MAC0460 – EP3

DCC / IME-USP — Primeiro semestre de 2019

Entrega até: 15/04 — Este EP pode ser feito em dupla

O objetivo do EP3 é implementar o algoritmo da regressão logística, usando a técnica de gradiente descendente para otimização (Lecture 09 / capítulo 3 do Mostafa).

O que entregar:

- um arquivo `ep3.py` contendo uma função para encontrar o vetor de pesos (*fitting*) e outra para fazer a predição (*prediction*). Detalhes das funções estão mais abaixo.
- um relatório simples descrevendo a sua implementação, listando as fontes consultadas, e comentando os testes realizados e os resultados observados
- Todo e qualquer código que escreveu/usou para testar as duas funções
- Uma entrega por dupla

Especificação das duas funções:

1. função de *Fitting*: use obrigatoriamente o seguinte protótipo

   ```
   def logistic_fit( X, y, w = None, batch_size = None, learning_rate = 1e-2,
                     num_iterations = 1000, return_history = False )
   ```

   Entrada:

   - `X`: array 2D de dimensões $N \times d$ contendo N amostras (pontos) $\mathbf{x} \in \mathbb{R}^d$, $d > 0$
   - `y`: array 1D de tamanho N, contendo os *labels* (-1 se negativo e +1 se positivo) correspondentes a cada amostra $\mathbf{x}$
   - `w`: array 1D de tamanho $d + 1$, correspondendo a um vetor de pesos inicial; se `None`, deve ser inicializado com valores aleatórios
   - `batch_size`: quantidade de pontos usados para se fazer um *update* no vetor de pesos; se `None` ou maior que N, deve ser igualado a N (note que `batch_size=1` corresponde ao chamado *stochastic gradient descent*)
   - `learning_rate`: número real, um dos parâmetros do *gradient descent*
   - `num_iterations`: inteiro positivo (quantidade de vezes que o conjunto inteiro X é percorrido)
   - `return_history`: booleano (ver descrição da Saída)

   Saída:

   - array 1D de tamanho $d + 1$ correspondendo ao vetor de pesos $\mathbf{w}$ ao final das iterações
   - somente quando `return_history==True`: lista contendo o valor da função custo a cada *update*, desde o correspondente ao vetor de pesos inicial, antes do primeiro *update*, até ao peso após o último *update*.

2. função de *Prediction*: use obrigatoriamente o seguinte protótipo

   `def logistic_predict(X, w)`

   Entrada:

   - `X`: array 2D de dimensões $N \times d$, contendo N amostras (pontos) $\mathbf{x} \in \mathbb{R}^d$, $d > 0$
   - `w`: array 1D de tamanho $d + 1$, correspondente a um vetor de pesos $\mathbf{w}$

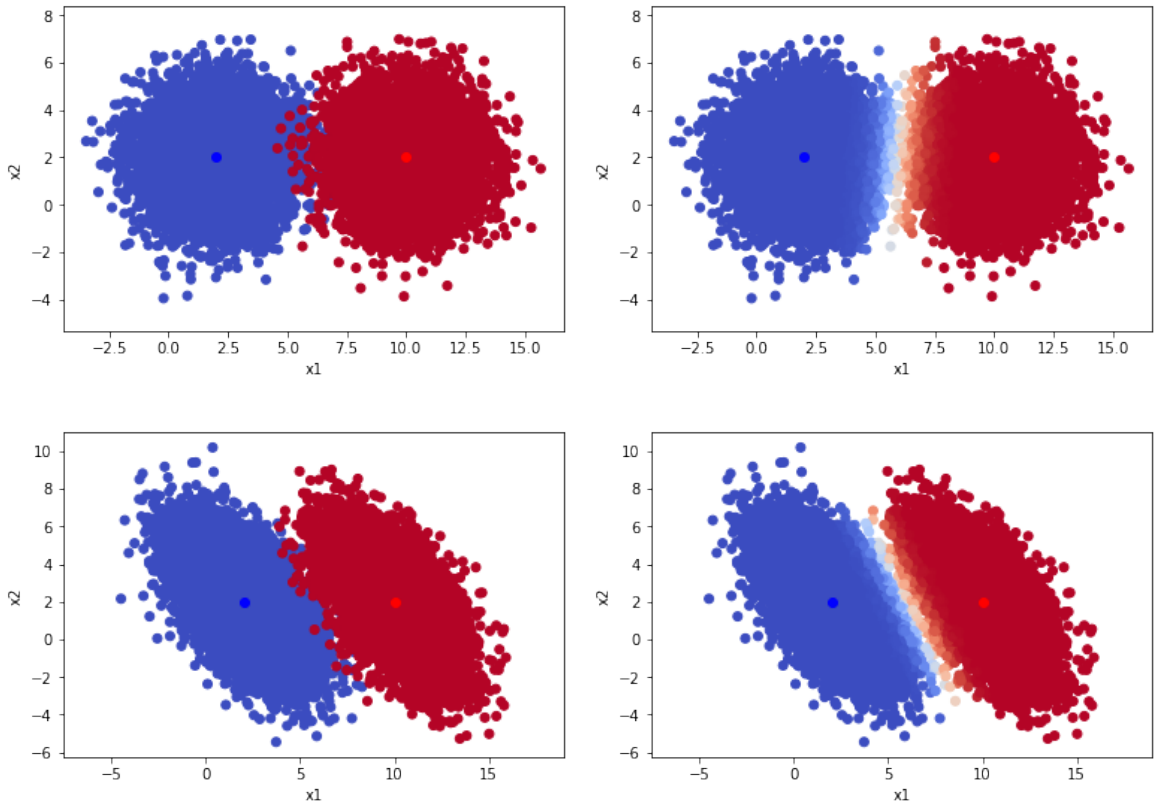
   Saída:

   - um array 1D de tamanho N com as predições (i.e., $\hat{P}(y = +1|\mathbf{x}) = sigmoid(w^T\mathbf{x})$) para cada $\mathbf{x}$

Como testar o seu algoritmo: gere dois aglomerados de pontos, de acordo com uma mesma distribuição normal multivariada[1]. Para testar o seu algoritmo, varie os parâmetros da distribuição, a quantidade de amostras (total e por classes), e o grau de sobreposição das classes, e verifique as superfícies de separação geradas. Varie também o `batch_size`, e outros hiperparâmetros do algoritmo. O efeito deles pode ser examinado, por exemplo, plotando-se a evolução do custo ao longo das iterações.

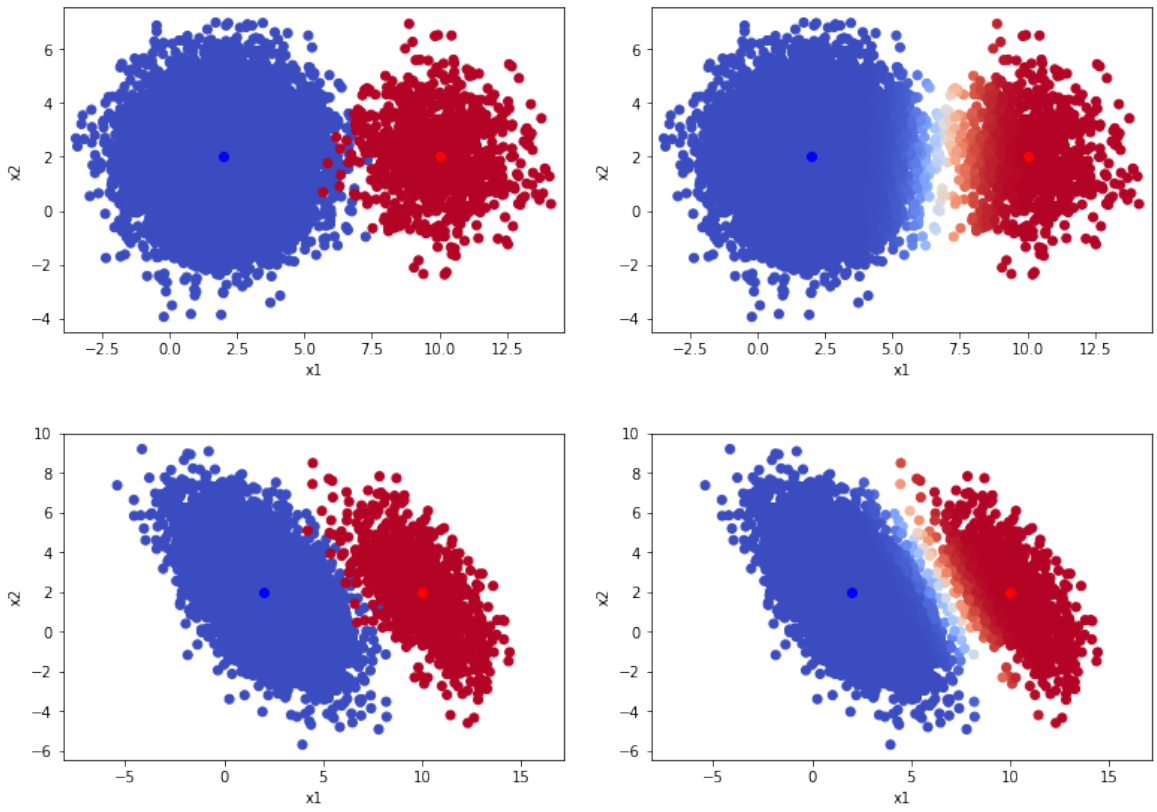
De forma geral, se as duas classes seguem uma mesma distribuição normal, onde deveria estar a superfície de decisão ótima? O seu algoritmo está conseguindo encontrar aproximadamente essa superfície ótima ?

As figuras abaixo mostram duas classes de amostras; ao lado esquerdo as amostras iniciais e ao lado direito uma indicação da separação obtida. A quantidade de amostras na classe azul é a mesma da quantidade na classe vermelha. As amostras de cada classe foram geradas a partir de uma mesma distribuição normal (a menos do centro da distribuição).



[1]Para gerar amostras de uma distribuição normal multivariada pode-se usar, por exemplo, `np.random.multivariate_normal`

Já as figuras a seguir mostram uma situação na qual as amostras de uma classe são dez vezes mais frequentes que as da outra classe. Da mesma forma do caso anterior, as amostras de cada classe foram geradas a partir de uma mesma distribuição normal (a menos do centro da distribuição).



Observações finais:

- Os exemplos acima são de pontos 2D para facilitar a visualização, porém seu algoritmo deve funcionar para pontos de qualquer dimensão d.
- Use as funções da biblioteca `NumPy` e implemente cálculos matriciais.
- Os testes não precisam ser restritos aos aglomerados do tipo acima.