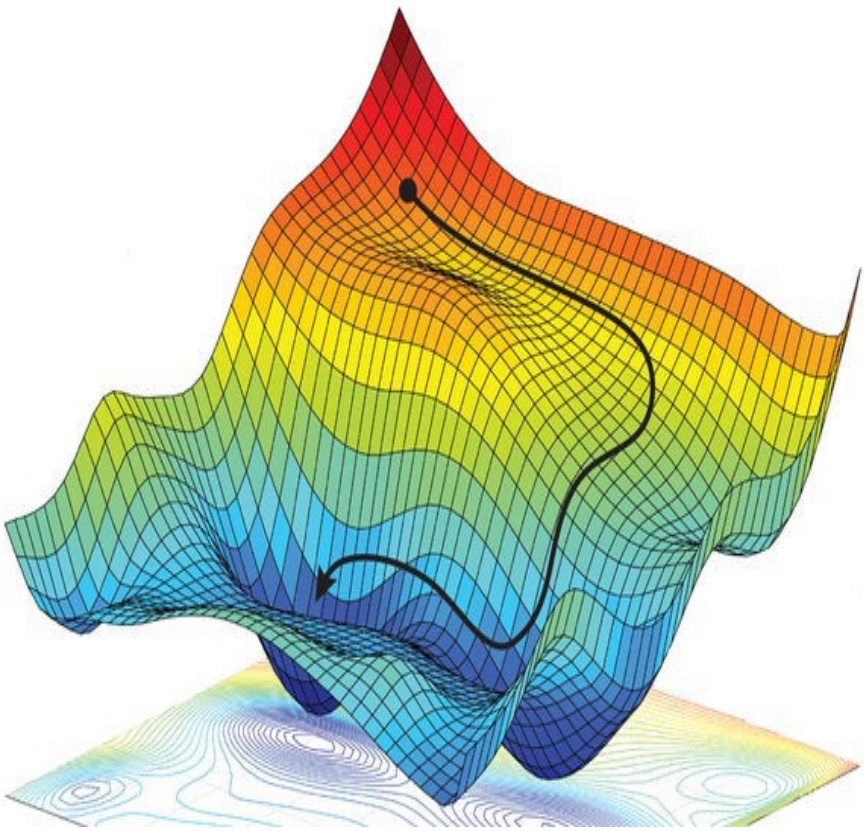
Para o problema abordado achamos adequada a utilização da regressão linear/não-linear, tal técnica consiste em traçar uma reta (Ou curva para o caso não-linear) de maneira que a mesma sirva como uma função que relacione váriaveis diferentes, ou seja, uma função que estime um valor y a partir de variáveis de entrada. Abaixo, um exemplo de regressão linear:



Para o caso não-linear, podemos fazer aproximações utilizando polinômios de graus variados, exponenciais, logaritmos e entre outros. Um exemplo de uma aproximação logarítmica se encontra abaixo:



Com intuito de obter a melhor curva possível que se adeque aos dados, utilizaremos como algoritmo o método do gradiente descendente, tal método tem como base a utilização do negativo do gradiente, ou seja, a direção e o sentido onde se tem o maior incremento em direção ao mínimo local/global, tal incremento é regulado também pela taxa de aprendizado, de forma iterativa, ou seja, a cada iteração, o algoritmo se aproxima mais do ponto de otimização. Para o caso de busca pela melhor curva de ajuste aos dados, o que estamos tentando minimizar é a diferença entre os pontos "originais" pré oferecidos, e os pontos que serão encontrados pela função $h(x)$ que melhor os ajusta, ou seja, utilizando o erro quadrático médio, teremos algo da forma $J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$, e utilizando o gradiente descendente para minimizar o custo e obter os θ ideais, teremos $\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{j=1}^{m}(h_\theta(x^{(i)}) - y^i)x^i$, sendo alfa uma taxa de aprendizado que pode ser ajustada. Com esse algoritmo, e um número de iterações bom, podemos encontrar uma curva que se adeque bem ao problema proposto.
Abaixo temos um exemplo de como o algoritmo, quando bem ajustado, se comporta:



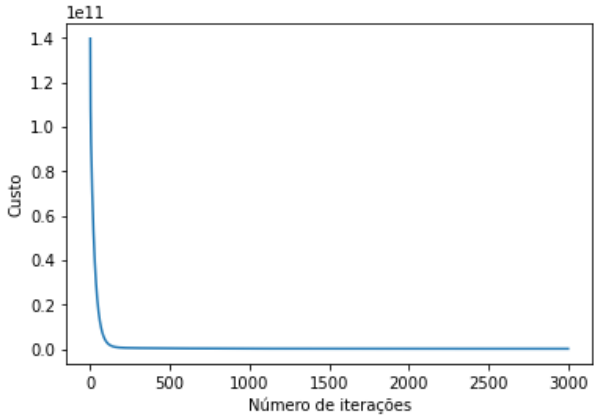
3 Primeira Questão da Primeira Parte:

3.1 Abordagem:

Para este primeiro problema, desenvolvemos um algoritmo em Python-3 que permite ao usuário a escolha do grau do polinômio, entre 1 e 10, que será utilizado na modelagem da curva que melhor se adequa aos dados disponibilizados acerca do início da COVID-19 no Brasil, tal algoritmo também permite ao usuário a visualização do gráfico da curva obtida pelo modelo, ou a curva de custos em função do número de iterações, além de uma previsão sobre o número de casos em um dado dia através da função $h(x)$ obtida pelo modelo. Alguns dados extras encontram-se na seção de anexos.

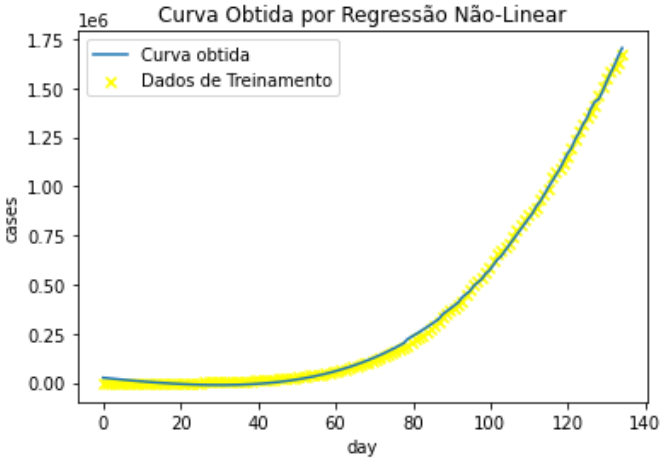
3.2 Análise dos Polinômios de Terceiro, Quinto e Décimo Grau:

Para todos os casos, iniciamos com uma abordagem de normalização, pela fórmula $x_{norm} = \dfrac{x_j - min(x_j)}{max(x_j) - min(x_j)}$ apenas para os valores de entrada (dias) X, porém, tal abordagem, por conta da alta propagação de erros gerada em decorrência do tamanho e discrepância dos valores de Y, ocasionou um valor de custo J consideravelmente alto, como podemos vislumbrar no gráfico abaixo para o caso de um polinômio de grau dez com taxa de aprendizado fixada em 0.88, qualquer valor superior da taxa de aprendizado fazia o modelo crescer os custos constantemente com 3000 iterações:

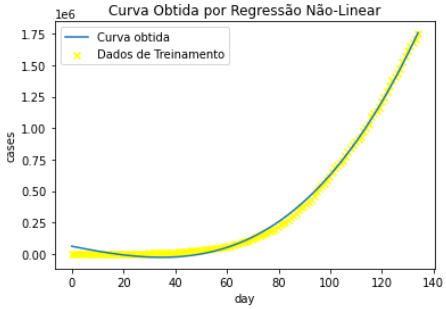


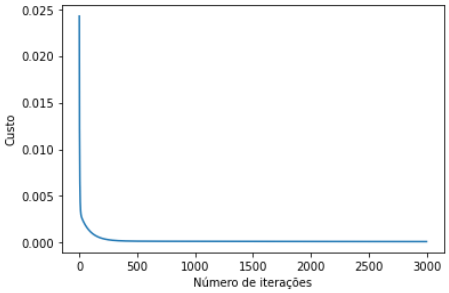
Como podemos ver pela escala do gráfico, o custo chegou ao patamar de 10^{11}, apesar de decrescer com uma boa velocidade e gerar uma curva que se ajusta bem aos dados, uma função $h(x) = 15817.3 - 283227x + 310355x^2 + 646478x^3 + 655612x^4 + 477915x^5 + 223297x^6 - 45615.3x^7 - 298001x^8 + 29160.5x^9 -$

$9705.36x^{10}$, como podemos ver no gráfico seguinte, onde os valores dos dias já foram ajustados para melhor visualização:

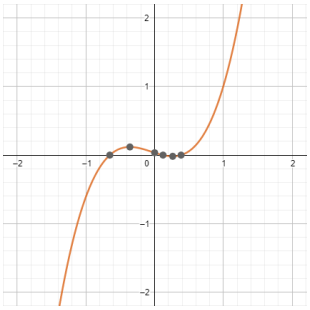


Assim, como forma de corrigir essa questão do alto valor de custo, decidimos também normalizar os valores de saída (casos) Y com a mesma fórmula anterior, $y_{norm} = \dfrac{y_j - min(y_j)}{max(y_j) - min(y_j)}$, e com isso, reduzimos bruscamente a escala do custo J, além de acelerar consideravelmente o processo do algoritmo do gradiente de descida, tornando assim, o algoritmo melhor otimizado. Tais detalhes acerca do funcionamento do código como um todo serão tratados futuramente em uma seção própria na qual abordaremos cada função implementada.
Com isso, iniciamos a análise primeiro para o polinômio de terceiro grau, onde através do modelo obtivemos um vetor $Theta$=[0.034618, -0.319849, 0.16427 , 1.12596], o que nos leva para uma função $h(x) = 0.034618 - 0.319849x + 0.16427x^2 + 1.12596x^3$, e um gráfico de curva, novamente já ajustado em suas escalas para melhor visualização, e outro de custos com a seguinte faceta:



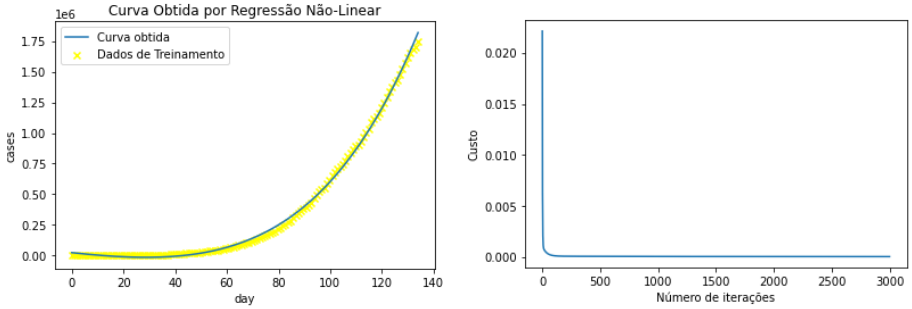


Novamente os parâmetros utilizados foram de taxa de aprendizado fixada em 0.88 e 3000 iterações. Como é possível de se observar, o ajuste para os dados foi adequado, porém, encontramos uma faixa, entre aproximadamente 10 e 50 dias, onde a curva se ajustou de tal forma que os valores obtidos pela função $h(x)$ são negativos, algo que é completamente irreal quando tratamos sobre o acumulado de casos de COVID-19 em uma pandemia, assim, consideramos que uma previsão utilizando o modelo é melhor acurada a partir de uma faixa de dias maior ou igual a 50. Para uma melhor análise do modelo, plotamos também, com a utilização do software GeoGebra, a curva geral da função obtida (Repare que nesse caso, a escala do eixo x e y não está ajustada para a do problema tratado, mas sim para os valores normalizados), e não somente a faixa onde estão inseridos os dados de entrada, com isso, obtivemos:

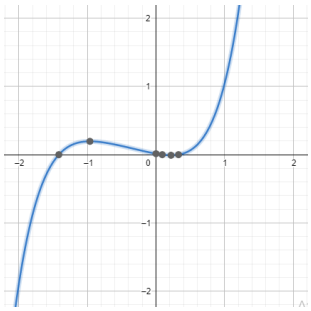


Assim, conseguimos visualizar que o modelo satisfaz de forma considerável o fenômeno retratado, visto que a função não possui pontos de inflexões em sua extensão, o que simula bem o comportamento exponencial da propagação pandêmica para previsões futuras.

Tratando agora para o caso do polinômio de quinto grau, ou seja, $n = 5$, obtivemos um vetor $Theta$ =[0.0142429, -0.180597, 0.233018, 0.42388, 0.369441, 0.178665], o que nos leva à função $h(x) = 0.0142429 - 0.180597x + 0.233018x^2 + 0.42388x^3 + 0.369441x^4 + 0.178665x^5$, com isso, o gráfico de curva da função ajustado aos dados iniciais e o gráfico de custos foram os seguintes:

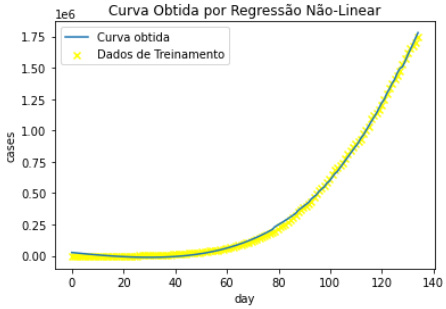


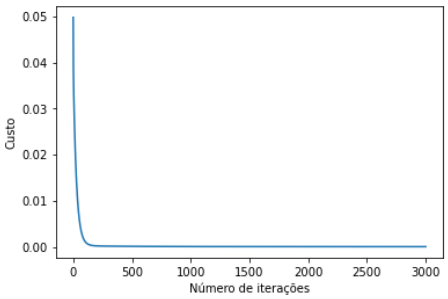
Como padrão, os parâmetros utilizados foram os mesmos do caso anterior, assim é facilitada a comparação. Novamente, se trata de uma boa aproximação para os dados de entrada e saídas oferecidos, moldando bem a curva sobre eles, com uma precisão inclusive maior do que para o caso de $n = 3$, e assim como no caso anterior, nesse também plotamos o gráfico com a curva geral da função, obtendo:



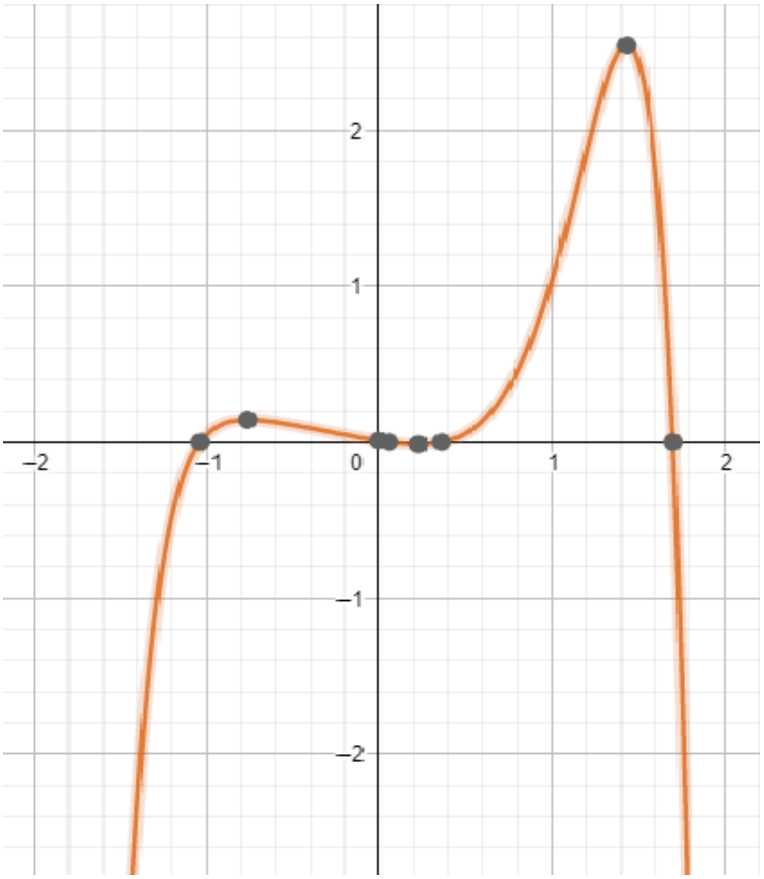
Assim como no caso anterior, é observável a ausência de pontos de inflexão na faixa de extensão para previsões futuras, o que auxilia na correspondência com o fenômeno exponencial esperado em uma pandemia, mas novamente, observamos o mesmo problema retratado no caso anterior, para uma faixa de dias (Aproximadamente entre 10 e 50 novamente) a curva foi ajustada de forma que a função devolve valores negativos para o x correspondente, o que indica uma imprecisão do modelo.

Utilizando novamente os mesmos parâmetros, tratamos acerca do polinômio de décimo grau. Para tal polinômio, obtivemos $Theta$ =[0.00944107, -0.169064, 0.185258, 0.385896, 0.391349, 0.285277, 0.133291, -0.0272287, -0.177883, 0.0174065, -0.00579334], o que nos leva para uma função $h(x) = 0.00944107 - 0.169064x + 0.185258x^2 + 0.385896x^3 + 0.391349x^4 + 0.285277x^5 + 0.133291x^6 - 0.0272287x^7 - 0.177883x^8 + 0.0174065x^9 - 0.00579334x^{10}$, obtendo assim uma curva ajustada para os dados do problema e um gráfico de custos da seguinte forma:





É visível que se trata de uma ótima aproximação, com uma precisão considerável dada a dispersão dos dados, no entanto, observa-se novamente a questão de uma faixa de dias, que apesar de nesse caso de $n = 10$ ser menor, onde o valor entregue pela função obtida é negativo, além de, constatarmos um problema relacionado a utilização desse modelo para previsões futuras, visto que o mesmo não segue continuamente o comportamento exponencial relatado em contextos de curvas pandêmicas, como podemos observar no gráfico geral da curva plotado:



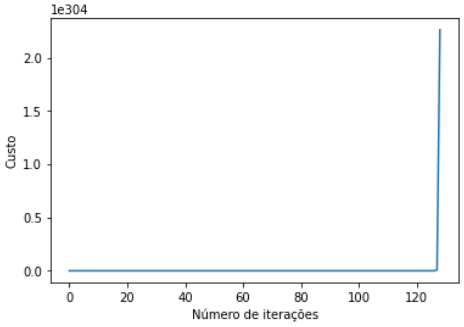
Dessa forma, tomamos a derivada de $h(x)$ e igualamos a zero, e assim, encontramos que o ponto de inflexão visível no gráfico ocorre em $x = 1.43778$, o que ao utilizarmos a fórmula de reconversão da normalização, $x_j = x_{norm}(max(x_j) - min(x_j)) + min(x_j)$, nos dá aproximadamente o dia 192, ou seja, apesar de um ótimo ajuste para os dados de treino, tal modelo é extremamente impreciso para previsões futuras feitas posteriormente ao dia 192, assim, recomendamos que o mesmo seja utilizado apenas em uma faixa de $49 \leq x \leq 192$, para tal faixa, sua precisão será considerável, oferecendo uma ótima curva para análise do fenômeno pandêmico.

Dessa forma, constatamos que todos os modelos se adequam relativamente bem ao problema proposto, gerando curvas que relatam com precisão considerável os dados utilizados para construção do modelo, porém, isso não ocorre com perfeição (Algo esperado dada a dispersão de dados e o comportamento exponencial), e podemos observar que por exemplo, para o caso do polinômio de décimo grau, apesar de uma ótima adequação quando observamos a faixa de dados oferecidas para a modelagem do problema, temos problemas quanto a previsões futuras, assim, ao avaliarmos o todo, talvez seja mais interessante a adoção do polinômio de quinto grau em relação aos outros dois abordados, ele possui um ótimo ajuste aos dados e um comportamento geral condizente com um fenômeno exponencial.

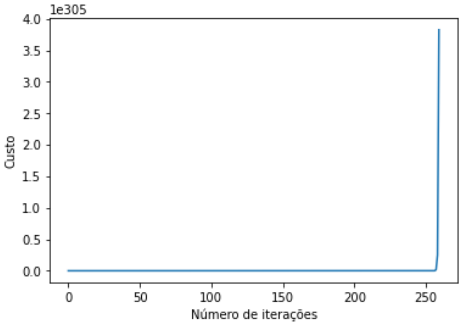
Nos anexos, possuímos dados referentes a análises com outros graus de polinômio.

3.3 Análise Acerca do Custo em Função da Taxa de Aprendizado no Caso Para Polinômios:

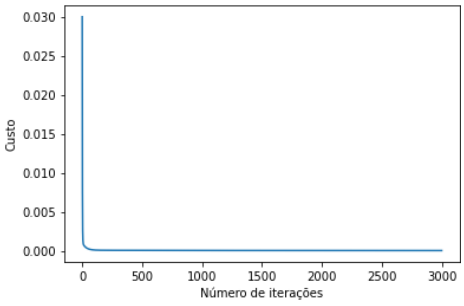
Para tal análise, decidimos escolher o polinômio de quinto grau e fixamos o número de iterações em 3000, assim, seguem-se os gráficos obtidos:



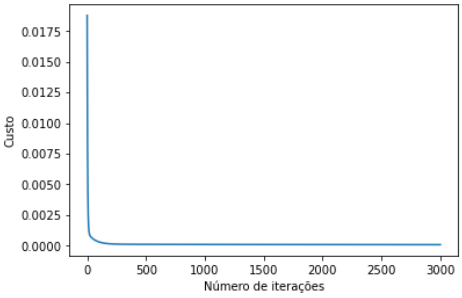
(a) Taxa de aprendizado: 10



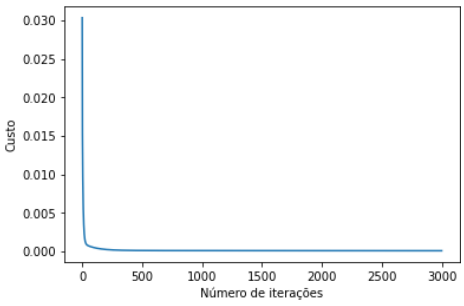
(b) Taxa de aprendizado: 3



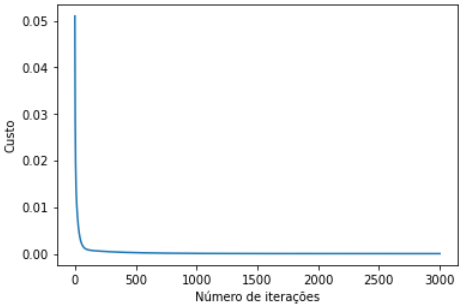
(c) Taxa de aprendizado: 1



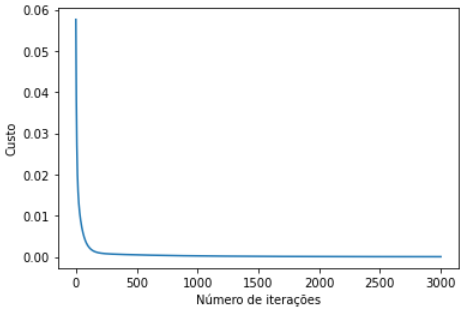
(d) Taxa de aprendizado: 0.5



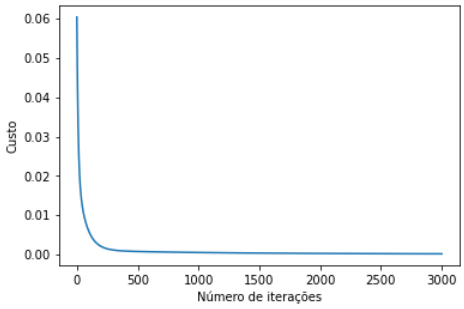
(e) Taxa de aprendizado: 0.3



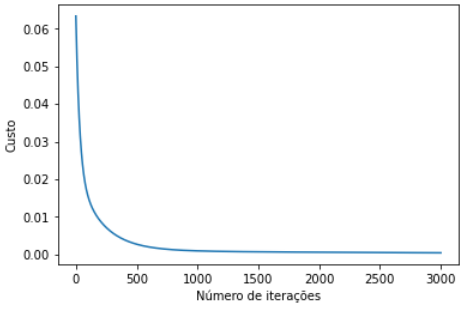
(f) Taxa de aprendizado: 0.1



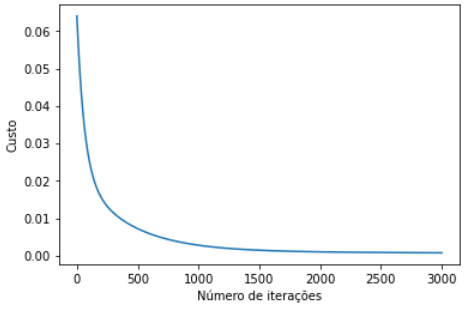
(a) Taxa de aprendizado: 0.05



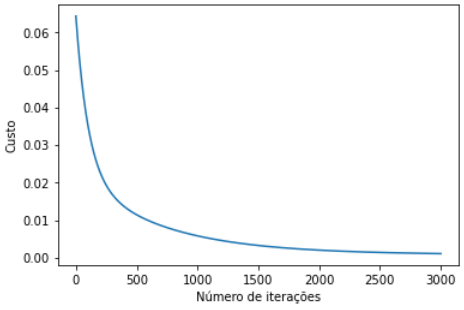
(b) Taxa de aprendizado: 0.03



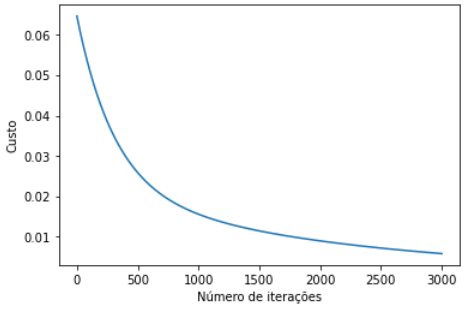
(c) Taxa de aprendizado: 0.01



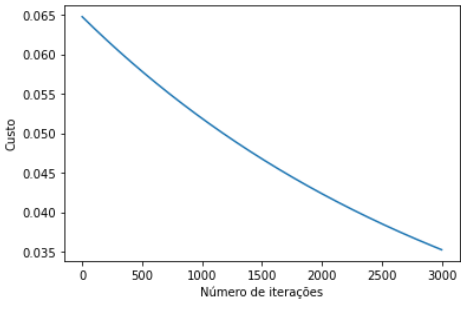
(d) Taxa de aprendizado: 0.005



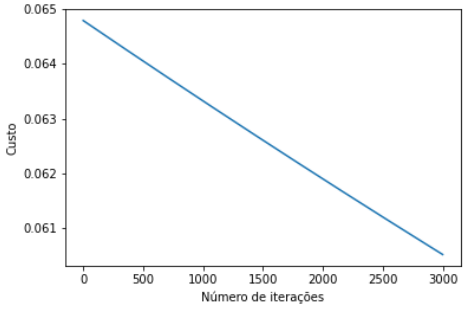
(e) Taxa de aprendizado: 0.003



(f) Taxa de aprendizado: 0.001



(g) Taxa de aprendizado: 0.0001



(h) Taxa de aprendizado: 0.00001

Assim, observando os gráficos, conseguimos constatar que a escolha de uma taxa de aprendizado muito alta, inadequada para o problema abordado, acaba por fazer com que o custo entre em um crescimento exponencial ao invés de diminuir continuamente e convergir para um dado valor, é possível também que o custo entre em uma oscilação, com períodos de crescimento e outros de decrescimento, tal possibilidade não foi observada em nossa análise, mas ela existe e se a taxa de aprendizado for escolhida de forma incorreta, pode ocorrer. É interessante observarmos também a curvatura da queda do custo, para um valor "alto" de taxa de aprendizado, ela tende a ser bem acentuada, com um comportamento de queda exponencial, em certos pontos formando até uma espécie de "reta vertical", já para valores pequenos, o comportamento se assemelha consideravelmente com algo linear, e se diminuirmos ainda mais o valor da taxa de aprendizado, se observará a curva se tornando cada vez mais horizontal, no entanto, devemos ressaltar que mesmo em menor proporção, o custo continua diminuindo, indicando assim o bom funcionamento do modelo. Com nossa rotina de testes onde utilizamos inúmeros valores para a taxa de aprendizado, pudemos constatar que valores ≤ 1.2 garantem convergência para este caso.
Segue-se um exemplo com uma taxa de aprendizado extremamente pequena, onde é visível o fenômeno de "horizontalização" da reta de queda de custos por iteração:

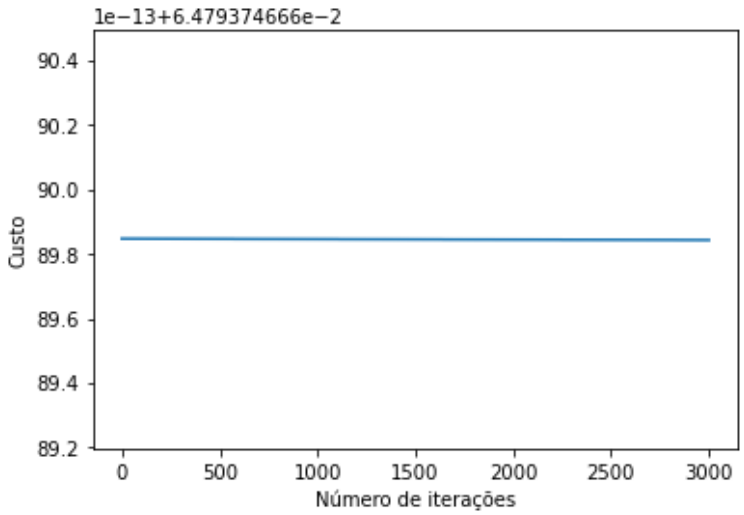


Figure 3: Taxa de aprendizado: 0.0000000000000000001

3.4 O Algoritmo em Python-3 Para o Caso dos Polinômios:

Para a implementação do modelo, utilizamos a linguagem de programação Python-3 auxiliada das bibliotecas NumPy e Pandas, a primeira principalmente para a realização de operações matemáticas com matrizes, e a segunda, para a extração e análise de dados, que foram obtidos através de um arquivo CSV com uma coluna de dias e outra de casos acumulados. Assim, segue o algoritmo implementado:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

def Creation(X, Y, n):
    A=X.reshape(m,1)
    Y=Y.reshape(m,1)
    X=np.append(np.ones([m,1]), (X).reshape(m,1),  axis=1)
    for i in range(2,n+1):
        X=np.append(X, A**(i), axis=1)
    for j in range(1,n+1):
        X[:,j]=(X[:,j]-np.amin(X[:,j]))/(np.amax(X[:,j])-np.amin(X[:,j]))
    Y=(Y-np.amin(Y))/(np.amax(Y)-np.amin(Y))
    return X, Y

def Grad_Descent(X, Y, z, c, n):
    theta=np.zeros((n+1,1))
    J=[]
    for i in range(z+1):
        b=np.dot(X.transpose(),(X.dot(theta)-Y))
        theta=theta-(c/m)*b
        J.append(Compute_Cost(X, Y, theta))
    return theta, J

def Compute_Cost(X, Y, theta):

  h = X.dot(theta)
  J = (1/(2*m))*np.sum((h-Y)**2)
  return J

def Prediction(v, a, b, theta, t, h):
    v=(v-a)/(b-a)
    K=[]
    for i in range(0,n+1):
        K.append(v**i)
    K=np.array(K)
    K=K.reshape(1,n+1)
    K=K.dot(theta)[0,0]
    return K*(h-t)+t


files=pd.read_csv(r'C:\Users\USUARIO\Desktop\projeto python\teste.csv',header=None)
X=files[0].values
Y=files[1].values
m=Y.shape[0]
a=np.amin(X)
b=np.amax(X)
t=np.amin(Y)
h=np.amax(Y)
n=int(input('Qual seria o grau do polinômio? (Entre 1 e 10) '))
r=float(input('Qual seria a taxa de aprendizado? (Para n>=7, recomendamos <=0.88) '))
o=int(input('Qual seria o número de iterações? '))
l=int(input('Tecle 1 para ver a curva obtida pelo método ou 0 para ver a curva de custos: '))
p=int(input('Tecle um dia para o qual gostaria de fazer uma previsão, caso o contrário, tecle 0: '))
X,Y=Creation(X, Y, n)
theta,J=Grad_Descent(X, Y, o, r, n)


if p!=0:
   print('\n')
   print('A previsão para o dia',p, 'é de:',int(Prediction(p, a, b, theta, t, h)), 'casos')

if l==1:
    plt.scatter(X[:,1]*134,Y*1.75*10**6,c='yellow',marker='x',label='Dados de Treinamento')
    plt.plot(X[:,1]*134,np.dot(X,theta)*1.75*10**6,label='Curva obtida')
    plt.ylabel('cases')
    plt.xlabel('day')
    plt.legend()
    plt.title('Curva Obtida por Regressão Não-Linear')
    plt.show()

if l==0:
    plt.plot(range(len(J)),J)
    plt.xlabel('Número de iterações')
    plt.ylabel('Custo')
    plt.show()
```

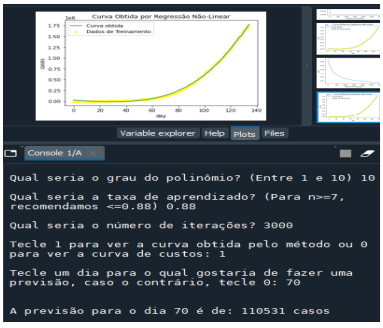
O algoritmo permite que o usuário escolha o grau do polinômio que o mesmo quer usar como base para o modelo, além do número de iterações, taxa de aprendizado e a visualização do gráfico da função sobrepondo os pontos de entrada ou gráfico de custo por número de iterações. Na "área de controle do código" (Parte onde são chamadas as funções) fazemos a leitura dos dados, e armazenamos os valores máximos e mínimos dos dados de entrada X e dos dados de saída Y, além de armazenarmos a variáveis escolhidas pelo usuário.

Em seguida, temos as funções utilizadas no código, a primeira delas, a função "$Creation$", a mesma recebe os dados X e Y, além do grau de polinômio escolhido pelo usuário, e constrói a matriz X com todos os dados normalizados coluna a coluna, o mesmo ocorre com Y.

A próxima função tratada é a "$GradDescent$", a mesma é responsável por encontrar o melhor $Theta$ possível, ou seja, os melhores coeficientes que se encaixem na função do polinômio requisitado, para tal, é iniciado um vetor coluna de zeros, tal vetor é atualizado iteração a iteração, reduzindo progressivamente o custo durante o processo, e assim, atualizando sempre para um $Theta$ melhor que o anterior (Se a escolha da taxa de aprendizado não for incorreta, claro). Inserida na função "$GradDescent$", temos a função "$ComputeCost$", a mesma é utilizada para o cálculo progressivo do custo a cada iteração, custo este que é armazenado e utilizado para a plotagem do gráfico posteriormente.

Seguindo, temos a função "$Prediction$", a mesma recebe os valores de máximo e mínimo de X e Y armazenados anteriormente, o $Theta$ obtido pelo modelo e um dia oferecido pelo usuário, assim, é calculada a normalização do valor oferecido pelo usuário, tal valor normalizado é utilizado na função $h(x)$, em seguida, o resultado é "reconvertido" para um valor não normalizado, e com isso, uma previsão é oferecida ao usuário.

Ao final temos os plots dos gráficos ajustados para que o usuário possua uma melhor visualização acerca das curvas obtidas. Por fim, segue um exemplo de resultado do código rodado na IDE "Spyder":



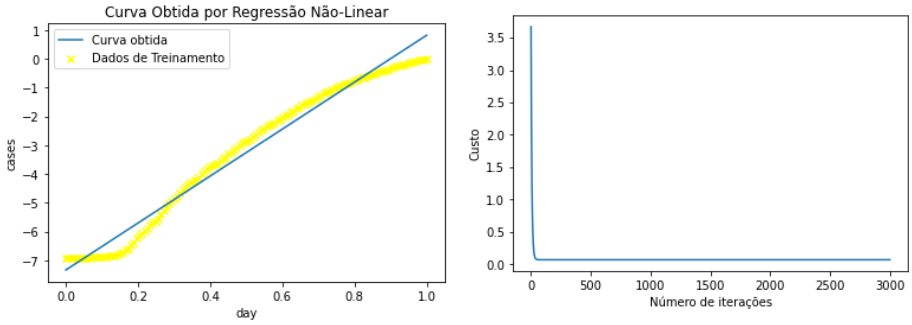
4 Segunda Questão da Primeira Parte:

4.1 Abordagem:

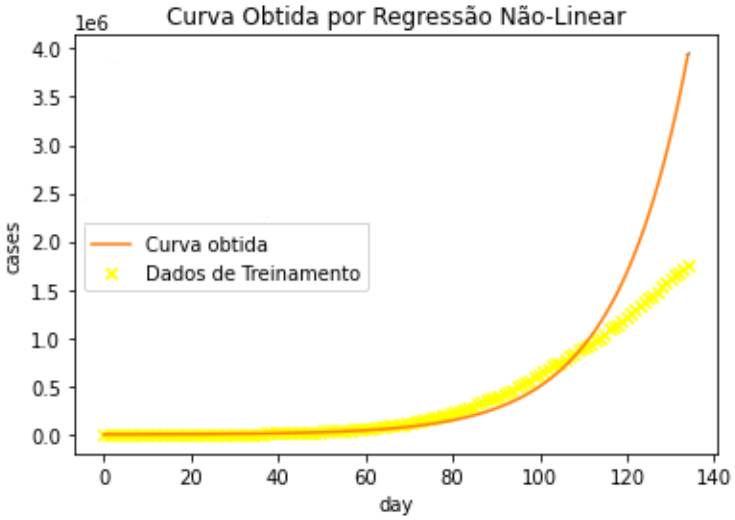
Para esta segunda questão, nossa abordagem foi semelhante a primeira, ou seja, implementamos um modelo de regressão não linear, porém, neste caso, utilizamos como parâmetro para a modelagem uma exponencial da forma $h(x) = \theta_0 e^{\theta_1 x}$, dessa forma, ao invés de utilizarmos uma aproximação de y e $h(x)$ diretamente, fizemos por $ln(y)$ e $ln(h(x))$, assim, obtivemos $ln(h(x)) = ln(\theta_0 e^{\theta_1} x) \rightarrow ln(\theta_0) + \theta_1 x$, com isso, pudemos analisar o problema como uma regressão linear simples, bastando fazer de forma que, dado um formato $f(x) = ax + b$, $a = \theta_1$ e $b = ln(\theta_0)$.

4.2 Análise Acerca da Aproximação Exponencial:

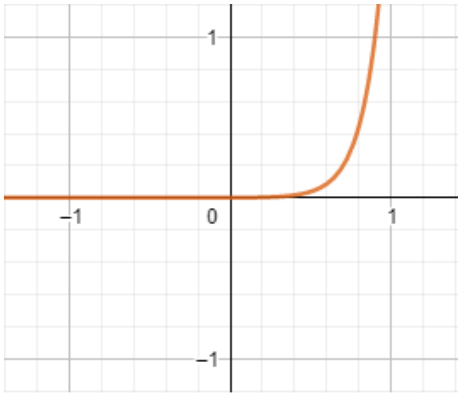
Primeiramente, realizamos a normalização dos dados de entrada X e dos dados de saída Y, seguindo a fórmula padrão $x_{norm} = \dfrac{x_j - min(x_j)}{max(x_j) - min(x_j)}$, e sua versão para Y, em seguida, aplicamos o logaritmo natural para todos os elementos de Y, tomando o cuidado de somar um "valor de folga" 0.001, visto que, com a normalização, alguns valores de Y tornaram-se zero, e isso ocasionaria um problema no cálculo do logaritmo natural. Com isso, utilizando uma taxa de aprendizado de 1.0 e 3000 iterações, obtivemos um $Theta$=[-7.33056, 8.14269], e os seguintes gráficos de curva e de custos:



Tal gráfico de curvas está ajustado nos parâmetros de minimização e $ln(y)$, ou seja, a função $h(x)$ é da forma $h(x) = -7.33056 + 8.14269x$, colocando a mesma em sua forma exponencial, teremos algo como $h(x) = 0.000655207e^{8.14269x}$, visto que $a = \theta_1$ e $b = ln(\theta_0)$, assim, ajustando os eixos e plotando a curva obtida sobreposta aos dados originais, teremos:

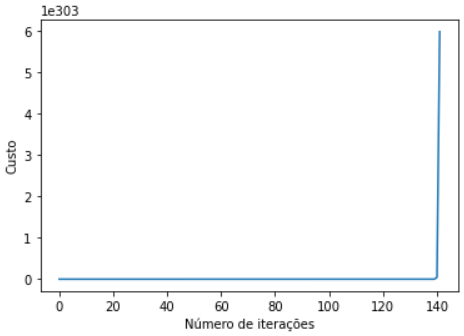


Com isso, observamos que, apesar de conseguir representar bem o fenômeno pandêmico proposto, em alguns trechos (Principalmente no final dos dados originais, a partir de 100 dias) a divergência é considerável, e ao compararmos com a representação obtida por polinômios (Focando principalmente nos que possuem um maior grau), constatamos que o mesmos conseguem se ajustar melhor ao problema. No entanto, não observamos nesse modelo o problema com faixas de valores se tornando negativas por conta da função encontrada, tal ponto é algo extremamente positivo quando analisamos um contexto de casos acumulados diários. Em um geral, podemos adequar tal modelo como uma boa representação, principalmente para visualizações futuras, dado o caráter exponencial da propagação da pandemia.
Segue-se um gráfico expandido da função obtido no GeoGebra com os parâmetros normalizados:

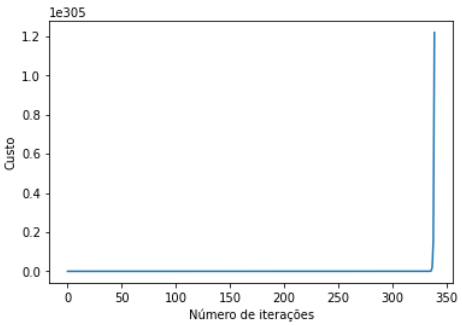


4.3 Análise Acerca do Custo em Função da Taxa de Aprendizado no Caso Exponencial:

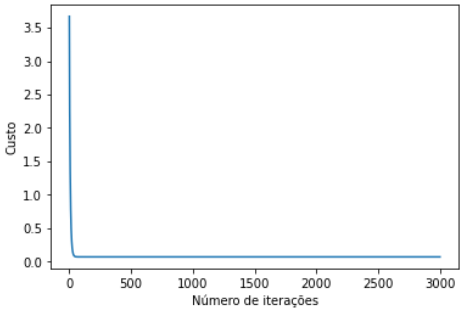
Para tal análise, fixamos o número de iterações em 3000, tornando assim o único parâmetro variável a taxa de aprendizado. Com isso, seguem-se os gráficos obtidos:



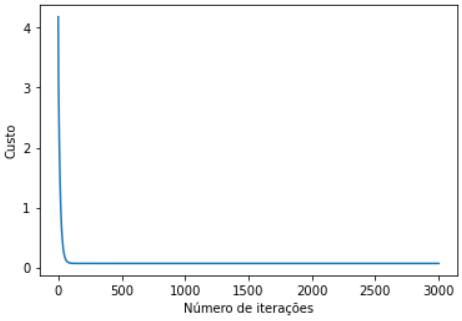
(a) Taxa de aprendizado: 10



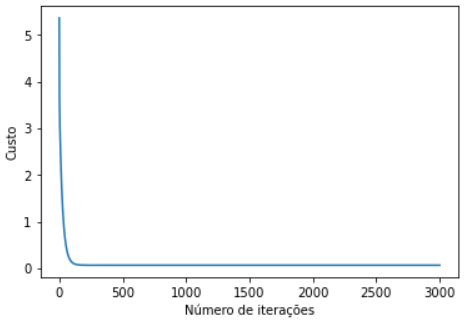
(b) Taxa de aprendizado: 3



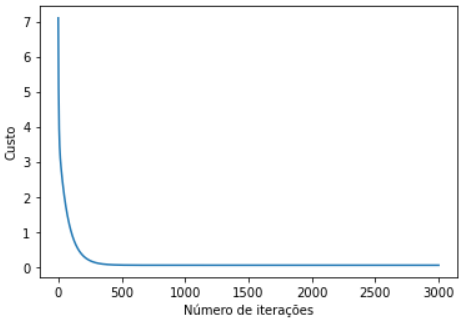
(c) Taxa de aprendizado: 1



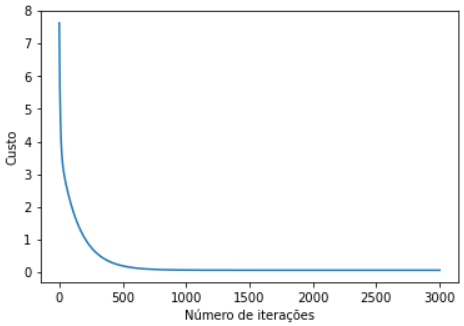
(d) Taxa de aprendizado: 0.5



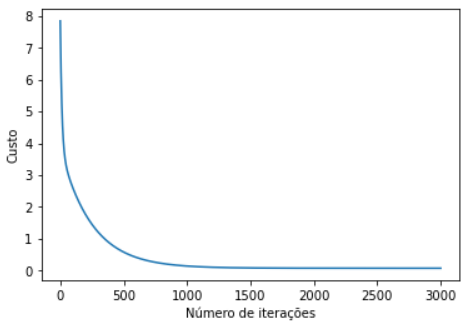
(e) Taxa de aprendizado: 0.3



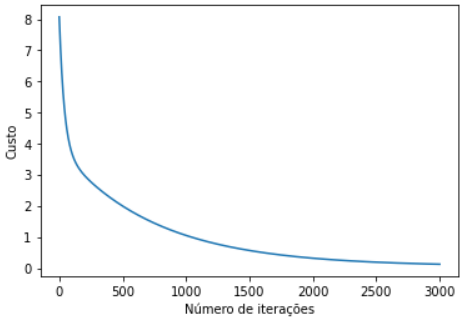
(f) Taxa de aprendizado: 0.1



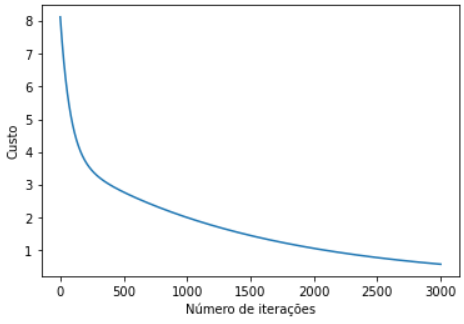
(a) Taxa de aprendizado: 0.05



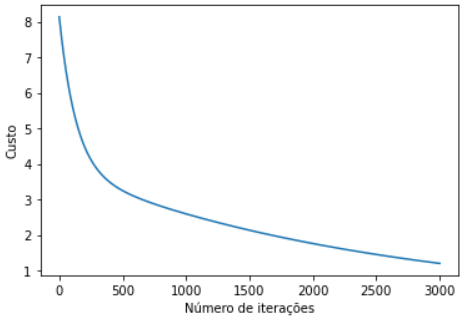
(b) Taxa de aprendizado: 0.03



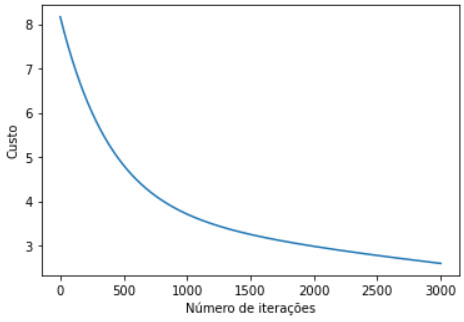
(c) Taxa de aprendizado: 0.01



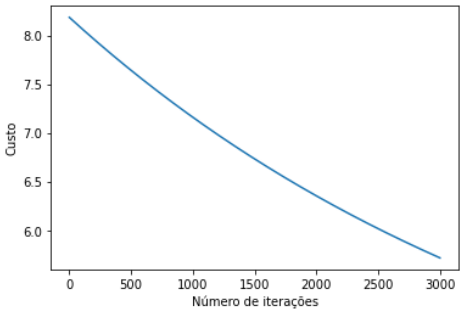
(d) Taxa de aprendizado: 0.005



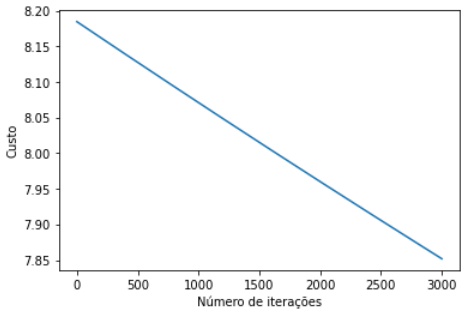
(e) Taxa de aprendizado: 0.003



(f) Taxa de aprendizado: 0.001



(g) Taxa de aprendizado: 0.0001



(h) Taxa de aprendizado: 0.00001

Observando os gráficos, constatamos que o fenômeno observado é o mesmo que visualizamos no caso para polinômios, ou seja, a escolha de uma taxa de aprendizado inadequada, muito alta, ocasiona um crescimento descontrolado do custo, enquanto que, a escolha de uma taxa de aprendizado demasiado pequena, apesar de não interferir no fator de diminuição contínuo do custo, faz com que a velocidade que o gradiente descendente atinge o melhor $Theta$ seja menor, tendo um comportamento linear, e em casos mais extremos, um comportamento quase que "horizontal". Dada as rotinas de testes que executamos com inúmeros valores possíveis, chegamos a conclusão de que uma taxa de aprendizado ≤ 1.5 garante convergência para este caso.
Assim como no caso dos polinômios, segue-se um gráfico com um caso extremo onde essa reta quase que horizontal é plenamente visível:

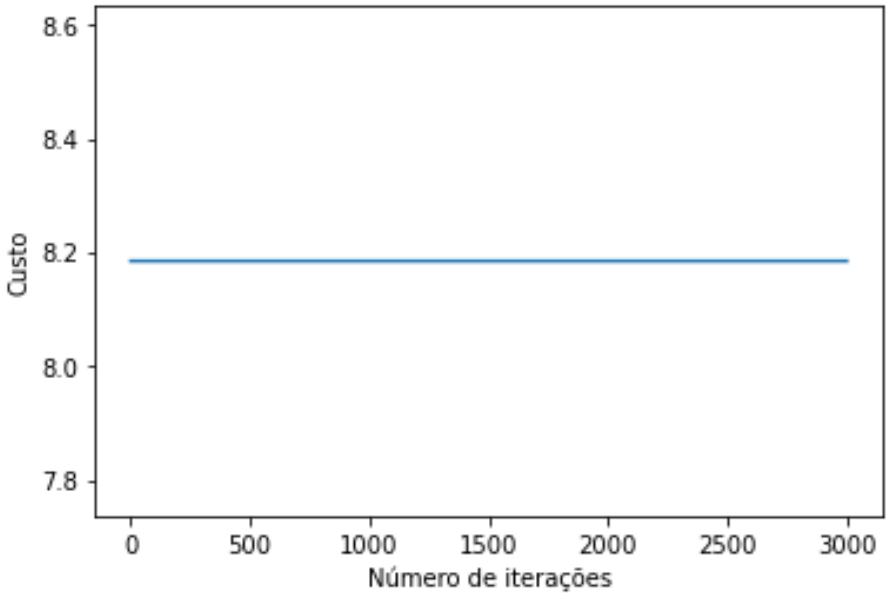


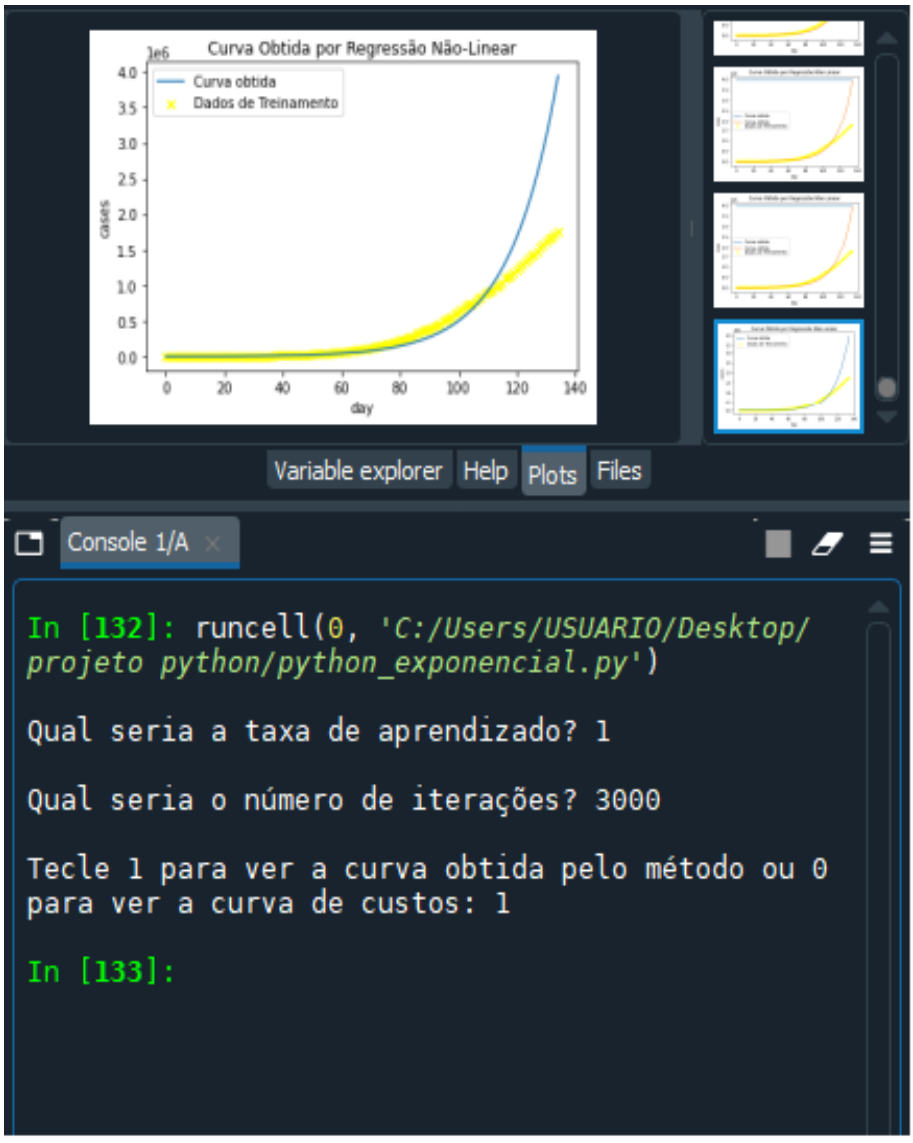
Figure 7: Taxa de aprendizado: 0.0000000000000000001

4.4 O Algoritmo em Python-3 Para o Caso Exponencial:

O algoritmo em um geral possui a mesma estrutura do caso para polinômios, a maior diferença se dá no processo de preparo de X e Y, e na ausência da função "*Prediction*", assim, o código ficou da seguinte forma:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

def Creation(X, Y, n):
    Y=Y.reshape(m,1)
    X=np.append(np.ones([m,1]), (X).reshape(m,1),  axis=1)
    for j in range(1,n):
        X[:,j]=(X[:,j]-np.amin(X[:,j]))/(np.amax(X[:,j])-np.amin(X[:,j]))
    Y=(Y-np.amin(Y))/(np.amax(Y)-np.amin(Y))
    Y=np.log((Y+0.001))
    return X, Y

def Grad_Descent(X, Y, z, c, n):
    theta=np.zeros((n,1))
    J=[]
    for i in range(z+1):
        b=np.dot(X.transpose(),(X.dot(theta)-Y))
        theta=theta-(c/m)*b
        J.append(Compute_Cost(X, Y, theta))
    return theta, J

def Compute_Cost(X, Y, theta):

  h = X.dot(theta)
  J = (1/(2*m))*np.sum((h-Y)**2)
  return J

files=pd.read_csv(r'C:\Users\USUARIO\Desktop\projeto python\teste.csv',header=None)
X=files[0].values
Y=files[1].values
m=Y.shape[0]
r=float(input('Qual seria a taxa de aprendizado? '))
o=int(input('Qual seria o número de iterações? '))
l=int(input('Tecle 1 para ver a curva obtida pelo método ou 0 para ver a curva de custos: '))
X,Y=Creation(X, Y, 2)
theta,J=Grad_Descent(X, Y, o, r, 2)

if l==1:
    plt.scatter(X[:,1]*134,np.exp(Y)*(1.75*10**6),c='yellow',marker='x',label='Dados de Treinamento')
    plt.plot(X[:,1]*134,(0.000655207*np.exp(8.14269*X))*(1.75*10**6),label='Curva obtida')
    plt.ylabel('cases')
    plt.xlabel('day')
    plt.legend()
    plt.title('Curva Obtida por Regressão Não-Linear')
    plt.show()
if l==0:
    plt.plot(range(len(J)),J)
    plt.xlabel('Número de iterações')
    plt.ylabel('Custo')
    plt.show()
```

Como já citado anteriormente, a maior mudança se encontra na função "*Creation*", retiramos o vetor A que servia como um auxiliar para a construção de X com todos os polinômios e adicionamos uma linha que calcula o logaritmo natural de cada valor de Y adicionando um valor de folga 0.001, que como já citado, funciona como uma forma de proteção contra possibilidades do caso $ln(0)$. Por fim, um exemplo do resultado obtido rodando o algoritmo na IDE "Spyder":



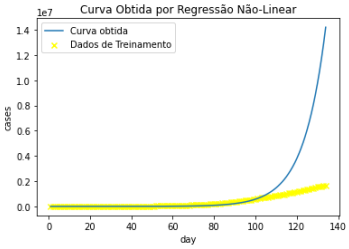
5 Terceira Questão da Primeira Parte:

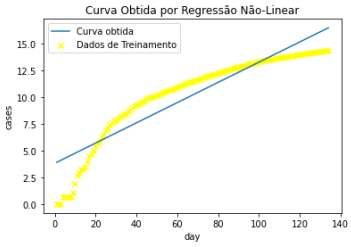
5.1 Abordagem:

Para esse problema, organizamos X e Y para que os mesmos pudessem computar o produto de matrizes $(X^T X)^{-1} X^T Y$. A grande vantagem deste método é que não precisamos nos preocupar em definir taxa de aprendizado e número de iterações adequados, para o problema retratado, apenas computamos o logaritmo natural de cada valor de Y e realizamos o produto acima.

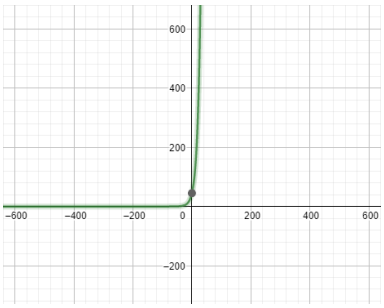
5.2 Análise do Resultado Obtido Por Equação Normal:

O algoritmo nos deu como resultado um $Theta$=[3.80887, 0.0944832], o que nos leva à função $h(x) = 3.80887 + 0.0944832x$, o que pela substituição $a = \theta_1$ e $b = ln(\theta_0)$, nos leva a função exponencial $h(x) = 45.0994e^{0.0944812x}$ e por consequências, aos seguintes gráficos:





O da esquerda se trata da exponencial plotada sobre os dados originais, enquanto o da direita se trata da reta obtida quando aproximamos $ln(Y)$ por $ln(h(x))$, o resultado obtido é condizente com o que esperávamos, uma boa abordagem para previsões futuras dado o caráter exponencial da pandemia, porém, com algumas discrepâncias consideráveis com os dados originais, muito ocasionado por conta da alta variação entre eles, o que acaba por "propagar erros" e dificultar o ajuste.
Segue-se um gráfico obtido no Geogebra onde conseguimos observar o comportamento da função obtida:



5.3 Algoritmo em Python-3 Para o Caso de Equação Normal:

O algoritmo é consideravelmente mais simples que os anteriores, mantivemos apenas a função "*Creation*" e adicionamos uma linha que calcula o produto de matrizes da equação normal, assim, obtivemos o seguinte código:

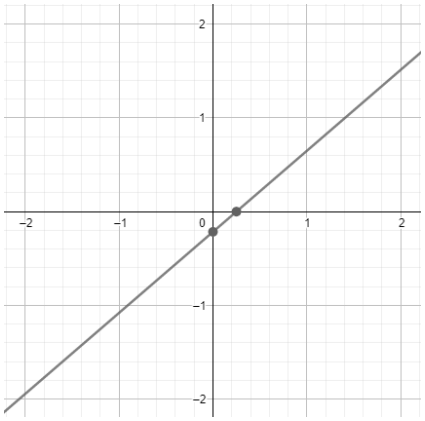
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

def Creation(X, Y, n):
    Y=Y.reshape(m,1)
    X=np.append(np.ones([m,1]), (X).reshape(m,1),  axis=1)
    Y=np.log((Y))
    return X, Y

files=pd.read_csv(r'C:\Users\USUARIO\Desktop\projeto python\teste.csv',header=None)
X=files[0].values
Y=files[1].values
m=Y.shape[0]
X,Y=Creation(X, Y, 2)
A=X.transpose()
theta=np.dot(np.dot(np.linalg.inv(np.dot(A, X)),A),Y)

plt.scatter(X[:,1],np.exp(Y),c='yellow',marker='x',label='Dados de Treinamento')
plt.plot(X[:,1],45.0994*np.exp(0.0944812*X[:,1]),label='Curva obtida')
plt.ylabel('cases')
plt.xlabel('day')
plt.legend()
plt.title('Curva Obtida por Regressão Não-Linear')
plt.show()
```

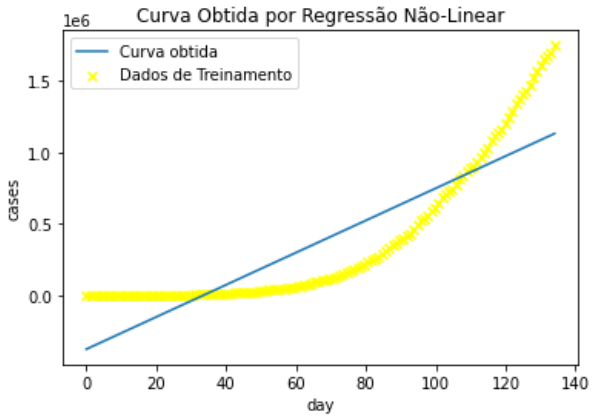
Como é fácil de observar, ele é de extrema facilidade na implementação, sendo uma ótima alternativa para problemas não tão grandes, visto que o cálculo da inversa e dos produtos matriciais demanda um considerável custo computacional. É necessário observarmos que para este caso retratado, não normalizamos os valores, enquanto que, na implementação do Jupyter notebook, sim.

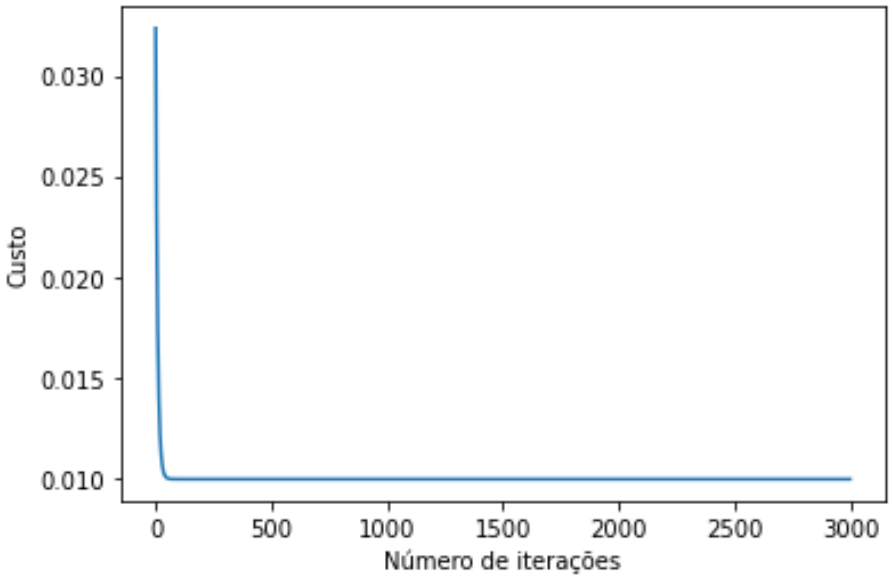
6 Anexos da Primeira Parte:

6.1 Anexos da Primeira Questão:

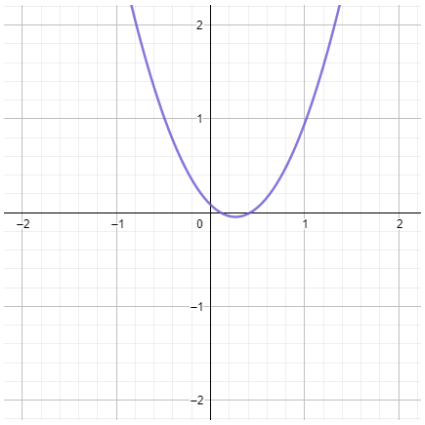
Para o polinômio de primeiro grau, com um teste de 3000 iterações e taxa de aprendizado de 0.88, obtivemos um $Theta$=[-0.215846, 0.863215], o que nos leva a uma função $h(x) = -0.215846 + 0.863215x$ e aos seguintes gráficos:

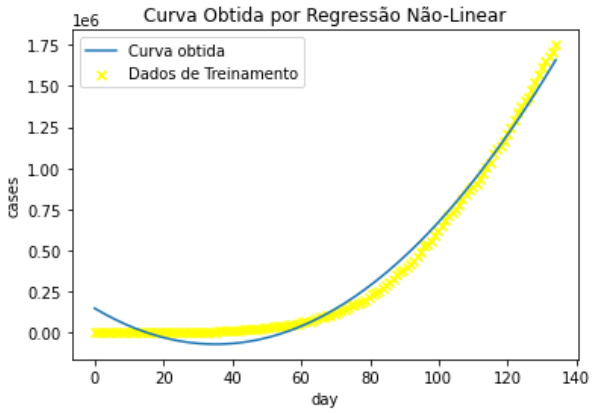


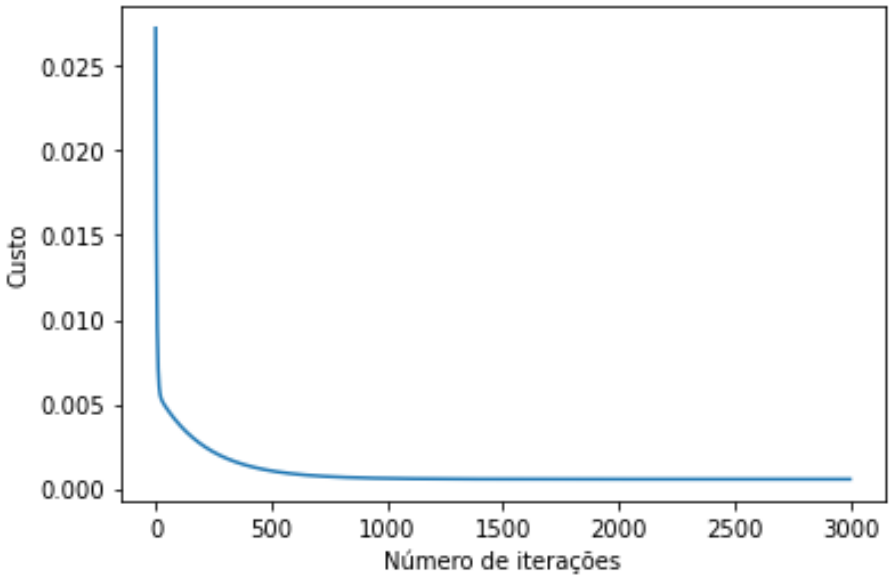




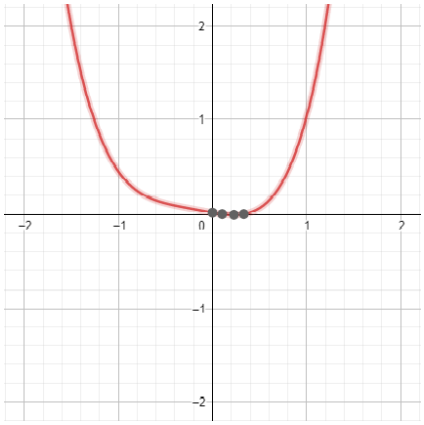
Para o polinômio de segundo grau, com um teste de 3000 iterações e taxa de aprendizado de 0.88, obtivemos um $Theta$=[0.0842579, -0.978417, 1.84167], o que nos leva a uma função $h(x) = 0.0842579 - 0.978417x + 1.84167x^2$ e aos seguintes gráficos:

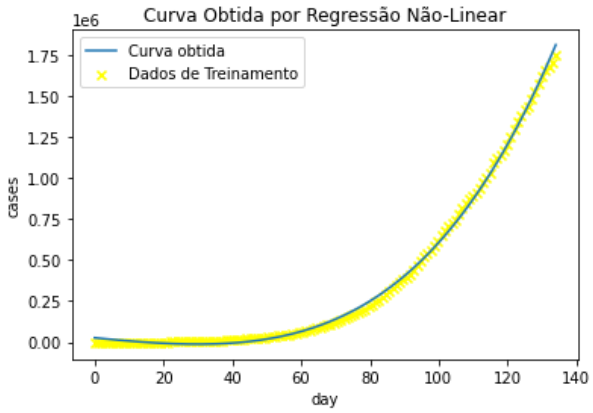


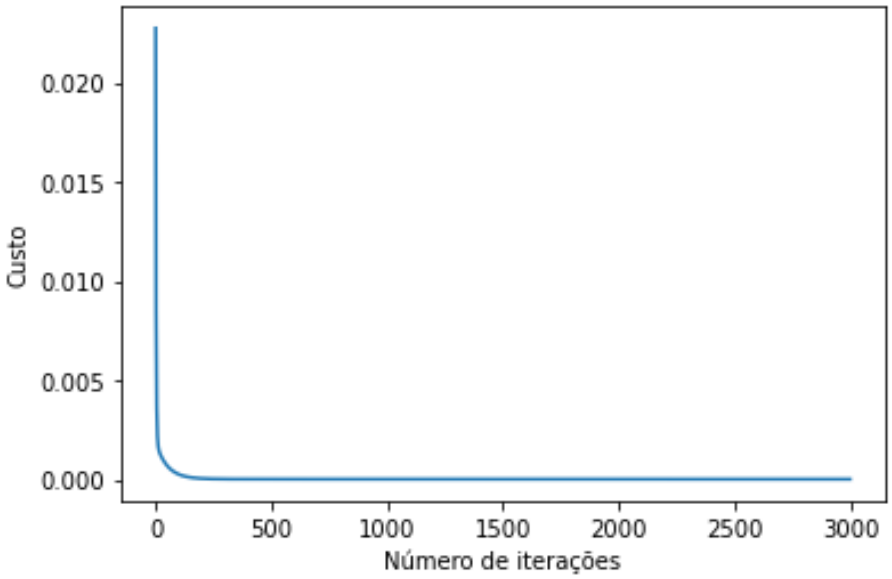




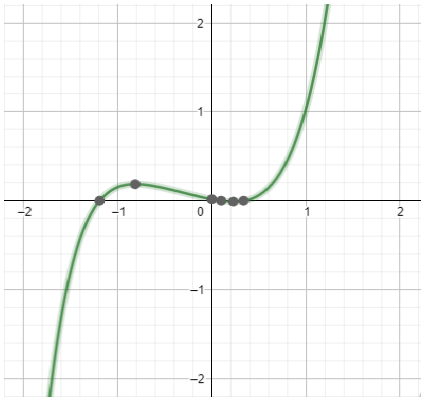
Para o polinômio de quarto grau, com um teste de 3000 iterações e taxa de aprendizado de 0.88, obtivemos um $Theta$=[0.0149335, -0.165108, 0.144472, 0.457395, 0.582225], o que nos leva a uma função $h(x) = 0.0149335 - 0.165108x + 0.144472x^2 + 0.457395x^3 + 0.582225x^4$ e aos seguintes gráficos:

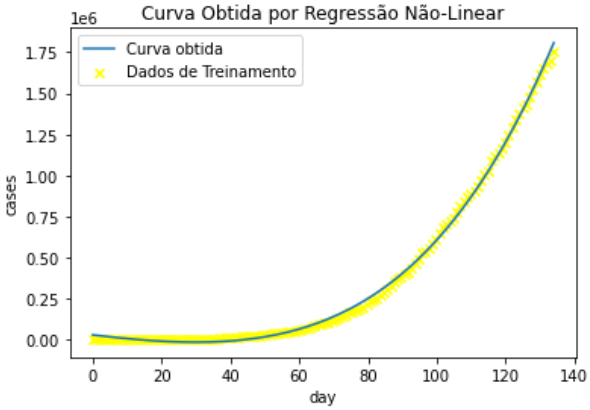


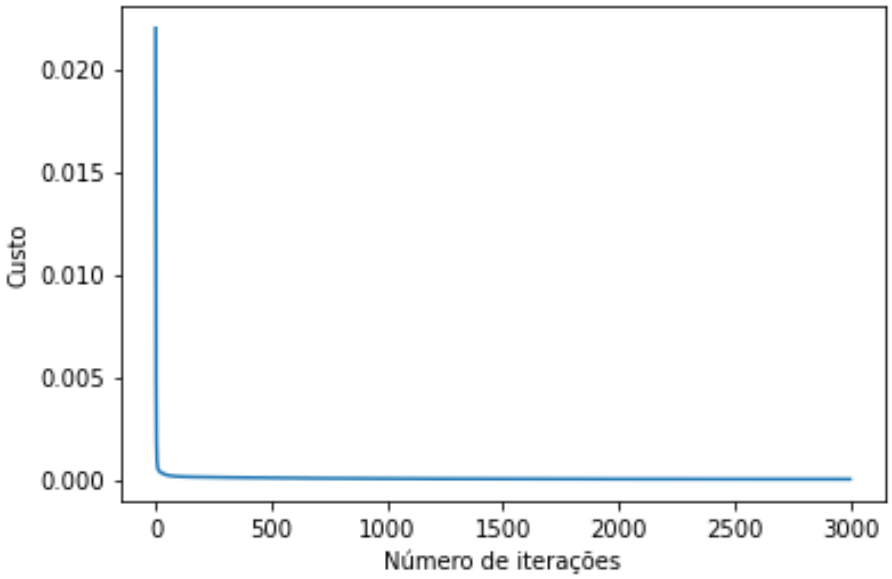




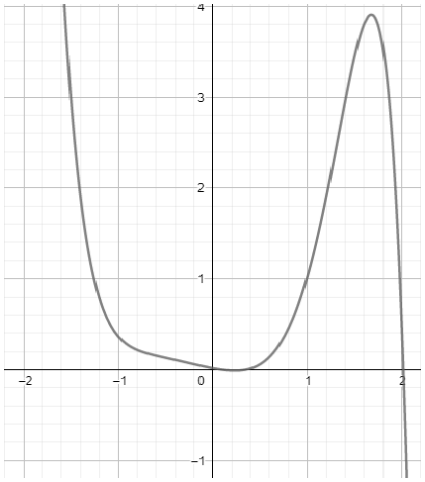
Para o polinômio de sexto grau, com um teste de 3000 iterações e taxa de aprendizado de 0.88, obtivemos um $Theta$=[0.0170822, -0.199887, 0.240266, 0.444346, 0.391418, 0.196015, -0.0563695], o que nos leva a uma função $h(x) = 0.0170822 - 0.199887x + 0.240266x^2 + 0.444346x^3 + 0.391418x^4 + 0.196015x^5 - 0.0563695x^6$ e aos seguintes gráficos:

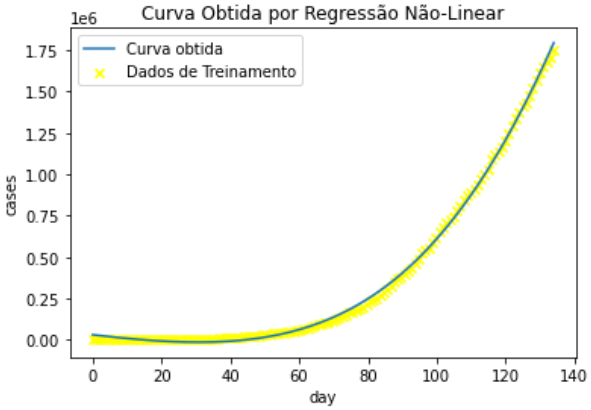


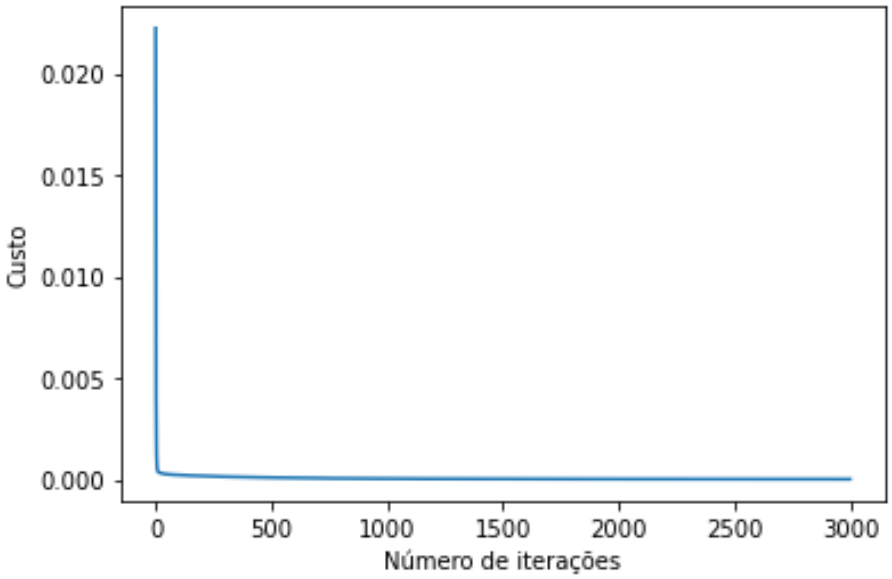




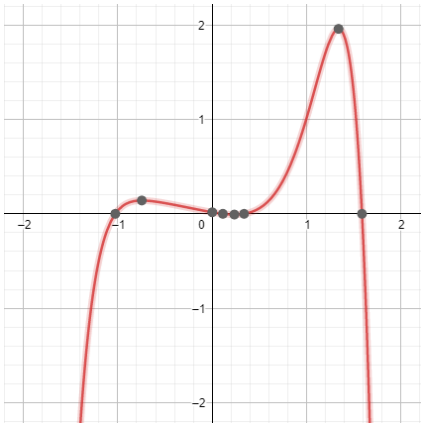
Para o polinômio de sétimo grau, com um teste de 3000 iterações e taxa de aprendizado de 0.88, obtivemos um $Theta$=[0.0177606, -0.192436, 0.206038, 0.428725, 0.416739, 0.265743, 0.0541009, -0.171953], o que nos leva a uma função $h(x) = 0.0177606 - 0.192436x + 0.206038x^2 + 0.428725x^3 + 0.416739x^4 + 0.265743x^5 - 0.0541009x^6 - 0.171953x^7$ e aos seguintes gráficos:

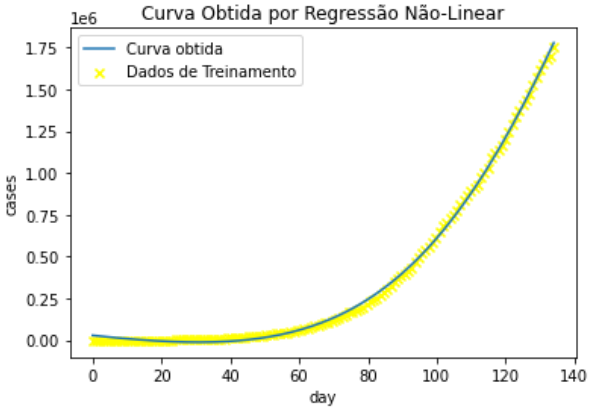


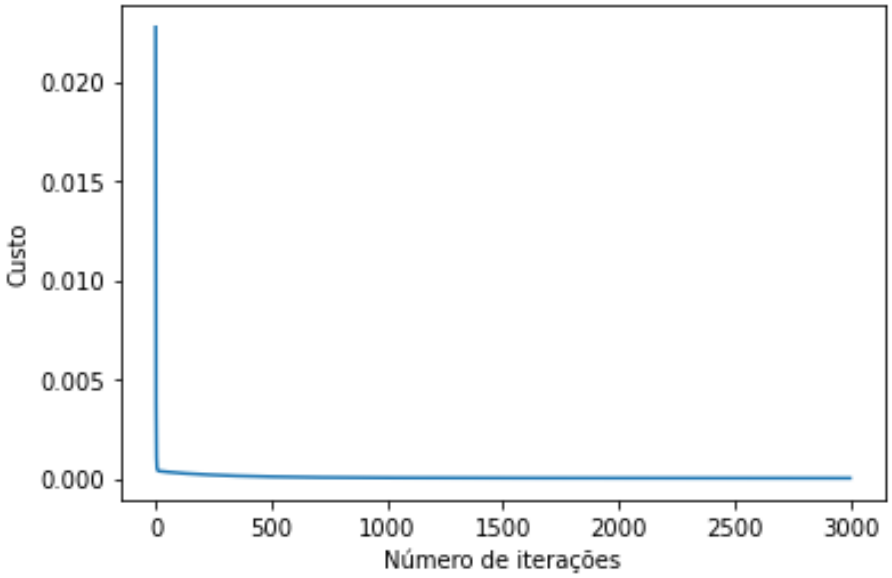




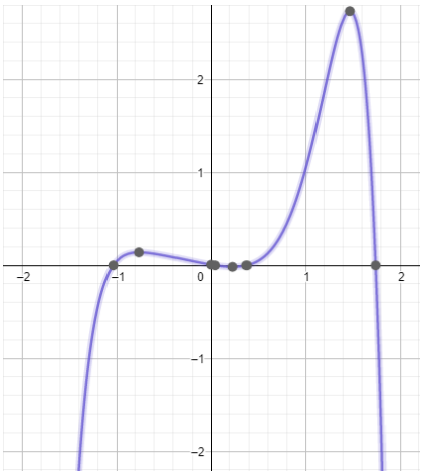
Para o polinômio de oitavo grau, com um teste de 3000 iterações e taxa de aprendizado de 0.88, obtivemos um $Theta$=[0.0164742, -0.171157, 0.169644, 0.393456, 0.414696, 0.308558, 0.143137, -0.0398639, -0.21831], o que nos leva a uma função $h(x) = 0.0164742 - 0.171157x + 0.169644x^2 + 0.393456x^3 + 0.414696x^4 + 0.308558x^5 + 0.143137x^6 - 0.0398639x7 - 0.21831x^8$ e aos seguintes gráficos:

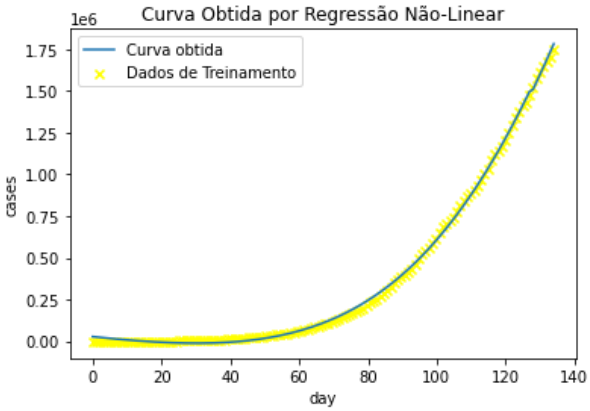


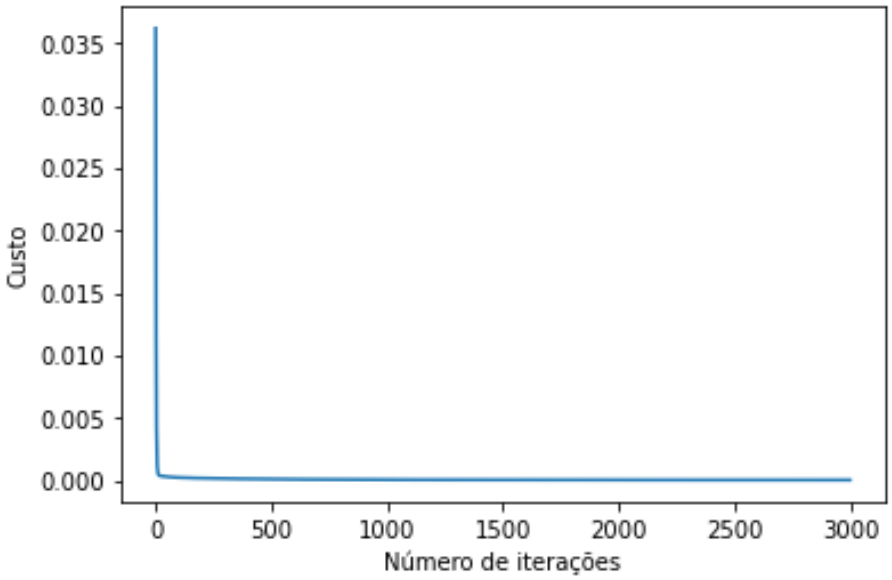




Para o polinômio de nono grau, com um teste de 3000 iterações e taxa de aprendizado de 0.88, obtivemos um $Theta$=[0.00696653, -0.170295, 0.18479, 0.387369, 0.393199, 0.286401, 0.133302, -0.028302, -0.179852, 0.0168987], o que nos leva a uma função $h(x) = 0.00696653 - 0.170295x + 0.18479x^2 + 0.387369x^3 + 0.393199x^4 + 0.286401x^5 + 0.133302x^6 - 0.028302x7 - 0.179852x^8 + 0.0168987x^9$ e aos seguintes gráficos:





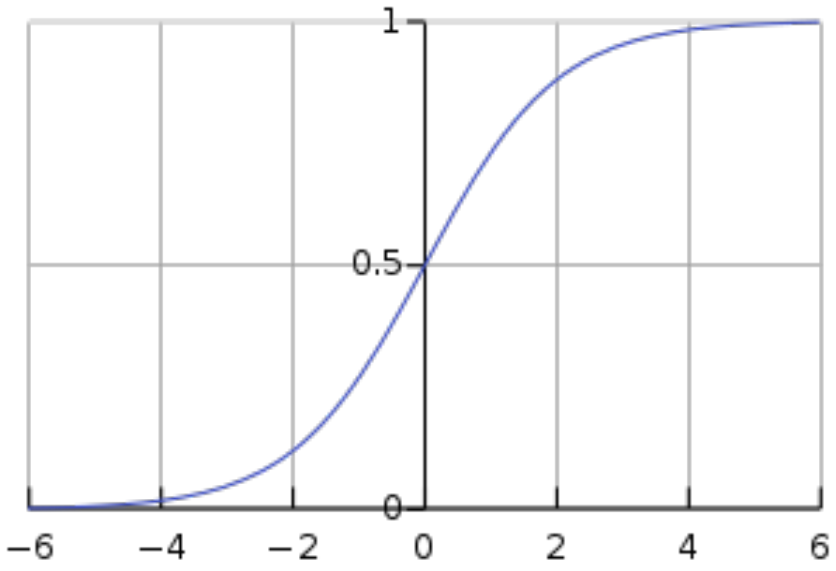


7 Introdução da Segunda Parte:

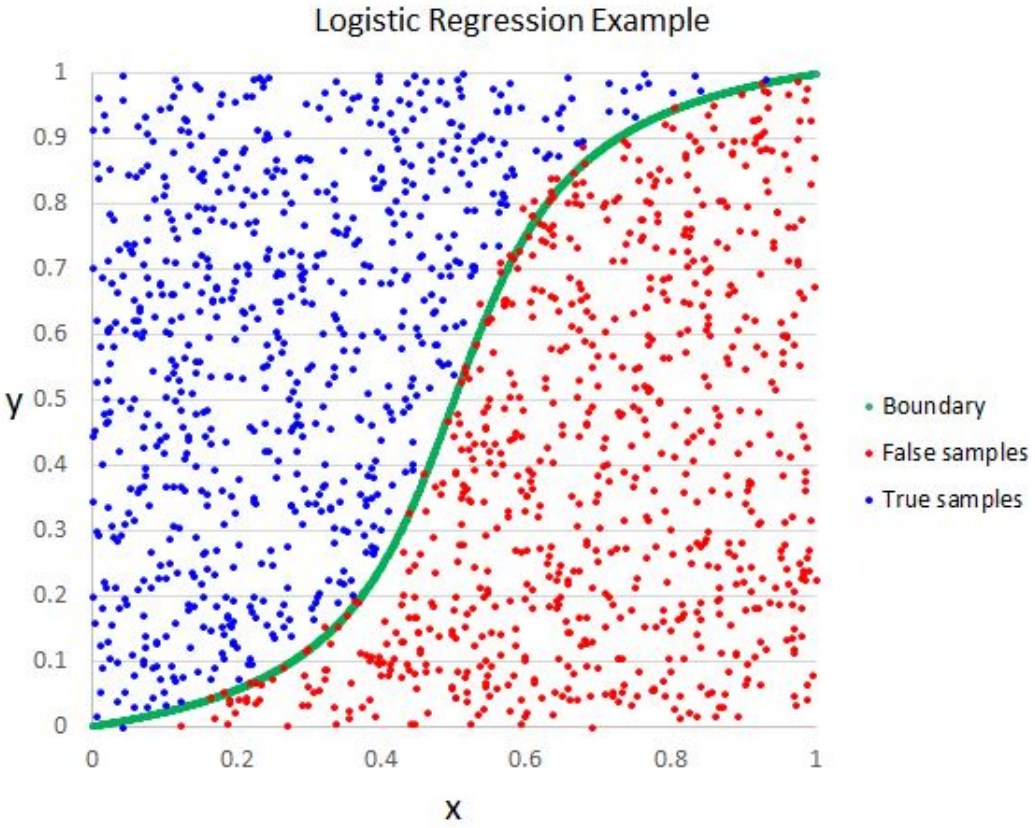
Tal projeto consiste na implementação de um algoritmo baseado em regressão logística para a identificação de dígitos manuscritos, sendo que, as imagens que contém tais dígitos foram linearizadas para vetores de comprimento 400. Os dados que utilizamos para tal implementação consistem em uma matriz onde cada linha representa uma das imagens da base MNIST já transformada em vetor, e em um vetor onde o dígito de cada linha é o valor real representado em cada imagem.

8 Regressão Logística:

Como forma de simplificar a compreensão, podemos visualizar a regressão logística como uma "versão" da regressão linear/não-linear voltada para problemas de classificação, visto que, a mesma nos oferece um modelo de predição de valores através de um grupo de dados de treino. Suas aplicações se extendem por inúmeras áreas e contextos, entre eles podemos citar o campo da medicina (Identificação de tipos de câncer e entre outros), financeiro e bancário (Fraudes econômicas e entre outros) e também no campo de reconhecimento de dígitos, o qual será abordado no modelo que desenvolvemos.
A implementação da regressão logística consiste em definirmos nossa função buscada $h_\theta(x)$ como $h_\theta(x) = g(\theta^T X)$, sendo a função $g(x)$ definida como $g(z) = \frac{1}{1+e^{-z}}$, ou seja, $h_\theta(x) = \frac{1}{1+e^{-\theta^T X}}$, que ao plotarmos, obtemos um gráfico da forma:



Com isso, é observável que os valores da função se limitam a $0 \leq h_\theta(x) \leq 1$, e isso ocorre pelo fato da função oferecer uma probabilidade para dada predição, ou seja, se definirmos 1 como "ocorre" e 0 como "não ocorre", poderíamos ter algo como $h_\theta(x) = P(y = 1|x; \theta)$, que traduzido para a linguagem comum, seria algo como "A probabilidade de y ocorrer, dado x, parametrizado por θ". Nosso método para a busca da função $h_\theta(x)$ ótima é o já citado gradiente descendente, suas bases de atuação são as mesmas da regressão linear/não linear, a maior mudança se dá na função de custo, que passa a ser $J(\theta) = \frac{1}{m}\sum_{i=1}^{m}[y^{(i)}log(h_\theta(x^{(i)})) + (1 - y^{(i)})log(1 - h_\theta(x^{(i)}))]$. Para o caso de classificação não binária, executamos um método conhecido como "Um contra todos", onde possuímos k classificadores binários, sendo k o número de variáveis que se quer classificar, e o modelo é treinado para que possa qualificar e predizer da melhor forma possível cada item que se quer classificar (O problema dos dígitos manuscritos é um bom exemplo de implementação multi-classes).
Segue-se um exemplo de formato de resposta para uma implementação de regressão logística:



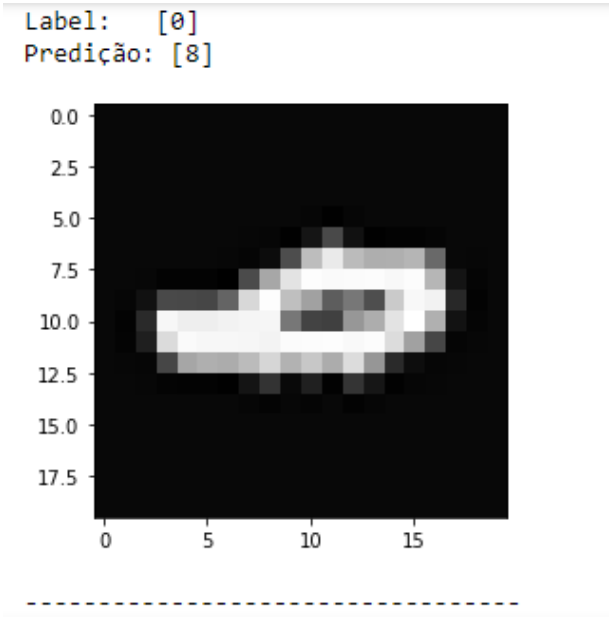
9 Primeira Questão da Segunda Parte:

9.1 Abordagem:

Para este problema, desenvolvemos um modelo de regressão logística multi-classes não regularizada com vistas de obter um bom classificador para dígitos manuscritos, para tal, realizamos dois tipos de testes, um utilizando todo o conjunto de dados para o treinamento, e outro dividindo tal conjunto de dados de forma aleatória em uma parcela para treinamento e outra para testes, na proporção de $\frac{1}{5}$ para teste e $\frac{4}{5}$ para treino.

9.2 Análise do Resultado Obtido Pelo Modelo:

Com o modelo desenvolvido, atingimos uma taxa de 95.32% de acertos para o caso onde utilizamos todo o conjunto de dados para treino e teste, e uma taxa de 86.30% nos dados de teste com o conjunto de dados sendo dividido, em ambos os casos utilizamos 3000 iterações e uma taxa de aprendizado de valor 3. Os resultados obtidos eram esperados, ou seja, uma maior taxa de acertos utilizando todo o conjunto para treino e teste, porém, para o caso onde ocorre a divisão do conjunto de dados, isso pode significar um menor overfitting dos dados de treino, o que permite uma maior generalização do modelo, algo que buscamos. Ao final, conseguimos também comparar os dígitos erroneamente identificados pelo modelo com seus valores reais.
Segue-se um exemplo de exibição de um dígito (No caso, 0) erroneamente identificado pelo modelo como 8:



9.3 O Algoritmo em Python-3 Para o Caso de Regressão Logística:

Para a implementação de um modelo de regressão logística, desenvolvemos o seguinte código:

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

def Load():
    X = data_2.values
    m = X.shape[0]
    n = X.shape[1]
    nbr_classes = 10
    X = np.append(np.ones([m,1]),X,axis=1)
    y = data_3.replace(10,0).values
    aux = []
    for i in range(m):
        aux.append(np.array([1 if y[i] == j else 0 for j in range(nbr_classes)]))
    Y = np.array(aux).reshape(-1, nbr_classes)
    return X, Y, m, n, nbr_classes

def sigmoid(z):
    return 1/(1+np.exp(-z))

def costFunctionLogReg(X, Y, Theta):
    m = len(Y)
    h = sigmoid(X.dot(Theta))
    J = (-1/m)*np.diag(Y.T @ np.log(h) + (1-Y).T @ np.log(1-h))
    return J

def gradientDescentLogReg(X, Y, Theta, alpha, nbr_iter):
    J_history = []
    m = len(Y)
    for i in range(nbr_iter):
        h = sigmoid(X.dot(Theta))
        Theta = Theta - (alpha/m)*(X.T.dot(h-Y))
        J_history.append(costFunctionLogReg(X, Y, Theta))
    return Theta,J_history

def predict(X, theta):
    m = X.shape[0]
    aux = sigmoid(X.dot(theta))
    pred = []
    for row in aux:
        pred.append(np.argmax(row))

    pred = np.array(pred).reshape((m, 1))
    return pred

def evaluatePrediction(Y, pred):
    m = len(Y)
    Y_labels = []
    for row in Y:
        Y_labels.append(np.argmax(row))

    Y_labels = np.array(Y_labels).reshape((m, 1))
    ratio = (Y_labels == pred).sum()/m
    return ratio

def missFeedback(X, Y, pred):
    m = len(Y)
    y_labels = []
    for row in Y:
        y_labels.append(np.argmax(row))
    y_labels = np.array(y_labels).reshape((m, 1))
    for i in range(m):
        if not (y_labels == pred)[i]:
            print(f'Label: \t {y_labels[i]}')
            print(f'Predição: {pred[i]}')
            pixels = X[i, 1:].reshape((20, 20))
            plt.imshow(pixels, cmap='gray')
            plt.show()
            print('---------------------------------')


data_2 = pd.read_csv(r'C:\Users\USUARIO\Desktop\projeto python\imageMNIST.csv', sep=',', decimal=",")
data_3 = pd.read_csv(r'C:\Users\USUARIO\Desktop\projeto python\labelMNIST.csv')
X,Y,m,n,nbr_classes=Load()
Theta = np.zeros([n+1,nbr_classes])
nbr_iter = 3000
alpha = 3
new_Theta, J_history = gradientDescentLogReg(X, Y, Theta, alpha, nbr_iter)
pred=predict(X,new_Theta)
percentage=evaluatePrediction(Y,pred)
print(f'Taxa de acerto = {100*percentage:.2f}%')
a=int(input("Caso queira ver os números que o modelo identificou erroneamente, tecle 0: "))
if a==0:
    missFeedback(X, Y, pred)
```

Primeiramente, temos uma função "*Load*", tal função é responsável por estruturar X e Y de forma que tais matrizes possam ser utilizadas no modelo, em seguida, temos uma função "*sigmoid*", responsável for fazer automaticamente o cálculo $\frac{1}{1+e^{-z}}$.
Semelhante ao que temos para o caso de regressão linear/não-linear, seguimos com as funções "$costFunctionLogReg$" e "$gradientDescentLogReg$", a diferença se encontra na primeira, onde a função custo J está adequada para o caso de regressão logística.
Por fim, temos três funções que nos permitem estimar a taxa de eficácia do modelo e avaliar onde se encontram seus erros, são elas a função "*predict*", responsável por realizar a predição dos dígitos manuscritos através dos valores de θ obtidos no treinamento, a função "$evaluatePrediction$", onde se é comparado o resultado obtido na função anterior com os valores reais dos dígitos, permitindo assim o cálculo de uma taxa de eficácia geral do modelo, e ao final, a função "$missFeedback$", responsável pelo print do dígito erroneamente predito pelo modelo e seu valor real.
Para o caso de divisão do conjunto de dados, o que fazemos é aplicar a seguinte função "*div*", que já prepara adequadamente os dados:

```
def div():
    data_joined = data_2.join(data_3)
    data_random = data_joined.sample(frac=1)
    m_train = int(m*0.8)
    m_test = m - m_train
    df_train = data_random.iloc[:m_train,:]
    X_train, y_train = df_train.iloc[:,:-1].values, df_train.iloc[:,-1].values
    X_train = np.append(np.ones([m_train,1]),X_train,axis=1)
    df_test = data_random.iloc[:m_test,:]
    X_test, y_test = df_test.iloc[:,:-1].values, df_test.iloc[:,-1].values
    X_test = np.append(np.ones([m_test,1]),X_test,axis=1)
    aux = []
    for i in range(m_train):
        aux.append(np.array([1 if y_train[i] == j else 0 for j in range(nbr_classes)]))
    Y_train = np.array(aux).reshape(-1, nbr_classes)
    aux = []
    for i in range(m_test):
        aux.append(np.array([1 if y_test[i] == j else 0 for j in range(nbr_classes)]))
    Y_test = np.array(aux).reshape(-1, nbr_classes)
    return X_train, X_test, Y_train, Y_test, m_train, m_test
```

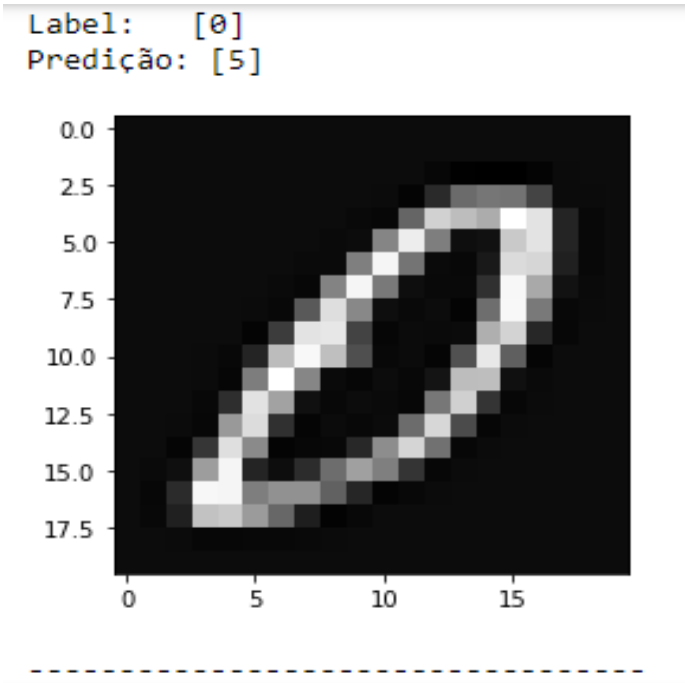
10 Segunda Questão da Segunda Parte:

10.1 Abordagem:

Em um geral, o modelo que usamos é o mesmo da questão tratada anteriormente, porém, adicionando o termo de regularização nas funções de custo e gradiente descendente.

10.2 Análise do Resultado Obtido Pelo Modelo:

Para o caso onde não dividimos o conjunto de dados, obtivemos uma taxa de acertos de 92.96% com um número de iterações igual a 3000, uma taxa de aprendizado 3 e um valor de beta (Nosso λ de regularização) igual a 3. Como podemos observar, é um valor inferior ao obtido com o modelo sem regularização, o que indica que possivelmente um valor de beta menor seja mais adequado. Para o caso onde dividimos o conjunto de dados, obtivemos uma taxa de acertos de 83.40% nos dados de teste, novamente inferior ao modelo não regularizado, e novamente, como esperado, inferior ao caso onde não dividimos o conjunto de dados.
Segue-se um exemplo de exibição de um dígito (No caso, 0) erroneamente identificado pelo modelo como 5 no caso regularizado:



10.3 O Algoritmo em Python-3 Para o Caso de Regressão Logística Regularizada:

Para a implementação de um modelo de regressão logística regularizada, desenvolvemos o seguinte código:

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

def Load():
    X = data_2.values
    m = X.shape[0]
    n = X.shape[1]
    nbr_classes = 10
    X = np.append(np.ones([m,1]),X,axis=1)
    y = data_3.replace(10,0).values
    aux = []
    for i in range(m):
        aux.append(np.array([1 if y[i] == j else 0 for j in range(nbr_classes)]))
    Y = np.array(aux).reshape(-1, nbr_classes)
    return X, Y, m, n, nbr_classes

def sigmoid(z):
    return 1/(1+np.exp(-z))

def costFunctionLogRegReg(X, Y, Theta, beta):
    J = costFunctionLogReg(X, Y, Theta) + np.diag((beta/(2*m))*(Theta[1:].T @ Theta[1:]))
    return J

def costFunctionLogReg(X, Y, Theta):
    m = len(Y)
    h = sigmoid(X.dot(Theta))
    J = (-1/m)*np.diag(Y.T @ np.log(h) + (1-Y).T @ np.log(1-h))
    return J

def gradientDescentLogRegReg(X, Y, Theta, alpha, beta, nbr_iter):
    J_history = []
    m = len(Y)
    Theta_aux = Theta.copy()
    Theta_aux[0] = 0
    for i in range(nbr_iter):
        h = sigmoid(X.dot(Theta))
        Theta_aux = Theta.copy()
        Theta_aux[0] = 0
        Theta = Theta - (alpha/m)*(X.T.dot(h-Y) + beta*Theta_aux)
        J_history.append(costFunctionLogRegReg(X, Y, Theta, beta))
    return Theta, J_history

def predict(X, theta):
    m = X.shape[0]
    aux = sigmoid(X.dot(theta))
    pred = []
    for row in aux:
        pred.append(np.argmax(row))

    pred = np.array(pred).reshape((m, 1))
    return pred

def evaluatePrediction(Y, pred):
    m = len(Y)
    Y_labels = []
    for row in Y:
        Y_labels.append(np.argmax(row))

    Y_labels = np.array(Y_labels).reshape((m, 1))
    ratio = (Y_labels == pred).sum()/m
    return ratio

def missFeedback(X, Y, pred):
    m = len(Y)
    y_labels = []
    for row in Y:
        y_labels.append(np.argmax(row))
    y_labels = np.array(y_labels).reshape((m, 1))
    for i in range(m):
        if not (y_labels == pred)[i]:
            print(f'Label: \t {y_labels[i]}')
            print(f'Predição: {pred[i]}')
            pixels = X[i, 1:].reshape((20, 20))
            plt.imshow(pixels, cmap='gray')
            plt.show()
            print('---------------------------------')


data_2 = pd.read_csv(r'C:\Users\USUARIO\Desktop\projeto python\imageMNIST.csv', sep=',', decimal=",")
data_3 = pd.read_csv(r'C:\Users\USUARIO\Desktop\projeto python\labelMNIST.csv')
X,Y,m,n,nbr_classes=Load()
Theta = np.zeros([n+1,nbr_classes])
nbr_iter = 3000
alpha = 3
beta = 0.01
new_Theta, J_history = gradientDescentLogRegReg(X, Y, Theta, alpha, beta, nbr_iter)
pred=predict(X,new_Theta)
percentage=evaluatePrediction(Y,pred)
print(f'Taxa de acerto = {100*percentage:.2f}%')
a=int(input("Caso queira ver os números que o modelo identificou erroneamente, tecle 0: "))
if a==0:
    missFeedback(X, Y, pred)
```

Como podemos visualizar, as funções em um geral são as mesmas, com exceção da adição de uma denominada "$costFunctionLogRegReg$", responsável pelo cálculo da parte regularizada da função custo, e alterações na função do gradiente descendente, permitindo a adequação da mesma para a inclusão do termo de regularização

11 Bibliografia

[1] M. A. Gomes Ruggiero, V. L. da Rocha Lopes. Cálculo Numérico - Aspectos Teóricos e Computacionais, 2ª edição, Editora Pearson, 1997.

[2] https://jornal.usp.br/universidade/equacao-de-vida-como-a-matematica-modela-a-pandemia/

[3] http://repositoriocovid19.unb.br/repositorio-projetos/modelagem-matematica-da-propagacao-do-covid-19/

[4] https://towardsdatascience.com/gradient-descent-in-python-a0d07285742f

[5] https://www.baeldung.com/cs/gradient-descent-logistic-regression

[6] https://pt.coursera.org/learn/machine-learning

[7] Slides e Jupyter Notebooks disponibilizados pelo professor da disciplina (MS960), João Batista Florindo.