Universidade do Minho
Escola de Engenharia

Optimização em Machine Learning
Relatório do Projeto Prático

Rui Costa A79947

Julho 2020

Resumo

Quando falamos em *Machine Learning*, o que nos salta à cabeça é aprendizagem por parte do sistema informático em utilização. No fundo a aprendizagem, no geral, consiste em pegar em exemplos de acontecimentos e, com base nestes mesmos prever, com um conjunto de dados de entrada semelhantes, o resultado correto. O objectivo deste trabalho para a unidade curricular de Otimização em Machine Learning foi a implementação de um *Logistic Classifier* para três tipos de classificadores: Primal, Dual e Dual com Kernels. Este problema é fundamental pelo que aborda uma das principais vertentes do *Machine Learning* sendo que é muito usado no ramo das Ciências Sociais, Medicina, entre muitos outros.

Conteúdo

1 Introdução

Um *Logistic Classifier* é usado para determinar a probabilidade de uma certa classe ou evento existirem, como por exemplo: passar ou falhar, ganhar ou perder, saudável ou doente, entre outros. Isto pode também ser estendido para que o sistema possa prever a probabilidade de numa imagem existir um gato ou outra entidade, sendo que a cada objecto a ser previsto é-lhe atribuída uma probabilidade entre 0 e 1 de existir.

No âmbito deste projecto foi proposto a implementação de classificadores para prever valores de vários *datasets* sendo estes linearmente ou não linearmente separáveis. Para isto foi implementado um classificador Primal, Dual e uma versão do último que recorre ao uso de k*kernels* de modo a conseguir separar *datasets* não linearmente separáveis.

Tendo isto em conta, foi feita uma posterior análise de resultados da aplicação de cada um destes classificadores a 3 dos *datasets* sendo que um deles é linearmente separável e os outros dois não, conseguindo assim comparar as diferenças dos classificadores binários do classificador multi-classe.

Seguimos agora para a explicação de cada um dos classificadores.

2 Noções principais

Uma vez que este trabalho recorre a um algoritmo de aprendizagem supervisionada, vamos ter um *training set* sendo que este possui o valor tanto das variáveis independentes como das dependentes. Assim, de modo a obtermos previsões corretas, o modelo é treinado de forma a obter uma função que tenta construir a relação entre as variáveis independentes e a variável dependente do *training set*. Para um novo conjunto de valores das variáveis independentes (*test set*) o modelo infere o valor da variável dependente.

Neste projecto foi usado um **Classificador binário** em que a variável dependente é do tipo Sim/Não. Estes classificadores podem ser lineares ou não lineares sendo que os primeiros podem ser separados por uma reta e os segundos por uma curva. Temos também os classificadores multi-classe em que a variável dependente pode assumir n valores.

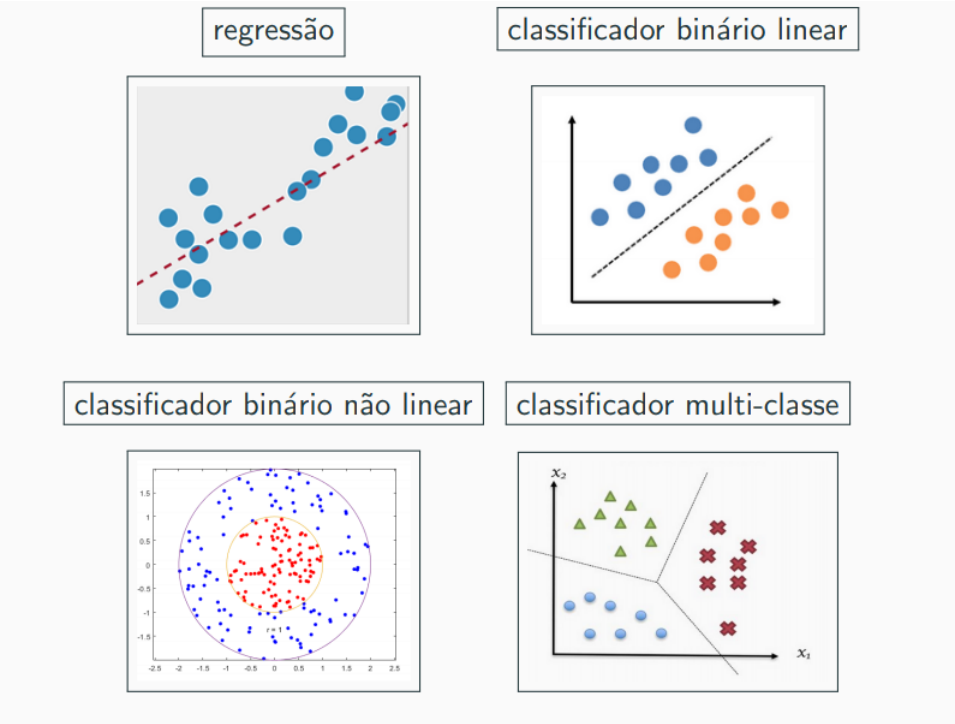

Figura 1: Diferentes tipos de classificadores

2.1 Separabilidade Linear

Para conseguirmos aplicar métodos de separabilidade linear temos o seguinte conjunto de atributos:

- espaço dos atributos (*feature space*): $\mathcal{A} = \mathbb{R}^I$
- espaço das classes: $\mathcal{C} = \{-1, +1\}$
- evento ou ocorrência: $\rho^n = (x^n, y^n) \in \mathbb{R}^I \times \{-1, +1\}$
  - $x^n = (x_1^n, \ldots, x_I^n)$: atributos (*features*) do evento n
  - y^n: rótulo ou etiqueta (*label*) do evento n
- base de dados (*dataset*): $D = (x^n, y^n)_{n=1}^N$

Uma base de dados diz-se linearmente separável se existir um valor pertencente ao *feature set* tal que o produto desse valor com um valor pertencente ao *feature set* satisfaz, para qualquer valor da base de dados as seguintes condições:

- $y^n = +1, p(x^n; \widetilde{w}) > 0;$
- $y^n = -1, p(x^n; \widetilde{w}) < 0.$

ou seja,

$p(x^n; \widetilde{w})y^n > 0, n = 1, ..., N$

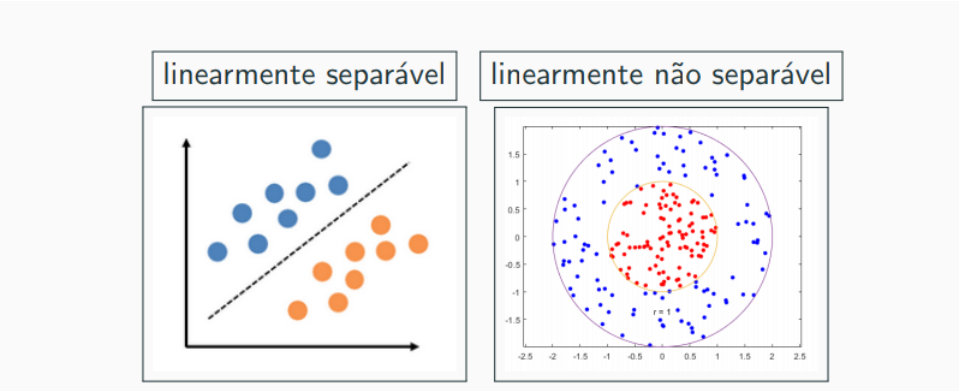

Figura 2: Datasets linear e nao linearmente separáveis

2.2 Hiperplano

Seja w um conjunto de valores pertencentes ao *feature space* e b um número real. Chama-se hiperplano no *feature space* ao conjunto de pontos:

$$\mathcal{H}_{w,b} = \{(x_1, \dots, x_l) \in \mathbb{R}^l : \underbrace{w_1 x_1 + \cdots + w_l x_l + b}_{p(x;w,b)} = 0\}.$$

Na definição anterior, w é um vetor normal a H em que a definição do hiperplano generaliza a definição do plano:

- $\mathbf{l = 1}$: o hiperplano é um ponto;
- $\mathbf{l = 2}$: o hiperplano é uma reta;
- $\mathbf{l = 3}$: o hiperplano é um plano

Notação:

- versão afim — w, b
  - variável: $x = (x_1, \ldots, x_I) \in \mathbb{R}^I$
  - parâmetros: $w = (w_1, \ldots, w_I) \in \mathbb{R}^I, b \in \mathbb{R}$
  - hiperplano: $p(x; w, b) = 0$, com

$$p(x; w, b) = w_1 x_1 + \cdots + w_I x_I + b$$

- versão vetorial — $\tilde{w}$
  - variável: $\tilde{x} = (1, x_1, \ldots, x_I) \in \mathbb{R}^{I+1}$
  - parâmetros: $\tilde{w} = (w_0, w_1, \ldots, w_I) \in \mathbb{R}^{I+1}$
  - hiperplano: $p(x; \tilde{w}) = 0$, com

$$\begin{aligned} p(x; \tilde{w}) &= w_0 \times 1 + w_1 x_1 + \cdots + w_I x_I \\ &= \tilde{w} \cdot \tilde{x} \end{aligned}$$

3 Dados

De modo a modelar uma Machine Learning temos de ter em conta o tipo de dados que vamos interpretar em que estes dados seguem a seguinte notação: e = (x,y) em que e é um evento, x são os atributos ou input e o y são os labels ou outputs. Uma base de dados é uma sequência de dados com o N elementos com o formato anteriormente definida. Para classificar os dados, a Machine Learning vai agrupa-los consoante a sua classe, tendo em conta o valor dos seus outputs.

Para este trabalho foram escolhidos três bases de dados:

- **Quadrados Círculos**: Dataset Linearmente separável
- **Line 600**: Dataset não Linearmente separável
- **Rectangle 600**: Dataset não Linearmente separável

Estes três conjuntos de dados consistem em um conjunto de classes em que o objectivo é que o classificador as identifique correctamente. Foram usados datasets tanto linearmente como não linearmente separáveis de modo a conseguir obter resultados diferentes e identificar vantagens e desvantagens entre as diferentes arquitecturas usadas visto que apenas com o uso de kernels é possível a separação total dos dois últimos conjuntos de dados.

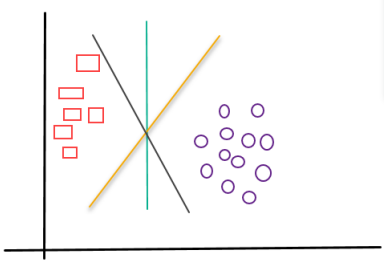

Figura 3: Dataset Quadrados Círculos

Como podemos ver acima, este *dataset*, contrariamente aos outros, é linearmente separável o que possibilita e facilita a convergência para as arquiteturas primal e dual do classificador.

Numa perspectiva de interpretação dos dados, todos estes datasets se baseiam em valores entre -1 e 1, em que para que fossem correctamente interpretados pelo algoritmo foi implementada uma função de normalização de valores que os restringe entre 0 e 1, pelo que todos os valores -1 passam a ser 0 permitindo assim uma correta leitura da parte do classificador.

4 Arquitetura

4.1 Versão Primal

Neste trabalho, para atingir o objectivo pretendido, usamos o *Logistic Classifier*. Este é um modelo matemático inspirado no comportamento dos neurónios sendo que cada neurónio recebe um conjunto de sinais de entrada e emite um sinal de saída. os sinais de entrada são pesados e combinados de modo a produzir um sinal de saída. Este sinal de saída depende também do *bias* que é o nível de activação existindo também uma função de activação sendo o sinal de saída uma probabilidade entre 0 e 1 sendo que se p = 0 a o valor de saída é -1 e se p = 1 o valor de saída é +1.

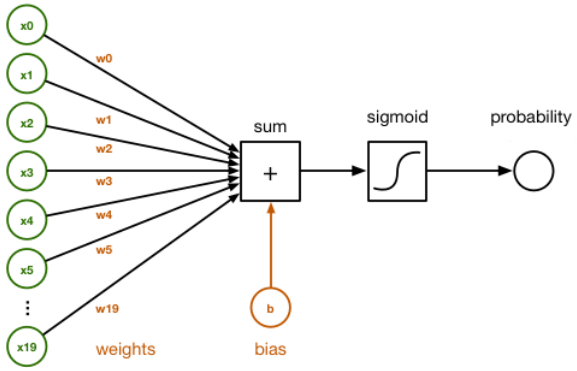

Figura 4: Esquema do Classificador

Para conseguirmos criar um Perceptron é necessário um conjunto fundamental de elementos:

- Base de dados;
- Arquitetura da machine learning;
- Função sinal;
- Confrontação do valor real y^n do evento n e o seu valor inferido;
- Erro relativo ao evento n;
- Objetivo: minimizar a função de custo tal que o erro é 0 para todos os eventos da base de dados, o que significa que todos os eventos são bem classificados. Quantos mais eventos mal classificados, maior será o erro.

Com isto, para obter previsões corretas precisamos de definir uma regra de aprendizagem de modo a que o sistema possa aprender.

Para determinar uma lista de previsões é construída uma sequência de previsões tal que o dataset converge, sendo que quanto mais rápida a convergencia, melhor, sendo que numa situação ideal converge num número finito de iterações. Dada a lista das previsões, de modo a calcular a seguinte é aplicada a função signal ao produto da previsão com o correspondente dado de entrada. Tudo isto é feito através de comparações entre o valor de saída real e o previsto.

Se o resultado da previsão for positivo, não é necessário actualizar os pesos, caso contrário é efectivamente necessário actualizá-los.

Seguindo todas as indicações acima descritas obtemos o seguinte algoritmo:

$$\widehat{y} = \sigma(\widetilde{w}^T x) = \sigma(\sum_{i=0}^{I} wi.xi)$$
$$\widetilde{w} \in \mathbb{R}^{I+1}; \widetilde{w} = \lambda 1.\widetilde{x}1 + \lambda 2.\widetilde{x}2 + ... + \lambda N.\widetilde{x}N$$

Figura 5: Algoritmo da versão Primal

4.2 Versão Dual

No que toca a resultados, tanto na versão primal ou dual do Perceptron, estes são iguais. No entanto, o algoritmo dual apresenta algumas alterações que mudam o modo de como os dados são processados. Assim, enquanto que no primal o calculo era feito com base em $\widetilde{w}$, no dual vamos ter em conta λ.

Tendo em conta todas estas alterações, ficamos então com o seguinte algoritmo:

$$\widehat{y}(x;\lambda) = \sigma(\widetilde{w}^T.x)$$
$$\widehat{y}(x;\lambda) = \sigma(\left\{\sum_{n=1}^{N} \lambda n.xn\right\}^T .x)$$
$$\widehat{y}(x;\lambda) = \sigma(\sum_{n=1}^{N} \lambda n.(x^n)^T.x)$$

Figura 6: Algoritmo da versão Dual

4.3 Versão Kernel

Como referido anteriormente, a capacidade dos das duas versões anteriormente descritas aplica-se apenas a bases de dados linearmente separáveis sendo que no mundo real os problemas que satisfazem esta restrição são muito poucos.

Assim, temos o objetivo de tratar bases de dados que nã são linearmente separáveis atravéz do modelo linear do perceptron sendo para isto aumentado o espaço dos atributos.

O objetivo é o algoritmo trabalhar numa nova base de dados que transforma todos os atributos da antiga base de dados (D) na nova D' da mesma forma (*feature map*). Esta nova base de dados é então definida por:

$$D' = (z^n, y^n)_{n=1}^N,\ z^n = \Phi(x^n) \in \mathbb{R}^J$$

Mas evitando o cálculo explícito de $\Phi(x^n)$ tendo como objetivo remover do algoritmo a dependência explícita de $\widetilde{z}$ via $\widetilde{\Phi}$.

Como estamos a trabalhar num espaço aumentado, computacionalmente pode ser mais custoso visto que precisamos de calcular o seu produto interno. Com isto recorremos ao **Kernel Trick** que calcula o produto interno no espaço transformado (J) recorrendo apenas ao espaço original(I).

tendo tudo isto em conta obtemos o seguinte algoritmo:

$$\widehat{y}(x;\lambda) = \sigma(\sum_{n=1}^{N} \lambda n.(x^n)^T.x)$$

$$\widehat{y}(x;\lambda) = \sigma(\sum_{n=1}^{N} \lambda.[\widetilde{x}^n.\widetilde{x}]^P); (\widetilde{x}^n)^T.\widetilde{x} = \widetilde{x}^n.\widetilde{x}$$

Figura 7: Algoritmo da versão Kernel

4.4 Activatoin Function

De modo a definir o que é positivo ou não, ou seja, classificar entre dois valores, definimos a função de ativação. Esta determina se um neurónio deve ou não ser ativado consoante o input do neurónio à previsão do modelo. Estas ajudam a normalizar o output de cada neurónio que, no nosso caso foi entre 0 e 1. Para isto utilizamos a função *sigmoid* que para a nossa implementação trouxe muitas vantagens como a existência de um gradiente contínuo prevenindo saltos nos valores, normalização dos valores de output e previsões claras em que se a previsão estiver perto do 1 ou 0 é uma boa previsão. Esta função tem então o seguinte aspecto:

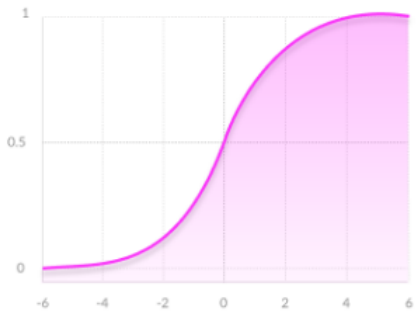

Figura 8: Gráfico da função Sigmoid

Para obter um resultado correto foi usada a seguinte fórmula: $\frac{1}{1+exp(-s)}$ em que s é o vetor das previsões.

4.5 Loss Function

De modo a avaliar o desempenho do nosso algoritmo foi necessário implementar uma forma de calcular o custo do calculo das previsões, ou seja, calcular o custo entre uma previsão correta e uma incorrecta, entre o valor real e o previsto. Para a versão Primal esta função é dada pela seguinte fórmula:

$$E(\widetilde{w}, D) = \sum_{m=1}^{N} y^m(log(\widehat{y^m})) + (1 + y^m)log(1 - \widehat{y}^m)$$
$$\widehat{y} = \widehat{y}(x, \widetilde{w}); \quad \widehat{y}^m = (x^m, \widetilde{w})$$

Figura 9: Loss function na versão Primal

Na versão Dual da Machine Learning a Loss Function é muito semelhante, sendo apenas alterado o espaço de atributos ficando então com a seguinte formula:

$$E(\lambda, D) = \sum_{m=1}^{N} y^m(log(\widehat{y^m})) + (1 + y^m)log(1 - \widehat{y}^m)$$
$$\widehat{y} = \widehat{y}(x, \lambda); \quad \widehat{y}^m = (x^m, \lambda)$$

Figura 10: Loss function na versão Dual

Na versão com Kernel a loss function usada foi igual à usada na versão dual.

O objetivo geral do treino da Machine learning será minimizar o valor de perda (loss) o que se traduz numa maior precisão de resultados.

5 Simulação

5.1 Tratamento dos Dados

Com todo o algoritmo definido e implementado começamos então a testar para os diferentes *datasets* sendo estes o *Quadrados Circulos*, *Line 600* e *Rectangle 600*. Como apenas temos uma base de dados de cada tipo, separamos os dados em duas partes:

- **Base de dados para treino**: Ficou com 80% dos dados;
- **Base de dados para teste**: Ficou com os restantes 20% dos dados.

Esta separação é muito importante pois permite-nos verificar se a nossa Machine Learning aprende com os dados reservados para o treino e generaliza bem para os dados reservados para o teste.

Posteriormente, tanto aos dados de treino como aos dados de teste, foi feita uma normalização em que todos os valores inferiores a 0 foram convertidos em 0 de modo a que o algoritmo pudesse ler os dados no formato previsto.

5.2 Avaliação do Desempenho

De modo a facilitar a visualização dos resultados utilizamos uma **Matriz de Confusão**. Esta é uma matriz quadrada de dimensão 2x2 que contém o somatório dos valores consoante as suas classificações. Esta matriz segue a seguinte forma:

$$\begin{bmatrix} True\ positives & False\ positives \\ False\ negatives & True\ negatives \end{bmatrix}$$

Figura 11: Formato da Matriz de Confusão

Desta forma podemos fácil e rapidamente ter a visão e fazer uma análise prévia da precisão do nosso sistema.

Ainda para recolher dados em relação ao desempenho do nosso sistema foi também calculado, a partir dos resultados contidos na Matriz de Confusão, o grau de precisão do nosso sistema. Este valor pode ser calculado através da seguinte formula:

$$Acc = \frac{True\ positives + True\ negatives}{Total} * 100$$

Figura 12: Fórmula do calculo da precisão

5.3 Análise de Resultados

De modo a testar as diferentes versões da nossa Machine Learning nos diferentes *datasets* dividimos os testes em três fases diferentes:

- **Teste com Quadrados/Círculos**: Conjunto de dados linearmente separável;
- **Teste com Line 600**: Conjunto de dados não linearmente separável, sendo que com o uso de dois ou mais kernels conseguimos separar todos os valores fácilmente;
- **Teste com Rectangle 600**:Conjunto de dados não linearmente separável, sendo que com o uso de dois ou mais kernels, ainda que com alguma dificuldade, conseguimos separar todos os valores.

Tendo isto em conta começamos então a efetuar todos os testes e a retirar os valores obtidos.

Quadrados/Círculos

Como este é, dos três, o único *dataset* linearmente separável, obtivemos os melhores resultados para o testing tanto como a convergência mais rápida no training:

Quadrados/Circulos	Primal	Dual	Kernel / k=2	Kernel / k=3
Training	100% Acc LR: 0.5 | Iter: 200 LR: 0.1 | Iter: 150	100% Acc LR: 1 | Iter: 300 LR: 0.2 | Iter: 200	100% Acc LR: 0.5 | Iter: 800 LR: 0.1 | Iter: 200	100% Acc LR: 1 | Iter: 800 LR: 0.5 | Iter: 200
Test	100% Acc True positive: 164 True negative: 136 False positive: 0 False negative: 0	100% Acc True positive: 164 True negative: 136 False positive: 0 False negative: 0	100% Acc True positive: 164 True negative: 136 False positive: 0 False negative: 0	100% Acc True positive: 164 True negative: 136 False positive: 0 False negative: 0

Figura 13: Resultados obtidos para as diferentes versões do algoritmo com o dataset Quadrados/-Círculos

Como podemos observar pela tabela, este dataset facilmente converge em todas as versões do algoritmo obtendo uma separação perfeita conseguindo assim ter uma precisão de teste de 100% em todas as versões.

Line 600

Este dataset é o primeiro não linearmente separável a ser analisado e é também o mais simples de obter convergência. Esta não acontece nem na versão primal nem dual devido ao facto deste conjunto de dados não ser, como referido anteriormente, linearmente separável, convergindo apenas para versões do algoritmo com dois ou mais kernels. Estes resultados podem ser visualizados na tabela seguinte:

Line 600	Primal	Dual	Kernel / k=2	Kernel / k=3
Training	72% Acc LR: 1 | Iter: 2500 LR: 0.5 | Iter: 2000 LR: 0.1 | Iter: 1000 LR: 0.05 | Iter: 1000	82% Acc LR: 1 | Iter: 1000 LR: 0.5 | Iter: 500 LR: 0.2 | Iter: 500 LR: 0.05 | Iter: 400	100% Acc LR: 1 | Iter: 100	100% Acc LR: 1 | Iter: 100
Test	63% Acc True positive: 48 True negative: 28 False positive: 23 False negative: 21	70% Acc True positive: 41 True negative: 44 False positive: 20 False negative: 15	100% Acc True positive: 61 True negative: 59 False positive: 0 False negative: 0	100% Acc True positive: 61 True negative: 59 False positive: 0 False negative: 0

Figura 14: Resultados obtidos para as diferentes versões do algoritmo com o dataset Line 600

A partir desta tabela podemos facilmente observar que tanto para a versão primal e versão dual do nosso algoritmo não conseguimos separação total, o que já era de esperar visto que se trata de um conjunto de dados não linearmente separável. Conseguimos também observar que com o uso de dois ou mais kernels é relativamente simples obter convergência total o que resulta numa precisão de teste também de 100%.

Rectangle 600

Este *dataset* é o último e mais complexo a ser utilizado para testes, sendo que se trata, como o anterior, de um conjunto de dados não linearmente separável e, de maior dificuldade de obter convergência com o recurso a kernels. Por isso será necessário muito mais poder computacional para obter convergência total, como podemos observar na tabela seguinte:

Rectangle 600	Primal	Dual	Kernel / k=2	Kernel / k=3
Training	70% Acc LR: 1 | Iter: 2500 LR: 0.5 | Iter: 2000 LR: 0.1 | Iter: 1000 LR: 0.05 | Iter: 1000 LR: 0.01 | Iter: 1000	70% Acc LR: 1 | Iter: 2000 LR: 0.5 | Iter: 1500 LR: 0.2 | Iter: 800 LR: 0.05 | Iter: 600 LR: 0.01 | Iter: 500	100% Acc LR: 1 | Iter: 1000 LR: 0.5 | Iter: 400 LR: 0.1 | Iter: 200	100% Acc LR: 1 | Iter: 1000 LR: 0.5 | Iter: 500 LR: 0.1 | Iter: 200
Test	65% Acc True positive: 50 True negative: 29 False positive: 21 False negative: 20	64% Acc True positive: 47 True negative: 30 False positive: 24 False negative: 19	96% Acc True positive: 68 True negative: 48 False positive: 3 False negative: 1	98% Acc True positive: 70 True negative: 48 False positive: 1 False negative: 1

Figura 15: Resultados obtidos para as diferentes versões do algoritmo com o dataset Rectangle 600

A partir desta tabela podemos observar que, em relação ao teste para o conjunto de dados anterior, obtivemos resultados piores para as versões primal e dual do algoritmo, levando mais iterações para o treino e não obtendo resultados melhores. No que toca aos resultados com o uso de kernels, podemos observar um aumento significativo no número de iterações necessárias para obter uma convergência total no treino e, ainda assim no test não conseguimos obter uma precisão de 100% o que leva a crer que existem situações no *dataset* de teste que não foram abordadas no treino, ou seja o sistema não reconhece aquele tipo de dados de entrada. Esta quebra na precisão pode ser também devido ao *over fitting* causado pelo uso de kernels, sendo que quando maior o número de kernels utilizado maior é o risco de obtermos erros na classificação.

6 Conclusão

Para este trabalho foi pedida a implementação de um *Logistic Classifier* em três diferentes versões, sendo estas a versão primal, dual e com o uso de kernels. Para isto foram usados todos os mecanismos anteriormente mencionados, obtendo assim um resultado favorável. Foram também realizados testes e análise de desempenho para verificar se os algoritmos desenvolvidos funcionam como previsto, em que observamos os valores previamente esperados. Para perspectivas de trabalho futuro, seria a implementação de uma loss function para a versão com kernel para obtermos valores ainda mais precisos e realistas. Em suma, com este trabalho consegui perceber o funcionamento de um classificador logístico, sendo que o resultado final é favorável.