UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS
PROGRAMA DE PÓS GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Leonardo dos Santos Schmitt

Atividade 1: Regressão Linear

Blumenau
2025

1 ATIVIDADE 1

A primeira atividade consiste na aplicação do algoritmo de Regressão Linear, tanto para uma variável quanto para duas. No caso da aplicação com uma única variável de entrada, o objetivo é prever os lucros de uma franquia de *food truck*. Já na aplicação com múltiplas variáveis, a atividade envolve o uso da Regressão Linear para prever os preços de uma casa.

Os arquivos utilizados para realizar a predição, tanto dos preços dos *food trucks* quanto dos preços das casas, são `ex1data1.txt` e `ex2data2.txt`, respectivamente. É importante ressaltar que esses arquivos foram disponibilizados pelo Prof. Dr. Mauro Roisenberg.

Destaca-se ainda que isso não faz parte do exercício principal, mas está incluído apenas para a prática da utilização de matrizes e vetores na linguagem de programação *Python*, utilizando o módulo `numpy`. Conforme apresentado na Figura 1, é necessário obter uma matriz identidade de dimensão 5×5.

Figura 1 – Saída desejada para uma matriz identidade de dimensão 5x5.

```
ans =

Diagonal Matrix

   1   0   0   0   0
   0   1   0   0   0
   0   0   1   0   0
   0   0   0   1   0
   0   0   0   0   1
```

Fonte: Dr.Prof.Mauro Roisenberg, 2025.

Para realizar a obtenção da matriz desejada, criou-se uma função denominada `def warmUpExercise(tamanho)`, na qual o usuário fornece o parâmetro `tamanho`, uma variável do tipo inteiro responsável por definir a dimensão da matriz identidade. Dentro da função, utiliza-se o comando `np.eye()` do módulo `numpy`, o qual permite criar a matriz identidade de ordem especificada e retorná-la. Essa aplicação está apresentada na Quadro 1.

Quadro 1 – Trecho de código com a implementação da função `warmUpExercise()` .

```python
def warmUpExercise(tamanho):
    A = np.eye(tamanho)
    return A
```

Fonte: Autor, 2024.

Além da função para obter a matriz identidade, foi implementada uma função para plotar os dados de forma 2D, referentes ao banco de dados. Nesse contexto, o eixo x refere-se à variável de entrada (ou variável independente), enquanto o eixo y representa a variável de saída (ou variável dependente). A função apresentada na Quadro 2 utiliza o módulo `matplotlib`, mais especificamente o submódulo `pyplot`, referenciado como `plt`.

No caso do quadro abaixo (Quadro 2), a função está sendo chamada para plotar os dados do problema monovariável. Entretanto, pode-se utilizar a mesma função para visualizar dados multivariáveis, desde que se atente ao fato de que ela irá plotar apenas uma variável de entrada em relação à variável de saída.

Quadro 2 – Trecho de código com a implementação da função `plotData()`.

```
def plotData(x,y):

    fig = plt.figure()# open new figure

    plt.plot(x, y, 'ro', ms=10, mec ='k')
    plt.ylabel('Profit in $ 10.000')
    plt.xlabel('Population of CIty in 10,000s')
```

Fonte: Autor, 2024.

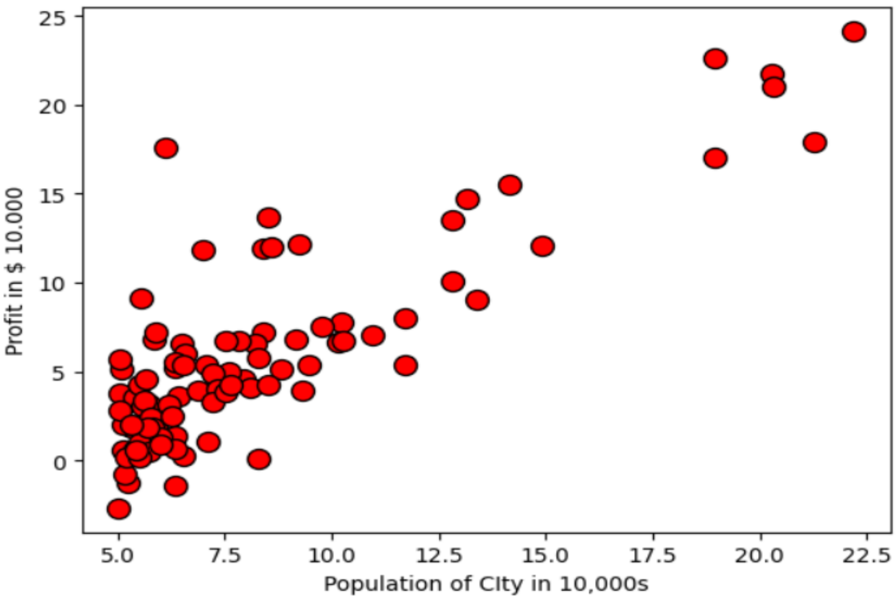
Ressalta-se que, para este documento, serão apresentados em forma de código apenas a função que gera a matriz identidade e a função responsável por plotar os dados em 2D, uma vez que fazem parte da avaliação e podem ser utilizadas tanto para o caso monovariável quanto multivariável. As demais funções utilizadas para resolver o problema de regressão serão descritas em forma de texto, acompanhadas das respectivas equações matemáticas, visto que o objetivo é apresentar os conceitos por trás do funcionamento da Regressão Linear e sua fundamentação matemática.

1.1 RESOLUÇÃO COM VARIÁVEL ÚNICA

A Regressão Linear com uma única variável é relativamente simples de ser aplicada, pois ao treinar o modelo, a complexidade computacional é reduzida. O primeiro problema abordado consiste em estimar o lucro de uma franquia de *food truck*, dada a população de uma determinada cidade. A relação entre essas duas variáveis pode ser visualizada na Figura 2, a qual foi gerada utilizando a função `plotData()`, apresentada anteriormente.

A ideia da regressão linear é traçar várias retas com o objetivo de encontrar aquela que melhor representa a separação linear dos dados. Ou seja, considerando a equação Equação (1), conhecida como equação da reta, e comparando com a equação Equação (2), que representa a hipótese modelada por um neurônio — o *perceptron* — conforme ilustrado na Figura 3, percebe-se que, ao associarmos θ_0 (denominado *bias*) ao termo b, correspondente ao intercepto no eixo y, e θ_1 ao coeficiente angular m, a hipótese de um único neurônio pode ser interpretada como a equação de uma reta.

Figura 2 – Gráfico de População x Lucros.

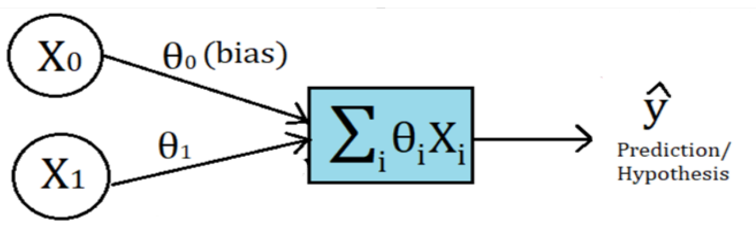


Fonte: Autor 2025.

$$y = m \cdot x + b \tag{1}$$

$$h = \theta_0 + \theta_1 \cdot x \tag{2}$$

Figura 3 – Gráfico de População x Lucros.



Fonte: Autor 2025.

A implementação do cálculo da hipótese é realizada por meio das funções `computeCost()` e `gradientDescent()`, que serão descritas adiante. Antes de qualquer implementação, entretanto, é necessário realizar o carregamento dos dados.

No ambiente *Google Colab*, o dataset pode ser carregado diretamente para a pasta de arquivos e lido utilizando a função `loadtxt()` do módulo `numpy`. Os dados devem ser atribuídos a uma variável — neste caso, foi utilizada a variável `data`.

Após o carregamento, procede-se à separação das variáveis x e y. A coluna correspondente à população pode ser acessada com `data[:,0]`, enquanto a coluna de lucros é acessada com `data[:,1]`. Por padrão, os dados carregados dessa forma são representados como arrays unidimensionais (1D). Para que possam ser utilizados em operações matriciais, é necessário convertê-los para arrays bidimensionais (2D), utilizando a função `expand_dims()` do `numpy`. Essa conversão é fundamental, uma vez que a variável x contém múltiplos valores, sendo necessário que cada um deles seja tratado como uma linha da matriz.

A equação Equação (2) representa a hipótese para um único valor de entrada. Contudo, ao lidar com múltiplos exemplos, a hipótese é calculada por meio da multiplicação matricial entre a matriz de entrada x e o vetor de parâmetros θ_S. Para que essa operação seja válida, deve-se incluir uma coluna adicional composta apenas por 1s, garantindo que o termo de *bias* seja corretamente considerado. Essa construção resulta na forma matricial apresentada na Equação (3).

$$h_\Theta(X) = X \cdot \Theta = \begin{bmatrix} 1 & x^1 \\ 1 & x^2 \\ 1 & x^3 \\ \vdots & \vdots \\ 1 & x^m \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \tag{3}$$

A multiplicação entre a matriz de entrada X e o vetor de parâmetros θ pode ser realizada utilizando o operador `@`, que representa a multiplicação matricial em *Python*, ou por meio da função `dot()` do módulo `numpy`. Já para incluir a coluna de 1s em X, inicialmente é necessário obter a quantidade de linhas dos dados. Isso pode ser feito, por exemplo, com o comando `m = yṡize`, onde m representa o número de amostras. Em seguida, cria-se uma matriz de 1s com dimensões $m \times 1$ utilizando `np.ones((m, 1))`. Por fim, a matriz de 1s é concatenada com a matriz de entrada original por meio da função `hstack()`, formando a nova matriz X com a coluna de *bias* incorporada.

Sabe-se que, ao variar os valores dos parâmetros θ, diferentes hipóteses podem ser geradas. No entanto, não é suficiente adotar uma hipótese de forma arbitrária; é necessário utilizar uma métrica que permita avaliar a qualidade das predições realizadas. Essa métrica é conhecida como *função de perda* (*Loss Function*) ou *função de custo* (*Cost Function*), e tem como objetivo quantificar o erro entre os valores previstos pelo modelo e os valores reais observados nos dados.

A função de custo tem como objetivo medir o quão distante a predição está dos valores reais, ou seja, quantifica o erro cometido pelo modelo ao ajustar uma reta aos dados disponíveis. Entre as métricas existentes, destacam-se o *L1 score*, que representa o erro absoluto médio (*Mean Absolute Error* – MAE), e o *L2 score*, que corresponde ao erro quadrático médio (*Mean Squared Error* – MSE), conforme apresentado nas equações

Equação (4) e Equação (5).

Neste relatório, não serão apresentadas as deduções matemáticas dessas métricas, partindo-se do pressuposto de que o leitor já possui conhecimento prévio sobre o assunto. Ressalta-se, entretanto, que para os propósitos deste trabalho será utilizado o *L2 score*, cuja implementação encontra-se na função `computeCost()`.

$$L_1 = \frac{1}{m} \sum_{i=1}^{m} \left| y^{(i)} - h_\theta(x^{(i)}) \right| \tag{4}$$

$$L_2 = \frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)}) \right)^2 \tag{5}$$

Como é possível visualizar na equação, a função de custo depende da hipótese h. Dessa forma, tanto o cálculo da hipótese quanto o do erro L_2 podem ser implementados diretamente na função `computeCost()`, sendo necessário passar como parâmetros as variáveis x, y e os valores de θ_0 e θ_1. Para realizar o somatório, utiliza-se o comando `np.sum()`, envolvendo toda a equação do erro L_2, exceto a constante $\frac{1}{2 \cdot m}$, que é aplicada posteriormente.

Para verificar se a função está operando corretamente, são realizados testes parciais. No primeiro teste, a função é chamada com o valor zero para ambos os parâmetros θ, resultando em um erro de 32,07, conforme apresentado na Figura 4. No segundo teste, utiliza-se o valor de -1 para θ_0 e 2 para θ_1, cujo resultado, também exibido na mesma imagem, é de 54,24.

Figura 4 – Resultados prévios para teste da função `computeCost()`.

```
With theta = [0, 0]
Cost computed = 32.07
Expected cost value (approximately) 32.07

With theta = [-1, 2]
Cost computed = 54.24
Expected cost value (approximately) 54.24
```

Fonte: Autor 2025.

Em vista do que foi apresentado, pode-se partir para a explicação do Gradiente Descendente, que é um algoritmo de otimização utilizado para ajustar os parâmetros de forma iterativa, com o objetivo de encontrar os valores de θ_0 e θ_1 que minimizem a função de interesse — neste caso, a função de perda (*Loss*). Dessa forma, busca-se obter a reta que melhor se ajusta aos dados.

A função `gradientDescent()` é responsável por implementar esse algoritmo. São passados como parâmetros os dados de entrada (x), os dados de saída (y), o número de iterações — conhecido como épocas (*epochs*), que representa quantas vezes o modelo irá percorrer todo o conjunto de dados durante o treinamento — e o valor de α, chamado de taxa de aprendizado. A taxa de aprendizado é responsável por controlar o tamanho do passo dado em cada iteração para atualizar os valores de θ na direção que minimiza a função de custo.

Através da Equação (6), é possível visualizar como encontrar os valores de θ_s utilizando o conceito de gradiente descendente. Dentro da função `gradientDescent()`, está implementada grande parte do que já foi apresentado sobre a hipótese. No entanto, nesse caso, é necessário criar algumas variáveis para armazenar o histórico dos valores de L_2, θ_0 e θ_1.

Além disso, para a implementação do algoritmo dentro da função, deve-se criar um laço de repetição que varia de 0 até o número de iterações definido pelo usuário. Dentro desse laço, são realizados o cálculo da hipótese, o cálculo do erro e a aplicação do gradiente. A cada iteração, após a atualização dos valores de θ_s, a função `computeCost()` é chamada para calcular o valor do *Loss*. Ressalta-se que tanto o valor da função de custo quanto os parâmetros θ_s são armazenados em listas que registram seu histórico ao longo das iterações.

$$\theta_j = \theta_j - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)} \tag{6}$$

Após a obtenção dos melhores valores de θ, é possível plotar o gráfico dos dados utilizando a função `plotData()`, apresentada anteriormente, juntamente com a reta de regressão. Também pode ser gerado um gráfico que mostra a quantidade de épocas necessárias para a convergência da função `Loss`.

Além disso, é possível representar, de forma tridimensional, a superfície da função de custo e visualizar sua convergência até o mínimo local.

Por fim, os valores de θ obtidos podem ser utilizados para realizar predições. Por exemplo, para estimar o lucro em uma cidade com população 3,5, pode-se utilizar o seguinte comando:

```
predict1 = np.dot([1, 3.5], theta)
```

Os gráficos gerados, bem como os resultados das predições, serão apresentados na seção de resultados.

1.2 RESOLUÇÃO MULTIVARIÁVEL

A resolução multivariável não difere conceitualmente da monovariável, pois as funções `computeCost()` e `gradientDescent()` permanecem as mesmas. A principal distinção está no fato de que, em vez de se utilizar um vetor com apenas dois parâmetros θ, é necessário considerar um vetor com dimensão equivalente à quantidade de variáveis de entrada, conforme ilustrado na equação Equação (7). Isso é possível devido à multiplicação matricial entre X e θ, conforme apresentado na Equação (3).

$$h = \theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \theta_3 \cdot x_3 + \cdots + \theta_n \cdot x_n \tag{7}$$

A principal diferença entre a aplicação multivariável e a monovariável está no fato de que, para o primeiro caso, é necessário normalizar os dados de entrada entre $[0, 1]$ ou $[-1, 1]$. Isso ocorre porque as variáveis podem estar em escalas distintas, em função de suas unidades — como pés, quilômetros, entre outras — o que pode dificultar o processo de modelagem. Um exemplo dessa situação é abordado no artigo *"How to Use StandardScaler and MinMaxScaler Transforms in Python"* (BROWNLEE, 2020).

Imagine que os dados de entrada apresentem valores elevados, com dispersão da ordem de centenas ou milhares de unidades. O modelo, ao tentar aprender com base nessas amplitudes, pode se tornar instável, apresentar baixo desempenho durante o treinamento, e ser sensível a pequenas variações nos dados, resultando em erros de generalização.

Para contornar essa limitação, pode-se implementar uma função denominada `featureNormalize()`, a qual recebe como parâmetro a variável X, contendo todas as entradas do conjunto de dados. No interior dessa função, é realizado o cálculo da média e do desvio padrão ao longo das colunas de X, conforme apresentado em Equação (8) e Equação (9), respectivamente. A implementação dessas expressões pode ser realizada com o uso das funções `mean()` e `std()` do módulo `NumPy`.

Com os valores de média e desvio padrão em mãos, aplica-se a Equação (10) para normalizar cada uma das entradas da matriz X. Após a normalização, o treinamento do modelo pode prosseguir normalmente, utilizando as mesmas funções de custo e de descida do gradiente anteriormente definidas.

$$\mu_j = \frac{1}{m} \sum_{i=1}^{m} x_j^{(i)} \tag{8}$$

$$\sigma_j = \sqrt{\frac{1}{m} \sum_{i=1}^{m} \left(x_j^{(i)} - \mu_j \right)^2} \tag{9}$$

$$x_{j,\mathrm{norm}}^{(i)} = \frac{x_j^{(i)} - \mu_j}{\sigma_j} \tag{10}$$

É importante ressaltar que, após o treinamento do modelo, pode-se realizar a previsão para uma nova casa. Por exemplo:

```
house = np.array([1650, 3]).
```

Entretanto, deve-se ter cuidado, pois esses dados também precisam ser normalizados antes da predição; caso contrário, o resultado poderá divergir do esperado.

A normalização é particularmente importante quando se utiliza algoritmos como a descida do gradiente ou funções de custo, pois esses métodos são sensíveis à escala das variáveis. No entanto, ao empregar o método dos mínimos quadrados, não é necessário normalizar as entradas. Conforme apresentado na Equação (11), observa-se que essa fórmula é válida independentemente da escala das variáveis, uma vez que não depende de um processo iterativo de otimização. Ou seja, a escala das variáveis não afeta a convergência, já que o método fornece uma solução analítica direta. Ainda assim, embora a normalização não interfira no cálculo da solução, a presença de variáveis em escalas muito distintas pode dificultar a interpretação dos coeficientes. Por exemplo, se uma variável estiver em milhões e outra em unidades, o coeficiente da variável com maior escala poderá parecer menor apenas devido à sua magnitude — o que pode distorcer a análise sobre a importância relativa de cada variável no modelo.

$$\theta = (X^T X)^{-1} X^T y \tag{11}$$

Em vista do que foi explicado, a implementação do código foi realizada por meio da função `normalEqn()`, à qual devem ser passados os valores de entrada X e saída y. A equação apresentada na Equação (11) pode ser implementada em *Python* utilizando o seguinte comando:

```
theta = np.linalg.inv(Xin.T @ Xin) @ Xin.T @ Yin
```

2 RESULTADOS

Neste capítulo, serão apresentados os resultados obtidos tanto para a solução aplicada no caso monovariável quanto para o multivariável. Para uma melhor representação dos resultados, serão exibidas apenas as imagens de forma clara e sucinta.

2.1 RESULTADOS CASO MONOVARIÁVEL

Os resultados para o caso monovariável foram obtidos após o treinamento do modelo, conforme apresentado no capítulo anterior, onde foram testados valores fixos para θ_0 e θ_1. No entanto, como explicado anteriormente, essa abordagem manual é limitada, sendo necessário utilizar métodos automáticos de otimização para encontrar a melhor reta que se ajuste linearmente aos dados. A Tabela 1 apresenta os hiperparâmetros utilizados para o treinamento do modelo com o algoritmo de descida do gradiente.

Tabela 1 – Análise de execuções para diferente valores de hiperparâmetros caso monovariável.

Valores para Hiperparâmetros		
Hiperparâmetro	Execução 1	Execução 2
Taxa de aprendizado (α)	0.01	0.001
Número de épocas	1500	1500

Fonte: Autor, 2024.

Na Tabela 1, observa-se que o número de épocas — ou seja, o número de iterações — foi mantido constante em ambas as execuções. Essa escolha tem como objetivo avaliar se a mesma quantidade de iterações é suficiente para a convergência do modelo, mesmo com diferentes valores de α.

Como pode ser visto nas Figura 5 e Figura 6, quando $\alpha = 0{,}01$, com 1500 iterações, o modelo converge adequadamente, e os dados são separados de forma linear, conforme o esperado.

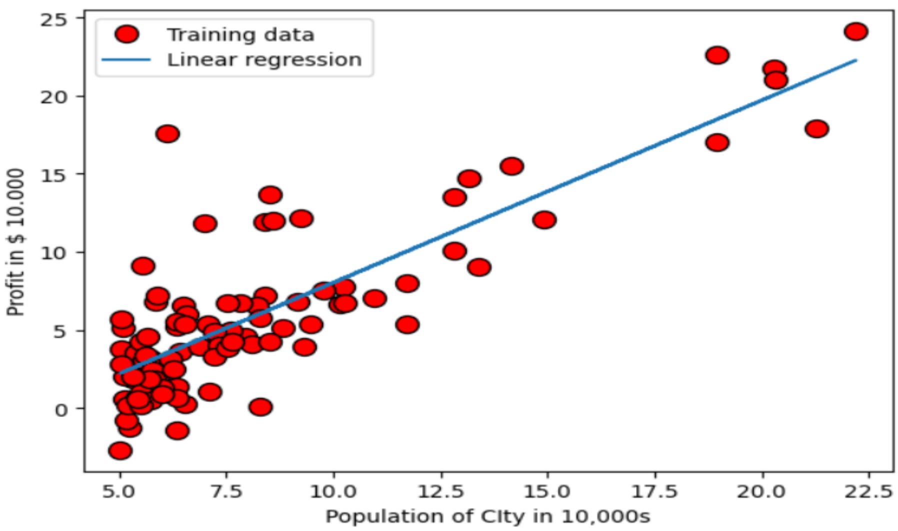
Além disso, pode-se observar, por meio da Figura 7, a evolução da função de custo ao longo das iterações. Outro gráfico, utilizado para uma análise mais criteriosa, é o da superfície, o qual apresenta tanto a forma da superfície gerada pela função de custo quanto a trajetória da descida do gradiente em direção ao ponto de mínimo local. Esse gráfico pode ser visualizado na Figura 8.

Figura 5 – Resultados obtidos através da execução 1.

```
Theta computed from the gradient descent: [[-3.63029144]
 [ 1.16636235]]
Expected theta values (approximately):  [-3.6303, 1.1664]
Mean Squared Error for training data computed from obtained Thetas: 4.483388256587726
```

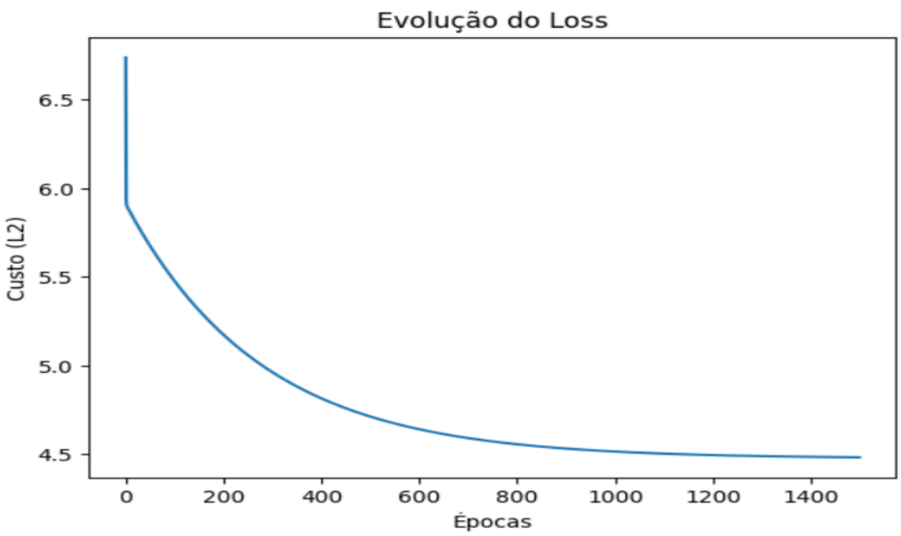
Fonte: Autor 2025.

Figura 6 – Gráfico gerado através da execução 1.



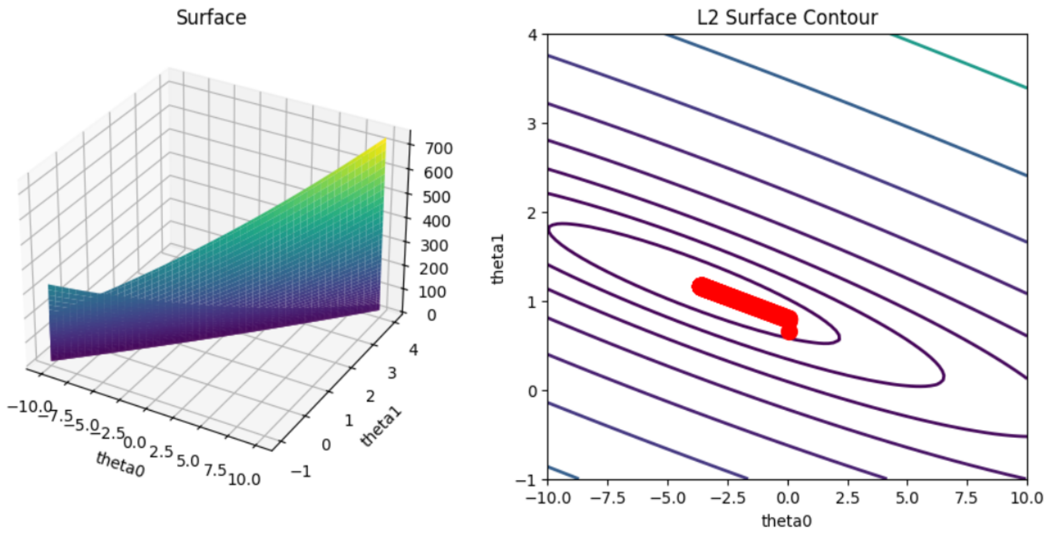
Fonte: Autor 2025.

Figura 7 – Evolução do *Loss* através da execução 1.



Fonte: Autor 2025.

Figura 8 – Gráficos de superficie.



Fonte: Autor 2025.

Já os resultados obtidos quando $\alpha = 0{,}001$, para o mesmo número de iterações, não se mostraram eficazes, conforme apresentado na Figura 9, uma vez que o modelo não foi capaz de separar os dados de forma suficientemente linear. Esse problema ocorre devido ao fato de que o número de iterações é pequeno em comparação ao tamanho do passo, como pode ser observado na Figura 10. Diante disso, ao invés de aumentar a quantidade de iterações, optou-se por prosseguir com $\alpha = 0{,}01$ para realizar novas predições.

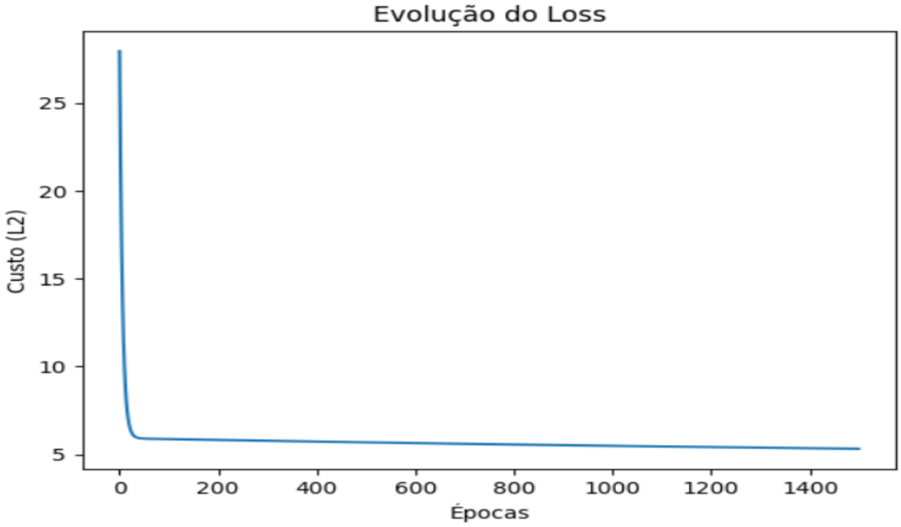
Figura 9 – Resultados obtidos através da execução 2.

```
Theta computed from the gradient descent: [[-0.86221218]
 [ 0.88827876]]
Expected theta values (approximately):  [-3.6303, 1.1664]
Mean Squared Error for training data computed from obtained Thetas: 5.314765150593782
```

Fonte: Autor 2025.

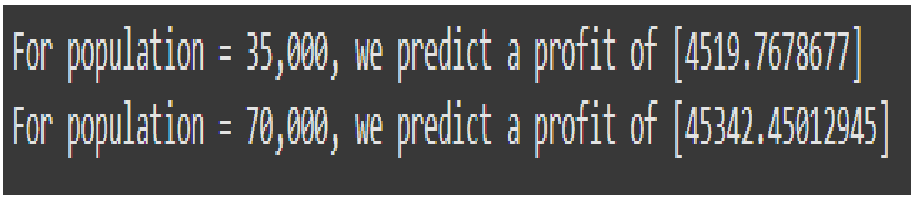
Ao realizar a predição de novos dados, como, por exemplo, para uma população de 35.000 e 70.000 habitantes, é possível observar, por meio da Figura 12, que o modelo consegue estimar corretamente os lucros. Esse resultado deve ser analisado com base na comparação com os dados presentes no banco de dados, concluindo-se, assim, que o método foi implementado com sucesso.

Figura 10 – Evolução do *Loss* através da execução 2.



Fonte: Autor 2025.

Figura 11 – Evolução do *Loss* através da execução 2.



Fonte: Autor 2025.

2.2 RESULTADOS CASO MULTIVARIÁVEL

Conforme mencionado anteriormente, o caso multivariável se assemelha ao monovariável; contudo, é necessário realizar a normalização dos dados de entrada. Os dados normalizados não serão apresentados neste documento, sendo exibidos apenas os resultados obtidos para dois valores distintos de α, conforme apresentado na Tabela 2. Além disso, será realizada uma comparação entre o método do gradiente descendente e o método dos mínimos quadrados.

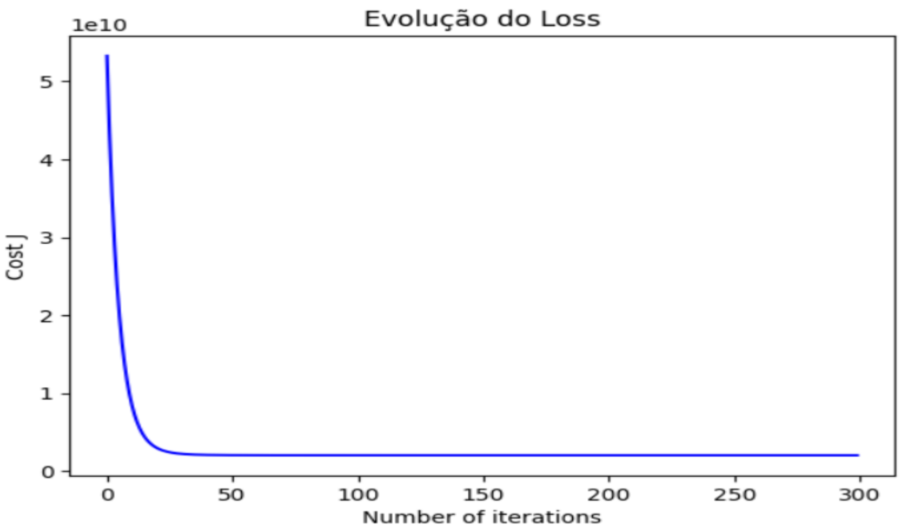
Ao compararmos a Figura 13 e a **??**, é possível perceber a diferença entre os valores de $\alpha = 0{,}1$ e $\alpha = 0{,}3$. Observa-se que, em ambos os casos, o modelo converge para o mínimo da função de custo. Contudo, quando $\alpha = 0{,}3$, o valor mínimo é alcançado mais rapidamente, evidenciando uma convergência mais eficiente.

Tabela 2 – Análise de execuções para diferente valores de hiperparâmetros caso multivariável.

Valores para Hiperparâmetro		
Hiperparâmetro	Execução 1	Execução 2
Taxa de aprendizado (α)	0.1	0.3
Número de épocas	300	300

Fonte: Autor, 2024.

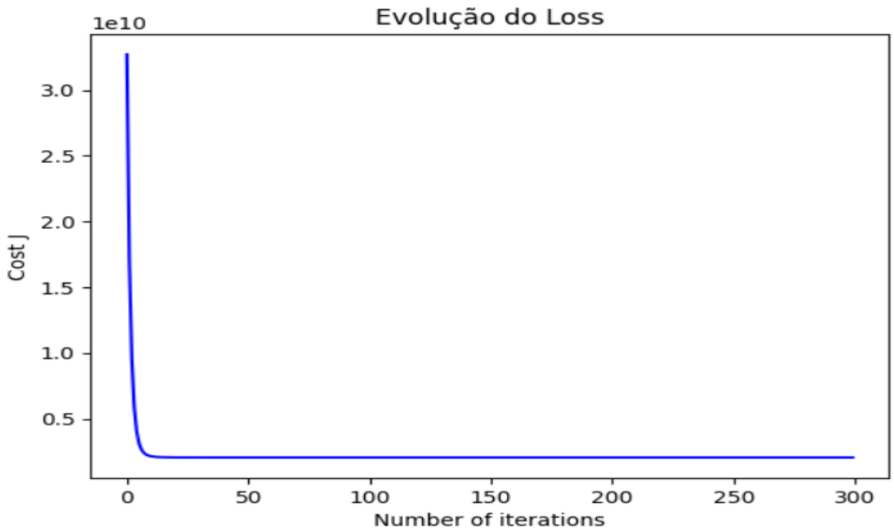
Figura 12 – Evolução do *Loss* através da execução 2.



Fonte: Autor 2025.

Ao realizar a predição do valor de uma nova casa, com tamanho de 1650 *ft* e 3 (três) quartos, o valor estimado foi de $293.081,48. Quando utilizado o método dos mínimos quadrados, o mesmo valor foi encontrado, conforme apresentado na **??**. Conclui-se, portanto, que ambos os métodos foram implementados com sucesso.

Figura 13 – Evolução do *Loss* através da execução 2.



Fonte: Autor 2025.

Figura 14 – Evolução do *Loss* através da execução 2.

```
Theta computed from the normal equations: [89597.9095428   139.21067402 -8738.01911233]
Predicted price of a 1650 sq-ft, 3 br house (using normal equations): $293081
```

Fonte: Autor 2025.

REFERÊNCIAS

BROWNLEE, Jason. **How to Use StandardScaler and MinMaxScaler Transforms in Python**. [*S.l.*: *s.n.*], 2020. Acessado em abril de 2025. Disponível em: `https://machinelearningmastery.com/standardscaler-and-minmaxscaler-transforms-in-python/`.