Unicamp

Disciplina de MS960: Aprendizado de Máquinas

Relatório: Projeto 1

Jorge Henrique de Andrade Pacheco Reis (237966)
Lucas Matheus dos Santos Nascimento (220838)

Os códigos usado nesse projeto estão num repositório do GitHub:
`https://github.com/jhapreis/MS960_MachineLearning`.

Sumário

1 Exercício 1

A Regressão Linear é uma proposta de análise preditiva que se pode fazer para um determinado conjunto de dados que supõe que é possível obter uma função linear que consiga prever um resultado esperado a partir do conjunto de dados inicial. Os dados analisados o número de pessoas contaminadas referente aos primeiros 134 dias da pandemia da Covid-19 no Brasil. Na Figura 1 é possível observar o comportamento do número de casos acumulados ao longo dos primeiros 134 dias iniciais da pandemia no Brasil.

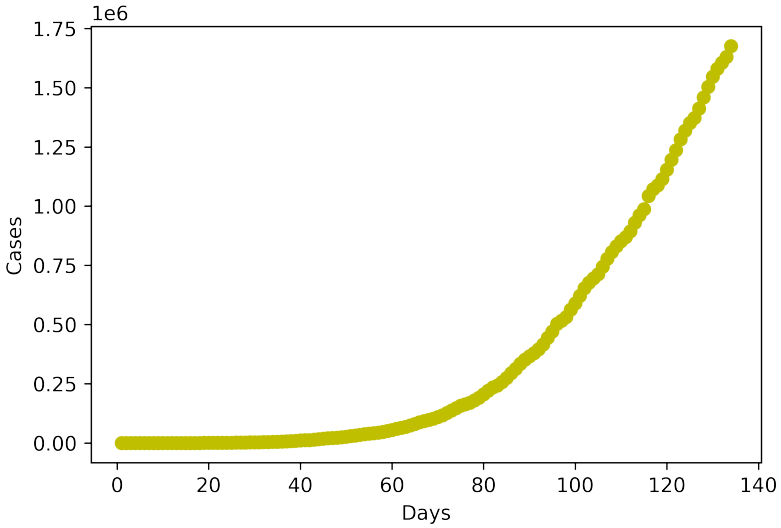

Figura 1: Casos acumulados nos primeiros 134 dias da pandemia de Covid-19 no Brasil.

Neste exercício, iremos tentar encontrar modelos que se sejam capazes capazes de prever a quantidade de casos de Covid-19 no Brasil quando solicitado o número de dias a partir do primeiro caso. Iremos abordar duas metodologias de ajuste, inicialmente, através de um ajuste polinomial e após isso, um ajuste exponencial. Ambos serão detalhados em suas respectivas secções e em ambos é utilizado o método de otimização Gradiente Descendente (GD), em que iterativamente, busca-se um mínimo global da função de custo (J) utilizada.

1.1 Ajuste Exponencial

Inicialmente, ao olharmos a distribuição de pontos, a primeira impressão foi uma curva exponencial. Dessa forma, ao aproximarmos a função *log* y por log $h_\theta(x)$, é possível obter θ_0 e θ_1 utilizando gradiente descendente.

Traremos alguns códigos da implementação das principais funções do trabalho, no entanto, algumas das funções "chamam"outras funções dentro delas. Assim sendo, a primeira função a ser comentada é a função Gradiente Descendente em que já no início, é verificada se os dados estão normalizadas e, caso não estejam, a função de normalização é chamada. Além disso, cabe comentar que os novos valor de θ_j são implementados simultaneamente. Por último, os valores para o máximo de tentativas $(max - tries)$ é igual a 1.10^5, o valor do resíduo mínimo $(min - residual)$ é igual a 1.1^{-4} e o coeficiente de aprendizado $(learning - rate)$ utilizado iniciou em 10 e terminou em 1.10^3. A função Gradiente descendente retorna os valores dos coeficientes para a reta que melhor ajusta os dados na escala logarítmica e os resíduos encontrados com a aproximação realizada.

```
def Gradient_Descendent(x_data_init, y_data_init, initial_guess, learning_rate, min_residual
    =1E-3, max_tries=100, normalize=1):

    if normalize == 1:
        x_data, y_data = normalization(x_data_init, y_data_init)
    else:
        x_data, y_data = x_data_init, y_data_init

    theta_0  = initial_guess[0]
    theta_1  = initial_guess[1]

    residual = residual_function(x_data, y_data, [theta_0, theta_1])
```

```
    tries    = 0

    residuals_values = [residual]

    while (tries <= max_tries) and (residual > min_residual):

        gradient = gradient_residual_function(x_data, y_data, [theta_0, theta_1])

        theta_0  -= learning_rate*gradient[0]
        theta_1  -= learning_rate*gradient[1]

        residual = residual_function(x_data, y_data, [theta_0, theta_1])
        tries    += 1
        residuals_values.append(residual)

    if residual <= min_residual:
        print(f'\n   Success! The residual is under the minimal value.\nAfter {tries} steps
    .\n\n')
    elif tries > max_tries:
        print(f'\n   Failed! The number of tries was exceded.\nThe residual is {residual}\n\
    n')

    if normalize == 1:
        coefficients = undo_normalization_coefficients(x_data_init, y_data_init, [theta_0,
    theta_1])

    coefficients     = pd.DataFrame(coefficients, columns=['values'], index=['theta_0', '
    theta_1'])
    residuals_values = pd.DataFrame(residuals_values, columns=['residuals'])

    return( coefficients, residuals_values  )
```

Listing 1: Gradiente Descendente

Achados os melhores coeficientes que melhores ajustam os dados na escala logarítmica, construímos agora a função Gradiente Descendente Exponencial, em que os coeficientes que tentam modelar o conjunto de dados é dado pela equação 1. Além disso, calculamos novamente os resíduos, que agora representam os resíduos da modelagem exponencial dada pela equação 1.

$$f(x) = \theta_1 \exp(\theta_0 x) \tag{1}$$

```
def Gradient_Descendent_Exponential(x_data_init, y_data_init, initial_guess, learning_rate,
    min_residual=1E-3, max_tries=100, normalize=1):

    time_start = time()

    y_data_log = np.log( y_data_init )

    coefficients, residuals = Gradient_Descendent(
        x_data_init=x_data_init,
        y_data_init=y_data_log,
        initial_guess=initial_guess,
        learning_rate=learning_rate,
        min_residual=min_residual,
        max_tries=max_tries,
        normalize=normalize
    )

    theta_1_exp = np.e**(coefficients['values']['theta_1'])
```

Listing 2: Gradiente Descendente Exponencial

Os resultados encontrados estão dispostos na Figura 2, em que é possível observar a distribuição de casos ao longo dos dias, na curva em amarelo, mostrado anteriormente. O segundo gráfico, mostra o número de casos em escala logarítmica em função do número de dias. O terceiro gráfico mostra o decréscimo do valor do resíduo encontrado e o último gráfico mostra o ajuste exponencial de acordo com o modelo da equação 1.

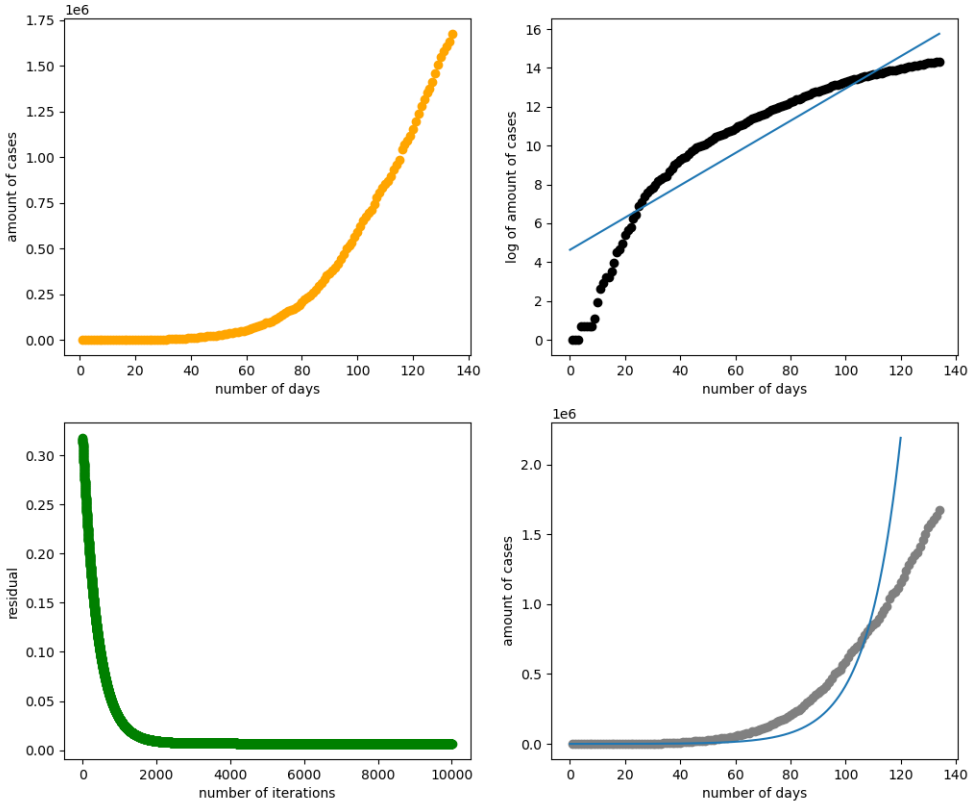

Figura 2: Ajuste Exponencial da curva de dados. Os valores encontrados para $\theta_0 = 0.083$ e $\theta_1 = 4.637$.

Por fim, cabe mostrar como foi a evolução dos resíduos a medida que a taxa de aprendizado ia diminuindo, Figura 3. Cabe observar que a partir da taxa de aprendizado menor ou igual a 1, temos que a escala do resíduo muda drasticamente para valores menores que 0.5.

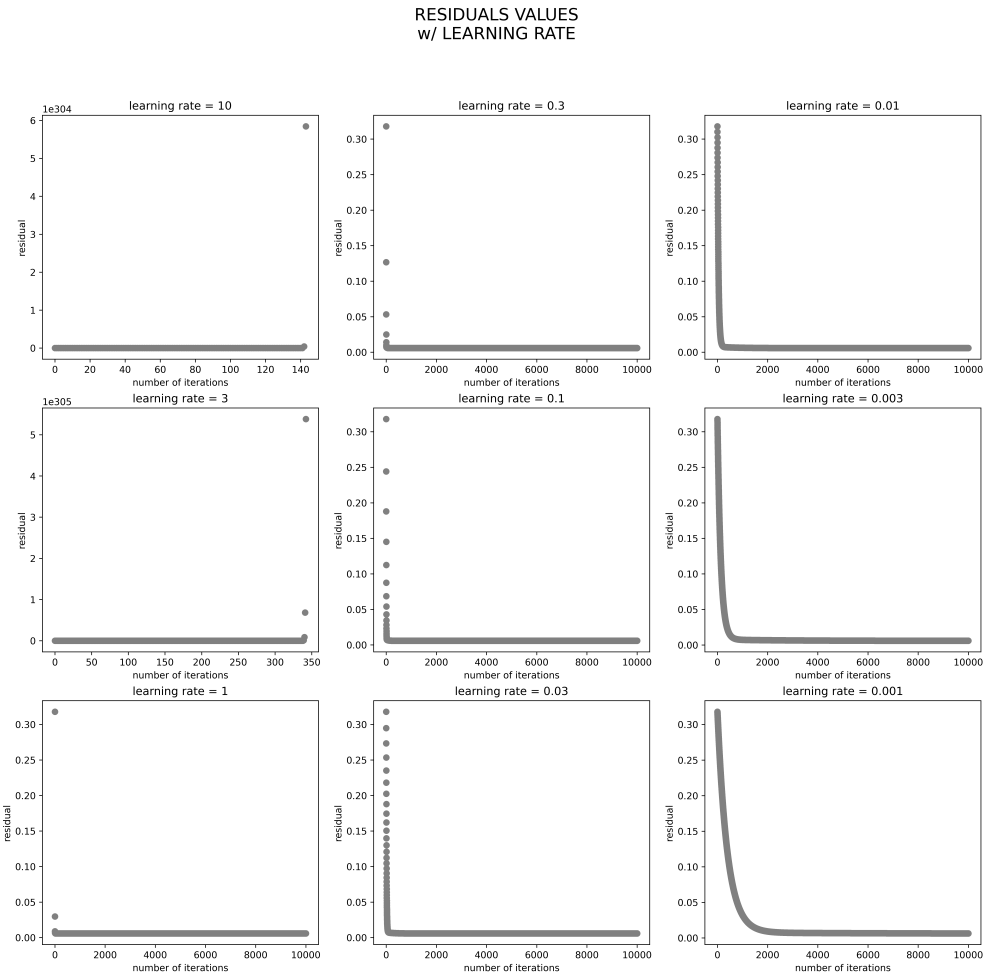

Figura 3: Resíduos obtidos para diferentes taxas de aprendizagem.

1.2 Ajuste Polinomial

O objetivo do ajuste polinomial é a implementação de um código que ajuste um polinômio de grau n à curva do número de casos em função da quantidade de dias utilizando o gradiente descendente. Sabemos que neste caso, trata-se de um ajuste não linear em que a nossa função de hipótese $h_\theta(x_j)$ é dada pela equação 2.

$$h_\theta(x) = \theta_0 + \theta_1 x^1 + \theta_2 x^2 + \theta_3 x^3 + ... + \theta_n x^n = \sum_{n=0}^{k} \theta_n x^n \tag{2}$$

Sabemos que é possível transformar a função de hipótese, descrita na equação 2, em uma função de hipótese que tenha linearidade entre os atributos x e a saída y. A equação 3 descreve a nossa nova função de hipótese do problema.

$$h_\theta(x_j) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + ... + \theta_n x_n \tag{3}$$

onde $x_0 = 1$ e para para todo n, temos $x_n = x^n$.

Escrevendo o problema como representado na equação 3, é possível resolvê-lo utilizando o gradiente descendente multivariado. Já apresentamos o Gradiente Descendente (Univariado), a única diferença para este é que este pode ser aplicado para polinômios de graus maiores, em resumo.

```python
def Multiple_Gradient_Descendent(x_data_init, y_data_init, initial_guess, learning_rate, min_residual=1E-3, max_tries=100, normalize=1):

    if normalize == 1:
        x_data, y_data = multiple_normalization(x_data_init, y_data_init)
    else:
        x_data, y_data = x_data_init, y_data_init


    thetas   = initial_guess
    residual = multiple_residual_function(x_data=x_data, y_data=y_data, coefficients=thetas)
    tries    = 0

    residuals_values = [residual]

    while (tries <= max_tries) and (residual > min_residual):

        gradient = multiple_gradient_residual_function(x_data, y_data, thetas)

        thetas = thetas - learning_rate*gradient[0]

        residual = multiple_residual_function(x_data=x_data, y_data=y_data, coefficients=thetas)
        tries    += 1
        residuals_values.append(residual)

    if residual <= min_residual:
        print(f'\n   Success! The residual is under the minimal value.\nAfter {tries} steps.\n\n')
    elif tries > max_tries:
        print(f'\n   Failed! The number of tries was exceeded.\nThe residual is {residual:e}.\n\n')

    if normalize == 1:
        coefficients = undo_multiple_normalization(x_data_init, y_data_init, thetas)
    else:
        coefficients = thetas

    coefficients     = pd.DataFrame(coefficients, columns=['values'], index=[('theta_'+str(i)) for i in range(coefficients.shape[0])])
    residuals_values = pd.DataFrame(residuals_values, columns=['residuals'])

    return( coefficients, residuals_values  )
```

Listing 3: Python example

Os resultados encontrados mostram que a medida que as curvas polinomiais com grau menor ou igual a 8 se ajustam muito bem aos dados disponíveis. No entanto, polinômios de graus maiores não se ajustam adequadamente aos dados. Temos que a medida que o grau do polinômio cresce, o valor dos coeficientes de maior grau chegam a valores muito pequenos, insignificantes em sua contribuição para o polinômio. Além disso, cabe ressaltar que a medida que os polinômios aumentavam, maiores eram os problemas de operações numéricas com *overflow/underflow*.

Na Figura 4, é possível analisar o ajuste polinomial para valores de n= 3, 5, 10 e observar o a qualidade do ajuste para os polinômios de graus 3 e 5, sendo este último o que possui o melhor ajuste dentre os três. O último, polinômio de grau 10 não é capaz de modelar os dados, sendo visualmente inadequado.

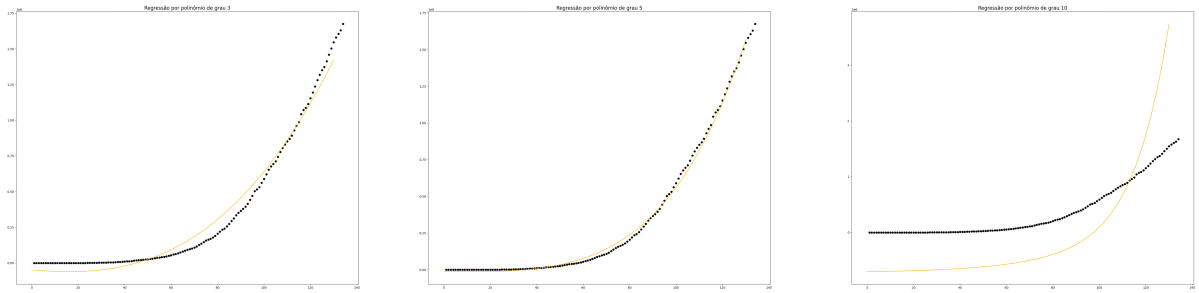

Figura 4: Ajuste Polinomial dos dados.

Por fim, temos os valores para os coeficientes de cada grau mostrado na Figura 4 mostrados na tabela abaixo.

	Grau 3	**Grau 5**	**Grau 10**
θ_0	-47908.73	-10012.723	-702617.00
θ_1	-1468.60	-452.84	573.18
θ_2	34.07	11.81	13.46
θ_3	0.49	0.17	0.124
θ_4		0.00172	0.00099
θ_5		1.515e-05	7.66e-06
θ_6			5.89e-08
θ_7			4.56e-10
θ_8			3.54e-12
θ_9			-3.66e-14
θ_{10}			2.05e-15

2 Exercício 2

No exercício 2, o objetivo é classificar um conjunto de imagens vetorizadas MNIST, com tamanho *20x20*, de acordo com a regressão logística. O algoritmo desenvolvido associa a cada imagem vetorizada uma probabilidade de pertencer a uma classificação. Ele classifica e atribui uma probabilidade diferente para cada classificação, sendo que a classificação final é a que possui a maior probabilidade.

2.1 Selecão de dados

Os dados originais foram repartidos em dois conjuntos: uma amostra aleatória com tamanho específico foi selecionada e reservada como amostra de teste, sendo que o restante foi reservado para ser utilizado como conjunto de aperfeiçoamento. Para os fins do projeto, a amostra foi de 25% do tamanho total[1]. Cada novo conjunto de dados foi salvo em um arquivo *.csv* para posterior análise.

Ao final da regressão com os dados de teste, os coeficientes obtidos (por meio da regressão logística com o gradiente descendente) serão aplicados de volta nos dados de aperfeiçoamento, de onde sairá a taxa de acerto do algoritmo. Ela será calculada ao se comparar as categorias previstas pelo algoritmo com os valores exatos já previamente dados.

2.2 Regressão logística e categorização dos dados

O algoritmo de regressão logística foi montado, como já mencionado anteriormente, fazendo uso do gradiente descendente. O objetivo final desse bloco é devolver uma coleção de coeficientes que maximizem a verossimilhança (isto é, que minimize o "erro" nos dados).

Definindo a função logística como

$$h_\theta(X) = \frac{1}{1 + e^{-\theta^T X}} \tag{4}$$

onde X é a matriz com os dados das imagens vetorizadas e θ é a matriz com os dados dos coeficientes, o valor $h_\theta(X)$ representa o valor da probabilidade associada. No caso, X tem a dimensão de *pixels vs imagens* (ex.: 400 x 1250), enquanto θ tem a dimensão de *pixels vs categorias* (ex.: 400 x 10). O produto entre essas matrizes é o produto matricial usual, sendo que o resultado das demais operações é aplicado elemento-a-elemento (*element-wise*). Assim, a matriz retornada pela função é uma matriz de dimensão *categorias vs imagens* (ex.: 10 x 1250).

Com isso, pode-se montar o cálculo do "resíduo" (no caso, da função de custo associada à verossimilhança da *sigmóide*). Esse resíduo foi calculado como *log-odds*, isto é, o *log* da verossimilhança. Esse resultado é como segue:

$$J(\theta) = \sum_{i=1}^{i=n^o\ de\ imagens} y_i \cdot ln(h_{\theta,i}(X)) + (1 - y_i) \cdot (1 - ln(h_{\theta,i}(X))) \tag{5}$$

Tal valor corresponde, matricialmente falando, a realizar as operações de

$$[r_{ij}] = Y \cdot ln(h_\theta(X)) + (1 - Y) \cdot (1 - ln(h_\theta(X)))$$

termo-a-termo, resultando numa matriz com a mesma dimensão de $h_\theta(X)$ (que também é a mesma dimensão de Y[2]. Após isso, somar todos os elementos das colunas, já que os resíduos, para uma mesma categoria, se adicionam, resultando numa matriz de dimensão *categorias vs 1*. Ou seja,

$$J(\theta) = \sum_{i=1}^{i=n^o\ de\ imagens} [r_{ij}]$$

O cálculo do gradiente associado foi feito tomando o produto matricial[3]

$$\Delta\theta = \frac{1}{n} \cdot X \times (h_\theta(X) - Y)^T$$

[1] Vale notar que a seleção, embora aleatória, se mostrou quase que uniformemente distribuída, de maneira tal que o total de elementos em cada classe era aproximadamente igual.

[2] Importante: a matriz Y é uma espécie de "matriz de classificação", com dimensões de $h_\theta(X)$, *categorias vs imagens*, preenchida com 0 ou 1, conforme a correspondência de "dada uma classificação, essa imagem pertence originalmente a ela?".

[3] A ordem do produto matricial e dos termos pode variar dependendo de qual a forma original dos dados, mas a ideia é a mesma.

onde n é o número de imagens, resultando em uma matriz com a mesma dimensão da matriz θ – *pixels vs categorias* (ex.: 400 x 10).

Depois disso, segue a repetição do algoritmo até ele possuir um resíduo menor que o estabelecido ou até passar do número de tentativas:

$$\theta := \theta - \alpha \cdot \Delta\theta \tag{6}$$

onde α é o valor da taxa de aprendizado.

```
# ============================================================================
def Logistic_Gradient_Descendent(x_data_init, y_data_init, initial_guess, learning_rate,
    min_residual=1E-3, max_tries=100, normalize=1):

    if normalize == 1:
        x_data, y_data = logistic_normalization(x_data_init), y_data_init
    else:
        x_data, y_data = x_data_init, y_data_init


    thetas   = initial_guess
    residual = residual_function_logistic(x_data=x_data, y_data=y_data, coefficients=thetas)
    tries    = 1

    residuals_values = residual


    while (tries < max_tries) and (np.all(residual) > min_residual):


        gradient = logistic_gradient_residual_function(x_data, y_data, thetas)

        thetas  -= learning_rate*gradient

        residual = residual_function_logistic(x_data=x_data, y_data=y_data, coefficients=
    thetas)
        tries    += 1
        residuals_values = np.append( residuals_values, residual, axis=1 )


    if np.all(residual) <= min_residual:
        print(f'\n   Success! The residual is under the minimal value.\nAfter {tries} steps
    .\n\n')
    elif tries >= max_tries:
        print(f'\n   Failed! The number of tries was exceeded.\n\n')


    if normalize == 1:
        coefficients = undo_logistic_normalization(x_data_init, thetas)
    else:
        coefficients = thetas


    coefficients     = pd.DataFrame(coefficients, columns=['label_'+str(i+1) for i in range(
    coefficients.shape[1])], index=[('theta_'+str(i)) for i in range(coefficients.shape[0])
    ])

    residuals_values = pd.DataFrame(residuals_values)
    residuals_values.columns = ['try_'+str(i) for i in range(residuals_values.shape[1])]
    residuals_values.index   = coefficients.columns


    return( coefficients, residuals_values  )
```

Listing 4: Gradiente Descendente para a função logística

Após encontrar a matriz θ otimizada pelo algoritmo, o próximo passo é aplicá-la nos dados de aperfeiçoamento e classificar os resultados de acordo com as categorias originais. Comparando os resultados, uma taxa de acertos é obtida.

```
# ============================================================================
def classification_result(x_data, thetas_regression):
    '''
    This function is built to...

    Parameters
    ----------
    x_data:
        .
    thetas_regression:
        .
    '''


    logistic = logistic_function(x_data, thetas_regression)

    logistic = pd.DataFrame(logistic, columns=x_data.columns)

    logistic.index = [ (i+1) for i in range(logistic.shape[0]) ]

    classification = pd.DataFrame( logistic.idxmax(axis=0), columns=['label'] )

    return(classification)



# ============================================================================
def Correct_Classification(x_data, thetas_regression, original_labels):
    '''
    This function is built to...

    Parameters
    ----------
    x_data:
        .
    thetas_regression:
        .
    original_labels:
        .
    '''

    classification = classification_result(x_data, thetas_regression)

    if classification.shape != original_labels.shape:
            raise ValueError(f"\n\'classification\' {classification.shape} has not the same
    shape as \'original_labels\' {original_labels.shape}\n")

    correct = (classification == original_labels).astype(int)

    return(classification, correct)
```

Listing 5: Categorização conforme os resultados da regressão logística

2.3 Normalização dos dados

A normalização aparece como uma ajuda para acelerar a obtenção do resultado. No caso, cada coletivo de dados x_i tem seu correspondente normalizado $\bar{x_i}$ tal que

$$\bar{x_i} = \frac{x_i - min(x_i)}{max(x_i) - min(x_i)} \tag{7}$$

A normalização dos dados requer uma eventual ´´des-normalização" dos coeficientes obtidos, já que eles precisam ser transformados de volta para o conjunto de dados originais, nos quais se está trabalhando. Cada função tem a sua des-normalização associada, porém, diferentemente da regressão linear, a regressão logística apresenta maior complexidade para execução da tarefa. Contudo, tomando em consideração que os dados de categorização representam os dados y, é importante notar que, dado que $y = 0, 1$, o conjunto de dados

y está normalizado. Isto é, $y = \bar{y}$. Sendo assim, se queremos manter as igualdades entre esses dois valores, mostra-se que

$$\theta X = \bar{\theta}^T \bar{X}$$

o que implica que, para encontrar os valores de θ correspondentes, precisamos resolver o sistema acima. Usando as propriedades de transposição de um produto matricial, chega-se que

$$X^T \theta = (\bar{\theta}^T \bar{X})^T \tag{8}$$

o que parece muito mais parecido com resolver um sistema com variável θ, matriz dos coeficientes X e matriz dos resultados $(\bar{\theta}^T \bar{X})^T$.

No caso, esse sistema não é trivial de solucionar, porque não possui a mesma quantidade de linhas e colunas. De fato, esse sistema possui múltiplas soluções e pode ser resolvido numericamente pelo método dos mínimos quadrados ao se minimizar o termo $||Ax - b||$. A solução implementada envolveu a biblioteca do *numpy*, com o método *numpy.linalg.lstsq*, já que o escopo do projeto não se tratava disso. Como resultado disso, os coeficientes θ foram encontrados.

2.4 Resultados

Os resultados da regressão logística foram obtidos, como dito, aplicando os valores da matriz θ, encontrada a partir dos dados de teste, nos dados de aperfeiçoamento. Os resultados foram

	Resultados	Percentual(%)
0	804	21,44
1	2946	78,56
total	3750	100

Como pode-se ver, o resultado da taxa de acerto foi de, aproximadamente, 79%. Esse resultado sofre uma melhora desprezível quando se aumenta o conjunto de dados de 25% do total para 50%: a taxa de acerto vai para 80%.

Analisando os histogramas que resultaram da análise de dados, conforme a Fig. 5 pode-se perceber que o algoritmo padece de conseguir reconhecer corretamente o número 5, assim como parece se confundir consideravelmente quando os números são 3, 8 ou 9. Isso possivelmente está relacionado com a própria dificuldade visual em reconhecer os números, já que os três possuem certas semelhanças quanto a suas grafias. No caso, os valores similares na grafia se mostraram bem confusos.

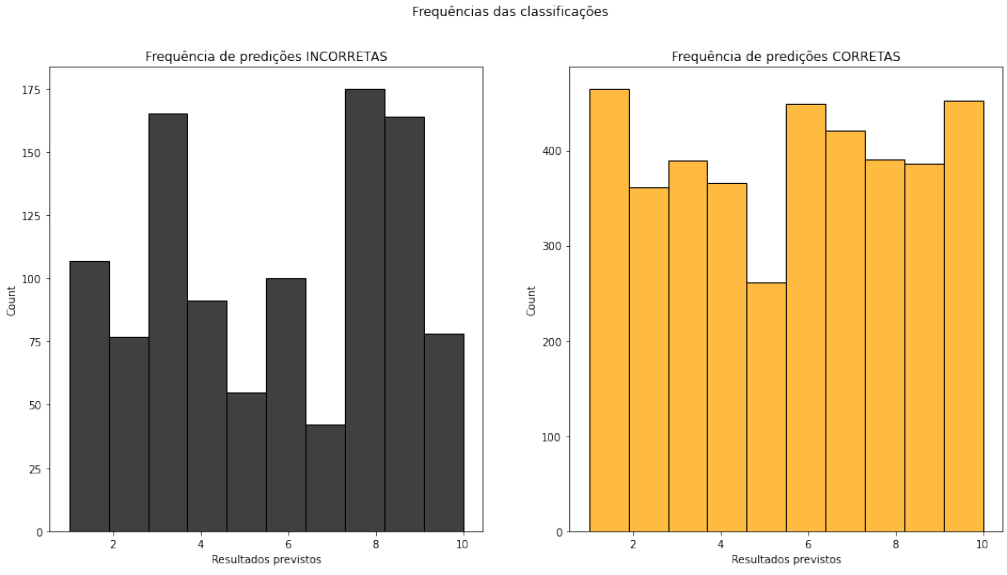

Figura 5: Histograma das previsões conforme acertos e erros.

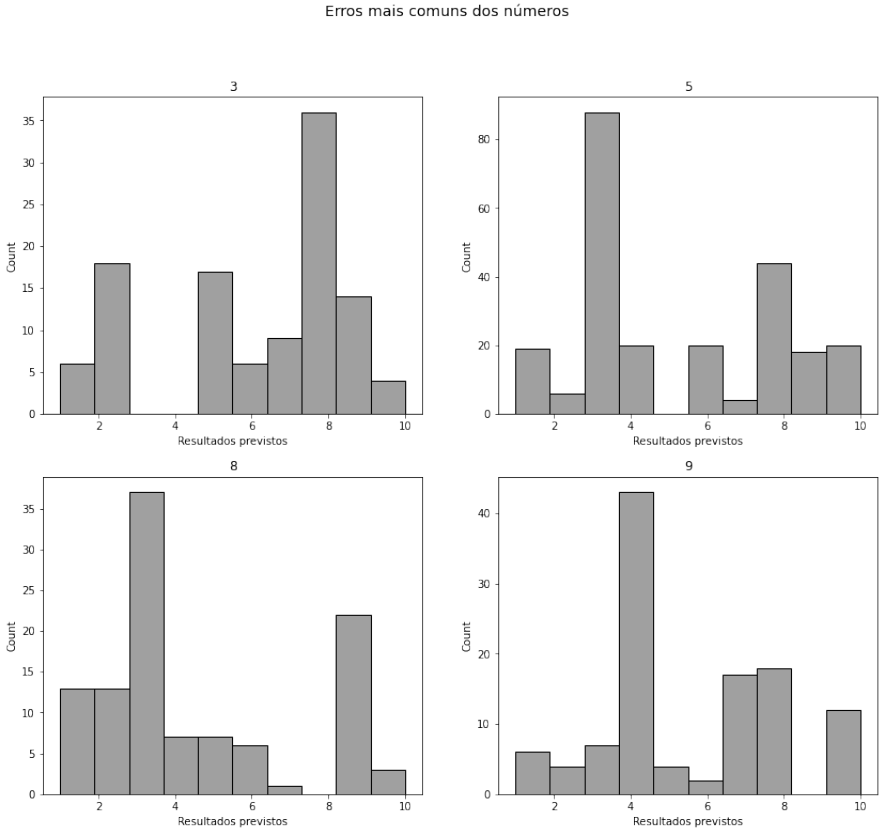

Figura 6: Distribuição dos erros mais comuns.

- No caso dos valores 3, as classificações erradas apontaram mais para categorizar as imagens como 8;
- Já no caso do 5, as classificações erradas apontaram as imagens mais o 3;
- Com o valor 8, as categorizações incorretas classificaram o resultado majoritariamente como 3 e, em segundo lugar, como 9;
- Com o 9, o resultado foi classificado erroneamente como 4.

2.5 Problemas

Alguns problemas ocorreram na resolução desse exercício. O principal deles é o *overflow*/*underflow* encontrado nas operações numéricas. Como nos dados originais já constam valores numéricos ínfimos (10^{-10}), as operações ficam no limiar e, por vezes, podem acabar dando *underflow* (ou *overflow*, na hora de calcular o inverso de um número, como faz a função logística).

Colocando um grande número de tentativas, é necessário ter uma taxa de aprendizado pequena também. Similarmente, por não poder avançar muito na regressão do gradiente descendente sem que esse tipo de erro apareça, os resíduos quase nunca ficaram abaixo do requerido, retornando um valor numérico e uma informação de que o resíduo total teria ficado acima do requerido. Isso significa que os valores de resíduo não puderam descer muito, ficando por volta de $0,2$, o que é bem alto. Um problema similar acontece mesmo ao se normalizar os dados, na hora de des-normalizar e obter as classificações dos dados de aperfeiçoamento.

3 Conclusão

Dessa forma, temos que este trabalho foi capaz de construir modelos iniciais de *machine learning* a partir de dados referentes a pandemia de Covid-19 no Brasil e de um banco de imagens com dígitos manuscritos. Em ambos os casos, é evidente a importância de aprendizado de máquinas, visto que compreender como uma doença evolui ao longo do tempo e possuir capacidade de predição se constitui como uma ferramenta importante para se antever com medidas protetivas. Além disso, assim como modelos para pandemias, ser capaz de identificar imagens a partir de um banco de dados pode ser significativamente importante em áreas como diagnóstico de doenças a partir de imagens médicas.

Assim sendo, foi possível determinar dois modelos, um exponencial e outro polinomial para a evolução temporal dos casos de Covid-19 no Brasil. O modelo exponencial consegue descrever os dados de maneira satisfatória, no entanto, aparentemente, se houvessem mais dados disponíveis referentes a uma quantidade maior de dias, seria possível inferir melhor sobre a qualidade do modelo exponencial. Já o modelo polinomial se adequou perfeitamente aos dados, desde que os polinômios tivessem no máximo grau 8. Para graus maiores, o modelo polinomial não foi capaz de se ajustar aos dados. Além disso, começaram a surgir problemas de divisão numérica por 0, por exemplo. Ademais, o algoritmo de classificação de imagens foi bem mais trabalhoso, porém produziu ótimos resultados, com um grau de acerto de quase 80%.

4 Textos de apoio

Géron, A. (2019). Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems (2nd ed.). O'Reilly.