Estudos em Regressão Linear

Matheus Mortatti Diamantino
RA 156740
matheusmortatti@gmail.com

José Renato Vicente
RA 155984
joserenatovi@gmail.com

I. Introdução

Este projeto teve como intuito o estudo prático do método de regressão linear em Machine Learning. Foram utilizado os algoritmos de *Gradient Descent* conhecidos como *Stochastic, Batch e Mini Batch* bem como a *Equação Normal*, que resolve o problema da Regressão Linear por uma equação matricial fechada, de modo a compara-los em termos de complexidade e acurácia. [1]

Foi feito um estudo de predição de preço de diamantes, utilizando uma base de dados com 54000 exemplos, em que são apresentados seus preços e nove features como tamanho, cor e número de quilates.

II. Atividades

A. Regressão Linear [2]

Regressão Linear é um método muito conhecido de Machine Learning, utilizado para predizer o valor de uma variável dependente baseado em valores de variáveis independentes. Essa regressão é chamada linear porque se considera que a relação da resposta às variáveis é uma função linear de alguns parâmetros. Desta forma, dado um vetor Theta de tamanho igual ao número de features, cujo valor queremos determinar, temos que:

$$PreçoAlvoEsperado = \sum_{i=1}^{m} \theta_i X_i = h_\theta(x) \tag{1}$$

Em que X é um vetor com os valores das features para um dado diamante, cujo preço queremos determinar. Para encontrar esse valor de Theta, utilizaremos alguns algoritmos e compararemos os resultados obtidos com cada um.

Cada algoritmo utilizado é baseado no método de *Descrida de Gradiente (ou Gradient Descent)*. Este é um método utilizado para achar o ponto mínimo de uma função, aproximando gradativamente seu valor até um ponto quando não é possível ser diminuido mais (i.e. a derivada da função neste ponto é zero). Este método consegue apenas achar mínimos locais e, com isso, não é garantido que o resultado obtido é o melhor para o dado problema.

Para medirmos a eficácia do algoritmo, utilizamos uma *Função de Custo* que nos diz o quão perto do resultado desejado estamos, dado um conjunto de dados. Esta função é definida por:

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

Como queremos minimizar a função de custo, queremos que cada passo da nossa descida de gradiente se aproxime mais do mínimo local. Para extrairmos a direção que temos que ir, utilizamos a derivada da função de custo:

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^i) - y^i) x^i \tag{3}$$

Logo, para aproximarmos os valores de θ de modo a nos aproximar do mínimo local, utilizamos a fórmula

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^i) - y^i) x^i \tag{4}$$

, onde $0 \leq j \leq m$, $x_0 = 1$ e α é o que chamamos de *Learning Rate* que define o quão agressivamente tentaremos nos aproximar do mínimo. Este método de descida de gradiente é chamada de *Batch Gradient Descent*. A seguir, veremos duas outras variações deste algoritmo e uma forma não iterativa para o problema.

a) Stochastic Gradient Descent: Neste método, utiliza-se apenas um exemplo de treino para cada passo da descida de gradiente. Deste modo, a equação 4 se transforma em

$$\theta_j := \theta_j - \alpha (h_\theta(x^i) - y^i) x^i \tag{5}$$

Utiliza-se este método quando procura-se rapidez de execução. Contudo, um ponto negativo deste método é que, como usamos como amostra apenas um exemplo de treino, um passo pode nos levar a um custo mais alto. Como o número de iterações é alta, porém, o método converge ao mínimo e mais eficientemente do que o *Batch* até um certo limite.

b) Mini Batch Gradient Descent: Para obtermos um resultado balanceado, utiliza-se uma mistura dos métodos *Batch* e *Stochastic*, em que define-se um tamanho para o lote de exemplos de treino que serão utilizados para atualizar cara θ.

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=k}^{k+b-1} (h_\theta(x^i) - y^i) x^i \tag{6}$$

, onde b é o tamanho do lote, $k = 0, b, 2b, ..., m-1$.

c) Equação Normal: Para a regressão linear, é possível derivar uma fórmula direta para o ponto de mínimo local que desejamos. Para isso, utilizamos manipulações matriciais na forma [3]

$$\theta = (X^T X)^{-1} X^T y \tag{7}$$

, onde X é a matriz de features, y é a matriz dos dados que queremos prever e θ é a matriz dos coeficientes de $h(X)$.

B. Normalização de Features

Como cada feature tem seu valor em uma escala diferente (i.e. algumas estão na ordem de milhares e outras na ordem de centenas), o processo de descida do gradiente poderá acontecer de forma lenta. Isso se dá pelo fato de que a atualização dos θs não ocorrerá de forma uniforme entre as features, já que a distância de um dado θ_i a seu valor esperado pode ser maior do que de outro $\theta_j, j \neq i$. Assim, realizamos o que é chamado de *Normalização de Features*, onde colocamos todas as features x_i em um valor entre $0.5 \leq x_i \leq 0.5$. Isso é feito através da fórmula:

$$x_i = \frac{x_i - \frac{size(x_i)}{2}}{size(x_i)} \tag{8}$$

C. Transformação de Features Com Valores Não Reais

Em alguns casos, podemos ter features que não possuem um valor no domínio dos números reais. Por exemplo, uma feature pode representar a cor de um dado elemento. Deste modo, é necessário realizar uma transformação de tais features para o domínio dos reais. Para realizar tal transformação, criamos uma nova feature para cada possível valor da feature que queremos transformar, de modo que, se para o exemplo de dado e_i temos que esta feature possui um valor x_j, a nova feature correspondente a x_j terá o valor 1 e as demais features criadas terão o valor 0. Tomemos como exemplo uma feature de Cor que pode receber os valores *Azul, Amarelo, Vermelho e Verde*. Assim, o resultado da transformação acontecerá da seguinte forma:

Feature x_i	Azul	Amarelo	Vermelho	Verde
Azul	1	0	0	0
Amarelo	0	1	0	0
Vermelho	0	0	1	0
Verde	0	0	0	1

Onde cada coluna representa a nova feature criada e cada linha representa o valor original da feature x_i. Chamamos este método de *Grid*.

D. Regularização

Regularização é um método utilizado para *evitar* overfitting dos dados. É relizado de forma a penalizar os valores de θ para que estes mantenham valores pequenos, sendo menos propenso ao overfitting. Isto é feito na forma de um novo parâmetro λ que definirá o quanto cada θ será penalizado:

$$\theta_j := \theta_j * (1 - \lambda * \frac{\alpha}{m}) - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^i) - y^i)x^i \tag{9}$$

Este valor deve ser modificado de forma a melhorar o resultado final. Se o valor for muito alto, a influência da regularização pode ser negativa, criando um aumento no erro final. Se muito baixo, não terá o efeito desejado. Vale mencionar que a regularização na equação normal se da pela forma

$$\theta = (X^T X + \lambda I)^{-1} X^T y \tag{10}$$

, onde I é a matriz identidade.

E. Base de Dados

Foi utilizado uma base de dados sobre Diamantes, a qual descreve diferentes aspectos sobre o diamante como cor, claridade e quilate, e apresenta seu preço. O objetivo deste projeto foi de utilizado o método de *Regressão Linear* descrito acima para prever o preço de um dado diamante, dado suas características.

III. Soluções Propostas

A. Tratamento dos Dados

Para obtermos melhores resultados, começamos por realizar o tratamento dos dados. Para isso, foram aplicados os métodos de *Normalização* descritos na seção anterior. Também, foi aplicado o método de *Grid* nas features *Cut, Color e Clarity*, pois seus valores são descritos por uma string.

B. Algoritmos Implementados

Com o objetivo de compara-los, foram implementados os algoritmos *Batch, Mini Batch, Stochastic e Equação Normal* como solução para a Regressão Linear.

a) Algoritmos Iterativos: Algoritmos iterativos para a *Descida de Gradiente* precisam de um critério de parada para sua execução. Tais critérios foram, demonstrados pela Listagem 1:

- Número máximo de iterações.
- Valor máximo da a diferença entre os θs originais e novos.

```
while(iterations < max_iterations and not
    done):
   # Do Gradient Descent
   ...
   iterations = iterations + 1
   if iterations >= max_iterations:
      break
   # If the change in value for new thetas is
       too small, we can stop iterating
   done = True
   for k in range(len(thetas)):
      done = abs(thetas[k] - new_thetas[k]) <
          stopCondition and done
   if done:
      break
```

Listing 1: Criterios de Parada para os algoritmos de Descida do Gradiente

Foi implementado também uma solução para o caso onde a função de custo diverge devido a um α muito alto. Caso o custo corrente seja maior do que o último custo calculado, o valor de α é diminuido por um fator pré definido, de modo que a descida aconteça com menores riscos de divergência, como demonstrado pela Listagem 2.

```
if len(costs) > 0 and cost > costs[-1]:
   learningRate *= alphaFactor
   if retryCount < retryMax:
      retryCount += 1
   else:
      done = true
```

Listing 2: Modo como o fator α é modificado

b) Regularização: Para todas as soluções implementadas, foi utilizado o método de regularização para melhorar os resultados e evitar *Overfitting*. Foram seguidos os métodos descritos acima para a implementação tanto na forma da *Equação Normal* quanto para os algoritmos iterativos.

IV. Experimentos e Discussão

Os experimentos foram realizados em uma máquina que possui um processador Intel Core i7-6700HQ com 4 cores rodando a 2.60GHz e 16GB de RAM, com Ubuntu 16.04.

A. Comparação de Tratamento de Dados

a) Aplicação de Normalização: Os resultados obtidos rodando Stochastic Gradient Descent para dados sem normalização foram de 812.20 para o erro absoluto nos dados de treino e 743406.95 para o custo.

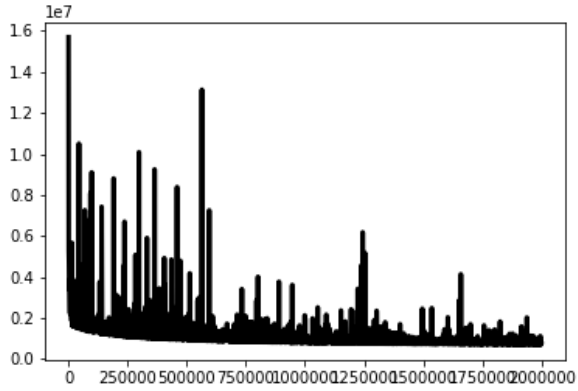

Figure 1: Custo x Iterações Para Stochastic Sem Normalização

Em acordo com os resultados obtidos, era esperado que os resultados fossem melhores para os dados normalizados, que se encontram na Tabela I. A diferença foi pequena, pois os valores encontrados nas diversas features nao são muito diferentes (mesma ordem para todos os dados).

A Figura 1 nos mostra que o caminho para a minimização do custo sem normalização teve muito mais ruido, sendo necessário o ajuste do *Learning Rate* quando o valor divergia. Isso se dá pois atualizamos os valores dos coeficientes para cada exemplo de treino e, como os valores não estão normalizados, isso ocorre de forma não uniforme, fazendo com que o custo varie muito mais.

b) Aplicação de Grid: Os resultados obtidos utilizando a Equação Normal para dados sem *Grid* foram de 794.47 para o erro absoluto nos dados de treino. Com a aplicação de *Grid*, os resultados foram de 729.33 para o erro absoluto.

Os resultados melhores para a aplicação de grid mostram que a aplicação de uma ordem de valores para as features não reais não funciona muito bem, pois isto infere que os valores para tais features possuem uma relação de grandeza crescente ou decrescente, o que pode não ser o caso.

B. Aplicação da Regularização

Os resultados obtidos utilizando a Equação Normal para dados com e sem a *Regularização* foram feitos sem a aplicação de *Grid*, com *Normalização*.

Tais resultados foram de 794.47 com $\lambda = -1.5$ e de 805.15 com $\lambda = 0$ nos dados de treino. A pequena diferença ocorre pelo fato de que a regressão linear, por utilizar apenas termos de ordem primária em sua função $h(\theta)$, não sofre de um *Overfitting* significativo, não necessitando da aplicação da regularização para evitá-lo.

Com a aplicação de todos os métodos de tratamento de dados descritos até aqui, foram testados valores para λ de modo a acharmos seu valor ótimo. Isso pode ser ilustrado pela Figura 2.

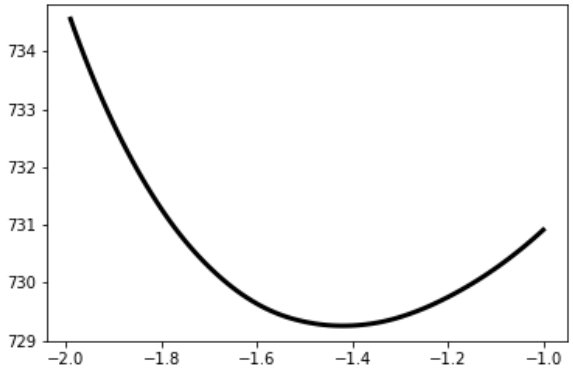

Figure 2: Erro X Lambda na Equação Normal

C. Comparação entre Algoritmos

Os algorítmos de Gradient Descent mostraram ter uma diferença de performance muito grande. Para um tempo de execução consideravelmente menor, Stochastic obteve resultados muito melhores do que as duas outras variantes, que demoraram muito mais para convergir, como podemos ver pela Tabela I. Por observar apenas um diamante do dataset de treino a cada atualização de Theta, a atualização deste vetor é muito mais rápida. Desta forma, ele chega mais próximo do mínimo local da função do que Batch e Mini Batch.

Porém, caso o algorítmo fosse executado por tempo suficiente, as outras variantes obteriam resultados com erros menores, já que, por utilizar mais do que um diamante do dataset de treino para atualizar Theta, suas atualizações são

mais precisas e, no caso do Batch, tem certeza que o novo vetor de thetas leva a um resultado mais proximo do minimo local, o que não ocorre com o Stochastic.

O método da equação normal foi o de melhor desempenho, tendo tempo de execução muito inferior aos demais, além de ser o método com os melhores resultados. Isso se deve pois, enquanto os outros métodos necessitam de diversas iterações para convergirem ao o minimo local, a equação normal já dá o resultado exato do mínimo local.

	Absolute Error	Final Cost	Run Time(s)	Iterations
Stochastic	753.24	645820.77	676	2.000.000
Mini Batch	991.72	1002188.59	1693.64	2.000
Batch	2554.09	6426353.60	4142.068	1000
Equação Normal	729.33	561271.499184	0.455	——
Scikit Learn	739.09	——	3561.999	200.000

Table I: Tabela de Resultados de Treino Para Cada Algoritmo

	Absolute Error	λ	α
Stochastic	743.73	0.0000005	0.01
Equação Normal	726.32	-1.5	——
Scikit-Learn	731.10	0.0000005	0.01

Table II: Tabela de Resultados de Validação Para Os Algoritmos Mais Eficientes

Comparando os valores obtidos com a implementação para o algoritmo *Stochastic* feita pelos autores deste projeto e a feita pela framework *Scikit-Learn* [4], vemos que eles são muito similares em termos do erro final obtido, tanto para os dados de treino (Tabela I) quanto para os dados de validação (Tabela II). Contudo, vemos divergências no número de iterações e no tempo de execução. Enquando o scikit obteve resultados muito próximos da Equação Normal em apenas *200.000* iterações, nossa implementação precisou de $2x$ mais. Em contrapartida, scikit teve um tempo de execução aproximadamente $5x$ maior, o que pode ser explicado pelos recursos que o algoritmo utiliza para evitar overfitting, divergência, entre outros detalhes de implementação. Vale notar também que a base de dados utilizada é relativamente simples e não necessita de um algoritmo com a complexidade do encontrado na framework, explicando os bons resultados obtidos pela nossa implementação.

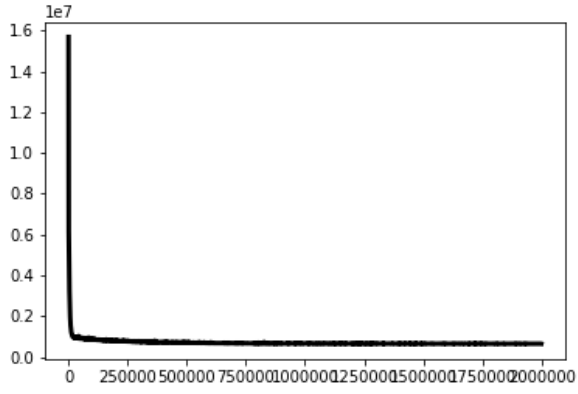

Figure 3: Custo x Iterações Para Stochastic

Observando a Figura 3, podemos notar a rapidez com que o algoritmo *Stochastic* converge no início de sua operação. Contudo, após aproximadamente 250.000 iterações, a taxa de convergência diminui significativamente. A figura também nos mostra como um *Learning Rate* bem definido afeta o resultado final. Se compararmos com a Figura 1, podemos ver que esta possui um ruído muito significativo, proveniente, dentre outros motivos, de uma taxa de aprendizado muito agressiva para a função.

D. Regressão Polinomial

Foram feitos breves experimentos com *Regressão Polinomial* para verificar seus impactos nos resultados. Para isso, foi utilizado o método da Equação Normal e as features foram modificadas de modo a incluir termos polinomiais. Como podemos ver pela Tabela III, os resultados para a regressão polinomial não são significativamente melhores que os da regressão linear. Isso pode ser explicado pelo *Erro Irredutível* inerente dos dados utilizados. Isto significa que a qualidade dos dados utilizados não é boa o suficiente para realizar um treino e uma predição precisa. Vale lembrar também que o quanto mais aumentamos o coeficiente polinomial das features, maiores as chances de *Overfitting*, mesmo que minimizados pelo processo de Regularização.

	Absolute Error (Validation Data)
X	726.32
$X + X^2$	720.94
$X + X^2 + X^3$	705.90
$X + X^2 + X^3 + X^4$	705.34

Table III: Tabela de Resultados da Regressão Polinomial

V. Conclusões

Após extensiva análise dos resultados, é seguro dizer que os estudos realizados em *Regressão Linear* foram profundos o suficiente para obter um entendimento maior sobre o assunto. Pode-se ver os efeitos de que um bom tratamento de dados, como normalização e aplicação de grids, pode trazer no resultado final, com melhora de 7.2% com o uso do primeiro e X% com o segundo. Analisando as diferenças de performance e resultados de cada método, pode-se entender quando cada abordagem é mais adequada. Os métodos que melhor desempenharam nesse projeto foram a equação normal e Stochastic GRadient Descent. Os resultados foram em linha com o esperado mesmo com erros relativamente altos. Para obter erros menores, novas abordagens de solução do problema podem ser utilizadas, seja um outro tipo de regressão, como a polinomial que obeteve resultados um pouco melhores, seja com técnicas mais poderosas.

References

[1] S. Avila. [Online]. Available: https://www.ic.unicamp.br/~sandra/

[2] A. Géron, "Hands-on machine learning with scikit-learn and tensorflow." [Online]. Available: https://www.oreilly.com/library/view/hands-on-machine-learning/9781491962282/ch04.html

[3] R. Kapur. [Online]. Available: https://ayearofai.com/rohan-3-deriving-the-normal-equation-using-matrix-calculus-1a1b16f65dda

[4] "scikit-learn, machine learning in python." [Online]. Available: http://scikit-learn.org/