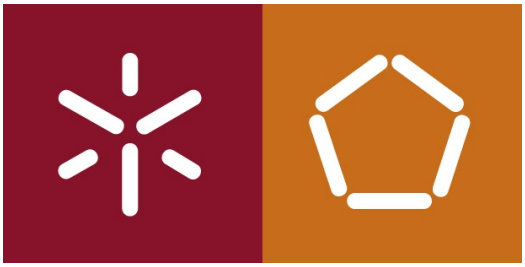

UNIVERSIDADE DO MINHO

MESTRADO EM ENGENHARIA INFORMÁTICA

PERFIL: MACHINE LEARNING:FUNDAMENTOS E APLICAÇÕES

OTIMIZAÇÃO EM MACHINE LEARNING

T1

VERSÃO DUAL DO LOGISTIC CLASSIFIER COM KERNEL

Autores:
Diogo Rodrigues (PG37150)
Luis Bouça (PG38933)
Luís Costa (A74819)
Hugo Gonçalves (PG38928)

2 de Junho de 2019

Conteúdo

1 Introdução

A construção de um modelo de Logistic Classifier na sua versão dual, possibilitando assim a implementação com kernel.

O objetivo com a construção do modelo, é possibilitar uma comparação com a sua versão primal, e verificar o impacto que o uso do kernel tem no modelo construido.

O desenvolvimento deste modelo, sem recurso a bibliotecas externas, vai permitir dar uma perceção do seu comportamento em baixo nível.

Os resultados finais, vão ser apresentados usando o custo devolvido durante o treino do modelo, assim como as previsões obtidas durante o teste ao mesmo. Por fim uma matriz de confusão será também apresentada, possibilitando assim uma analise às classes com maior dificuldade na sua previsão.

2 Base de Dados

O treino e validação do modelo, vai ser feito com recurso a três bases de dados distintas.

Numa fase inicial com a base de dados **AND**, **XOR**, e posteriormente com a base de dados **line600**, que é bastante mais complexa.

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

AND

Figura 1: Tabela AND

A figura 1, é referente ao dataset **AND**, e tenta representar os pontos identificados num operador do tipo **AND**
A segunda tabela representada pela imagem 2, é referente a um operador do tipo **XOR**.

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

XOR

Figura 2: Tabela XOR

Comparativamente ao operador **AND**, o **XOR** apresenta uma maior dificuldade na classificação por parte dos perceptrons com apenas uma camada.

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1,
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1,
0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0,
```

Figura 3: Tabela line600

O terceiro e último dataset, é bastante mais complexo comparativamente aos restantes. O objetivo deste dataset é tentar identificar dígitos em imagens constituídas por um tamanho de 8x8.

O dataset é assim constituído por 600 imagens, que representam as linhas do dataset, e por 64 features, que representa a posição de cada uma das imagens de 8x8.

A classe binária, vai indicar apenas a ausência ou não de um digito. Um exemplo das linhas existentes neste dataset, pode ser observado na figura 3.

3 Logistic Classifier

Relativamente à versão primal, a versão dual do **Logistic Classifier**, distingue-se no modo de como faz a sua previsão, usando para isso um vetor, que vai apontar para a direção de maior descida, possibilitando assim que o algoritmo consiga convergir mais facilmente.

O vetor **alpha**, de tamanho **N**(nº de linhas), distingue-se do vetor de pesos usado na versão primal, de tamanho **I**(nº de colunas).

É assim fácil perceber, que a versão dual vai caracterizar a arquitetura com base nas linhas, enquanto que a versão primal caracteriza a arquitetura com base nas colunas mais o **bias** (I+1).

3.1 Função Estocástica

O treino do modelo, vai ser feito através de um método estocástico. Este método implica que a convergência do algoritmo, é feita com a escolha aleatória de pontos na base de dados, que vão permitir uma atualização constante do gradiente e do custo associado em cada uma das iterações do processo estocástico.

```
while not converge do
    pick up (xm, ym) FROM D;
    g = calculate gradient using (xm, ym, g);
    cost = calculate cost using (X, Y, g)
    if iteration >= max_iterations then
        stop ;
    else
end
```

Figura 4: Algoritmo Estocástico

A figura 4, apresenta o algoritmo estocástico, que foi implementado no desenvolvimento deste modelo.

O método implementado possui dois critérios de paragem. O primeiro critério, é baseado no custo, e permite que seja feita uma execução recorrente do método estocástico, até que o custo consiga atingir um threshold definido previamente.

O segundo critério de paragem, tem em conta um limite máximo de iterações a serem executadas pelo método estocástico. O objetivo neste segundo critério, é impedir uma paragem demasiado tardia, ou uma execução indefinida e continua do algoritmo, ou seja, nos casos do custo não baixar o suficiente para convergir.

```
def stocastic(X, Y, N, max_it, lr):

    alpha = np.zeros(N)

    error=[];error.append(cost(X,Y,N, alpha))

    epsi=0
    it=0
    while(error[-1]>epsi):

        idx = np.random.randint(0, N)

        Xm = X[idx, :]
        Ym = Y[idx]


        alpha = gradient(X, Xm, Y, Ym, N, alpha, lr)
        error.append(cost(X,Y,N, alpha))

        print('iteration %d, cost=%.2f    \r' %(it,error[-1]),end='')

        #Second second codition to stop
        it=it+1
        if(it>max_it): break
    return alpha, error
```

Figura 5: Implementação Função Estocástica

A figura 5, mostra um excerto do código implementado, referente ao método estocástico.

Inicialmente é instanciado o vetor **alpha**, sendo usado posteriormente para calcular o erro inicial.

Conforme referido anteriormente, em cada iteração do método estocástico, é feita uma atualização ao vetor **alpha**, usando para isso o gradiente, e posteriormente uma atualização ao custo, verificando assim se o mesmo está a convergir ou não em cada atualização do vetor **alpha**.

3.2 Funçáo Predictor

$$sgn(\sum_{n=1}^{N} \alpha^n * \left[((x^n)^T * x^m) + 1\right]^2)$$

Figura 6: Arquitetura Dual

A figura 6, representa uma arquitetura **Dual**, com **Kernel** do predictor.

A nova arquitetura implica que seja percorrida toda a base de dados, sendo realizada a multiplicação do vetor linha, escolhido previamente pelo método estocástico, pela coluna **X[n]** transposto, referente à linha da iteração atual. Ao vetor resultante desta operação, deve ser adicionado o **bias**.

A implementação do **Kernel**, implica obter o quadrado deste mesmo vetor.

O vetor resultante desta operação, vai ser multiplicado pelo **alpha[n]**, que é referente à posição do vetor **alpha** na iteração atual.

Após serem percorridas todas as linhas da base de dados, o resultado do sumátorio vai ser aplicado na função **sigmoid** que varia entre [0,1].

```
def predictor(X, Xm, N, alpha):
    s = 0
    for n in range(N):
        s = s + ((np.dot(Xm, X[n]) +1)**2)*alpha[n]
    sigma = sigmoid(s)
    return sigma


def sigmoid(s):
    large=30
    if s<-large: s=-large
    if s>large: s=large
    return (1 / (1 + np.exp(-s)))
```

Figura 7: Implementação Função Predictor

A figura 7, mostra um excerto do código, referente à implementação da função predictor na sua versão **Dual**, assim como a função **sigmoid**, que irá calcular o resultado devolvido pelo predictor.

3.3 Função Custo

O objetivo da função custo, é avaliar o comportamento do modelo em relação ao erro durante o processo de treino **in-sample error**, ou durante o processo de teste **out-sample error**.

O resultado da função custo, irá determinar se o modelo está a convergir ou não, definindo assim se a procura estocástica pode terminar.

$$-\frac{1}{N}\sum_{n=1}^{N} y^n \ln(\widehat{y}^n) + (1-y^n)\ln(1-\widehat{y}^n)$$

Figura 8: Função de Custo

Conforme é possível observar na figura 8, a função de custo implementada é baseada na entropia cruzada. Uma vez que é uma função não negativa, e que se aproxima de 0 à medida que o valor de previsão é próximo do valor real, tornam esta função possível de ser utilizada como função de custo.

Esta função vai permitir anular o primeiro termo no caso do **y[n]** ser 0, sendo calculado apenas o segundo termo, assim como anular o segundo termo no caso de **y[n]** ser 1, sendo calculado apenas o primeiro termo.

```
def cost(X,Y,N, alpha):
    En=0;epsi=1.e-12
    for n in range(N):
        y_pred =predictor(X, X[n], N, alpha)
        if y_pred<epsi: y_pred=epsi
        if y_pred>1-epsi:y_pred=1-epsi
        En=En+Y[n]*np.log(y_pred)+(1-Y[n])*np.log(1-y_pred)
    En=-En/N
    return En
```

Figura 9: Implementação Função de Custo

Na figura 9, está representado o excerto de código referente à implementação da função de custo.

A função irá implementar um **epsi**, funcionando como um threshold, que irá adaptar o valor previsto, de modo a que este seja possível de aplicar no logaritmo.

3.4 Função Gradiente

$$\sum_{n=1}^{N} (y^m - yp^m) * \left[((x^n)^T * x^m) + 1 \right]^2$$

Figura 10: Função Gradiente

A função gradiente, tem como objetivo atualizar as posições do vetor **alpha**, conforme é possível observar na figura 10, ao longo das **N** linhas existentes, que compõem o dataset.

Semelhante à função predictor, a função gradiente, vai implementar o **kernel**, como produto da multiplicação entre **X[n]** transposto, referente a linha atual da base de dados, com a linha escolhida pela função estocástica.

O vetor quadrático obtido, vai ser multiplicado pelo valor resultante da subtração entre a classe real, e classe prevista.

```
def gradient(X, Xm, Y, Ym, N, alpha, lr):

    for n in range(N):
        value = (Ym - predictor(X, Xm, N, alpha)) * ((np.dot(Xm, X[n]) +1)**2)
        alpha[n] = alpha[n] + (lr*value)
    return alpha
```

Figura 11: Implementação da Função Gradiente

A figura 11, apresenta o código implementado para a função gradiente, que recebe como parâmetros o **alpha**, **lr**(learning rate), assim como as linhas escolhidas aleatoriamente pela função estocástica.

A taxa de aprendizagem, vai permitir que os valores de **alpha**, sejam atualizados de maneira gradual, consoante o valor definido. No caso da taxa de aprendizagem ser 1, significa que o valor a ser atualizado, vai ser o valor total calculado para o gradiente naquela posição do **alpha**.

4 Resultados

4.1 Tabela AND

4.1.1 1º Teste

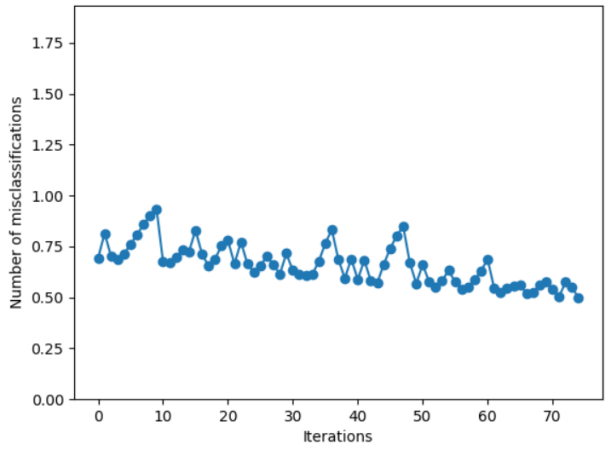

Figura 12: Teste 1 - Gráfico do Erro

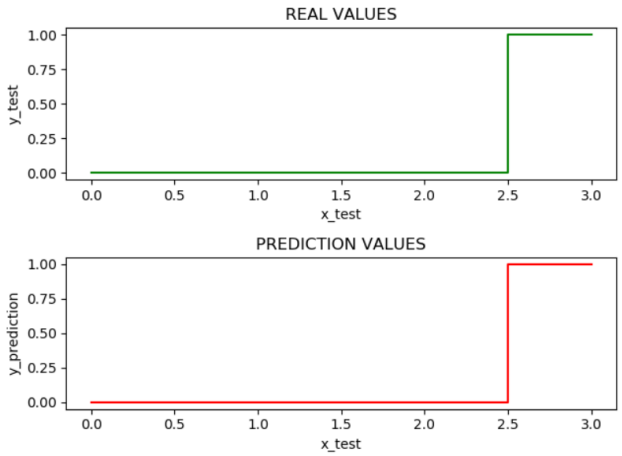

Figura 13: Teste 1 - Gráfico das Previsões

Os resultados do primeiro teste realizado sobre o dataset **AND**, pode ser observado nas figuras 12 e 13, que correspondem ao gráfico que apresenta a evolução do erro, e uma comparação entre o valor real, e o valor previsto.

É possível verificar na figura 12, que a convergência do modelo ocorreu ao fim de 70 iterações, atingindo um threnshold no custo de 0.5, definido previamente na função estocástica.

A figura 13, mostra a relação entre o valor previsto, e o valor real, no qual é possivel verificar que não existe diferenças entre ambos, significando assim que o modelo consegiu obter uma accuracy de 100% no primeiro teste, utilizando para isso uma taxa de aprendizagem de 0.03.

```
in-samples error=0.50
out-samples error=0.50

Accuracy: 1.00
Precision: 1.00
Recall: 1.00
####Confusion Matrix####
[[1 0]
 [0 3]]
########################
```

Figura 14: Teste 1 - Matriz de Confusão

Os resultados podem ser confirmados com a matriz de confusão apresentada na figura 14, em que mostram a inexistência de falsos positivos ou falsos negativos.

4.1.2 2º Teste

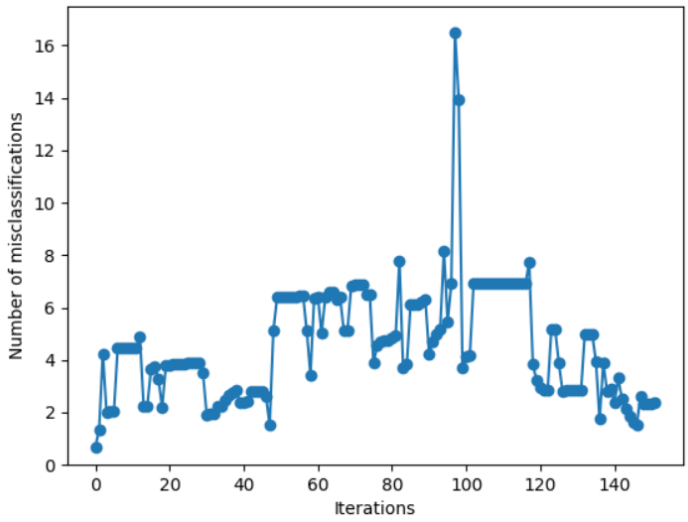

Figura 15: Teste 2 - Gráfico do Erro

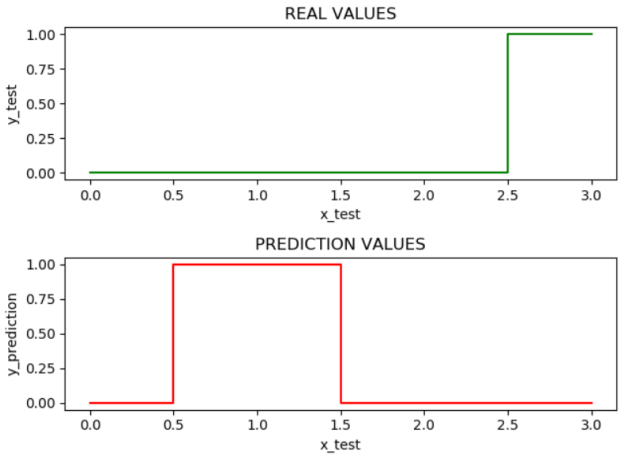

Figura 16: Teste 2 - Gráfico das Previsões

```
in-samples error=2.35
out-samples error=2.35

Accuracy: 0.50
Precision: 0.00
Recall: 0.00
####Confusion Matrix####
[[0 1]
 [1 2]]
########################
```

Figura 17: Teste 2 - Matriz de Confusão

O segundo teste, realizado sem taxa de aprendizagem, obteve um erro ligeiramente superior, como é possível verificar na figura 15.

O aumento significativo do erro, refletiu-se nas previsões obtidas, e respetiva matriz de confusão, como mostram as figuras 16 e 17.

4.1.3 3º Teste

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

Figura 18: Nova Tabela AND

O terceiro teste, implicou a adição de alguns pontos no dataset **AND**, conforme é possível observar na figura 18, e validar o modelo com estas novas linhas.

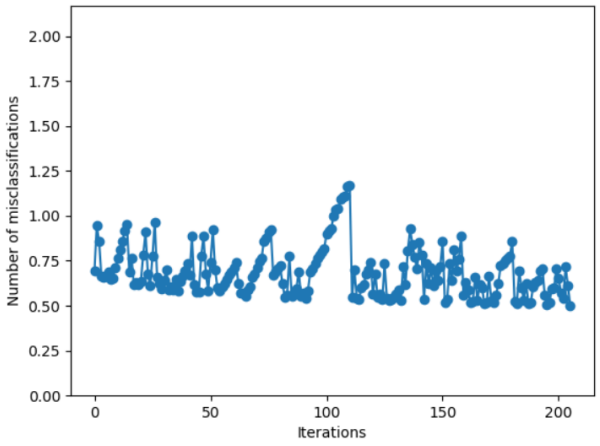

Figura 19: Teste 3 - Gráfico do Erro

A figura 20, mostra a evolução do erro após 205 iterações, no qual o modelo conseguiu convergir para um threshold de erro de 0.5, usando uma taxa de aprendizagem de 0.02.

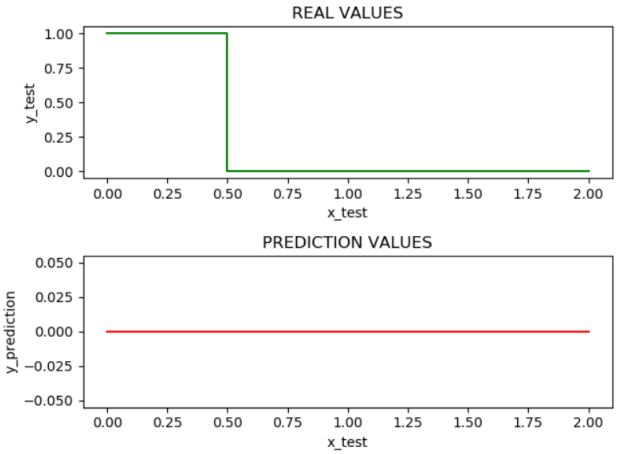

Figura 20: Teste 3 - Gráfico das Previsões

```
in-samples error=0.50
out-samples error=0.64

Accuracy: 0.67
Precision: 0.00
Recall: 0.00
####Confusion Matrix####
[[0 1]
 [0 2]]
########################
```

Figura 21: Teste 3 - Matriz de Confusão

A figura 20 e 21, mostra o resultado das previsões com o **Y_test**, referente às novas linhas adicionadas no dataset. É possível observar que existe uma maior dificuldade de prever a primeira linha de teste **[1 0 1]**.

4.2 Tabela XOR

4.2.1 1º Teste

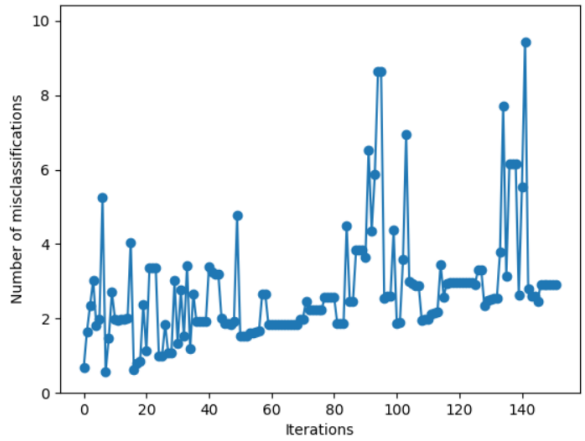

Figura 22: Teste 1 - Gráfico do Erro

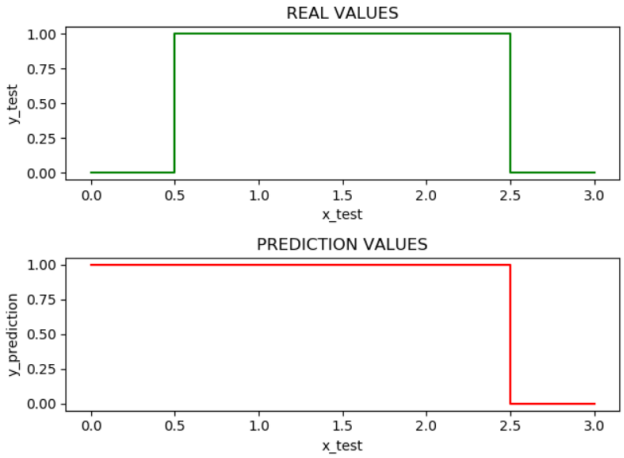

Figura 23: Teste 1 - Gráfico das Previsões

O primeiro teste sobre a tabela **XOR**, mostra que o modelo não conseguiu convergir, após as 150 iterações limite de execução para o método estocástico.

```
in-samples error=2.91
out-samples error=2.91

Accuracy: 0.75
Precision: 1.00
Recall: 0.67
####Confusion Matrix####
[[2 0]
 [1 1]]
########################
```

Figura 24: Teste 1 - Matriz de Confusão

A figura 24, mostra o valor exato do erro, e a matriz de confusão.

4.2.2 2º Teste

Um segundo teste foi realizado, na tentativa de descer o erro obtido pelo modelo, sendo alterada a taxa de aprendizagem para 0.4.

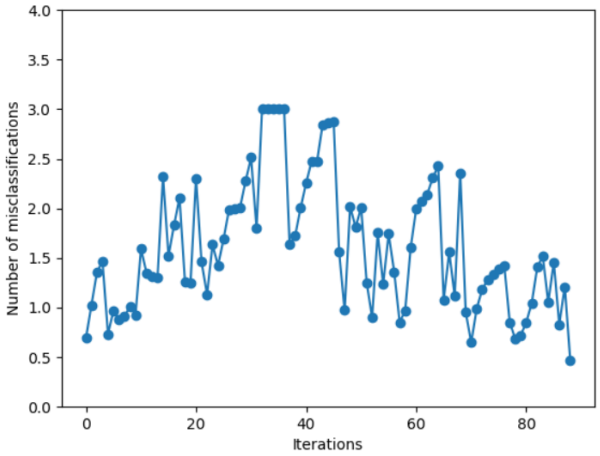

Figura 25: Teste 2 - Gráfico do Erro

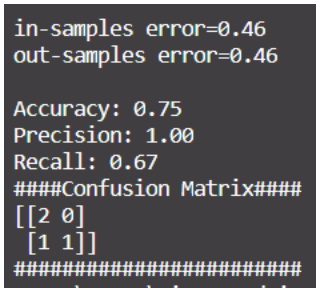

Figura 26: Teste 2 - Matriz de Confusão

Os resultados das figuras 25 e 26, mostram uma redução significativa do erro, apesar do mesmo, apresentar a mesma accuracy, significando assim que estamos na presença de um modelo mais preciso e robusto, quando comparado ao modelo do primeiro teste.

4.3 Tabela line600

4.3.1 1º Teste

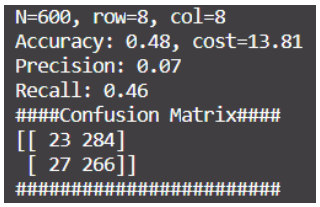

Figura 27: Teste 1 - Matriz de Confusão

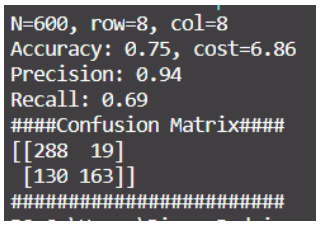

Figura 28: Teste 2(KERNEL) - Matriz de Confusão

As figuras 27 e 28, mostram as vantagens no uso do **Kernel** juntamente com a versão dual do classificador logístico. É possível observar na figura 28, uma accuracy relativamente superior quando comparado a um modelo sem o uso do **Kernel**.

4.3.2 2º Teste

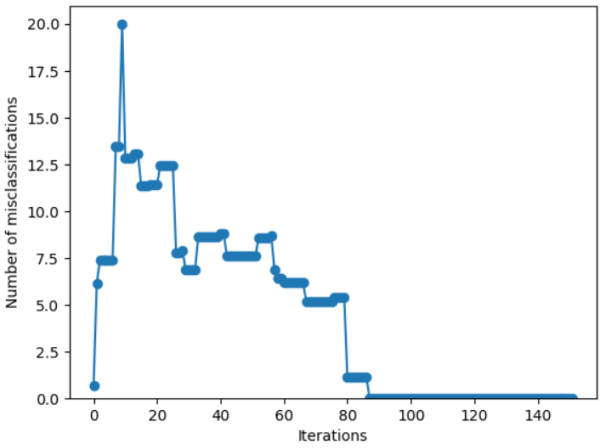

Figura 29: Teste 2 - Gráfico do Erro

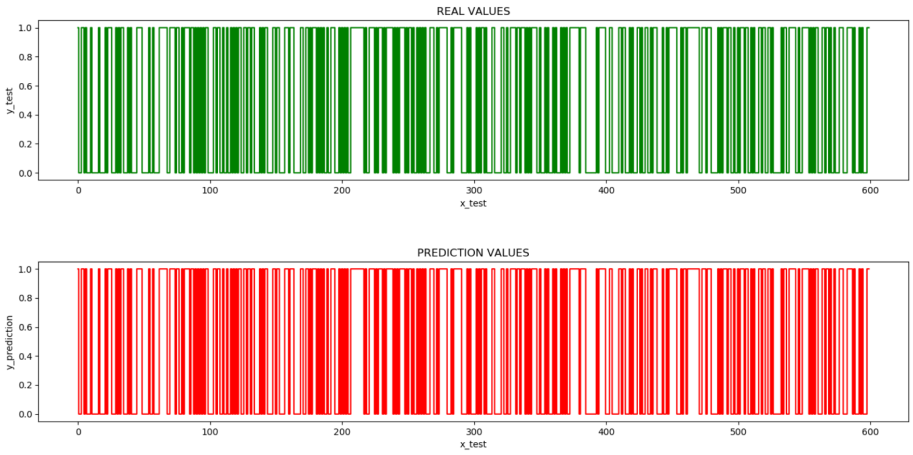

Figura 30: Teste 2 - Gráfico das Previsões

Os resultados do 2º teste, são obtidos após a alteração da taxa de aprendizagem para 0.5.

Observando a figura 29, é possível verificar um erro próximo de 0, indicando uma convergência total do modelo, que se pode confirmar fazendo uma comparação entre os valores reais e os valores previstos na figura 30.

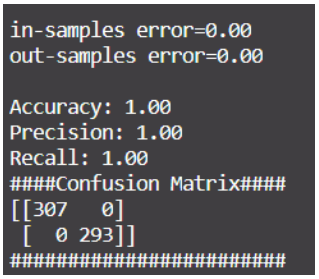

Figura 31: Teste 2 - Matriz de Confusão

A figura 31, denuncia uma possível situação de **overfitting** do modelo, denunciada pelo figura 29. Esta situação indica-nos que estamos na presença de um modelo demasiado ajustado ao conjunto de treino, e que poderá estar pouco preparado quando submetido a novos casos de teste. Existe assim a necessidade de parar o treino previamente, reduzindo o número de iterações ou aumentando o threshold do erro.

5 Conclusão

Este projeto teve com principal desafio a construção do algoritmo estocástico, e do predictor na sua versão dual, e a forma de como a atualização do vetor **alpha** é feita.

Durante a realização do projeto, os resultados mais interessantes, foi observar a diferença que o **Kernel** pode fazer no modelo, assim como a modificação na taxa de aprendizagem, fazendo com que o algoritmo consiga ou não convergir mais facilmente.

Os testes realizados, deram uma noção de que na construção do modelo, é necessário realizar vários testes, alterando parâmetros como o **epsi**, número de iterações ou taxa de aprendizagem, de modo a que o modelo fique preparado para o problema que estamos a tratar.

Futuramente seria interessante implementar mini-batch, fazendo um termo de comparação com o método estocástico, e finalmente aplicar **Softmax**, e verificar o comportamento do modelo na tentativa de solucionar problemas mais complexos.