Relatório 1: Regressão Logística

Diogo Rebelo, pg50327

27 de fevereiro de 2023

1 Introdução

Serve o presente documento para expor a análise efetuada no contexto do estudo da Regressão Logística iniciado na aula teórica, no âmbito da UC de Aprendizagem Profunda. Para além disso, o documento elucida o conjunto de alterações efetuadas a nível da solução, ou seja, alterações no código fornecido pela equipa docente. No código construido, optou-se por utilizar uma metodologia clara que permitisse perceber que funções novas foram construídas, que funções da equipa docente foram alteradas, e quais foram mantidas. Então, as funções inalteradas, não possuem qualquer comentário antes da sua assinatura. As funções da equipa docente que foram alteradas possuem a alteração efetuada (em comentário) antes da sua assinatura. As funções novas possuem também um comentário antes da sua assinatura: # nova função.

A principal ideia foi, então, perceber como funciona o algoritmo de Regressão Logística com e sem Regularização. Para isso, aproveitamos a maioria das funções que já foram fornecidas. Assim sendo, para o teste sem Regularização, é realizada Regressão Logística com estimação de parâmetros pelo Gradiente Descendente. Também se testa uma otimização em relação à descida máxima, com o algoritmo Nelder-Mead, que é um algoritmo de otimização não-linear que não requer o cálculo de derivadas. O algoritmo Nelder-Mead é um método de busca por coordenadas simples que opera por manter um conjunto de pontos que se movem em direção ao mínimo da função objetivo. Já em relação à Regularização, testa-se Regularização L2, com diferentes valores para o parâmetro lambda. O dataset utilizado como objeto de análise foi o *hearts-bin.data*.

Para efetuar os testes, basta usar o formato:

- Sem regularização & Sem Otimização Nelder-Mead:
  `comando: python3 logistic_regression.py <dataset> <noReg> <noOp>;`
- Sem regularização & Com Otimização Nelder-Mead:
  `comando: python3 logistic_regression.py <dataset> <noReg> <op>;`
- Com regularização:
  `comando: python3 logistic_regression.py <dataset> <reg>;`

2 Análise Crítica de Resultados

Em todos os casos, com e sem regularização, os dados sofreram primeiro um processo de Standardização (ajusta a média das variáveis para zero e o desvio padrão para 1), necessário tendo em conta que algoritmos de Gradiente Descendente podem funcionar pior com variáveis com escalas muito diferentes. Utiliza-se este método do Gradiente para estimar os parâmetros da Regressão. Como sabemos, o objetivo é minimizar a função de custo definida para cada caso.

2.1 Sem regularização

2.1.1 Sem otimização Nelder-Mead

- **`test_hearts_noOp(dataset):`** não usa otimização com Nelder-Mead.
  - Lê o dataset fornecido;
  - Efetua a divisão aleatória em dados de treino e teste (utilizou-se 0.3 para teste), usando a nova função solicitada no TPC;
  - Cria o modelo de Regressão Logística com Padronização dos dados de treino;
  - Aplica o Gradiente Descendente (com alpha=0.02 e iters=15000);
  - Efetua as previsões nos dados de teste, usando uma nova função solicitada no TPC;

O *output* apresentado foi:

```
comando: python3 logistic_regression.py hearts-bin.data noReg noOp

Train set has 189 rows and 13 columns
Test set has 81 rows and 13 columns

No Regularization.
No Optimization of GD.
Standardized Data.
Aplying Logistic Regression:
Initial Cost (not optimized): 0.6931471803599453
Running GD:
0.6931471803599453
0.32131140013581544
0.31640027103522705
0.3155908195693179
0.31541300176468784
0.3153689768981798
0.3153573214332958
0.3153540906322721
0.31535316150587195
0.31535288546649176
0.31535280096205753
0.3153527743687996
0.3153527657892939
0.31535276296058645
0.3153527620106516
Final Cost (with GD): 0.31535276168679827
Accuracy Score: 0.8272
```

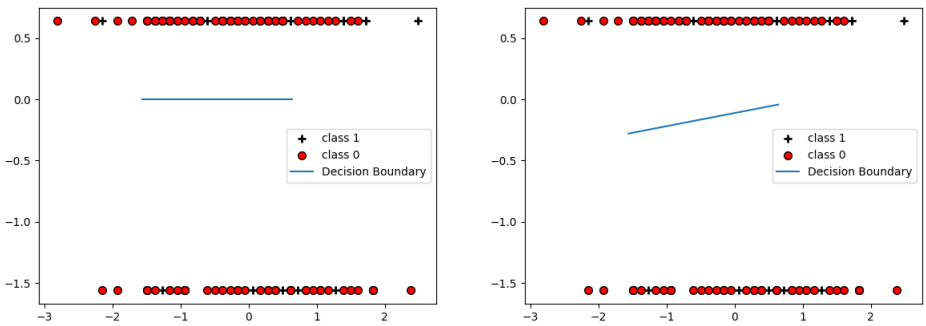

Figura 1: Representação Gráfica dos dados sem regularização e sem otimização (antes e depois do algoritmo).

Como se observa, há uma descida da função de custo, quando se utiliza o algoritmo de gradiente, de quase para metade. O valor do custo é rapidamente determinado, estabilizando ao longo do aumento de cada iteração. Nota-se que há uma descida mais significativa da primeira iteração para a segunda, não acontecendo isso nas restantes iterações.

Em relação ao gráfico, observa-se que a inclinação da reta de ajuste não é muito acentuada.

O valor da *accuracy* é razoavelmente bom, contudo, este valor, após se terem realizados vários testes, difere em todos os casos, devido à aleatoriedade inerente à divisão dos dados em cada teste.

2.1.2 Com otimização Nelder-Mead

- **`test_hearts_op(dataset):`** usa otimização com Nelder-Mead.
  - Lê o dataset fornecido;
  - Efetua a divisão em dados de treino e teste (utilizou-se 0.3 para teste), usando a nova função solicitada no TPC;
  - Cria o modelo de Regressão Logística com Padronização dos dados de treino;
  - Aplica a otimização de descida máxima;
  - Efetua as previsões nos dados de teste, usando uma nova função solicitada no TPC;

O *output* apresentado foi:

```
comando: python3 logistic_regression.py hearts-bin.data noReg op

Train set has 189 rows and 13 columns
Test set has 81 rows and 13 columns

No Regularization.
With Optimization of GD.
Standardized Data.
Aplying Logistic Regression:
Initial Cost (not optimized): 0.693147180359945
Optimization terminated successfully.
       Current function value: 0.366869
       Iterations: 3756
       Function evaluations: 4951
Final Cost (GD optimized): 0.36686928030550653
Accuracy Score: 0.7531
```

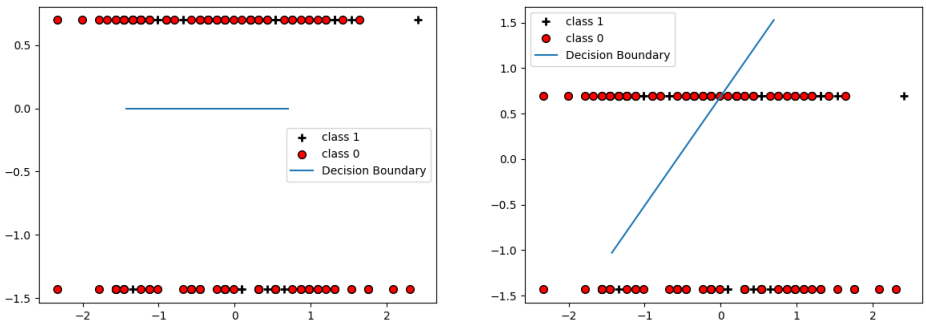

Figura 2: Representação Gráfica dos dados sem regularização e com otimização (antes e depois do algoritmo).

Como se observa, há uma descida da função de custo, quando se utiliza o algoritmo de otimização, para metade também. O número de iterações utilizado na função anterior foi 15000. Verificou-se que um número mais baixo como, 10000, não era suficiente. Observamos, neste caso, que são

apenas necessárias 3756 iterações para encontrar os valores ótimos dos parâmetros e, por isso, um valor substancialmente inferior ao caso que não utiliza otimização na descida máxima.

Em relação ao gráfico, observa-se que a inclinação da reta de ajuste é mais acentuada.

O valor da *accuracy* é razoavelmente bom, contudo, pior em relação ao caso que não utiliza otimização. Uma explicação é o facto de os dados serem totalmente aleatórios, aquando da divisão, o que também justifica a pior performance. A separação entre as classes parece menos bem conseguida.

Em ambas as funções, é apresentada a representação gráfica do modelo, usando a função de *plot*, antes e depois da aplicação do algoritmo, sendo apresentada a *decision boundary*, que separa as duas classes.

2.2 Com Regularização

Para este caso, testamos com vários valores para o parâmetro lambda (0, 0.1, 1, 10, 100).

- **`test_hearts_reg(dataset):`** usa Regularização L2. Socorre-se do BFGS, um algoritmo de otimização de segunda ordem que tenta encontrar o mínimo da função de custo iterativamente, usando informações sobre a curvatura da função para atualizar a direção de busca.
  - Lê o dataset fornecido;
  - Efetua a divisão em dados de treino e teste (utilizou-se 0.3 para teste), usando a nova função solicitada no TPC;
  - Cria o modelo de Regressão Logística com Padronização dos dados de treino;
  - Aplica a regressão usando o algortimo BFGS, para os vários lambdas;
  - Efetua as previsões nos dados de teste, usando uma nova função solicitada no TPC;

O *output* apresentado foi:

```
comando: python3 logistic_regression.py hearts-bin.data Reg

Train set has 216 rows and 13 columns
Test set has 54 rows and 13 columns

With Regularization.
Standardized Data.
Aplying Logistic Regression:
Regularization (lambda=0):
Final Cost : 0.3129255658751516
Accuracy Score: 0.7407
============================================================
Regularization (lambda=0.1):
Final Cost : 0.3125528926156651
Accuracy Score: 0.7407
============================================================
Regularization (lambda=1):
Final Cost : 0.3113360915618457
Accuracy Score: 0.7593
============================================================
Regularization (lambda=10):
Final Cost : 0.3278913460848603
Accuracy Score: 0.7778
============================================================
Regularization (lambda=100):
Final Cost : 0.43420949747454013
Accuracy Score: 0.7407
============================================================
```

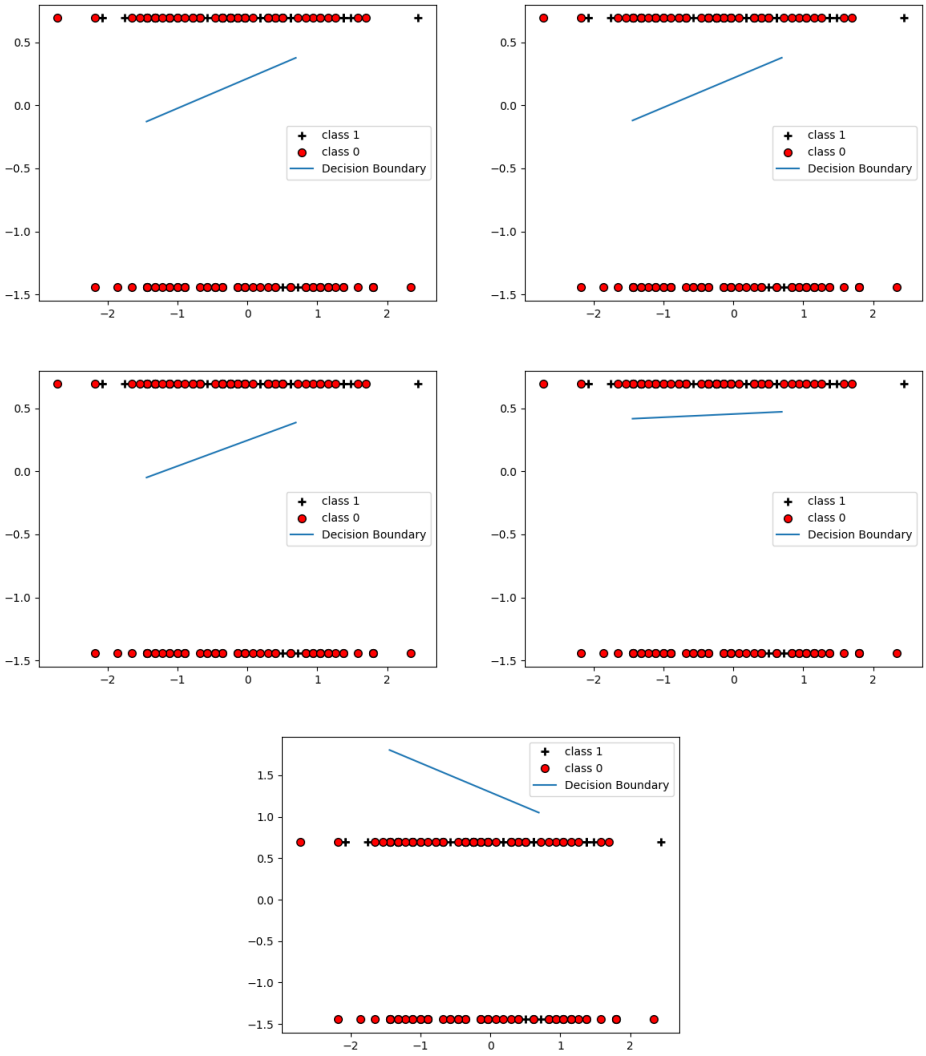

Figura 3: Representação Gráfica dos dados com regularização (para lambda = 0,0.1,10,100).

Como se observa, há uma descida da função de custo, à medida que o valor do lambda aumenta, até certo ponto. Quando o valor é superior a 0.1, a função de custo começa a aumentar, o que se traduz também numa diminuição da performance.

A regularização adiciona um termo de penalidade à função de custo que restringe os valores dos pesos do modelo, ajudando a evitar o overfitting, ou seja, a situação em que o modelo se ajusta muito bem aos dados de treino, mas não generaliza bem para novos dados. Conseguimos observar que, com regularização, a *performance* diminui, mas o ajuste da reta é muito mais preciso, o que permite concluir que, apesar de a *accuracy* ser inferior, o modelo é mais bem conseguido, já que o valor da função de custo também se mostra inferior em relação ao caso de teste sem Regularização (basta olhar para o primeiro *ouput*, onde o *score* é superior e a função de custo é mais elevada).

Em termos gráficos, verifica-se que o ajuste da reta vai melhorando até certo ponto (lambda = 10), a partir de onde vai piorando, como se vê na última figura, onde `lambda = 100`.

Esta função também constrói um gráfico que apresenta a variação do valor de lambda em relação à *performance/accuracy*, o que pode ajudar a perceber o valor ideal deste parâmetro e como este varia:

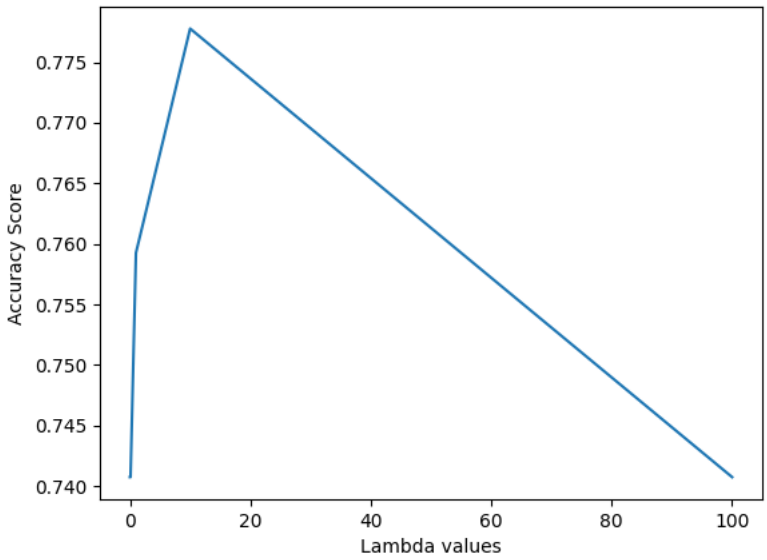

Figura 4: Variação do parâmetro lambda em função da *accuracy*.

Ao observar a figura anterior, comprova-se o que foi concluído no excerto anterior sobre a variação do valor do lambda:

- A *accuracy* aumenta até `lambda = 10`, a partir daí diminui;
- A *cost function* diminui até `lambda = 1`, a partir daí aumenta;

Apesar de no primeiro caso a *accuracy* ser maior do que no segundo, o valor da função de custo é inferior no segundo caso. Então, devo escolher que valor de lambda? Ora, se objetivo é minimizar o custo do modelo, então deve-se escolher o modelo com menor custo, mesmo que a *accuracy* seja ligeiramente menor. Por outro lado, se a prioridade é obter a maior *accuracy* possível, então, deve-se escolher o modelo com a maior *accuracy*, mesmo que o custo seja ligeiramente maior. Note-se que a distribuição e o tipo de dados também têm impacto.

2.2.1 *Tradeoff lambda* vs. *Overfitting*

Se lambda é muito grande, a penalidade é muito forte e os pesos do modelo são fortemente restringidos, levando a um modelo mais simples, mas menos preciso. Por outro lado, se o valor do lambda é muito pequeno, a penalidade é fraca e os pesos do modelo têm pouca restrição, levando a um modelo mais complexo e com mais sobreajuste.

Portanto, ao adicionar regularização, a forma do modelo resultante pode mudar de várias maneiras, dependendo dos valores dos pesos e do parâmetro de regularização, afetando a visualização dos dados, por exemplo, na separação das classes, que pode não ser tão clara quanto antes (depende da distribuição dos dados).

3 Conclusão

Este exercício permitiu compreender de uma forma mais aprofundada a Regressão Logística, contribuindo positivamente para a elucidação dos algoritmos utilizados na estimação de parâmetros e regularização.

Primeiro, observou-se a forma como a regressão é feita, de acordo com o método do gradiente descendente e com e sem otimização Nelder-Mead.

Posteriormente, testou-se a existência de regularização, com diferentes valores de lambda, compreendendo a sua variação com a *performance*. Fez-se também uma análise entre a *accuracy* e a função de custo (também em termos gráficos)

De uma forma geral, o trabalho foi bem conseguido.