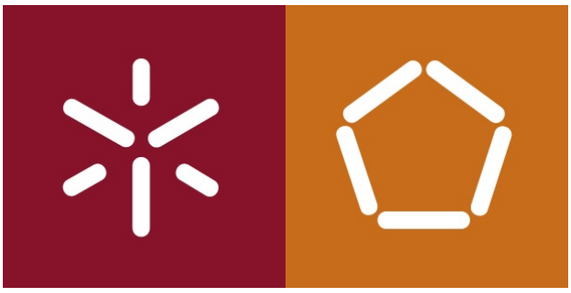
Mestrado Integrado em Engenharia Informática

Otimização em Machine Learning
Grupo A



Perceptron multi-classe

Bruno José Infante de Sousa, A78997
Bruno Manuel Macedo Nascimento, A67647
João Miguel Freitas Palmeira, A73864
Rafael Machado Silva, A74264
Ricardo Barros Pereira, A77045
Tiago Filipe Gonçalves Vilar Pereira, A61032

7 de Junho de 2020

Resumo

O presente relatório surge no âmbito da Unidade Curricular de Otimização em *Machine Learning*, do 4ºano do Mestrado Integrado em Engenharia Informática da Universidade do Minho. Este documento visa descrever o desenvolvimento do trabalho prático da UC, que tinha como tema o *Perceptron* multi-classe. O objetivo passou por comparar os resultados obtidos com diferentes abordagens, como o *One vs All*, *One vs One* e *Error-Correcting Output Codes* tendo em conta a relação entre o tempo computacional e a eficiência de aprendizagem. Cada versão foi usada com o método primal, método dual e dual com *kernel*, sendo todas as implementações escritas em *Python*.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

1 Introdução

No seguimento da Unidade Curricular de Otimização em *Machine Learning*, inserida no 4º ano do Mestrado Integrado em Engenharia Informática, da Universidade do Minho, foi realizado o seguinte trabalho que tem como tema o estudo do algoritmo *Perceptron* Multi-classe.

O objetivo deste trabalho é estudar, analisar e implementar este algoritmo utilizando a linguagem de programação *Python*, seguindo diferentes abordagens e métodos para assim comparar a sua eficiência e custo computacional. Assim sendo, seguindo os métodos primal, dual e dual com *kernel* codificamos este algoritmo de acordo com as versões *One vs All*, *One vs One* e *Error-Correcting Output Codes*.

Para fazer a comparação entre cada versão, a proposta pedia que fosse utilizada uma base de dados com 4 classes. Posto isto, depois de analisar diferentes opções, optamos por utilizar o *dataset Digits* do *Scikit Learn* e o *dataset Mnist* do *TensorFlow* estudando assim a problemática do reconhecimento de dígitos caligráficos.

De seguida vamos passar a uma análise e explicação do algoritmo *Perceptron* Multi-classe, contendo também uma análise a cada uma das diferentes abordagens foram exploradas. No capítulo do desenvolvimento iremos analisar e apresentar, de forma mais aprofundada, a base de dados utilizada. Este capítulo contém ainda uma explicação das opções que foram tomadas na parte prática deste trabalho com a codificação de cada versão do algoritmo, seguindo-se uma análise crítica dos resultados obtidos. Finalizamos este relatório com uma conclusão, fazendo uma revisão ao trabalho realizado, focando os pontos positivos e também aquilo que poderia ter sido melhorado.

2 *Perceptron*

É um algoritmo usado para aprendizagem supervisionada de classificadores binários, classificadores que decidem se um *input*, normalmente representado por uma série de vetores, pertence a uma classe específica.

De forma breve, *Perceptron* é uma camada de rede neuronal constituída por 4 elementos:

- *Input* e os seus valores
- Pesos e *bias*
- Somatório de vetores
- Função de ativação

2.1 Abordagem científica

O raciocínio humano deriva do cérebro e, neste órgão, existem por volta de 100 mil milhões de neurónios que são responsáveis pelo controlo de funcionalidades humanas.

Um neurónio recebe um sinal do exterior e propaga a reação a esse *input* por outros neurónios, através de sinais eléctricos, até chegar ao local adequado para iniciar o procedimento de atuação. Entre neurónios existem diferenciações na quantidade de sinais que recebem/enviam, sangue que flui, conexões e produção para o sistema. Estes fatores têm um peso nos envios de sinais porque existem diferentes *ativações*.

É implementado um neurónio artificial através de um algoritmo, que recebe *input* de outros neurónios, cada um com o seu grau de importância diferente, cada conexão realizada com os intervenientes contém um peso.

Ativação refere-se ao somatório de todos os valores de peso proveniente dos *inputs* num neurónio. Se ativação >0, o algoritmo continua a iterar para os próximos neurónios, se for <0 não haverá ativação.

$$a = \sum_{d=1}^{D} \omega_d \chi_d$$

É a fórmula que o algoritmo se baseia em que o vector $\chi = (x_1, x_2, ..., x_D)$ é o vector de *input*, e D contém os pesos associados ao neurónio, $\omega_1, \omega_2, \ldots, \omega_D$.

Instâncias com peso zero serão ignoradas, instâncias com peso positivo serão valorizadas como casos positivos e a seguir. Por fim, instâncias com peso negativo são indicativas de exemplos negativos com prejuízo no valor da ativação. É aconselhado utilizar um *Threshold* com valor diferente de 0, como pretendemos obter previsões com valores positivos, ou seja, $a > 0$. É necessário introduzir ***bias*** no neurónio, para a ativação ser

incrementada de todas as vezes de forma fixa e criada uma variável b que e adicionada ao valor do somatório, de tal forma que ficamos com a equação

$$a = \left(\sum_{d=1}^{D} \omega_d \chi_d \right) + b.$$

2.2 Orientação por Erro

Perceptron é um algoritmo clássico de aprendizagem neuronal que apesar de simples é eficaz. Este atua de forma diferente de outros algoritmos similares, p.e, *Árvores de Decisão* e *K Nearest Neighbours* (KNN).

A forma como atua é simples, o *dataset* é **particionado** e o algoritmo trata cada parte em vez de todo o dataset de uma só vez. Outra particularidade é ser **orientado pelo erro**. Ou seja, enquanto o cálculo efetuado não apresentar erros, não faz *update* aos parâmetros estabelecidos.

O processo de execução é descrito desta maneira, o algoritmo estima os parâmetros, pesos e *bias* que deve considerar para calcular a ativação. Calcula apenas uma parte dos dados, retorna a previsão e compara-a com o *dataset*. Como estamos em partição *training* temos acesso aos *labels* que identificam as previsões certas. Caso todo este raciocínio se verifique, o algoritmo permanece idêntico. Caso se verifique que a previsão não esteja correcta, os parâmetros são alterados para os valores que acertavam na previsão, e seguem para a próxima partição.

Algoritmo 1: Treino *Perceptron*

```
1  w_d ← 0, ∀d = 1...D;
2  b ← 0;
3  for iter = 1 ... MaxIter do
4  |  forall (x, y) ∈ D do
5  |  |  a ← Σ_{d=1}^{D} ω_d χ_d + b;
6  |  |  if ya ≤0 then
7  |  |  |  w_d ← w_d + yx_d, ∀d = 1...D ;
8  |  |  |  b ← b + y;
9  |  |  end
10 |  end
11 end
12 return w_0, w_1, ..., w_D, b
```

Algoritmo 2: Teste *Perceptron*

```
1  Σ_{d=1}^{D} ω_d x̂_d + b
2  return SIGN (a)
```

O algoritmo "Treino *Perceptron*", possui uma particularidade, na linha 6, quando é verificado se é necessária uma atualização, que é necessária se a previsão não estiver de acordo. O truque é multiplicar o valor do rótulo com a ativação e verificar com o valor 0. Como o rótulo possui valor $+1$ ou -1, apenas é preciso verificar o valor do produto ya, é positivo quando a e y possuem o mesmo sinal, ou seja quando a previsão é correta, o produto ya é positivo.

O peso ω_d é aumentado por yx_d e o *bias* é incrementado por y. O objectivo do *update* é ajustar os parâmetros para que futuras iterações possuam previsões certas. Quando ocorre um erro e é calculada uma nova ativação, é necessário atualizar os valores dos pesos e *bias*, os novos pesos são $w'_1, w'_2, \ldots, w'_D, b$, e para calcular a nova ativação (a') efetuamos o seguinte cálculo

$$a = \sum_{d=1}^{D} \chi_d^2 + 1 > a.$$

A diferença entre a ativação inicial e esta nova reside no facto de que $\sum_{d=1}^{D} {\chi_d}^2 + 1$, mas ${\chi_d}^2 \geq 0$, uma vez que é ao quadrado, portanto este valor terá sempre o valor de 1, além do facto que a ativação nova é a anterior mais 1.

O único *hiperparâmetro* do algoritmo *Perceptron* é *MaxIter*, número de iterações realizadas no algoritmo de treino, se efetuarmos demasiadas passagens, poderá ocorrer *overfitting*, se não passarmos as suficientes para treino poderemos incorrer num caso de *underfitting*. Outra dica, é permutar a ordem dos exemplos ao algoritmo com intuito de impedir que exemplos com a mesma previsão sejam demonstrados todos de forma seguida, de forma a não *viciar* as previsões e consequente resultado.

2.3 Interpretação Geométrica

Neste sub capítulo vamos abordar como o limite de decisão do algoritmo *Perceptron* é assumido. Tal como vimos anteriormente, a decisão é efetuada consoante o valor da ativação, $+1$ e -1. Por outras palavras, os valores dos pontos de $\mathcal{X}$ que obtêm ativação 0. Estes pontos são neutros para incorporar na fórmula de limite de decisão, caso o valor de b seja

$$B = \left\{ x : \sum_{d=1}^{D} \omega_d \chi_d = 0 \right\}.$$

Agora é possível aplicar álgebra linear. Relembrando, $\sum_d \omega_d \chi_d$, é o produto entre o vetor de pesos, $\omega = \langle w_1, w_2, \ldots, w_D \rangle$, e o vetor de *input*, $\chi = (x_1, x_2, \ldots, x_D)$. É feito o produto $\omega \cdot \chi$, se o resultado for igual a 0 significa que são perpendiculares. O plano que resultar do produto é o limite da decisão entre os pontos positivos e negativos. A escala dos pesos é irrelevante da perspectiva da classificação. **Normalizar** os vetores de peso.

O papel do *bias* deve ser adaptado consoante a projeção dos pontos, previamente o valor de *Threshold* estava definido a 0, qualquer projeção no vetor ω com valor negativo seria classificada como negativa e vice-versa para casos positivos, as modificações do *bias* altera o valor do *Threshold* comparado ao original, se o valor $b \geq 0$, então , o limite de decisão afasta-se do vetor ω, caso contrário se $b \leq 0$ então o limite de decisão aproxima-se do vetor ω, e consoante as movimentações, diferentes serão as classificações.

2.4 Interpretação dos Pesos do *Perceptron*

A forma como o *Perceptron* aprende a classificar modificações durante a última classificação, podem ser respondidas com alterações na derivação. A forma como fazê-lo é ordenar desde os maiores valores positivos até aos maiores valores negativos, e de seguida extrair os 10 maiores positivos e os 10 maiores negativos. Este raciocínio pode ser representado na equação de limite de decisão.

$$x \mapsto \text{sign} \left(\sum_{d=1}^{D} \omega_d \chi_d + b \right) \tag{1}$$

No final obtêm-se um conjunto com os 10 maiores positivos, que contem características que influenciam o modelo a fazer previsões positivas, tal como os 10 maiores negativos possuem características para fazer previsões negativas.

2.5 Convergência do *Perceptron* e Separabilidade Linear

Diz-se que dois subconjuntos X e Y de dimensão 2 são linearmente separáveis se existir um hiperplano, de maneira que os elementos de X e os de Y se encontrem em lados opostos. A Figura 1 mostra um exemplo do conjunto de pontos linearmente separáveis e outro não linearmente separáveis.

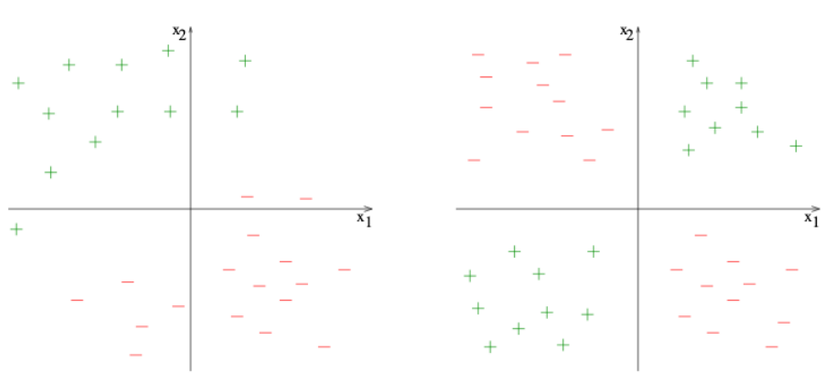


Figura 1: *Datasets* de dimensão 2 — O *dataset* da esquerda é linearmente separável, o da direita não.

No lado esquerdo da Figura 1, podemos desenhar uma linha de modo que todos os pontos marcados a $\{+\}$ fiquem em um lado da linha, e todos os pontos marcados a $\{-\}$ estejam no outro lado. Também é visível que nenhuma linha com a mesma propriedade pode ser desenhada para o conjunto de pontos rotulados mostrado à direita. A propriedade em questão é separabilidade linear. De modo geral, no espaço d-dimensional, um

conjunto de pontos com rótulos de $\{+, -\}$ é separável linearmente se existir um hiperplano no mesmo espaço, de modo que todos os pontos rotulados a $\{+\}$ fiquem em um lado do hiperplano e todas os pontos marcados a $\{-\}$ fiquem do outro lado do hiperplano.

Um *Perceptron* com seus parâmetros fixos pode de fato ser visto como um hiperplano centrado na origem que divide o espaço em duas regiões. Concretamente, os parâmetros (ou pesos) do *Perceptron* podem ser interpretados como os componentes de um vetor normal para o hiperplano. Observe que um vetor normal é suficiente para consertar um hiperplano centrado na origem. De fato, o comprimento do vetor normal é imaterial; somente a sua direção é importante.

Dado um *dataset* rotulados como a $\{+\}$ e $\{-\}$, o algoritmo de aprendizagem do *Perceptron* é um procedimento iterativo para atualizar os pesos de um *Perceptron*, de modo que, eventualmente, o hiperplano correspondente contenha todos os pontos marcados a $\{+\}$ de um lado e todos os pontos marcados a $\{-\}$ do outro.

No *Perceptron*, se os dados forem linearmente separáveis, eles convergirão para um vetor de pesos que os separa. Intuitivamente, o algoritmo de *Perceptron* convergirá mais rapidamente para problemas mais simples e mais lentamente para problemas mais complexos.

Pode-se definir essa complexidade usando a noção de *margin*, que é a distância entre o hiperplano e o ponto mais próximo. Problemas com margens grandes são simples, porque há muito espaço de manobra para encontrar um hiperplano de separação. Problemas com margens pequenas são difíceis porque precisa-se ter um vetor de pesos muito específico e bem ajustado.

$$\text{margin}(D, w, b) = \begin{cases} \min_{(x,y)\in D} = y(w^T x + b), & \text{se } w \text{ separa } D \\ -\infty, & \text{caso contrário.} \end{cases}$$

A margem é definida apenas se w, b separar os dados D. Se for definido, a margem será o ponto com ativação mínima, após a ativação ser multiplicada pela *label*.

Para calcular a margem de um *dataset*, tenta-se todos os pares w, b possíveis. Por cada par, calcula-se a margem e considera-se a maior delas como a margem geral dos dados. A margem é normalmente indicada por γ.

2.6 Generalização Melhorada: Votação e Média

Vanilla Perceptron é um algoritmo eficaz no entanto pode sofrer melhorias para torná-lo mais competitivo. O problema está no treino do *Perceptron*, em que uma modificação nos *inputs* que apareça no final do treino pode tornar inválida toda aprendizagem até esse ponto.

A solução passa por introduzir **votação** nos hiperplanos que maior tempo de vida tiveram consoante o número de iterações do algoritmo que continuaram estáveis. A forma tradicional por voto funciona de forma eficaz até ocorrer uma situação previamente descrita, a equação seguinte é o cálculo tradicional do algoritmo *Vanilla Perceptron*.

$$\hat{y} = \text{sign}\left(\sum_{k=1} kc^{(k)} \,\text{sign}(\omega^{(k)} \cdot \hat{x} + b^{(k)})\right)$$

Uma proposta de resolução trata de trocar o voto único por uma média de classificações, o conceito passa por guardar os pesos e o seu tempo de sobrevivência, mas ao mesmo tempo compara com a média obtida até esse ponto e não pela votação do melhor resultado. A seguir é apresentada a proposta de resolução para cálculo da média.

$$\hat{y} = \text{sign}\left(\sum_{k=1} kc^{(k)} (\omega^{(k)} \cdot \hat{x} + b^{(k)})\right)$$

A principal diferença reside de *sign* não aparecer no interior do somatório, a vantagem desta abordagem de pesos médios ao *perceptron* é que é possível manter a soma até em qualquer parte do cálculo do hiperplano. O *Perceptron* médio, em maior parte dos casos comparado com o *Perceptron* normal é melhor, devido á generalização dos casos. No entanto não impede que ocorra paragem antes das iterações previamente definidas e incorrer em *overfitting*.

Algoritmo 3: Treino *Perceptron* com Pesos Médios

$\omega \leftarrow \langle 0, 0, \ldots, 0 \rangle, b \leftarrow 0$
$u \leftarrow \langle 0, 0, \ldots, 0 \rangle, \beta \leftarrow 0$
$c \leftarrow 1$
for $iter = 1 \ldots MaxIter$ **do**
  forall $(x, y) \in D$ **do**
    if $y(\omega \cdot \chi + b) \leq 0$ **then**
      $w \leftarrow w + y\ x, ;$
      $b \leftarrow b + y;$
      $u \leftarrow u + y\ c\ \chi;$
      $\beta \leftarrow \beta + y\ c;$
    end
    $c \leftarrow c + 1;$
  end
end
return $\omega - \frac{1}{c}u, b - \frac{1}{c}\beta$

2.7 *Perceptron* e os seus limites

As redes *Perceptron* devem ser treinadas com adaptações, que apresenta os vetores de entrada para a rede, um de cada vez, e faz correções na rede com base nos resultados de cada representação. Adaptar dessa maneira garante que qualquer problema linearmente separável seja resolvido num número finito de representações de treino. *Perceptrons* também podem ser treinados com funções de treino. Quando se utiliza o treino para *Perceptrons*, ele apresenta as entradas para a rede em *batches*, e faz correções na rede com base na soma de todas as correções individuais. Infelizmente, não há provas de que esse algoritmo de treino convirja para *Perceptrons*.

As redes *Perceptron* têm várias limitações. Primeiro, os valores de saída de um *Perceptron* podem assumir apenas um de dois valores (0 ou 1) devido à função de transferência com limite rígido. Segundo, os *Perceptrons* só podem classificar conjuntos de vetores linearmente separáveis. Se uma linha reta ou um plano puder ser desenhado para separar os vetores de entrada em suas categorias corretas, os vetores de entrada são linearmente separáveis. Se os vetores não forem linearmente separáveis, o aprendido nunca chegará a um ponto em que todos os vetores sejam classificados corretamente. No entanto, se os vetores são linearmente separáveis, os *Perceptrons* adaptadamente treinados sempre encontrarão uma solução em tempo finito. Com alguns exemplos de *datasets* consegue-se verificar a dificuldade de classificar vetores de entrada que não são linearmente separáveis.

2.8 Classificação Multi-classe

A classificação multi-classe é uma extensão natural da classificação binária. A diferença é que se tem $K > 2$ classes para escolher. O objetivo é o mesmo: temos um classificador binário para resolver o problema de classificação em várias classes. Uma abordagem muito comum é a técnica *one versus all* (OVA). Para realizar esta técnica, treinamos K classificadores binários e cada classificador vê o treino todo dos dados. O classificador de uma determinada classe i retorna todos os exemplos da classe i como positivos e todos os outros exemplos como negativos. Mais à frente no relatório explicaremos mais detalhadamente este algoritmo. Outra técnica muito utilizada é *all versus all* ou *one versus one* (OVO) que consiste em criar classificadores para cada par de classes possível com o objetivo de testar se é de uma ou outra classe. Um determinado classificador da classe i contra a classe j retorna todos os elementos da classe i como positivos e todos os exemplos da classe j como negativos. Cada classe ganha um ponto por cada elemento que é testado e dá positivo para a sua classe. No final a classe com mais pontos vence. Mais à frente iremos também explicar mais detalhadamente este algoritmo. Por fim existe também uma abordagem com o método de *ECOC* que combina muitos classificadores para resolver problemas de Multi-classe. Este método baseia-se em gerar um código de erro distinto com comprimento fixo n, por cada uma das K classes. A esse código é atribuído o nome de *code-word*. Cada uma das *code-words* é comparada com uma das outras pertencente às K classes. Depois de treinar os classificadores, para classificar uma nova sequência temos de evoluir todos os n classificadores binários para obter a nova *code-word*. Por fim escolhe-se a classe cuja *code-word* seja a mais semelhante à de previsão já existente. Na secção relativa ao *ECOC* podemos encontrar maior detalhe sobre este algoritmo.

2.9 Kernel

Os modelos lineares são ótimos porque são fáceis de entender e otimizar. Eles sofrem porque só podem aprender limites de decisão muito simples. As redes neuronais podem aprender limites de decisão mais complexos, mas perdem as boas propriedades de convexidade de muitos modelos lineares. Uma maneira de fazer com que um modelo linear se comporte de maneira não linear é transformar o *input*. Por exemplo, adicionando pares de *features* como *inputs* adicionais. Aprender um modelo linear em tal representação é convexo, mas é computacionalmente proibitivo em todos os aspetos, excepto em dimensões muito baixas. A família de técnicas que torna isso possível é conhecida como abordagens de *kernel*.

No nosso caso em concreto o que interessa é conseguir fazer com que o algoritmo *Perceptron* consiga classificar modelos não lineares através do uso do *kernel*. Considere o algoritmo 1 (*perceptron*) usando notação de álgebra linear e usando notação de expansão de *features* de $\phi(x)$. Neste algoritmo, há dois lugares onde $\phi(x)$ é usado explicitamente. O primeiro está no cálculo da ativação e o segundo está na atualização dos pesos. O objetivo é remover a dependência explícita desse algoritmo em ϕ e no vetor de peso. Para fazer isso, é possível observar que, em qualquer ponto do algoritmo, o vetor de peso w pode ser gravado como uma combinação linear de dados de treino expandidos. Em particular, a qualquer momento, $w = \sum_n a_n \phi(x_n)$ para quaisquer parâmetros a. Inicialmente $w = 0$, então escolher $a = 0$ vai gerar isso. Se a primeira actualização ocorre no n-ésimo exemplo de treino, a resolução o vetor de peso é simplesmente $y_n \phi(x_n)$, equivalente à configuração $a_n = y_n$. Se a segunda actualização ocorrer no m-ésimo exemplo de treino, só é preciso atualizar $a_m \leftarrow a_m + y_m$. Isto verifica-se, mesmo percorrendo várias vezes os dados. Essa observação leva à conclusão de que o vetor de peso do *perceptron* está no intervalo dos dados de treino. Podemos observar o *perceptron* com *kernel* no algoritmo 4.

Algoritmo 4: *Perceptron* com *Kernel*

```
a ← 0, b ← 0
for iter = 1 ... MaxIter do
    forall (x_n, y_n) ∈ D do
        a ← Σ_m a_m φ(x_m).φ(x_n) + b
        if y_n a ≤ 0 then
            a_n ← a_n + y_n
            b ← b + y
        end
    end
end
return a, b
```

3 Classificadores Multi-Classe

3.1 *One vs All*

Em *Machine Learning* a *Multiclass Classification*, como se referiu anteriormente, representa o problema de classificar três ou mais classes numa instância, pois classificar duas classes numa instância representa uma classificação binária.

A estratégia *One vs All* (OVA) permite transformar problemas de classificação multiclasse em binários. Dado um problema de classificação com N possíveis soluções, a OVA utiliza um classificador binário por cada *label*, ou seja, L classificadores. Cada classificador prevê se um *label* pertence a um dado valor da classe.[1] O algoritmo desta abordagem é dado por:

1. For $t = (1 : L)$
2. Amostras positivas se $x_i : t \in y_i$
3. Amostras negativas se $x_i : t \notin y_i$
4. $f_t(x)$: decisão do classificador t (um valor de f_t grande indica uma elevada probabilidade de x pertencer à classe t)
5. Previsão: $f(x) = argmax_t f_t(x)$

A OVA envolve o treino de um único classificador por classe, utilizando todas as amostras dessa classe como amostras positivas e as amostras de todas as outras classes como

amostras negativas. Ou seja, toma uma classe como positiva e o resto como negativa e treina o classificador. Portanto, para os dados que possuem n-classes, ele treina n classificadores. De seguida, é exigido que os classificadores de base produzam uma pontuação de confiança com valor real da decisão, ao invés de utilizar apenas uma *label* da classe. Na fase de pontuação, todo o classificador n prevê a probabilidade de determinada classe e é selecionada a classe com maior probabilidade. É importante referir que as *labels* de classes discretas podem levar a ambiguidades, onde várias classes são previstas para uma única amostra. Se observarmos o algoritmo anterior, podemos afirmar que tomar decisões significa aplicar todos os classificadores a uma amostra invisível x e prever o rótulo t para o qual o classificador correspondente relata a maior pontuação de confiança. Esta estratégia contém vários problemas, entre os quais a escala dos valores de confiança. Pode diferir entre os classificadores binários e mesmo que a distribuição de classes seja equilibrada no conjunto de treino, as classificações que se irão ver aparentam ser distribuições desequilibradas, uma vez que normalmente o conjunto de negativos é muito maior que o conjunto de positivos.

A título de exemplo, consideremos que existem quatro classes diferentes: A, B, C e D. Aos classificadores para essas mesmas classes chamaremos de classificador_A, classificador_B, classificador_C e classificador_D. Realizada a previsão, cada classificador contém a seguinte probabilidade:

- classificador_A = 40%
- classificador_B = 30%
- classificador_C = 60%
- classificador_D = 50%

Relembrando que a probabilidade de uma classe é a mesma que a probabilidade do classificador respetivo e que a soma das probabilidades pode não ser 100%, a escolha irá recair pela classe B devido aos resultados anteriores.

3.2 *One vs One*

A estratégia *One vs One* (OVO) considera cada par binário de classes e treina o classificador no subconjunto de dados que contém essas classes. Utiliza $L(L-1)/2$ classificadores binários para um problema com L classes. Durante as fases de classificação, cada classificador prevê apenas uma classe e a classe que mais foi prevista é a resposta, o que contrasta com o OVA, onde cada classificador prevê uma probabilidade. Tal como foi referido, cada um recebe um par de classes que corresponde a todas as combinações do número de classes duas a duas, formando o classificador. Se na *label* estiver presente um par, este é identificado com um 1. A previsão é feita através da contagem da incidência dos elementos em cada um dos classificadores. Isto é, todos os classificadores $L(L-1)/2$ são aplicados a uma amostra invisível e a classe que obteve o maior número de previsões "+1"é prevista pelo classificador combinado.[1]

1. For $t = (1 : L(L-1))$
2. Amostras positivas se $x_i : s \in y_i$
3. Amostras negativas se $x_i : t \in y_i$
4. $f_{(t,s)}(x)$: se $f_{(t,s)}(x)$ for elevado, a classe s tem maior probabilidade que a classe t
5. $f_{(t,s)}(x) = -f_{(s,t)}(x)$
6. Previsão é dada por: $f(x) = argmax_s(\sum_t f_{(s,t)}(x))$

Tal como no capítulo anterior, a título de exemplo, consideremos que existe quatro classes diferentes: A, B, C e D. Os classificadores para essas mesmas classes são chamados de classificador_AB, classificador_AC, classificador_AD, classificador_BC, classificador_BD e classificador_CD, uma vez que cada um dos seis classificadores que se irão treinar contém as diferentes classes envolvidas. Realizada a previsão, os resultados levam que cada classificador seja atribuído a uma classe:

- classificador_AB atribui a classe A
- classificador_AC atribui a classe A
- classificador_AD atribui a classe A

- classificador_BC atribui a classe B
- classificador_BD atribui a classe D
- classificador_CD atribui a classe C

Neste caso, a escolha recai pela classe A pela maioria das atribuições apesar de não existirem as probabilidades.

3.3 *Error-Correcting Output Codes*

O *Error-Correcting Output Codes* (ECOC) é um método criado para problemas de classificação multi-classe. Este método combina vários classificadores binários com o objetivo de resolver os problemas multi-classe, sendo que a primeira tarefa passa por decidir o número de opções possíveis (sempre superior a 2, ou seja, $k > 2$) para classificar as várias classes. Considerando a tarefa de reconhecimento de dígitos, é necessário mapear cada dígito escrito à mão para uma das $k = 10$ classes. No entanto, existem vários algoritmos como as árvore de decisão, *naive bayes* e redes neuronais, que podem lidar diretamente com um problema de várias classes.[3] Os algoritmos baseados em árvores de decisão podem ser generalizados para resolver problemas de classificação multi-classe. Neste caso, dá-se o nome de abordagem multi-classe direta. Cada folha corresponde a uma classe e a descriminação é feita em cada nó da árvore. Por cada árvore de decisão ou unidade de rede neuronal pode ser vista como o cálculo da probabilidade de o novo exemplo pertencer à sua classe correspondente. A classe cuja árvore de decisão providencia a estimativa de maior probabilidade é escolhida como a classe prevista.

Para o *ECOC*, cada árvore de decisão pode ser vista como o cálculo da probabilidade do bit correspondente na respetiva *code-word* ser 1. Associando o valor da probabilidade sendo $B = \langle b1, b2, \ldots, bn \rangle$, sendo n o comprimento das *code-words* no *ECOC*. Para classificar um novo exemplo, calcula-se L_1 distância entre o vetor de probabilidade B para cada uma das *code-words* $W_i (i = 1, 2, ..., k)$ do ECOC. A L_1 distância entre B e W_i é definida por:

$$L_1(B, W_i) = \left(\sum_{j=0}^{L} b_j - W_{i,j} \right).$$

4 Desenvolvimento

Nesta secção iremos apresentar e analisar a componente prática deste projeto que consiste em aplicar, a um classificador *Perceptron* multi-classe/*single-layer* nas versões primal e dual com/sem *kernel*, as estratégias *One-vs-All*, *One-vs-One* e *Error-Correcting Output Codes*. Foi necessário definir um conjunto de dados para que fosse possível testar o classificador com as diferentes estratégias. No entanto optou-se por utilizar dois conjuntos: o *dataset digits*[4], que é composto por 1797 imagens de tamanho 8x8 e um vetor com 64 *features*, e o MNIST, que é composto por 60000 imagens de treino e 10000 imagens de teste com um tamanho de 28x28.[11] Cada imagem destes dois conjuntos de dados corresponde a um dígito escrito à mão.

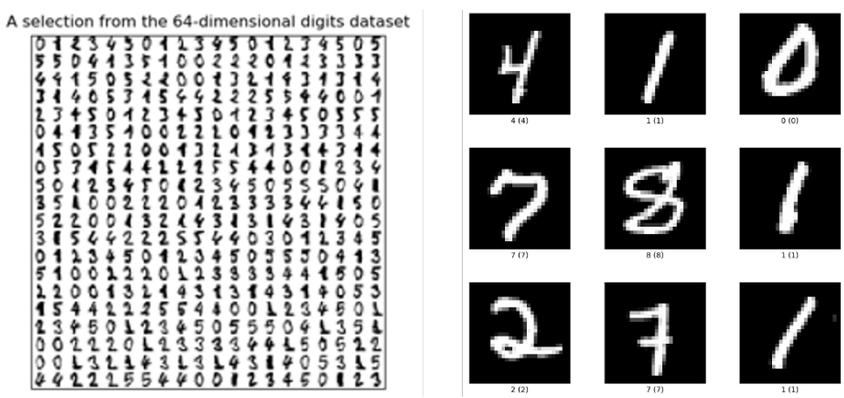


Figura 2: Conjuntos de dados: *digits*[5] e MNIST

4.1 Estrutura principal

Posto isto, foram desenvolvidos dois tipos de *perceptron*: primal e dual. Para cada um dos *perceptrons* aplicou-se uma das três estratégias e como resultado, pretende-se obter as respetivas matrizes de confusão. Na figura 3 pode-se observar o diagrama que representa este processo.

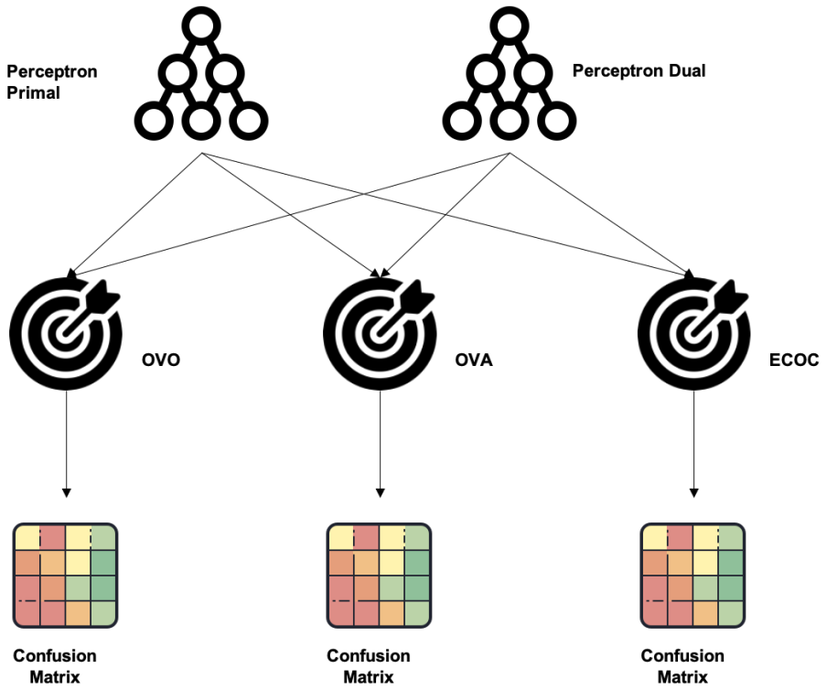


Figura 3: Estratégia do projeto

Para a realização da estratégia apresentada na figura 3 foram desenvolvidos três *scripts*: um *script* para o *perceptron primal* (***MCP_Primal.py***), um *script* para o *perceptron dual* (***MCP_Dual.py***) e um *script* principal ao qual denominamos **server.py** onde são invocados os métodos definidos para os dois *perceptrons* e onde são calculadas as matrizes de confusão, *score* dos modelos e os tempos de execução, ou seja, o resultado final destes algoritmos.

No *script server.py* foram definidas várias funções para o cálculo do modelo e *score* para cada tipo de classificador (OVO, OVA e ECOC). Cada uma destas funções invoca a função para a multi-classe respetiva que está presente no respetivo *script*. Iremos abordar estas classes nas secções seguintes. No entanto, antes de abordarmos estas funções, os primeiros passos deste *script* dizem respeito à importação do conjunto de dados a utilizar através da função ***load_digits*** da livraria *scikit-learn* e são divididos os dados em dados de treino e teste, através da função ***train_test_split***. Para além do *dataset digits*, também se importa o conjunto de dados MNIST através da livraria *keras* e realiza-se a divisão em dados de treino e teste.

Feito isto, foi efetuada uma exploração de dados onde visualizamos que tipo de dados existem no *dataset* e aplicamos um PCA para verificar os *clusters* dos diferentes valores das classes que se irão utilizar, como é possível verificar na figura 4. Para além disso, é importante relembrar que o PCA é o *Principal Component Analysis* que é "um procedimento matemático que utiliza uma transformação ortogonal (ortogonalização de vetores) para converter um conjunto de observações de variáveis possivelmente correlacionadas num conjunto de valores de variáveis linearmente não correlacionadas chamadas de componentes principais."[10] Depois de analisar os dados inicia-se a execução dos diferentes classificadores consoante as funções que foram referidas anteriormente.

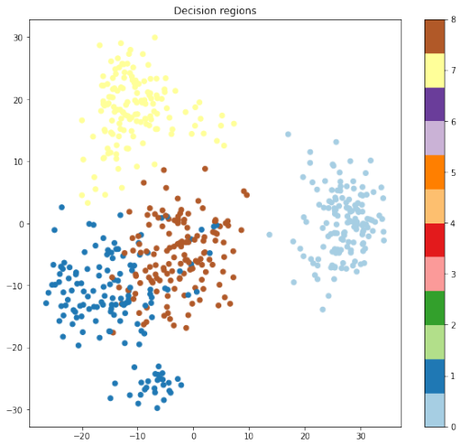


Figura 4: Regiões de decisão — *Clusters*

4.2 *Perceptron*

No *script server.py* estão definidas as funções dos diferentes *perceptrons*, para os diferentes classificadores. O primeiro passo é definir o número de *labels* de cada classe, que nos ajudará a definir o número de classificadores que treinaremos. Mediante o *perceptron*, cada função invocará o método relativo à multi-classe respetiva, ou seja, ou invocará a multi-classe primal ou dual (com e sem *kernel*).

Começando pela multi-classe primal, é importante referir que esta foi construída no *script MCP_primal.py*. Este *script* dispõe de duas classes: ***MCP_Primal*** e ***Perceptron_Primal***. Cada uma das classes dispõe dos métodos ***init***, ***fit*** e ***predict***, no entanto a classe *Perceptron_Primal* é invocada na classe *MCP_Primal*. Desta feita, começamos por explicar em que consiste a classe *Perceptron_Primal*, ou seja, esta classe corresponde ao algoritmo do *perceptron primal* onde é definido o *learning rate*, o número máximo de iterações e o classificador a receber para realizar o *fit* e o *predict* do *perceptron*. Quanto ao *fit* é realizado enquanto não se atingir o número máximo de iterações ou não existir convergência. Terminado o *fit*, é efetuado o *predict* consoante o classificador pretendido, iremos explicar mais detalhadamente para cada um dos classificadores nas secções 4.3.1, 4.3.2 e 4.3.3. O resultado do *fit* depende de cada classificador, embora seja sempre o resultado final um *array* com os *scores* finais. Relativamente à multi-classe dual, o processo é semelhante ao que acontece com o primal, onde no *MCP_dual.py* também existem as classes ***MCP_Dual*** e ***Perceptron_Dual***, embora com as devidas alterações para o algoritmo de *perceptron dual*. Neste caso, ao invés de ser definido o *learning rate*, o número máximo de iterações e o classificador, apenas se define o *kernel* e o classificador. O *fit* não termina enquanto não forem percorridos todos os dados e enquanto o nosso classificador não convergir, relativamente ao *kernel* escolhido (se o *kernel* for "linear", diz-se que corresponde ao dual sem *kernel*). Para cada *kernel* foi definida uma função para ser invocada no *fit* e os *kernels* (k) definidos para além do **linear** ($k(x, y) = x * y$) são o **RBF** ($k(x, y) = e^{-\alpha||x-y||^2}$) e o **Polinomial** ($k(x, y) = (x * y)^d$). Por último, o *predict* também é realizado consoante o *kernel* escolhido que calcula os *scores* finais e consoante o classificador coloca-os num *array* para ser retornado.

Voltando ao *script server.py*, os dados retornados pelo respetivo *perceptron* são retornados em forma de modelo em que a este lhe é aplicado um *fit*, de modo a treinar o modelo com os dados de treino. De seguida, após ter sido treinado o modelo, utiliza-se o modelo para obter as previsões com um grupo de dados de teste, que, posteriormente, serão comparadas (as previsões) com os dados reais através da função *accuracy_score* da livraria *scikit-learn*. Obtidos os resultados, é possível obter novos dados conclusivos como vários gráficos de erros onde é possível observar em que altura cada classificador convergiu, uma figura com todas as previsões obtidas e ainda a matriz de confusão do modelo criado. Estes resultado serão apresentados na secção 5.

4.3 Multi-Classe Perceptron

4.3.1 *One vs All*

Relativamente ao OVA é importante referir que o algoritmo e toda a informação referida na secção 3.1 foi aplicada para utilização deste classificador multi-classe. Como foi referido anteriormente, tanto no *script MCP_primal.py* como no *script MCP_dual.py*

existem duas classes, a classe *perceptron* e a *multi-classe perceptron*. Inicia-se o classificador OVA, na multi-classe *perceptron* é invocado o *perceptron* (primal ou dual) com os respetivos atributos após serem criados os classificadores consoante as classes, ou seja, para cada classificador é aplicado o *perceptron*.

De seguida, realiza-se o *fit* de cada classificador onde inicialmente são alterados os valores binários para valores de "+1"ou -1"e posteriormente é aplicado o *fit* ao classificador. Por último, é realizado a previsão dos dados que são fornecidos (*predict*) através de todos os classificadores disponíveis, criando uma matriz de previsões em que o retorno do *predict* corresponde ao melhor classificador consoante o valor do *argmax* dessa matriz no conjunto de *labels*, isto é, no conjunto de *labels* com os valores que foram fornecidos para treino.

4.3.2 *One vs One*

Quanto ao classificador OVO, o processo foi semelhante ao classificador anterior, onde se iniciou por definir o número de classificadores, uma vez que, para o *One vs One*, o número de classificadores corresponde a metade do número de classes multiplicado pelo número de classes subtraído por um $((n(n-1))/2)$. De seguida, aplica-se o respetivo *perceptron* (primal ou dual) a cada classificador definido. No entanto, como este tipo de classificador multi-classe utiliza mais classificadores, a matriz de *labels* que contém dados de cada classe, será utilizada para construir uma nova com mais dados, ou seja, se a matriz inicial tinha informação de 4 classes, esta nova matriz terá informação de 6 classificadores, uma vez que $(4 \times (4-1))/2 = 6$.

Assim sendo, iniciado o classificador, realiza-se o *fit* onde são alterados os valores binários para valores de "+1"ou "−1"e também definidos os dados de treino e o *target* a fornecer ao *fit* e, por fim, é realizado o *fit*. Por último, é realizado o *predict* com base na estratégia de voto [12], ou seja, para realizar este *predict* definiram-se duas funções: a função ***ovo_voting_both*** que é auxiliada pela função ***ovo_class_combinations***. Estas funções vão criar as amostras positivas e negativas, tal como foi explicado na secção 3.2, e como recebem como argumento as previsões de cada classificador calculadas da mesma forma que o OVA, verifica para cada um dos classificadores se tem mais amostras positivas ou negativas, isto é, mais valores "+1"ou -1"através de um *argmax*, para determinar qual a classe predominante. No final é retornado valor das previsões no conjunto de *labels* da mesma forma que acontece no classificador OVA.

4.3.3 *Error-Correcting Output Codes*

Em relação à implementação do ECOC, a primeira fase do processo é condizente com os outros dois classificadores referidos anteriormente até à *multi-classe perceptron* invocar o *perceptron* (primal, dual) com os respetivos atributos passados após a criação dos classificadores mediante o número de classes pretendidas. Os número de classificadores corresponde à potencia de 2 elevado ao número de classes subtraindo um, e por fim, novamente subtraindo um a esse resultado, basicamente corresponde a $2^{n-1} - 1$ classes onde n está contido entre $(0 < n \leq 10)$. De seguida cria-se uma matriz *random* de valores entre 0 e 1 para o número de classificadores necessário mediante as classes, por consequência alteramos os valores dessa matriz baseado numa distribuição onde todos os valores superiores a 0.5, são alterados para 1, e todos os outros para -1. Pode-se verificar na Tabela 1 um exemplo da matriz para 4 classes. Posteriormente aplica-se o *fit* aos classificadores das classes correspondentes onde estão identificadas com valor 1. Por fim, aplica-se a previsão dos dados que são retirados através do*predict* dos classificadores, colocando-se numa matriz de previsões. É retornado o melhor classificador depois de calcular a distância euclidiana consoante o *argmin* da matriz de previsões.

Tabela 1: 7 ECOC para 4 classes

Classes	***Code Book***						
	1	**2**	**3**	**4**	**5**	**6**	**7**
C_1	1	−1	−1	1	−1	1	−1
C_2	−1	1	−1	−1	1	1	1
C_3	−1	−1	1	1	1	−1	−1
C_4	1	−1	1	−1	1	−1	1

Por exemplo no classificador 4 separa se em dois grupos $\{C_1, C_3\}$ de $\{C_2, C_4\}$ para o treino.

4.4 Classificadores *Scikit-learn*

Posto isto, ainda foram desenvolvidos três novos *scripts* com o auxílio da livraria *scikit-learn*. Nesta livraria está disponível o algoritmo *perceptron* onde este é um algoritmo de classificação que compartilha a mesma implementação subjacente com o *SGDClassifier*.[6] Para além do *perceptron* esta livraria dispõe dos classificadores OVA, em que a estratégia consiste em ajustar um classificador por classe [7], OVO em que a estratégia consiste em ajustar um classificador por par de classes [8] e ECOC em que a estratégia consiste em representar cada classe com um código binário (uma matriz de zeros e uns).[9]

Cada um dos *scripts* corresponde a uma estratégia diferente e o intuito de se realizar estas novas versões do projeto foi a possível comparação que se pode fazer com o que tinha sido desenvolvido até então. Ambos os *scripts* seguem o mesmo processo, onde o primeiro passo corresponde ao carregamento dos dados (*load* do *dataset digits*), por exemplo, e divisão em dados de treino e dados de teste através da função ***train_ test_ split*** também da livraria *scikit-learn*. De seguida, aplicou-se o classificador ao *perceptron* do *scikit-learn*, como é possível ver no pedaço de código a seguir (exemplo do classificador OVA).

```
OVA = OneVsRestClassifier(Perceptron(max_iter=num_iter, random_state=0))
```

Obtido o resultado do classificador, aplica-se o ***fit*** ao modelo seguido de um ***predict***, calcula-se também o ***score*** para o classificador em questão para se saber o resultado da *acurracy* do modelo. Com o resultado do *predict* é possível calcular a matriz de confusão através da função ***confusion_ matrix***.

```
cmOVA = metrics.confusion_matrix(y_test_thres, predictionsOVA)
```

De seguida, é construído o *heat map* para as diferentes classes existentes. Os resultados obtidos foram, para além do *heat map* e o *score*, o tempo de execução do classificador e um *plot* com os diferentes *scores* para as diferentes classes.

Na secção 5.2, serão analisados os diferentes resultados e comparados os resultados dos nossos modelos com os que foram obtidos com a implementação dos algoritmos da livraria do *scikit-learn*.

4.5 Interface Gráfica

Como extra, assim como a utilização dos classificadores da livraria *scikit-learn* conforme foi explicado na secção 4.4, o grupo decidiu construir uma interface gráfica para proporcionar uma experiência diferente ao utilizador quando este pretender testar os diferentes tipos de classificador multi-classe elaborados, queremos que tenha uma experiência mais agradável desde a seleção dos classificadores, dos parâmetros, das classes, dos conjuntos de dados, entre outros aspetos.

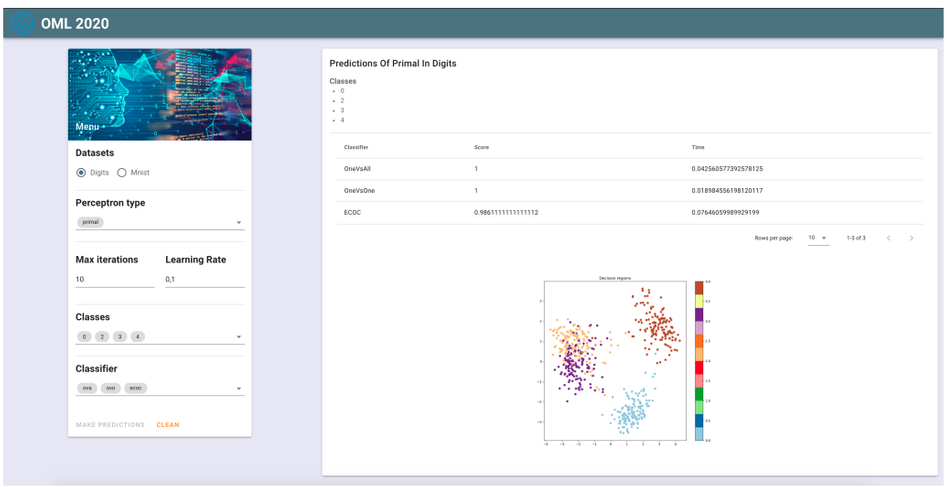


Figura 5: Interface desenvolvida — página principal

Como se pode observar nas figura 5, do lado esquerdo o utilizador pode selecionar o conjunto de dados que quer utilizar para classificar, o tipo de *perceptron*, os parâmetros do *perceptron*, as classes a utilizar e os classificadores que pretende utilizar. Já do lado direito é apresentado o resultado do que foi pedido e clicando num dos resultados dos classificadores, são apresentados resultados mais detalhados tal como está apresentado na figura 6, em que podemos observar a *accuracy*, o tempo de execução, gráficos de convergência dos diferentes classificadores, matriz de confusão e previsões obtidas.

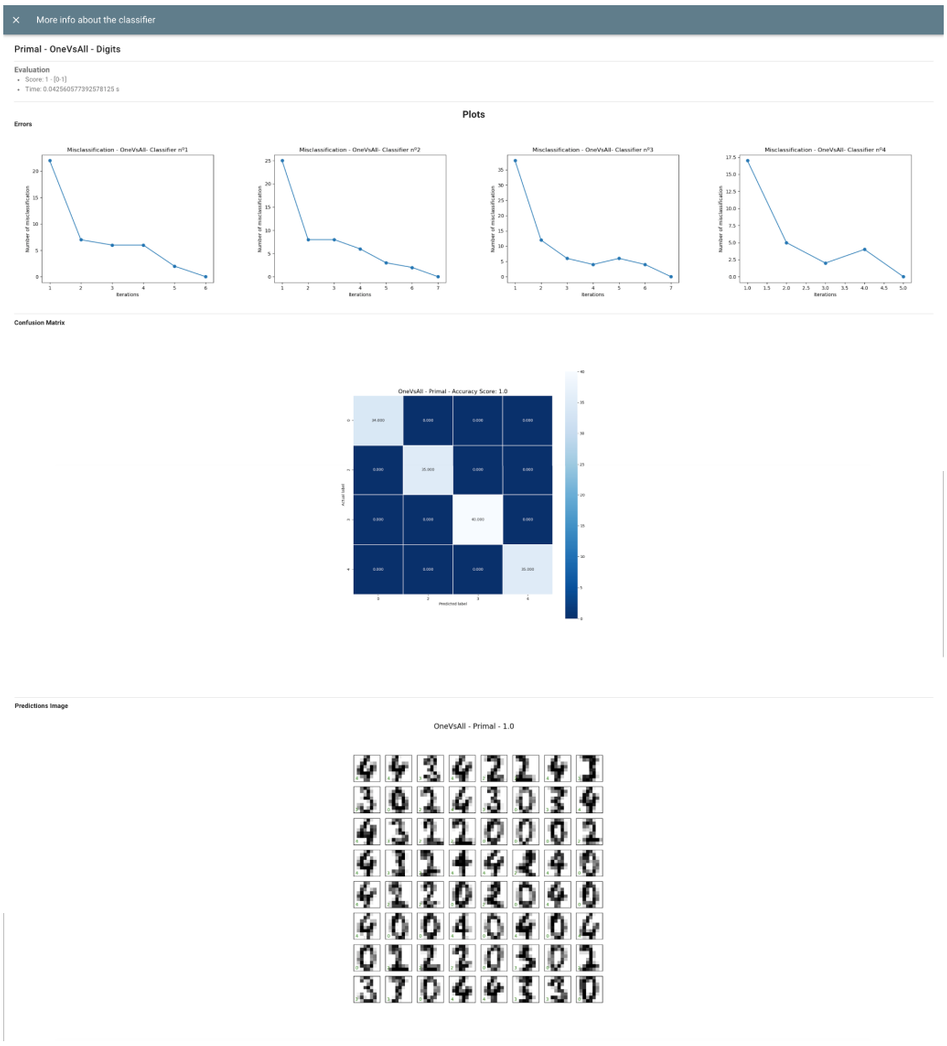


Figura 6: Interface desenvolvida — página de resultados pormenorizados

Esta interface encontra-se *online*, uma vez que foi desenvolvido um servidor para a mesma, que foi copulado na AWS (*Amazon Web Services, Inc.*) que é uma plataforma de serviços de computação em *cloud*, onde é formada uma plataforma de computação na *cloud* oferecida pela *Amazon.com*.[13]

5 Análise de resultados

5.1 Classificadores Multi-Classe

Nesta secção iremos apresentar os resultados obtidos em vários testes efectuados aos classificadores desenvolvidos pelo grupo, bem como as diferentes versões do algoritmo *Perceptron*.

As Tabelas 2, 3, 4, 5 mostram os resultados obtidos da aplicação das diferentes combinações de execução entre as diferentes versões do algoritmo *Perceptron*, os diferentes tipos de classificadores e diferentes grupos de classes. De referir ainda que os testes efectuados foram aplicados a dois *datasets* diferentes, já anteriormente mencionados, e os próximos sub-capítulos corresponderão a cada um dos *datasets*. Iremos observar que os resultados situam-se acima dos 90% onde alguns deles atingem os 100%, o que é bastante bom e mostra o quão bem desenvolvidos estão os nossos algoritmos.

No conteúdo deste relatório apenas apresentamos os resultados de *accuracy* e *time*, no entanto, nos Anexos podem ser observados os gráficos de erro e de *accuracy* correspondestes a cada classificador. É ainda importante referir que também foram determinados os valores das iterações (épocas) para os quais cada classificador demorou a convergir, ou seja, a não fazer uma classificação com erro.

5.1.1 *Dataset Digits*

Tabela 2: Resultados *Perceptron* utilizando o *Dataset Digits* e as classes 0123

Perceptron	**Classificadores**	***Accuracy***	***Time***
Primal	*OneVsAll*	98.6%	0.05s
	OneVsOne	99.3%	0.02s
	Error-Correcting Output Codes	98.6%	0.1s
Dual	*OneVsAll*	98.6%	38.2s
	OneVsOne	99.3%	5.2s
	Error-Correcting Output-Codes	97.2%	67s
Dual com Kernel RBF	*OneVsAll*	96.5%	45.1s
	OneVsOne	99.3%	17.3s
	Error-Correcting Output-Codes	97.9%	71.9s
Dual com Kernel Polinomial	*OneVsAll*	100%	20.3s
	OneVsOne	99.3%	6.3s
	Error-Correcting Output-Codes	100%	35.2s

Tabela 3: Resultados *Perceptron* utilizando o *Dataset Digits* e as classes 3679

Perceptron	**Classificadores**	***Accuracy***	***Time***
Primal	*OneVsAll*	96.5%	0.06s
	OneVsOne	97.2%	0.02s
	Error-Correcting Output Codes	96.5%	0.1s
Dual	*OneVsAll*	96.5%	42.6s
	OneVsOne	97.2%	5.8s
	Error-Correcting Output-Codes	97.2%	79.1s
Dual com Kernel RBF	*OneVsAll*	97.2%	49.5s
	OneVsOne	96.5%	17.6s
	Error-Correcting Output-Codes	97.2%	92.5s
Dual com Kernel Polinomial	*OneVsAll*	98.6%	24.1s
	OneVsOne	97.2%	6.3s
	Error-Correcting Output-Codes	96.5%	41.8s

5.1.2 *Dataset Mnist*

Tabela 4: Resultados *Perceptron* utilizando o *Dataset Mnist* e as classes 0123

Perceptron	**Classificadores**	***Accuracy***	***Time***
Primal	*OneVsAll*	91.2%	0.2s
	OneVsOne	96.8%	0.2s
	Error-Correcting Output Codes	93.1%	0.6s
Dual	*OneVsAll*	91.2%	156.8s
	OneVsOne	96.8%	18.4s
	Error-Correcting Output-Codes	93.7%	285.6s
Dual com Kernel RBF	*OneVsAll*	94.3%	83.5s
	OneVsOne	93.7%	30.3s
	Error-Correcting Output-Codes	95.6%	126.6s
Dual com Kernel Polinomial	*OneVsAll*	96.2%	65.1s
	OneVsOne	93.1%	12.2s
	Error-Correcting Output-Codes	96.8%	76.1s

Tabela 5: Resultados *Perceptron* utilizando o *Dataset Mnist* e as classes 3679

Perceptron	**Classificadores**	***Accuracy***	***Time***
Primal	*OneVsAll*	87.8%	0.09s
	OneVsOne	91.5%	0.05s
	Error-Correcting Output Codes	88.5%	0.1s
Dual	*OneVsAll*	91.5%	146.6s
	OneVsOne	89.1%	24.6s
	Error-Correcting Output-Codes	90.1%	258.3s
Dual com Kernel RBF	*OneVsAll*	90.9%	90.9s
	OneVsOne	92.7%	32.2s
	Error-Correcting Output-Codes	90.3%	147.9s
Dual com Kernel Polinomial	*OneVsAll*	94.5%	98.3s
	OneVsOne	91.5%	17.8s
	Error-Correcting Output-Codes	92.1%	116.9s

5.2 Classificadores *Scikit-learn*

Utilizando estes resultados obtidos para estes classificadores da livraria *Scikit-learn* para comparar com os mesmos obtidos pelos classificadores desenvolvidos, podemos verificar que os resultados obtidos pelos nossos foram superiores tanto a nível de *accuracy* como a nível de tempo, logo conseguem ser superiores a nível de classificação e execução.

Tabela 6: Resultados com *Scikit-learn* e *Perceptron* Primal utilizando o *Dataset Digits* e as classes 0123

Perceptron	**Classificadores**	***Accuracy***	***Time***
Primal	*OneVsAll*	95.7%	0.31s
	OneVsOne	95.7%	0.29s
	Error-Correcting Output Codes	95.0%	0.27s

6 Conclusão

Concluindo, achamos relevante ter em consideração alguns pontos chave deste trabalho. Relativamente à investigação realizada para este trabalho, todo o grupo defende que foi realizada uma investigação preparatória ao longo do projeto, que permitiu que o projeto fosse realizado da melhor forma possível, e que foi detalhada nos primeiros capítulos deste documento.

Quanto ao desenvolvimento, todos os objetivos deste projeto foram alcançados. Achamos ainda que é importante realçar que, para além desses objetivos que foram alcançados, o grupo foi capaz de ser autónomo e desenvolver componentes extras de maneira a complementar o projeto, tais como o teste com dois conjuntos de dados e não apenas um como foi pedido, utilizar as ferramentas da livraria *Scikit-learn* para utilizar os classificadores pré-definidos como termos de comparação com os que foram desenvolvidos pelo grupo e, por último, o desenvolvimento de uma interface gráfica que pode ser consultada *online*, algo que apesar do trabalho que originou traz bastante valor ao projeto.

Por fim, em relação aos resultados apresentados, o grupo encontra-se bastante satisfeito com o trabalho desenvolvido, uma vez que analisando os resultados dos algoritmos do grupo, estes tiveram valores de *accuracy* bastante bons, sempre acima dos 90%, em alguns casos a 100%, e comparando com os classificadores da livraria *Scikit-learn*, os classificadores que o grupo desenvolveu obtiveram melhores resultados. Quanto à comparação de *performance* e tempos de execucação entre os vários classificadores aplicados aos diferentes *Perceptrons*, verificamos que o OVO é o mais eficiente pois é o que tem melhores tempos de execução em todos os casos testados e em maior parte é também aquele que obtém melhores resultados a nível de *accuracy*. Logo, no geral o grupo considera que o trabalho está bastante bom.

Referências

[1] Cho-jui Hsieh. ECS289 : Scalable Machine Learning. 2015.

[2] Daumé III,H., "A Course in Machine Learning,", pp. 41–54,77–81, 141–154, January 2017

[3] Error-Correcting Output Codes, http://www.ccs.neu.edu/home/vip/teach/MLcourse/4_boosting/lecture_notes/ecoc/ecoc.pdf

[4] Pen-Based Recognition of Handwritten Digits Data Set, https://archive.ics.uci.edu/ml/datasets/Pen-Based+Recognition+of+Handwritten+Digits

[5] sklearn.datasets.load_digits, https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html

[6] sklearn.linear_model.Perceptron, https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Perceptron.html

[7] sklearn.multiclass.OneVsRestClassifier, https://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html

[8] sklearn.multiclass.OneVsOneClassifier, https://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsOneClassifier.html

[9] sklearn.multiclass.OutputCodeClassifier, https://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OutputCodeClassifier.html

[10] Principal Component Analysis, https://pt.wikipedia.org/wiki/Análise_de_componentes_principais

[11] MNIST database, https://en.wikipedia.org/wiki/MNIST_database

[12] An Overview of Ensemble Methods for Binary Classifiers in Multi-class Problems: Experimental Study on One-vs-One and One-vs-All Schemes, https://sci2s.ugr.es/ovo-ova#Aggregations%20in%20One-vs-One

[13] Amazon Web Services, Inc. , https://aws.amazon.com/pt/