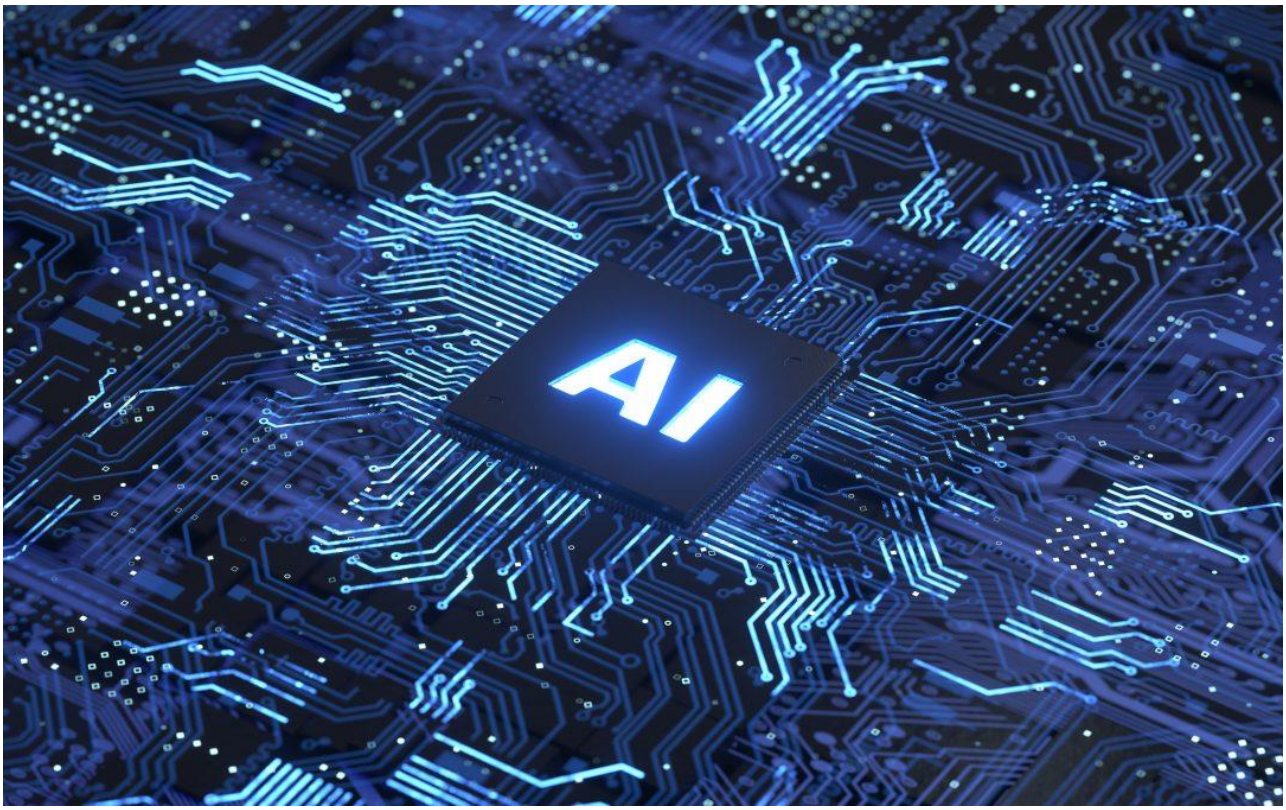
iscte
TECNOLOGIAS
E ARQUITETURA

INTRODUCTION TO MACHINE LEARNING – UNSUPERVISED LEARNING

21 DE DEZEMBRO DE 2020



Nome: Miguel Oliveira

Aluno nº: 96200

Email: mdlco@iscte-iul.pt

Docente: Professor Doutor Luís Nunes

Docente: Professor Doutor Sancho Oliveira

Introdução:

Supervised Learning ou aprendizagem supervisionada é um ramo de *machine learning* em que se tem um conjunto de dados X e o resultado esperado desse conjunto Y e recorre-se a um algoritmo que irá tentar replicar e mapear o comportamento existente nesse conjunto de dados. Depois tendo apenas o conjunto de dados teste, o algoritmo é capaz de prever qual o resultado final.

Este ramo, é assim denominado, pois muito como a relação entre aluno e professor, assemelha-se ao processo de aprendizagem do algoritmo, pois sabemos as respostas corretas e assim, é possível fazer previsões de resultados e ao mesmo tempo ir corrigindo as respostas obtidas, tal como de um professor se tratasse.

Neste relatório serão abordados vários aspetos deste ramo e este encontra-se separado em duas partes. A primeira corresponde à análise de redes neuronais e a segunda parte à analise de KNN e árvores de decisão.

Foi elaborado um programa em *python* para auxiliar na conceção e análise deste problema e, assim, tentar dar resposta às questões colocadas.

Todos os testes e resultados obtidos foram feitos num computador HP EliteBook 840 G6 com um processador i5-8365U, 16GB de RAM e um sistema operativo Windows de 64 bits.

De modo a que os resultados obtidos possam ser replicados, recorreu-se a um sistema de "*seeds*" que garantem a replicação dos resultados.

Para se facilmente executarem os resultados de cada exercício, cada um dos exercícios está num ficheiro *.py* respetivo, pois apesar de se ter replicado código comum aos vários exercícios, é assim mais fácil testar e replicar os resultados dos mesmos.

Cada *.py* conta com uma função indicativa com o nome do exercício respetivo e que funciona como "*main*", onde se podem definir alguns modos de execução do programa, por exemplo, analisar os tempos de execução, ou apenas a média de recompensas por estado.

```
if __name__ == '__main__':
    test_runner(operator_and=False, alphas=[0.1, 0.9], operator_xor=False)
```

Já as variáveis do ambiente, como por exemplo, os valores de alfa e discount terão que ser alterados dentro destas funções "*main*".

operator_and: *True* → operador AND | *False* → operador OR.

operator_xor: *True* → operador XOR | *False* → Outros operadores

alphas: Lista com os valores do Alfa a testar

Neural Networks (Parte 1):

Pretende-se desenvolver um algoritmo, ao qual sejam fornecidos exemplos, bem como os resultados esperados e que consiga emular o comportamento existente nos dados. Para tal, recorreu-se a uma "*network*" representada por um "*perceptron*" que recebe dois dados. Estes, podem ser representados pelas seguintes equações:

$$o = f(w0 + x1\,w1 + x2\,w2)$$
$$f(x) = 1, \text{se } x > 0$$
$$f(x) = 0, \text{se } x <= 0$$

Neste caso podemos escolher entre os operadores, "*AND*", "*OR*" e "*XOR*" recorrendo à construção de um vetor das combinações possíveis de dados para dois bits:

{{0, 0}, {0, 1}, {1, 0}, {1, 1}}

E respetivamente os resultados esperados para cada operador:

AND: {0, 0, 0, 1}, *OR*: {0, 1, 1, 1} *XOR*: {0, 1, 1, 0}

Já os valores de "*wX*" serão todos inicializados com pequenos valores aleatórios entre -1 e 1, e para cada valor de entrada, estes serão atualizados recorrendo ao seguinte processo:

$$\Delta w0 \mathrel{+}= \alpha\,(d - o)$$
$$\Delta w1 \mathrel{+}= \alpha\,x1\,(d - o)$$
$$\Delta w2 \mathrel{+}= \alpha\,x2\,(d - o)$$

Em que "*d*" representa o resultado esperado, "*o*" o resultado obtido e *"α"* um valor arbitrariamente atribuído, que varia consoante o teste (0 a 1). Para que o erro diminua na interação seguinte do algoritmo é necessário, depois, atualizar os valores de "*wX*".

$$w0 \mathrel{+}= \Delta w0$$
$$w1 \mathrel{+}= \Delta w1$$
$$w2 \mathrel{+}= \Delta w2$$

Foi, então, desenvolvido um programa em *Python* para auxiliar na conceção e análise deste problema e, assim, tentar responder às questões colocadas. *Δ α*

Exercício 1:

Neste exercício foram fornecidos ao algoritmo os valores de entrada, anteriormente referidos, e esperava-se obter como resultados, as combinações possíveis para os operadores *AND* e *OR*.

Para efeitos de testes e análise de comportamentos, o valor do alfa foi variado, para ambos os casos, entre 0.1 e 0.9.

Operador *AND*:

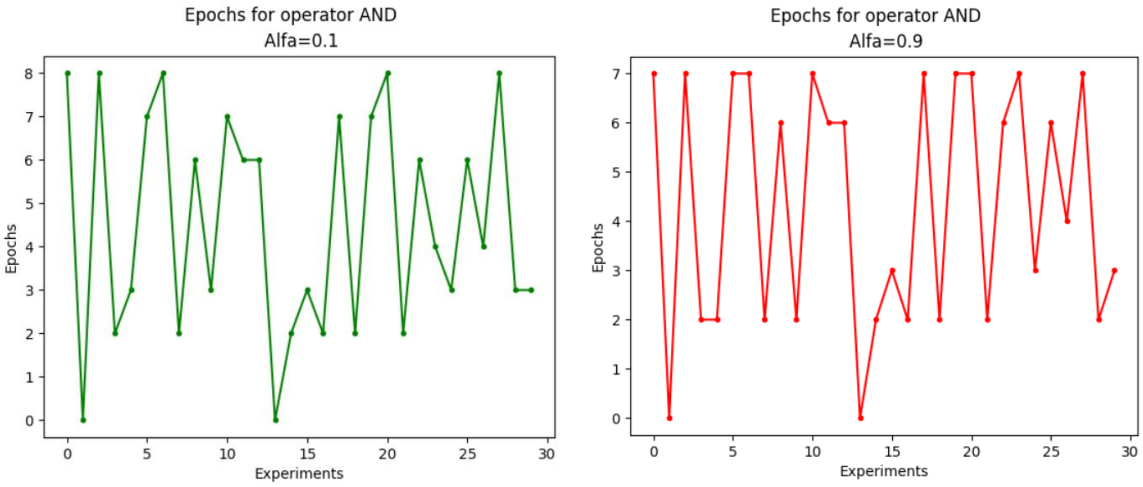


Fig. 1 – Desempenho de acordo com diferentes valores para o alfa

Operador OR:

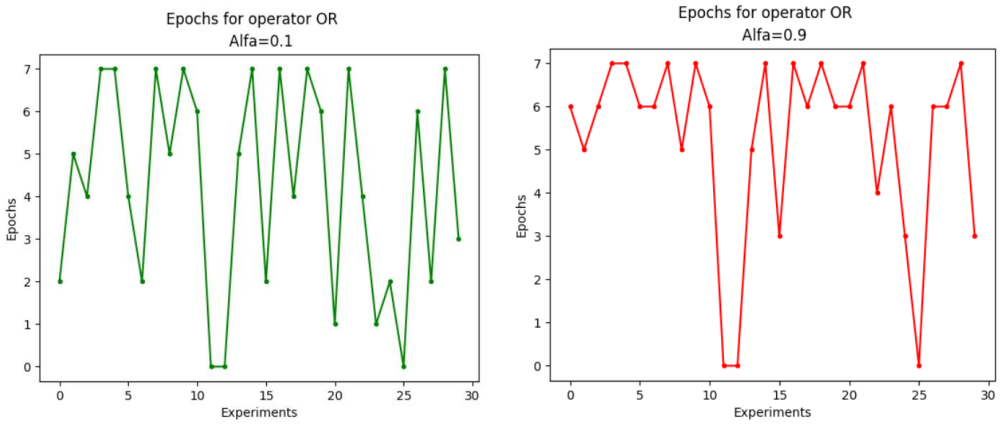


Fig. 2 – Desempenho de acordo com diferentes valores para o alfa

Com os resultados obtidos verificou-se que, para valores mais baixos do alfa, o operador *OR* conseguiu em média melhores resultados de *"epochs"*, enquanto que, para o operador *AND*, isso acontecia apenas quando o alfa tomava valores mais elevados.

Também se verificou que, de uma forma geral, tanto para o operador *AND* como para o operador *OR*, o número de "*epochs"* necessário para alcançar o resultado esperado, foi relativamente baixo, nunca excedendo nove "*epochs"*, ou seja, com esta "*network"* o algoritmo rapidamente conseguia aprender o comportamento inerente aos dados fornecidos para estes operadores.

Exercício 2:

Para o exercício 2 foi analisado o operador *XOR*, e pelos resultados, rapidamente se verificou que o "perceptron" usado para este operador não era adequado para estudar o comportamento do mesmo, pois o "perceptron" traça separações lineares.

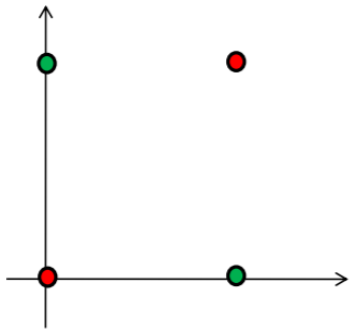


Fig. 5 – Comportamento do operador *XOR*

Assim, não se conseguiram obter dados capazes de medir o desempenho do algoritmo para este operador.

Exercício 3:

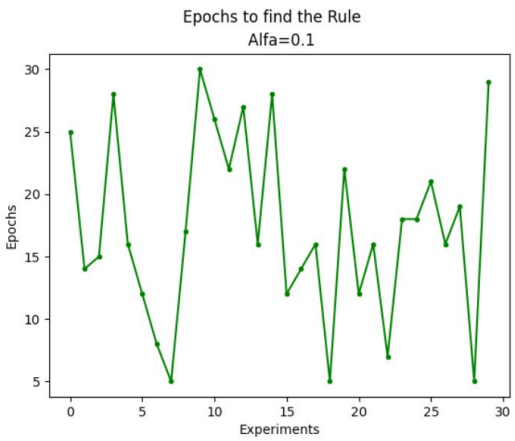
Para este exercício foram adicionadas mais variáveis de entrada, ou seja, em vez das duas usadas até agora, o algoritmo neste exercício usa dez, isto é, desta vez o algoritmo trabalhará com um conjunto de dados com 1024 *inputs* diferentes, ao contrário dos 4 anteriores. Isto envolveu também criar uma regra para assim definir o resultado esperado do programa. Como previsto, das regras criadas nem todas tiveram solução, tal como o caso analisado no exercício anterior do XOR. Contudo, foi concebida e utilizada para este exercício, a seguinte regra:

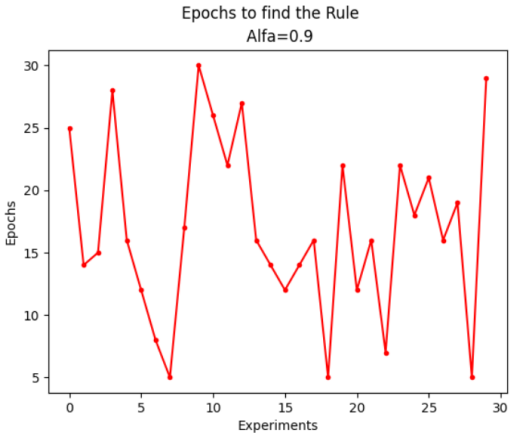
$se\ quantidade\ de\ 1\ na\ combinação\ x \geq 6$
  $retorna\ 1$
$se\ não$
  $retorna\ 0$

Com esta regra a distribuição de "1" e "0" no resultado esperado é:

Quantidade de 1: 386 (37.7%) | Quantidade de 0: 638 (62.3%) em 1024.

Posto isto, o algoritmo foi adaptado para trabalhar com as novas variáveis. Porém, a lógica de funcionamento continua a ser igual aos casos analisados anteriormente, através do "*perceptron*", e assim, obtiveram-se os seguintes resultados:

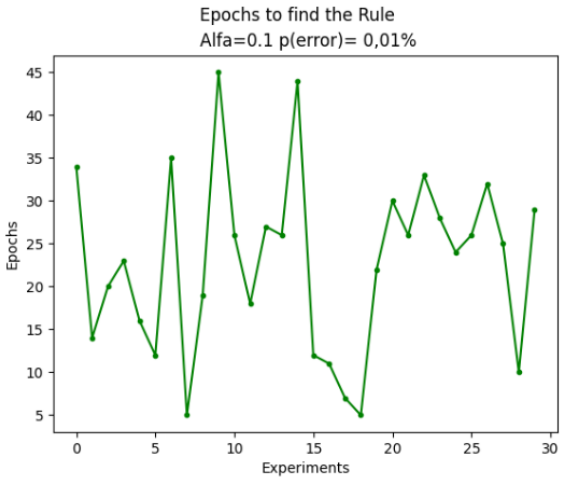


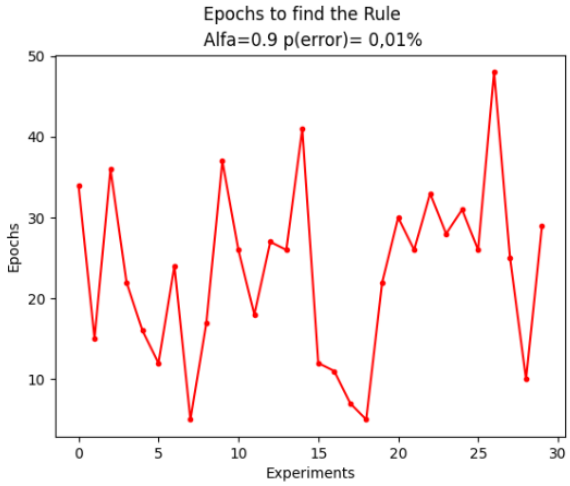


De seguida foi também introduzido o fator "erro", ou seja, com este fator, existe uma probabilidade de o valor retornado pelo *perceptron* seja diferente do "real", para que assim se possa testar o desempenho do algoritmo mediantes estas adversidades.

Para que a exigência computacional não aumente exponencialmente ao ponto de tornar a reprodução dos testes demasiado longa, foi apenas definida uma probabilidade de erro de 0.001%, isto é, 99,99% das vezes o resultado do *perceptron* será o "correto".

Eis os resultados obtidos:





Após a análise dos resultados, é percetível que o algoritmo consegue, de facto, aprender o comportamento inerente aos dados que lhe foram agora fornecidos e, assim, encontrar sempre o resultado esperado. É de notar também, que, desta vez, são precisas mais *epochs* até se alcançar o resultado, no entanto, o seu aumento parece proporcional ao aumento de variáveis, e assim esperado.

De notar também que a introdução de erro no sistema, ainda que com um probabilidade consideravelmente baixa, aumentou significativamente o número de *epochs* necessárias para se atingir o objetivo, mostrando que o algoritmo não está muito preparado para lidar com situações adversas.

KNN e Decision Trees (Parte 2):

Nesta parte irá ser usado como base dos exercícios e programas desenvolvidos o conjunto de dados de teste desenvolvido no último exercício da parte 1, ou seja, o conjunto de dados com todas as combinações de um possível operados com 10 variáveis, que resulta num conjunto de dados com 1024 elementos.

Exercício 1:

Para este exercício foi implementado o algoritmo KNN adaptado ao conjunto de dados que temos. Este algoritmo pode ser usado para classificação ou regressão, neste caso será para classificação. Para funcionar, o algoritmo requer um conjunto de dados de treino, onde irá aprender a analisar os dados e por fim um conjunto de dados de teste onde se irá avaliar a precisão do mesmo.

O algoritmo de um KNN *classifier* consiste no seguinte processo:

- **Passo-1:** Selecionar o número de K de vizinhos.
- **Passo-2**: Calcular a distância euclidiana a cada um dos vizinhos.
- **Passo-3**: Adicionar a distância e a sua posição a uma lista ordenada.
- **Passo-3:** Ordenar a lista de distância mais curta para maior.
- **Passo-4:** Retirar da lista ordenada os primeiros K elementos.
- **Passo-5:** Aplicar a moda à lista dos K elementos e assim obter a previsão para aquele caso.

A escolha do valor de K é crucial, pois é o que vai determinar a precisão do algoritmo bem como o seu desempenho. Assim sendo o valor de K deve ser grande para ter em conta mais vizinhos, mas é preciso ter em conta que isso também irá aumentar o tempo de execução do algoritmo. Ao mesmo tempo, o valor de K também não deve ser par, pois pode calhar que para uma dada previsão as distâncias entre dois conjuntos de dados podem ser iguais impedindo o algoritmo de tomar uma decisão.

Assim do conjunto de dados da parte 1 do exercício 3, foram retirados os aleatoriamente 70% dos elementos e usados como sendo o conjunto de treino. Os restantes 30% foram usados como conjunto de teste.

Neste sentido o algoritmo será encarregue de prever a classificação do conjunto de teste. Tais classificações são atribuídas segundo a regra desenvolvida no exercício 3 da parte 1, ou seja, ao gerar o conjunto de dados de treino, é aplicada a regra a cada um dos seus elementos para assim obter a sua classificação esperada, por exemplo para o dado {0,0,0,0,0,0,0,0,0,0} $\rightarrow$ 0.

Para efeitos de teste, foi testada a precisão do algoritmo para valores de K igual a 3, 7 e 11 e recorre-se à fórmula da moda para depois fazer a decisão da previsão de entre os K elementos obtidos no fim da execução do algoritmo.

K	*Previsões Corretas*	*Total*	*Percentagem*
3	225	308	73%
7	228	308	74%
11	266	308	86%

Analisando os dados conclui-se que no geral quanto maior é o valor de K, maior é a precisão do algoritmo na sua capacidade de classificar os dados fornecidos.

Exercício 2:

Neste exercício continua-se a usar o conjunto de dados obtido no exercício 3 da parte 1, contudo desta vez, os dados são divididos segundo o valor da primeira coluna, ou seja, todos os valores que tenham 0 vão para o um subconjunto e os que têm 1 vão para outro. Assim obtêm se 3 conjuntos, o conjunto 1, que contem todos os dados do conjunto de dados, o conjunto 2 que contem todos os valores cuja o valor da primeira coluna é 0 e o conjunto 3 e que contem todos os valores cuja o valor da primeira coluna é 1.

Este processo repete-se para cada uma das 10 colunas dos elementos do conjunto de dados, pois cada uma representa uma das dez variáveis. De seguida é calculada a entropia de cada um destes conjuntos recorrendo à seguinte fórmula:

$$\text{entropy}(S) = -(\text{p}+) * \log(\text{p})(\text{p})\log(\text{p})$$

	1	*2*	*3*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Data set	0.956	0.956	0.956	0.956	0.956	0.956	0.956	0.956	0.956	0.956	0.956
Data set 2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Data set 2	0.817	0.817	0.817	0.817	0.817	0.817	0.817	0.817	0.817	0.817	0.817

Posteriormente, é calculado o ganho da separação do conjunto por variável, e para isso usa-se a seguinte fórmula:

$$G(S,a) = entropy(S) - \sum_v(|S|Entropy(S_v))/|S|$$

Onde |S| é o número de elementos de um dado conjunto S e S_v o subconjuntos obtidos a partir de S após a separação segundo uma dada variável.

Assim obtiveram-se os seguintes ganhos:

	1	*2*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Ganho	0.047	0.047	0.047	0.047	0.047	0.047	0.047	0.047	0.047	0.047	0.047

Analisando os resultados obtidos, e devido à igualdade dos valores obtidos em todas as *features* para cada um dos subconjuntos, não é possível indicar que exista uma Feature em particular capaz de influenciar a classificação do conjunto de dados, também se nota que devido ao valor da entropia para o subconjunto 1 ser sempre 1, leva a que este não seja levado em consideração na hora de avaliar o conjunto de dados, isto significa que nestes casos não é possível prever o resultado de uma dade *feature* e o impacto que tem no conjunto de dados, por outras palavras estas features não seriam valorizadas numa *decision tree*.

Já quanto aos ganhos, e uma vez que todas as features obtiveram resultados iguais, não possível indicar qual delas teriam prevalência em relação a outra, na construção de uma *decision tree*, pois sendo os valores todos iguais, seria um indiferente a escolha de uma ou outra *feature*, sendo difícil indicar uma pela qual se poderia dividir para obter valores de entropia mais baixos.

Assim sendo, a *decision tree* teria o papel de tentar prever a avaliação do conjunto de dados. Para que isto aconteça a árvore procura sempre dividir o conjunto de dados, pelas *features* que se mostrem mais promissoras, isto é, com melhores resultados, calculando o ganho a cada divisão para assim escolher sempre as melhores *features*.