Trabalho prático 1

Regressão Linear

Julianny Favinha Donda
156059
julianny.favinha@gmail.com

José Henrique Ferreira Pinto
155976
joseh.fp@gmail.com

1. Introdução

Em complemento às aulas de MC886, e visando dar aos alunos um primeiro contato com aprendizado de máquina, o primeiro trabalho é proposto para explorar alternativas à técnica de Regressão Linear. O objetivo é prever preços de diamantes baseados entre uma e nove *features* como dimensões, cor e qualidade. A base de dados[1] a ser utilizada foi encontrada no site Kaggle[2].

2. Atividades

Regressão linear é um método que usa a soma ponderada dos elementos usados como *feature* e um termo de viés para criar uma função que pode ser usada para prever observações[5].

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n \quad (1)$$

Treinar um modelo linear envolve encontrar os parâmetros θ_n que formam com as *features*, a melhor estimativa mais próxima da observação real.

Função de custo

Para conseguir avaliar se o modelo proposto está chegando a um resultado satisfatório, definimos uma função de custo que é a medida de quão errado está o modelo em termos de estimar a relação entre as *features* e o *target* y. A função de custo utilizada foi a de erro quadrático médio

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

onde m é a quantidade de dados do conjunto de treinamento, h_θ é o valor estimado (hipótese, ou modelo) e y é o valor real para o conjunto de x.

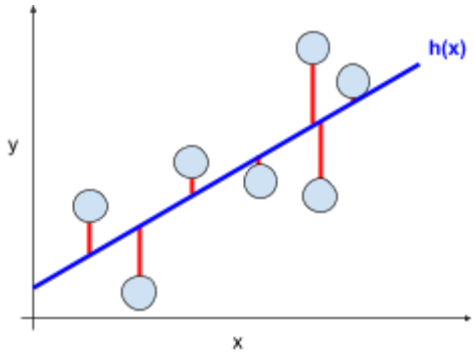

Figura 1 - Diagrama livre para exemplificar a função de custo. A soma das linhas vermelhas (verticais) representa o erro total, dado pela diferença da previsão h(x) e o valor real y(x), m é o número de observações (círculos).

Descida de Gradiente

O método de Descida de Gradiente encontra os parâmetros θ. A somatória dos erros em função dos parâmetros da função $h(x)$ nos dá uma ideia de quais parâmetros devem ser escolhidos. Encontrando o vetor gradiente da função, conseguimos saber para qual lado o erro é minimizado.

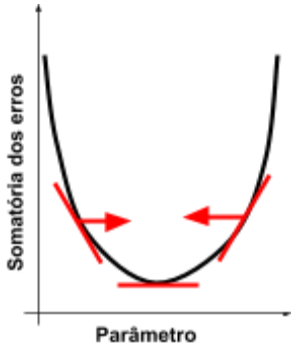

Figura 2 - Diagrama livre da descida do gradiente. O gradiente da função diz para qual lado seguir para minimizar o erro.

Iterando os valores dos parâmetros baseados nessa direção, conseguimos descer até um mínimo local, por isso o método se chama descida de gradiente. A equação utilizada é explicitada a seguir.

$$repeat \left\{ \theta_0 = \theta_0 - a\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \right\} \quad (3)$$

onde a é o *learning rate*, m é a quantidade de dados do conjunto de treinamento, h_θ é o modelo proposto, e y é o valor real para o conjunto de dados.

3. Soluções propostas

Para tentar melhorar o resultado de uma regressão linear, primeiro precisamos compreender a técnica estudada. Para isso, assumimos que os dados podem ser modelados através de uma função linear, exemplificada a seguir.

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 + \\ + \theta_5 x_5 + \theta_6 x_6 \quad (4)$$

Escolhemos 6 *features*: quilate, profundidade do corte, mesa, largura, altura e profundidade.

Para saber se estamos no caminho certo, vamos comparar os resultados encontrados com dados conhecidos como ideais: primeiro, com o resultado do SGDRegressor (*Stochastic Gradient Regressor*) do framework Scikit Learn[3], que aplica o método de descida de gradiente já estabelecido na literatura e descrito na seção 2.

Posteriormente, iremos comparar com os parâmetros fornecidos pela aplicação do método de Equação Normal. Este método utiliza cálculo matricial para encontrar os parâmetros exatos. Porém, por utilizar operações matriciais, é um método muito custoso (tempo $O(n^3)$) para usar em qualquer conjunto de dados. A equação normal é descrita a seguir.

$$\theta = (X^T X)^{-1} X^T y \quad (5)$$

onde X é a matriz $m \times (n+1)$ de m observações e $n+1$ features, e y é o vetor com os m targets observados respectivos às linhas em X.

4. Discussão e experimentos

Os dados foram normalizados usando a abordagem de Mínimo-Máximo[6], usando a seguinte equação

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}} \quad (6)$$

onde X é a matriz dos dados. Essa normalização levou em conta apenas os dados de treinamento ao encontrar os pontos mínimo e máximo, ou seja, os dados de teste foram normalizados com os mínimo e máximo encontrados no conjunto de treinamento. A normalização foi realizada para atenuar efeitos de *outliers*, ou seja, dados que fogem do padrão gerado pela maioria dos dados.

Seguindo as recomendações dadas em aula para evitar *overfitting*, separamos os dados em duas partes, sendo **8091** observações de teste e **45849** observações para o treinamento, separadas em **35849** para treino e **10000** para validação. Mantivemos os dados de teste ocultos até que finalizamos o nosso modelo, para evitar incluir viés no modelo.

Implementamos primeiramente a função (3). Percebemos que o tempo de execução aumentava drasticamente ao aumentar o número de execuções. O gargalo encontrado foi a implementação que realizava a regressão linear proposta utilizando comandos de repetição, o que acarretou na execução em tempo $O(n^3)$.

Ao alterar o código para utilizar operações vetoriais da biblioteca Numpy[4], que são otimizadas para trabalhar com *arrays*, conseguimos melhorar o tempo de execução drasticamente e iniciamos a avaliação de resultados. Com os resultados iniciais do método base, partimos então para as experimentações.

Para a função em (3), precisamos de valores θ iniciais. Todos foram inicializados com valor 1.

Descida do gradiente em modo *batch*

O intervalo de iterações escolhido foi $[5000,\ 100000]$, com passos de 50000.

Começamos o treinamento com um *learning rate* igual a 10^{-3}. A seguir, temos o gráfico da função de custo pelo número de iterações para o conjunto de validação.

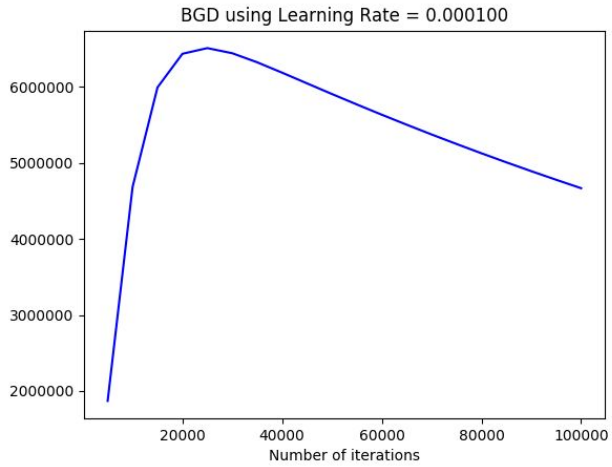

Figura 3 - Gráfico da função de custo (*mean squared error*) pelo número de iterações para um learning rate igual a 10^{-3}.

Podemos notar que o custo aumentou em um intervalo de iterações. Isso indica que, possivelmente, a descida do gradiente está divergindo. Aumentando o valor do *learning rate* para 10^{-1}, obtivemos a curva a seguir para o conjunto de validação.

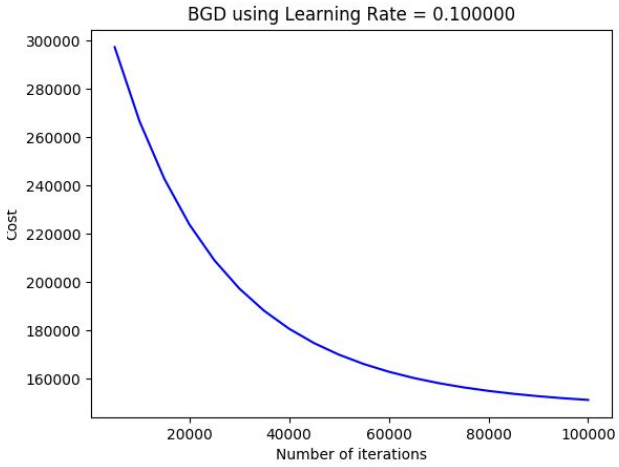

Figura 4 - Gráfico da função de custo (*mean squared error*) pelo número de iterações para um learning rate igual a 10^{-1}.

```
Applying BGD for 100000 iterations...
Coefficients: [ 7566.08078166 45246.68608135 -6523.26825651 -6476.62720318
 -4962.10023794   646.5590896  -2933.31599845]
Cost: 151336.56021574125
Elapsed time: 178.822971 s
```

Figura 5 - Captura de tela e trecho da execução do programa TrainLinearRegression.py.

A curva da figura xxx mostra que, usando o novo *learning rate*, o custo diminuiu ao aumentar a quantidade de iterações; dessa forma, a descida do gradiente convergiu para um mínimo local. Esse *learning rate* foi escolhido para o treinamento e gerou o modelo definitivo. Esse modelo foi usado de entrada para a execução do conjunto de teste, que pode ser vista a seguir.

```
python3 TestLinearRegression.py
Input file name of model: BGDModel
[ 7566.08078166 45246.68608135 -6523.26825651 -6476.62720318
 -4962.10023794   646.5590896  -2933.31599845]
MSE = 395156.2864277317
```

Figura 6 - Captura de tela da execução do programa TestLinearRegression.py. Note que, para executar o programa, é necessário inserir um modelo (no caso, BGDModel) pronto, pois o programa utiliza os dados de teste.

Validação de dados

Como proposto, utilizamos dois métodos para comparar com os dados encontrados com o método implementado. Equação Normal e o Regressor Linear Estocástico do Scikit Learn.

Equação Normal

A equação (5) foi implementada e sua execução pode ser vista a seguir.

```
python3 NormalEquation.py
Applying Normal Equation...
Coefficients: [  9076.72382566  46913.91453206  -8157.39055135  -6534.73174148
 -10980.38424697   4526.49973093   1623.3380558 ]
Elapsed time: 0.118795 s
```

Figura 7 - Captura de tela da execução do programa NormalEquation.py.

Com o modelo gerado pela execução da equação normal, podemos calcular o custo para o conjunto de teste, usando o programa TestLinearRegression.py. Sua execução pode ser vista a seguir.

```
python3 TestLinearRegression.py
Input file name of model: NormalEquationModel
[  9076.72382566  46913.91453206  -8157.39055135  -6534.73174148
 -10980.38424697   4526.49973093   1623.3380558 ]
MSE = 374206.6969988131
```

Figura 8 - Captura de tela da execução do programa TestLinearRegression.py, usando o modelo NormalEquationModel.

Descida do gradiente em modo estocástico

A descida do gradiente em modo estocástico foi implementada utilizando a biblioteca Scikit Learn. Ao executar usando os dados de treino e validação, obtivemos o resultado a seguir.

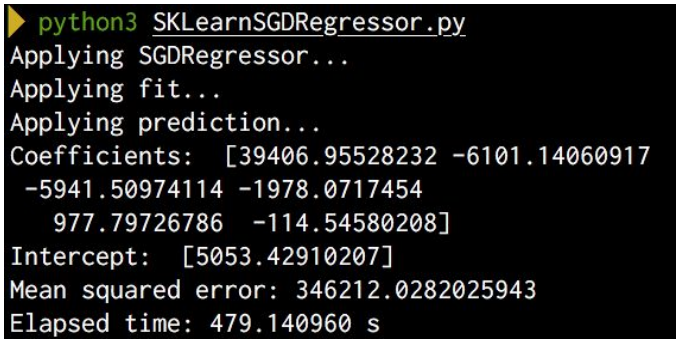

Figura 9 - Captura de tela da execução do programa SKLearnSGDRegressor.py.

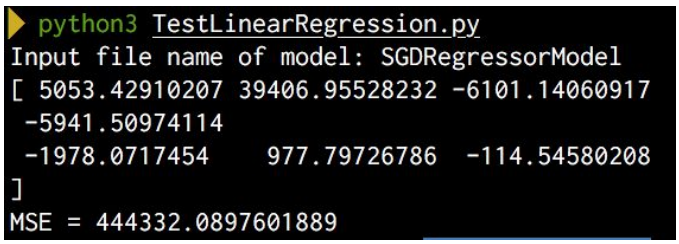

Figura 10 - Captura de tela da execução do programa TestLinearRegression.py.

5. Conclusões e trabalho futuro

Modelo	MSE
Descida gradiente modo batch (BGD)	395157
Equação normal (NE)	374207
Descida gradiente modo estocástico (SGD)	444332

Tabela 1 - Mean squared error alcançado com os dados de teste nos três modelos.

Modelo	Carat	Depth	Table	x	y	z
BGD	7566	45247	-6523	-6477	647	-2933
NE	9077	46914	-8157	-10980	4526	1623
SGD	5053	39407	-6101	-5942	978	-115

Tabela 1 - Parâmetros encontrados para os três modelos.

Tomamos o resultado da técnica da Equação Normal como o método mais próximo dos dados, por encontrar o mínimo global de forma algébrica. Utilizamos o SGDRegressor do Scikit Learn como controle, para verificar se nosso método está no caminho correto.

Comparando os valores apresentados nas tabelas 1 e 2, vemos que nossa implementação Batch chegou mais perto dos valores da Equação Normal do que o método do workbench. Isso pode ser devido ao fato de não termos fornecido iterações o suficiente para o método estocástico, já que este não se aproxima de forma tão direta da convergência como o método Batch. Percebemos uma variação muito grande nos resultados das *features* 5 e 6 nesse método, possivelmente em resposta a *outliers* em uma das últimas observações usadas para o recálculo dos parâmetros.

Comparando o método Batch, também percebemos que a maior diferença dos parâmetros se dá nas *features* 5 e 6.

Durante o treinamento, subimos o número de iteração para até 1000000, porém o gráfico MSE vs. iteração nos mostrou que após 100000 iterações não ocorria melhora significativa no modelo; dessa forma, utilizamos esse valor mais baixo de iterações para nosso modelo final.

Existem alguns tópicos a serem realizados no futuro. O primeiro é a detecção e remoção mais efetiva de outliers. Além disso, a utilização de outras *features* que não levamos em consideração nesse estudo, ou o conjunto de *features* que representasse melhor os dados. Outra coisa que desejávamos fazer, é avaliar o motivo das *features* 5 e 6, respectivamente largura e profundidade da gema, estarem gerando, em ambas as implementações, resultados tão divergentes do encontrado pela Equação Normal.

6. Referências

[1] **Diamonds dataset.** Disponível em: <https://www.kaggle.com/shivam2503/diamonds>. Acesso em: 31 ago 2018.
[2] **Kaggle website.** Disponível em: <https://www.kaggle.com> Acesso em: 31 ago 2018.
[3] **Scikit Learn website.** Disponível em: <http://scikit-learn.org/stable/#> Acesso em 31 ago 2018.
[4] **Numpy website.** Disponível em: <http://www.numpy.org> Acesso em 31 ago 2018.
[5] **Hands-on Machine Learning with Scikit-Learn and TensorFlow - Training Models.** Disponível em: <https://www.oreilly.com/library/view/hands-on-machine-learning/9781491962282/ch04.html> Acesso em 31 ago 2018.
[6] **About Min-Max scaling.** <https://sebastianraschka.com/Articles/2014_about_feature_scaling.html#about-min-max-scaling> Acesso em 2 set 2018.