Inteligência Artificial

Thiago Henrique Leite da Silva, RA: 139920

AULA10: Exercício teórico aprendizado de máquina

1) (0,3) Seja o seguinte dataset:

Nome	Febre	Enjôo	Manchas	Dores	Diagnóstico
João	Sim	Sim	Pequenas	Sim	Doente
Pedro	Não	Não	Grandes	Não	Saudável
Maria	Sim	Sim	Pequenas	Não	Saudável
José	Sim	Não	Grandes	Sim	Doente
Ana	Sim	Não	Pequenas	Sim	Saudável
Leila	Não	Não	Grandes	Sim	Doente

Ensinar uma rede do tipo Perceptron a distinguir:

- Pacientes potencialmente saudáveis

- Pacientes potencialmente doentes

Testar a rede para novos casos:

- (Luis, não, não, pequenas, sim)

- (Laura, sim, sim, grandes, sim)

Resolva o problema acima usando: Inicializar todos os pesos com o valor nulo e considerar η=0.5, limiar = -0,5. Considere 1 para 'sim'/'grandes' e 0 para 'não'/'pequenas' no dataset .

Para testar a rede para os novos casos, implementei o algoritmo do Perceptron em ruby, onde conseguimos inserir nosso dataset, valores para o bias/limiar, taxa de aprendizagem, testes e os pesos iniciais. A partir daí, o algoritmo irá realizar o treinamento com o dataset até os valores dos pesos convergirem, enquanto não convergir, continuamos atualizando os pesos conforme a função de atualização vista em aula.

Detalhando um pouco mais a solução, primeiramente definimos o peso w0 conforme o exemplo da aula, onde w0 recebe o inverso do limiar. No treinamento do algoritmo, percorremos todo o dataset realizando os seguintes métodos:

- Calculamos a saída da função de ativação para o indivíduo i
- Definimos sua classe com base na saída: -1 (doente) se a saída for menor que zero, caso contrário classificamos como 1 (saudável).
- Atualizamos os pesos se a classe obtida for diferente da esperada.

Terminado a passagem por todo o dataset, verificamos se ele convergiu, ou seja, passaremos novamente por todo o dataset verificando se, com os pesos atualizados, todos os elementos estão obtendo a saída adequada. Em caso positivo, finalizamos o treinamento, caso contrário, percorremos

novamente o dataset atualizando os pesos. Fazemos isso até que alcançarmos a conversão, ou um número k de iterações definida pelo programador, caso não haja conversão.

Após estarmos com o algoritmo treinado, podemos rodar os casos de teste informados, onde basicamente colocamos os dados fornecidos na função de ativação com os pesos resultantes da fase de aprendizagem.

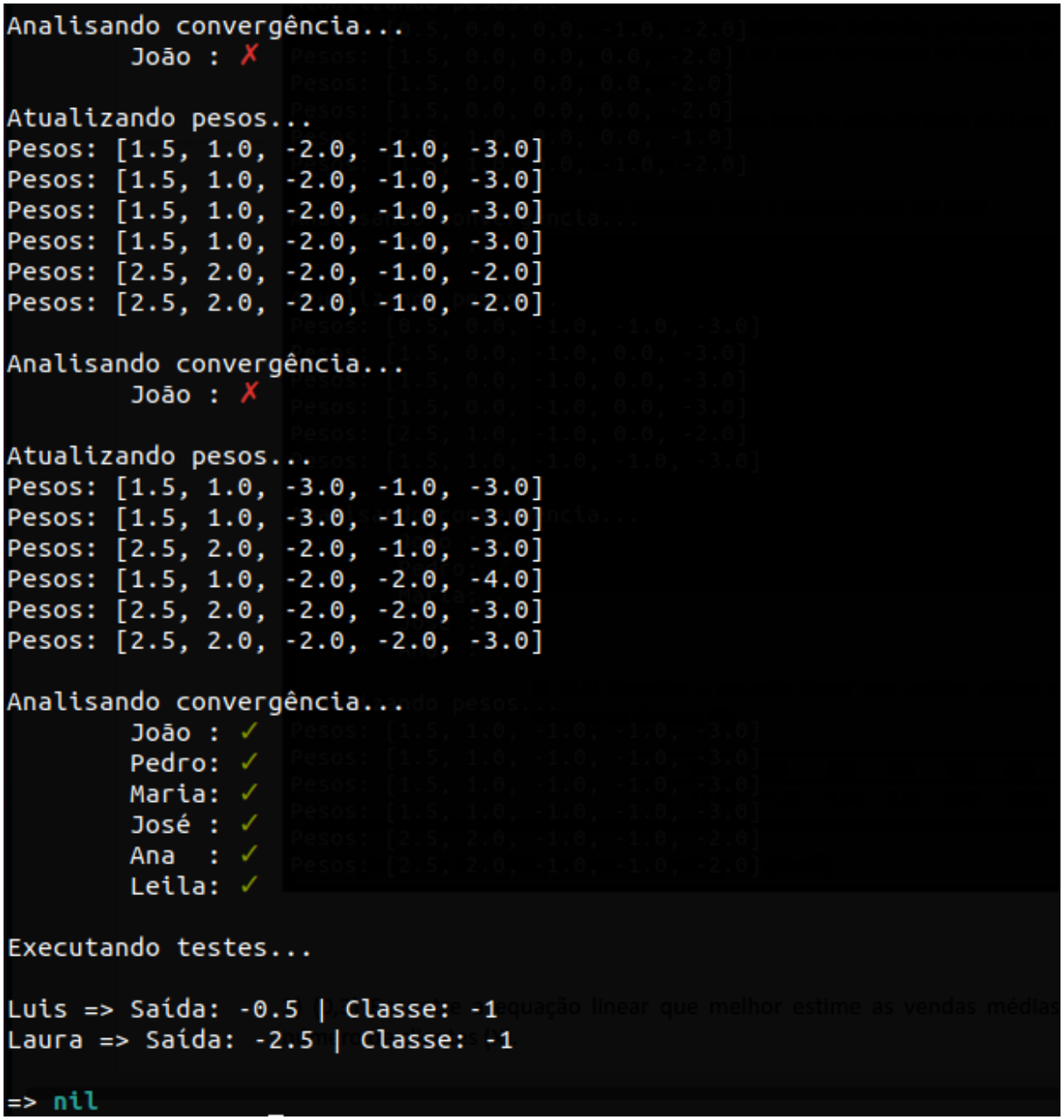
Por fim, classificamos com base na saída do teste qual sua classe equivalente.

Segue script em anexo.

Código em execução para o exemplo visto em aula:

Executando código para o dataset do exercício:

```
[9] pry(main)> Perceptron.perform(entrada, -0.5, 0.5, [0, 0, 0, 0], testes)

Treinando data set...

Pesos Iniciais: [0.5, 0, 0, 0, 0]

Analisando convergência...
        João : ✗

Atualizando pesos...
Pesos: [-0.5, -1.0, -1.0, 0.0, -1.0]
Pesos: [0.5, -1.0, -1.0, 1.0, -1.0]
Pesos: [1.5, 0.0, 0.0, 1.0, -1.0]
Pesos: [0.5, -1.0, 0.0, 0.0, -2.0]
Pesos: [1.5, 0.0, 0.0, 0.0, -1.0]
Pesos: [0.5, 0.0, 0.0, -1.0, -2.0]

Analisando convergência...
        João : ✓
        Pedro: ✗

Atualizando pesos...
Pesos: [0.5, 0.0, 0.0, -1.0, -2.0]
Pesos: [1.5, 0.0, 0.0, 0.0, -2.0]
Pesos: [1.5, 0.0, 0.0, 0.0, -2.0]
Pesos: [1.5, 0.0, 0.0, 0.0, -2.0]
Pesos: [2.5, 1.0, 0.0, 0.0, -1.0]
Pesos: [1.5, 1.0, 0.0, -1.0, -2.0]

Analisando convergência...
        João : ✗

Atualizando pesos...
Pesos: [0.5, 0.0, -1.0, -1.0, -3.0]
Pesos: [1.5, 0.0, -1.0, 0.0, -3.0]
Pesos: [1.5, 0.0, -1.0, 0.0, -3.0]
Pesos: [1.5, 0.0, -1.0, 0.0, -3.0]
Pesos: [2.5, 1.0, -1.0, 0.0, -2.0]
Pesos: [1.5, 1.0, -1.0, -1.0, -3.0]

Analisando convergência...
        João : ✓
        Pedro: ✓
        Maria: ✓
        José : ✓
        Ana  : ✗

Atualizando pesos...
Pesos: [1.5, 1.0, -1.0, -1.0, -3.0]
Pesos: [1.5, 1.0, -1.0, -1.0, -3.0]
Pesos: [1.5, 1.0, -1.0, -1.0, -3.0]
Pesos: [1.5, 1.0, -1.0, -1.0, -3.0]
Pesos: [2.5, 2.0, -1.0, -1.0, -2.0]
Pesos: [2.5, 2.0, -1.0, -1.0, -2.0]
```



Portanto, a saída obtida foi que tanto Luís, quanto Laura estão doentes de acordo com a classificação do dataset. Analisando a olhos humanos, pode ser que estamos gerando um falso positivo, já que Luís pode não estar doente, isso acontece pela quantidade pequena de entradas no nosso dataset.

2) (0,3) Encontre a equação linear que melhor estime as vendas médias semanais (Y) dado um número de clientes (X).

X	907	926	506	741	789	889	874	510	529	420
Y	11,20	11,05	6,84	9,21	9,42	10,08	9,45	6,73	7,24	6,12
X	679	872	924	607	452	729	794	844	1010	621
Y	7,63	9,43	9,46	7,64	6,92	8,95	9,33	10,23	11,77	7,41

Para encontrar a função que melhor estime as vendas médias semanais, com base nos dados de entrada acima, implementei um algoritmo de regressão linear. Com ela conseguimos encontrar uma função do tipo:

$$Y = a + bX$$

Onde **a** e **b** são os coeficientes que queremos encontrar. Eles são encontrados da seguinte forma:

$$a = \text{Média de Y} - (b * \text{Média de X})$$

$$b = \Sigma\, x[i] * (y[i] - \text{Média de Y}) / \Sigma\, x[i] * (x[i] - \text{Média de X})$$

O algoritmo implementado efetua estes cálculos com bases nos valores de X e de Y informados pelo usuário.

Segue script em anexo.

Na execução do código, iremos mostrar o resultado obtido com os dados utilizados em aula para provar a corretude, e posteriormente rodaremos o algoritmo com os dados do exercício para encontrar a solução.

Exemplo visto em aula:

x_values = [139, 126, 90, 144, 163, 136, 61, 62, 41, 120]
y_values = [122, 114, 86, 134, 146, 107, 68, 117, 71, 98]

```
[28] pry(main)> x_values = [139, 126, 90, 144, 163, 136, 61, 62, 41, 120];
[29] pry(main)> y_values = [122, 114, 86, 134, 146, 107, 68, 117, 71, 98];
[30] pry(main)>
[31] pry(main)> LinearRegression.perform(x_values, y_values)

        Valores X: [139, 126, 90, 144, 163, 136, 61, 62, 41, 120]
        Valores Y: [122, 114, 86, 134, 146, 107, 68, 117, 71, 98]

        Função: y = 0.4954 x + 52.6977

=> nil
```

Exercício:

x_values = [907, 926, 506, 741, 789, 889, 874, 510, 529, 420, 679, 872, 924, 607, 452, 729, 794, 844, 1010, 621]
y_values = [11.20, 11.05, 6.84, 9.21, 9.42, 10.08, 9.45, 6.73, 7.24, 6.12, 7.63, 9.43, 9.46, 7.64, 6.92, 8.95, 9.33, 10.23, 11.77, 7.41]

```
[38] pry(main)> x_values = [907, 926, 506, 741, 789, 889, 874, 510, 529
[39] pry(main)> y_values = [11.20, 11.05, 6.84, 9.21, 9.42, 10.08, 9.45
;

[40] pry(main)> LinearRegression.perform(x_values, y_values)

        Valores X: [907, 926, 506, 741, 789, 889, 874, 510, 529, 420,
        Valores Y: [11.2, 11.05, 6.84, 9.21, 9.42, 10.08, 9.45, 6.73,

        Função: y = 0.0087 x + 2.4445

=> nil
```

Portanto, a equação linear que melhor estima as vendas médias semanais (Y) é dada por:

$$Y = 0.0087 * X + 2.4445$$

Ou seja,

Vendas médias semanais = 0.0087 * número de clientes + 2.4445

Após encontrada a resposta, fiquei curioso em saber qual estava sendo a média de erro dos valores reais de Y para os valores de X informados, em relação aos valores de Y obtidos pela função que encontrei.

Para sanar esta dúvida, implementei um método para tirarmos a prova real se a função é realmente uma boa aproximação para estes valores de entrada.

Na prova real, o que fazemos é calcular o valor de Y agora utilizando a função encontrada, posteriormente, calculamos a diferença do Y esperado (fornecido no exercício) pelo Y obtido (através da função), este valor é nosso valor de erro, portanto, inserimos ele em um vetor.

Calculado todos os valores de Y, mostramos quais foram os valores de erro, o maior erro obtido e a média dos erros levando em consideração seu valor absoluto (módulo). Além disso mostramos para o usuário quais foram os valores de Y obtidos pela sua função.

O código ficou desta maneira:

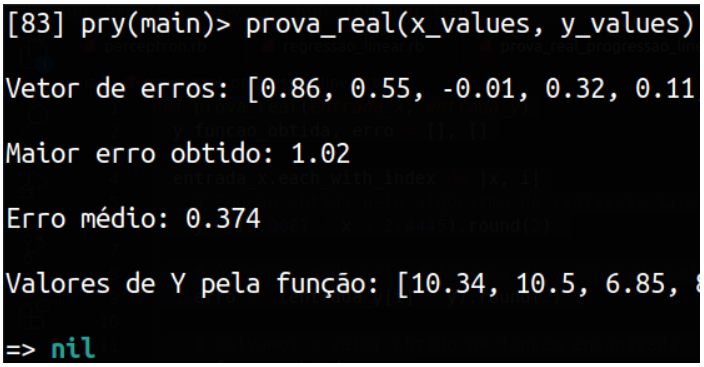
prova_real_progressao_linear.rb

```ruby
def prova_real(entrada_x, entrada_y)
  y_funcao_obtida, erro = [], []

  entrada_x.each_with_index do |x, i|
    # Função obtida pelo algoritmo de regressão linear
    y = (0.0087 * x + 2.4445).round(2)

    # O erro é dado pela diferença do y esperado pelo y obtido
    erro << (entrada_y[i] - y).round(2)

    # Salvamos o valor obtido na função encontrada
    y_funcao_obtida << y
  end

  puts ""
  puts "Vetor de erros: #{erro.inspect}"
  puts ""
  puts "Maior erro obtido: #{erro.map(&:abs).sort.last}"
  puts ""
  puts "Erro médio: #{average(erro)}"
  puts ""
  puts "Valores de Y pela função: #{y_funcao_obtida.inspect}"
  puts ""
end

def average(array)
  soma_total = 0
  array.each{ |n| soma_total += n.abs }
  soma_total.to_f / array.size
end
```

Resultado da execução:

```
[83] pry(main)> prova_real(x_values, y_values)

Vetor de erros: [0.86, 0.55, -0.01, 0.32, 0.11, -0.1, -0.6, -0.15, 0.19, 0.02, -0.72, -0.6, -1.02, -0.09, 0.54, 0.16, -0.02, 0.44, 0.54, -0.44]

Maior erro obtido: 1.02

Erro médio: 0.374

Valores de Y pela função: [10.34, 10.5, 6.85, 8.89, 9.31, 10.18, 10.05, 6.88, 7.05, 6.1, 8.35, 10.03, 10.48, 7.73, 6.38, 8.79, 9.35, 9.79, 11.23, 7.85]

=> nil
```

Saída ampliada:



3) (0,4) Pesquise quais dos algoritmos supervisionados vistos em aula para classificação de dados (KNN, Naive Bayes, Árvores de Decisão, SVM, MLP) podem ser adaptados para regressão e como funcionariam nesse caso.

KNN: O Algoritmo Knn pode sim ser adaptado a regressão linear, de forma que aproximaríamos a associação entre as variáveis independentes e o resultado contínuo através da média das observações na mesma vizinhança. O tamanho da vizinhança é definido pelo programador de forma a minimizar o erro médio. Segundo a pesquisa, o Knn para regressão é bastante eficiente, mas quando aumentamos as variáveis independentes, ele deixa de ser interessante. Agora se tratamento da parte prática da adequação deste algoritmo pra regressão, ao invés de utilizarmos os K vizinhos mais próximos para pegar a maior ocorrência de uma certa classe, atribuiríamos um valor inteiro para cada uma das classes dos nossos K vizinhos, e faríamos uma média ponderada pelo peso das distâncias, gerando uma função da seguinte forma:

$$f(t) = \frac{\sum_{i=0}^{K} v_i p_i}{\sum p_i}$$

Sendo que o peso P(i) é obtido assim:

$$p(x,t) = \frac{1}{\sum_{i=0}^{n} (x_i - t_i)^2}$$

Naive Bayes: O Algoritmo Naive Bayes até pode ser adaptado para regressão linear, mas seus resultados para esse tipo de problema são muito ruins, definitivamente não é uma boa opção utilizá-lo para regressão. Encontrei um artigo que fez um experimento com o Naive Bayes na solução de problemas de regressão, e a conclusão que chegaram foi que ele só deve ser aplicado a esses problemas de regressão quando o pressuposto de independência é valido. Ele performa melhor para regressão logística do que linear.

Árvores de Decisão: Para usar as Árvores de Decisão para regressão, nós precisamos levar em conta duas medidas de impureza, que são os mínimos quadrados, onde as divisões são escolhidas a soma dos quadrados; e o desvio mínimo absoluto, que minimiza o desvio absoluto médio da mediana dentro de um nó. O SKlearn tem até mesmo uma classe chamada DecisionTreeRegressor que faz exatamente

este trabalho, cada nó folha possui uma constante ou uma equação para o seu valor previsto; nestas árvores de regressão a podagem será em função da complexidade do erro mínimo. Em resumo, lendo o material dos links abaixo, pude notar que as árvores de decisão é um bom algoritmo na aplicação da regressão.

SVM: O SVM também pode ser adaptado a regressão, sendo conhecido como SVR (Support Vector Regression), nesta adaptação nós mantemos as principais propriedades e características do SVM, utilizando os mesmos princípios na hora da classificação, o que irá mudar é que teremos que determinar uma margem de tolerância, pois como provavelmente vamos trabalhar com saídas numéricas, definir essa margem é uma forma de impedir as infinitas possibilidades. Além dessa adaptação, existem algumas outras questões um pouquinho mais complicadas para implementarmos a regressão seguindo a mesma ideia: minimizar o erro, individualizando o hiperplano que maximiza a margem.

MLP: Este é mais um algoritmo que já possui uma adaptação implementada no SKLearn, o MLPRegressor que implementa um perceptron multicamadas e que treina usando um conceito chamado de retro propagação sem função de ativação na camada de saída, ele usa o erro quadrado como a função de perda e a saída é um conjunto de valores contínuos, conforme fala na documentação. Como o MLP é um aproximador de função universal, e apesar do perceptron realizar a classificação binária, os neurônios de um MLP, dependendo da função de ativação, podem ser utilizados tanto pra classificação quanto para regressão.

Fontes Questão 03:

- https://bookdown.org/tpinto_home/Regression-and-Classification/k-nearest-neighbours-regression.html
- https://profes.com.br/julio.c.p.rocha/blog/classificacao-e-regressao-com-k-nearest-neighbors
- https://link.springer.com/content/pdf/10.1023%2FA%3A1007670802811.pdf
- https://heartbeat.fritz.ai/implementing-regression-using-a-decision-tree-and-scikit-learn-ac98552b43d7
- https://arifromadhan19.medium.com/regrssion-in-decision-tree-a-step-by-step-cart-classification-and-regression-tree-196c6ac9711e
- https://www.saedsayad.com/support_vector_machine_reg.htm
- https://scikit-learn.org/stable/modules/neural_networks_supervised.html
- https://en.wikipedia.org/wiki/Multilayer_perceptron