APRENDIZADO DE MÁQUINA - TRABALHO 01

Jefferson Colares

Análise e conclusões sobre o código Python disponibilizado para este trabalho.
Itens 1 - 4 do enunciado.

Professor:
Eduardo Bezerra

Rio de Janeiro,
Junho, 2018

Sumário

Capítulo I Regressão Linear com uma Variável

I.1 Visualização dos Dados

O script plot_ex1data1.py possibilita a visualização dos dados que serão executados na parte 1 deste trabalho.
O arquivo fornecido para análise necessitou algumas modificações para poder executado
Segue uma análise do código:

1. A função importarDados lê os dados do arquivo para um dataframe do Pandas, acrescenta a esse dataframe uma nova coluna preenchida apenas com 1s.

```
path = os.getcwd() + filepath
data = pd.read_csv(path, header=None, names=names)
data.insert(0, 'Ones', 1)
```

2. Separa o dataframe em X (características) e y (alvo).

```
cols = data.shape[1]
X = data.iloc[:,0:cols-1]
y = data.iloc[:,cols-1:cols]
```

3. Converte X e y em arrays do Numpy.

```
X = np.array(X.values)
y = np.array(y.values)
```

4. Configura e plota o gráfico utilizando o método *pyplot* da biblioteca *matplotlib*

```
plt.scatter(X[:,1], y, color='blue', marker='x')
plt.title('Populacao da cidade x Lucro da filial')
plt.xlabel('Populacao da cidade (10k)')
plt.ylabel('Lucro (10k)')
plt.savefig('plot1.1.png')
plt.show()
```

I.2 Gradiente Descendente

I.2.1 Função do custo

A predição do lucro em função do tamanho da cidade pode ser obtida através de regressão linear, definida pela equação da reta $y = b + ax$, que será utilizada para representar nossa hipótese $h\theta(x)$. Ou seja, precisamos utilizar os dados do conjunto de treinamento para ajustar os parâmetros θ da equação abaixo até encontrar uma reta que possa ser utilizada para fazer predições.

$$h\theta(x) = \theta_0 x_0 + \theta_1 x_1$$

Para achar os parâmetros corretos precisamos de:

- Uma **função de custo**, utilizada para obter a distância (erro) entre a reta proposta (obtida através dos parâmetros que fornecemos à função linear) e os dados do conjunto de treinamento.
- Uma função de otimimzação, utilizada para mudar repetidas vezes os parâmetros fornecidos à função até encontrar os que façam a função de custo retornar o menor erro possível, no caso, a do **Gradiente Descendente**.

A função do custo é definida no arquivo **custo_reglin_uni.py**, com o nome **custo_reglin_uni()**. Sua função é traçar uma reta hipotética e medir a distânicia média entre ela e os pontos do conjuto de dados. A função recebe os seguintes parâmetros:

- As arrays X e y, contendo as coordenadas de cada ponto do conjunto de dados.
- O parâmetro theta, que na regressão com uma variável determina o ângulo de inclinação da reta (coeficiente angular da equação da reta).

Com esses três parâmetros, a função custo_reglin_uni() faz o cálculo do erro médio da seguinte forma:

1. Multiplica a array X pelo parâmetro theta (com o numpy, esse produto escalar pode ser obtido através do método .dot do objeto array).

   ```
   X.dot(theta)
   ```

2. De cada valor obtido, subtrai o valor do elemento correspondente na array y.

   ```
   (X.dot(theta) - y)
   ```

3. Eleva ao quadrado o resultado da subtração de cada elemento.

```
((X.dot(theta) - y)**2)
```

4. Soma todos os valores obtidos no passo anterior e divide pelo dobro do numero de elementos da array y (poderia ser len(X), tanto faz, ambas tem a mesma quantidade de linhas)

```
(np.sum((X.dot(theta) - y)**2)) / (2 * m)
```

Além do método sugerido no enunciado do trabalho, a correção da função de custo também pode ser avaliada através da execução do script **teste_custo_reglin_uni.py**, que forneço em anexo. Nele utilizo duas arrays iguais, e invoco a função de custo para calcular a distância entre elas. Se essa distância for diferente de zero, isso indica erro no código.

I.2.2 Função do Gradiente Descendente

O gradiente descendente tem o objetivo de encontrar o parâmetro que faz com que a função de custo retorne o menor valor.
Para conseguir isso, ela invoca a função de custo diversas vezes, informando diferentes valores de parâmetros, até encontrar aquele parâmetro que faça a função de custo retornar o menor valor possível. A seguir, analiso os passos executados pelo código da função **gd_reglin_uni()**

1. A função recebe 4 parâmetros:
    - X e y: Arrays com os valores de X e y do conjunto de dados de treinamento.
    - alpha: um hiperparâmetro que determina a taxa de aprendizado do gradiente descendente.
    - epochs: número inteiro que irá determinar a quantidade de passos que o GD dará para encontrar valor mínimo para theta.
    - theta: array de dois elementos contendo os valores iniciais de nossa hipótese (convém observar que theta é transposta, para possibilitar sua multiplicação por X posteriormente)
2. duas variáveis são inicializadas:
    - m: valor correspondente a quantidade de elementos no conjunto de dados de treinamento (len(y)).
    - cost: uma array com uma quantidade de itens igual à quantidade de epochs e preenchida completamente com zeros.
3. Os passos a seguir são repetidos tantas vezes quanto informadas no parâmetro epochs

(a) atribui à variável h (uma array) o produto da array X pela array theta, ambas recebidas como parâmetros da função.

```
h = X.dot(theta)
```

(b) atribui à variável loss o valor de h - y (o segundo parâmetro da função).

```
loss = h - y
```

(c) calcula o valor do gradiente, que é a array X transposta e multiplicada pela array loss e dividida pelo valor de m obtido no passo 1. Isso equivale ao cálculo da derivada parcial da fórmula do gradiente descendente.

```
gradient = X.T.dot(loss) / m
```

(d) atualiza o valor de theta com theta - alpha*gradiente. O passo do gradiente descendente está dado.

```
cost[i] = custo_reglin_uni(X, y, theta = theta)
```

(e) na ultima linha do loop, a função **custo_reglin_uni()** é invocada para obter o custo correspondente ao valor de theta. O valor retornado é armazenado na array cost, na posição correspondente à iteração atual.

4. a função **custo_reglin()** retorna o último (e menor) valor de theta obitdo e seu custo correspondente.

Obs.: Esta função utiliza um numero predeterminado de iterações para encontrar a hipótese theta de menor custo. Se esse número de iterações não for grande o suficiente a função poderá não retornar o valor correto para a regressão linear.

Uma vez obtido o valor ideal do parâmetro theta para a função linear, podemos utilizá-lo para traçar o gráfico da reta ou para fazer predições baseadas na regressão linear.

O programa **visualizar_reta.py** faz essas duas coisas, Traça uma reta e faz predições baseadas nesse modelo.

Em resumo, basta aplicar o parâmetro aprendido com o gradiente descendente à formula da hipótese (equação linear) , como no código a seguir:

```
#Prediz os valores do lucro para cidades de 35 e 70 mil habitantes
print("Predicao de lucro para cidades:")
print("com 35.000 habitantes: ", theta[0] + theta[1] * 35000)
print("com 70.000 habitantes: ", theta[0] + theta[1] * 70000)
```

Esse código pode ser encontrado e executado no arquivo **visualizar_reta.py**

I.3 Visualizacao de $J(\theta)$

Nessa parte do trabalho, vamos plotar os valores do custo J sobre uma grade de valores de $\theta 0$ e $\theta 1$.

Primeiramente, o contorno. Ele pode ser exibido executando-se o script **visualizar_J_contour.py**. Sua ideia básica é gerar uma grade de pontos x e y com intervalos regulares e utilizar as coordenadas de cada um deles como os parâmetros de entrada θ_0 e θ_1 da função de custo. Os outros dois parâmetros da função de custo vêm do cuonjunto de treinamento. Os valores obtidos são armazenados em uma matriz *J* que é plotada no gráfico.

Ao final, o que temos é uma avaliação de qual é o custo de cada combinação de parâmetros aplicada ao dataset do exercício.

Os detalhes interessantes do funcionamento desse script seguem abaixo:

Essas linhas geram os valores de theta0 e theta1 que iremos usar para o grid.

```
theta0 = np.arange(-10, 10, 0.01)
theta1 = np.arange(-1, 4, 0.01)
```

Aqui é gera uma array bidimensional de theta0 x theta1, preenchida completamente com zeros.

```
J = np.zeros((len(theta0), len(theta1)))
```

Esse loop prenche uma array bidimensional *t* com valores crescentes e sucessivos de theta0 e theta1 obtidos anteriormente

```
for i in range(len(theta0)):
    for j in range(len(theta1)):
        t = [[theta0[i]], [theta1[j]]]
```

Ainda dentro do mesmo loop, a função do custo é chamada para calcular o custo repetidamente para cada combinação em *t*. Todos os resultados são armazenados em *J*.

```
        J[i,j] = custo_reglin_uni(X, y, t)
```

Já fora do loop, o script faz a combinação de cada valor de theta0 com seu correspondente em theta1 e armazena nas variáveis theta0 e theta1, que posteriormente serão usadas para formar um grid.

```
theta0, theta1 = np.meshgrid(theta0, theta1)
```

As linhas seguintes definem parâmetros para o gráfico. A mais crucial delas é essa abaixo. Ela prepara os contornos que serão plotados. Os parâmetros theta0 e theta1 contém as coordenadas de cada ponto do grid e J contém o valor do custo correspondente a cada um deles.O parâmetro levels, que determina a posição das curvas de nível é alimentado pela função logspace, que fornece 20 elementos espaçados logaritimicamente de -1 a 4:

```
ax.contour(theta0, theta1, J, levels=np.logspace(-1, 4, 20), color='blue')
```

Através desse gráfico, podemos observar que o custo diminui conforme os parâmetros θ_0 e θ_1 se aproximam de -5 e 1,5, respectivamente.

O gráfico gerado pelo script visualizar_J_surface.py utiliza os mesmos dados do exemplo anterior, processados da mesma forma. A diferença é que, desta vez, foi utilizada uma biblioteca adicional *mpl_toolkits.mplot3d.Axes3D* que possibilitou a visualização dos dados em 3d. As linhas do script que possibilitaram esse resultado foram as seguites:

```
from mpl\_toolkits.mplot3d import Axes3D  # carrega o Axes3D
ax = fig.gca(projection='3d')  #define que o grafico e 3d
surf = ax.plot_surface(theta0, theta1, J)  #substitui a funcao contour()
```

Capítulo II Regressão Linear com Múltiplas Variáveis

II.1 Normalização das Características

A função **normalizarCaracteristica** reduz todos os valores de parâmetros para uma mesma escala. O objetivo é acelerar posteriormente o processamento do gradiente descendente, que não tem desempenho muito bom quando os valores de parâmetros estão em escalas muito diferentes. O código dessa função é simples. Ela recebe uma array qualquer, faz os cálculos e retorna três arrays. A primeira (0), contendo os valores normalizados de todas as colunas da array original, a segunda (1), a média de cada coluna da array original e a terceira (2), contendo o desvio padrão de cada uma das colunas.

Para utilizar a função, passe uma array qualquer como parâmetro e armazene o resultado em outra array. A chamada pode ter o formato abaixo, por exemplo:

```
resultado = normalizarCaracterstica(X).
```

Para acessar o resultado, informe o elemento desejado. Por exemplo:

```
print(resultado[0]) #imprime array com os valores normalizados
```

Outro exemplo:

```
media_X = resultado[2]
print(media_X[0])  Imprime a media da primeira coluna de X.
```

A implementação vetorizada permite que esses cálculos sejam feitos para arrays com o qualquer número de linhas ou colunas. Veja o código completo a seguir:

```
def normalizarCaracteristica(X):
    X_mean = np.array(np.mean(X, axis=0))
    X_std = np.array(np.std(X, axis=0))
    X_norm = np.array((X - X_mean) / X_std)
    return X_norm, X_mean, X_std
```

Em anexo a esse trabalho, no diretório correspondente à parte 2, coloquei um script chamado **testes_normalização.py**, que pode ser executado para testar o código dessa função.

II.2 Gradiente Descendente

O gradiente descendente para a regressão linear multivariada é um ligeiramente diferente da versão que utilizei na parte 1 desse trabalho para regressão com apenas uma variável. Como temos que encontrar a mínima individual de mais de uma variável, é necessário utilizar a derivada parcial para achar o próximo passo de cada uma. A equação que define o agoritimo e o código que implementei (tive que fazer diversas alteraçõres no código original) seguem abaixo:

$$\theta_{(j)} = \theta_{(j)} - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_{(j)}^{(i)}$$

```
def gd(X, y, alpha, epochs, theta):
    m = len(y)
    params = len(theta)
    for i in range(epochs):
        for j in range (params):
            h = X.dot(theta.T)
            dist = np.subtract(h, y)
            theta[j] = theta[j] - alpha / m * np.sum( dist * X[j])
            custo = custo_reglin_multi(X, y, theta)
    return  (theta, custo)
```

O código do gradiente descendente está vetorizado. Isso quer dizer que, em vez de utilizar loops para calcular a hipótese $h(\theta) = \Theta^T X$ podemos simplesmente multiplicar a matriz de características X pelo vetor de valores da hipótese. Além de ser mais elegante, esse método proporciona desempenho melhor que o tradicional, o que é muito importante em aplicações de Aprendizado de Máquina. Outra vantagem de utilizar matrizes é, como mencionado anteriormente, a de não termos que nos preocupar com a quantidade de parâmetros fornecidos.
Para testar o código implementado, disponibilizei o script **testes_gradiente.py**, no mesmo diretório que o anterior.

Capítulo III Regressão Logística

III.1 Visualização dos Dados

Assim como nos outros exercícios, utilizamos o Pandas para carregar os dados do arquivo. Após a leitura, os dados são carregados em duas arrays Numpy, X e y.
O vetor theta também é inicializado com os valores [0, 0, 0], mas não é utilizado na visualização de dados.
A função isin() é utilizada para separar os exemplos positivos e negativos, que são plotados em cores diferentes no gráfico.

III.2 Implementação

III.2.1 Função Sigmóide

A função sigmoide reduz qualquer número informado a uma faixa de valores entre zero e um. Ela é utilizada para representar a hipótese na regressão logística com a expressão $h\theta(x) = 1/(1 + e^{-h\theta(x)})$.
A função **exp()** do Numpy é utilizada para calcular o exponencial $e^{-h\theta(x)}$ da fórmula acima. Convenientemente, ela aceita como parâmetros tanto valores escalares como arrays. Desta forma, a implementação da função sigmoide fornecida no arquivo **sigmoide.py** quando recebe uma array, retorna uma array do mesmo tamanho contendo os resultados correspondentes a cada um dos elementos informados.

III.2.2 Função de Custo e Gradiente

Na regressão logística, a função do custo avalia apenas se o valor predito pela função está próximo ao valor de y (0 ou 1) esperado. Quando a predição está correta, o valor retornado pela função de custo é baixo, e vice-versa. A equação do custo é:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}[y^{(i)}log(h_\theta(x^{(i)})) - (1 - y^{(i)})log(1 - h_\theta(x^{(i)}))]$$

O cálculo do custo dos parâmetros é realizado através de duas funções implementadas no ar-

quivo **custo_reglog.py** da seguinte forma:

Função normalizar_caracteristicas():

Esta função normaliza os valores das características reduzindo-os a uma faixa de valores apropriada para os cálculos da função sigmoide.

Ela recebe duas arrays contendo os valores das características (X) e dos rótulos (y) e retorna:

- X_norm: array X normalizada
- X_mean: média dos valores de X
- X_stdev: desvio padrão de X
- y_norm: array y normalizada
- y_mean: média dos valores de y
- y_stdev: desvio padrão de y

Função custo_reglog():

Esta função calcula o custo (erro) para um conjunto de parâmetros informado.

1. X, y e theta são inicializados como matrizes numpy para podermos utilizar álgebra linear e evitar os loops for...end, como já fizemos em exemplos anteriores:

```
theta = np.matrix(theta)
X = np.matrix(X)
y = np.matrix(y)
```

2. A primeira parte da função, que avalia o custo da previsão de y=0:

```
grad0 = np.multiply(-y, np.log(sigmoide(X * theta.T)))
```

   Obs.: A fórmula acima, que constava no programa original apresentava erro na inversão do sinal do y. reescrevi como está abaixo e funcionou corretamente.

```
grad0 = np.multiply(-1 * y, np.log(sigmoide(X * theta.T)))
```

3. A segunda parte da função, que avalia o custo da previsão de y=1:

```
grad1 = np.multiply((1 - y), np.log(1 - sigmoide(X * theta.T)))
```

4. Aqui todas as linhas de grad0 e grad1 são combinadas e executadas. O somatório é calculado, dividido pela quantidade de exemplos de treinamento e o resultado é o custo retornado pela função:

```
return np.sum(grad0 - grad1) / (len(X))
```

Obs.: nos itens 2 e 3 a função **sigmoide()**, de que falamos anteriormente, é chamada para calcular o valor da hipótexe $h_\theta(x^{(i)})$.
O programa **teste_custo_reglog** executa o teste da função de custo, conforme a solicitação no enunciado do item 3.2.2 e retorna o valor correto (0,693) para theta = [0,0,0]. Abaixo segue uma descrição de suas principais ativiadades executadas:

1. Carrega os dados do dataset.

```
data = pd.read_csv('ex2data1.txt', header=None, names=['Prova␣1', 'Prova␣2', 'Aprovado'])
```

2. Separa as colunas do dataset em Características (X) e Rótulos (y).

```
X = data.iloc[:,0:2]
y = data.iloc[:,2:3]
```

3. Invoca a função de normalização e registra os dados obtidos em variáveis.

```
X_norm, X_mean, X_stdev, y_norm, y_mean, y_stdev  = normalizar_caracteristicas(X, y)
```

4. Insere uma coluna preenchida com 1's no conjunto de dados X.

```
X_norm = np.c_[np.ones((X.shape[0],1)), X_norm]
```

5. Inicia o vetor de parâmetros theta com zeros.

```
theta = np.array([0, 0, 0],ndmin=2)
```

6. Computa o valor do custo J.

```
J = custo_reglog(theta, X_norm, y)
```

7. Exibe o resultado do teste da função de custo.

```
print(J)
```

III.2.3 Aprendizado dos Parâmetros

Na regressáo logística, assim como na regressão linear, utilizamos o algoritimo do gradiente descendente para otimização. Apesar disso, a implementação é diferente nos dois casos, pois a representação da hipétese é diferente. O arquivo gd_reglog.py contém a implementação do gradiente descendente para a regressão logística. Esses são os principais passos executados pelo código fornecido:

1. Armazena na variável parametros a quantidade de itens existentes na matriz theta.

```
parametros = int(theta.ravel().shape[1])
```

2. Cria a array grad, preenchida com zeros e a mesma quantidade de itens que há em parâmetros.

```
grad = np.zeros(parametros)
```

3. Calcula a distância (erro) entre os valores obtidos na hiótese thetaX e os valores corretos (y) do conjunto de dadaos de treinamento.

```
erro = sigmoide(X * theta.T) - y
```

4. Calcula a derivada parcial de X

```
for i in range(parametros):
        term = np.multiply(erro, X[:,i])
        grad[i] = np.sum(term) / len(X)
```

5. Retorna o valor calculado

```
return grad
```

A função gd_reglog calcula o valor do gradiente, mas não executa as iterações para encontrar os parâmetros que minimizam o custo. Para fazer a implementação completa do algoritmo do gradiente descendente, criei o programa **teste_gd_reglog.py**, que lê os dados X e y do arquivo, inicializa os parâmetros theta e normaliza os valores de X. Nesse programa acrescentei o trecho de código fornecido na listagem 3 do enunciado do trabalho. Segue a análise desse código:

1. A chamada de uma nova biblioteca, Scipy, foi adicionada ao início do código.

```
import scipy.optimize as opt
```

2. Chama o método optimize.fmin_tnc, da biblioteca scipy. Ele implementa o algoritmo que, através de sucessivas iterações, encontra os parâmetros que minimizam uma função.

```
result = opt.fmin\_tnc(func=custo\_reglog, x0=theta, fprime=gd\_reglog, args=(X, y))
```

   Os parâmetros utilizados para calcular o gradiente descendente para a regressão logística foram:

   - *func=custo_reglog* : a função que será minimizada. No caso, queremos minimizar o custo calculado na função custo_reglog.
   - *x0=theta* : O parâmetro inicial (no caso, a array com os valores iniciais de theta) que será mmodificado a cada iteração.
   - *fprime=gd_reglog* : a função que é chamada para calcular o gradiente. No caso gd_reglog, que analisamos acima.

- *args=(X, y)* : os argumentos que são utilizados pela função. No caso, o conjunto de treinamento X,y.
- *disp=0* : suprime a exibição de mensagens durante a execução do algoritimo.

A função fmin_tnc retorna uma array com os parâmetros encontrados na otimização, o número de chamadas de funções e um codigo de retorno

III.2.4 Avaliação do Modelo

Nesse item, é solicitado predizer da probabilidade de aprovação de um aluno que tenha recebido as notas 45 e 85. O código que executa essa tarefa foi implementado no final do arquivo teste_gd_reglog e está reproduzido abaixo:

```
notas = np.array([1, (45 - X_mean[0]) / X_stdev[0], (85 - X_mean[1]) / X_stdev[1]])
pred = sigmoide(notas.dot(result))
print("Probabilidade de aprovacao com notas 45 e 85: ", pred)
```

Ele calcula o valor normalizado correspondente às notas 45 e 85, utilizando a média e desvio padrão obtidos anteriormente a partir do conjunto de treinamento.
Em seguida, faz o cálculo da função logística utilizando esses valores normalizados e os parâmetros obtidos anteriormente na otimização.
O valor resultante é impresso ao final (0.77615).
Para finalizar, executo um teste, conforme solicitado no enunciado, que calcula a acurácia do modelo em relação aos dados de treinamento.
Para fazer esse cálculo, são utlizadas duas funções, que estão no arquivo **predizer_aprovação.py**.
A primeira, se chama **predizer()**. Ela recebe como entradas o vetor theta com os valores mínimos obtidos após o gradiente descendente e o conjunto de dados X e calcula a sigmode utilizando esses parâmetros para cada linha do conjunto de dados X. O resultrado da sigmoide é sempre um número entre 0 e 1.

```
probabilidade = sigmoide(X.dot(theta))
```

Na última linha da função predizer(), os valores obtidos são convertidos para 0s ou 1s. Quando são menores ou iguais a 0,5, a função retorna 1, caso contrário, retorna 0.

```
return [1 if x >= 0.5 else 0 for x in probabilidade]
```

A segunda função, acuracia(), compara os valores obtidos em nossa predição (ou seja, a saída da função predizer()) com os valores reais do conjunto de treinamento (y). O resultado da função é o percentual correspondente ao número de vezes em que o modelo predisse o valor correto de y.
O código original foi modificado ligeiramente:

```
def acuracia(X, y, theta):
        predicoes = predizer(theta, X)
        corretas = [1 if ((a == 1 and b == 1) or (a == 0 and b == 0)) else 0 for (a, b) in
        zip(predicoes, y)]
        acc = (sum(map(int, corretas)) % len(corretas))
        print('Acuracia {0}%'.format(acc))
```

A função recebe X, y e o theta (obtido com o gradiente descente) e executa os seguintes passos:

1. utiliza a função predicoes() para obter o resultado das prediçoes do modelo (0s e 1s).
2. compara as predições do modelo com os valores de y correspondentes e armazena os acertos na variável *corretas*.
3. calcula o percentual de predições corretas e o exibe na tela.

O percentual de acurácia obtido pelo nosso modelo foi de 89%.

Capítulo IV Regressão Logística com Regularização

A visualiação dos dados solicitada nesse item do trabalho está implementada no arquivo **visualizarDados.py**.

A lógica de implementação é semelhante à do exercício 3.1. Para evitar a repetição, vou focar apenas nas particularidades desse exercício.

Aqui no exercício 4.1, diferentemente das vezes anteriores, não há preocupação em separar os dados em features (X) e rótulos(y), mas sim em separar os exemplos positivos dos negativos para poder representá-los de forma distinta no gráfico. Esse resultado é obtido através dos trechos de código reproduzidos abaixo:

```
#Separa os dados obtidos do arquivo em 2 conjuntos (dataframes),
# de acordo com  coluna "Resultado".
positivo = dados[dados['Resultado'].isin([1])]
negativo = dados[dados['Resultado'].isin([0])]
...
#Plota o grafico conforme solicitado no item 4.1.
...
ax.scatter(positivo['Test␣1'], positivo['Test␣2'], s=50, c='k', marker='+', label='y=1')
ax.scatter(negativo['Test␣1'], negativo['Test␣2'], s=50, c='y', marker='o', label='y=0')
```

IV.1 Mapeamento de características (feature mapping)

O mapeamento de características consiste em gerar novas características a partir das existentes.

Em nosso caso, transformei as duas características originais em 28. Para isso, computei todos os termos polinomiais de $x1$ e $x2$ até a sexta potência, conforme solicitado no exercício.

A função **mapFeatures()**, implementada no arquivo **mapFeatures.py** executa essa tarefa.

A estratégia escolhida foi utilizar o metodo *fit_transform*, do modulo *PolynomialFeatures* da biblioteca *ScikitLearn*.

O fit_transform recebe uma array com as características e gera os termos polinomiais. Basta informar a potência desejada.

Para contornar um problema de incompatibilidade identificado posteriormente, a função mapFeature() precisou receber as features x1 e x2 separadamente e juntá-los antes de passá-los como

parâmetros para o método *fit_transform*, como pode ser visto no código reproduzido abaixo:

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
def mapFeature(X1, X2):
    X = np.column_stack((X1,X2))
    poly = PolynomialFeatures(degree = 6 )
    Z = poly.fit_transform(X)
    return(Z)
```

IV.2 Função de custo e gradiente

No item 3 desse trabalho utilizei o método *fmin_tnc* do SciPy para fazer a otimização da função de custo. Esse método exigia que fossem utilizadas duas funções separadas para cálculo do custo e do gradiente. Aqui no item 4, realizei a mesma implementação, mas sem sucesso, porque o *fmin_tnc* apresentava erro de convergência, mesmo aumentando o limite de iterações possíveis.

Por esse motivo, ele foi substituído pelo método *minimize*, da mesma biblioteca. Diferente do *fmin_tnc*, o minimize exige apenas uma única função calcule o custo. Por esse motivo, na resolução da questão 4.2, a função **costFunctionReg()** retorna apenas cálculo do custo da regressão logística, mas não implementa cálculo do gradiente.

Abaixo, comento alguns trechos mais relevantes do código, que pode ser encontrado em sua totalidade no arquivo **costFunctionReg.py**.

A função do custo da regressão logística regularizada é semelhante à da regressão logística normal, mas contém um elemento adicional, que é o termo de regularização $\frac{\lambda}{2m}\sum_{j=1}^{n}\theta j^2$, como segue:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}[y^{(i)}log(h_\theta(x^{(i)})) - (1-y^{(i)})log(1-h_\theta(x^{(i)}))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta j^2$$

A fórmula foi implementada no Python da seguinte forma:

Quantidade de exemplos (m):

```
m = len(X)
```

Hipótese $h_\theta(x^{(i)})$:

```
h = sigmoide(np.dot(X, theta))
```

Lado esquerdo da equação do custo: (Custo para predição = 1) $y^{(i)}log(h_\theta(x^{(i)}))$:

```
custo0 = (-1*y).T.dot(np.log(h))
```

Lado direito da equação do custo (Custo para predição = 0) $(1 - y^{(i)})log(1 - h_\theta(x^{(i)}))$:

```
custo1 = (1 - y).T.dot(1 - np.log(h))
```

Termo de regularização:

```
reg = (lambd/2) * np.sum(np.dot(theta[1:].T,theta[1:]))
```

Custo regularizado:

```
custoReg = (1/m) * ( np.sum(custo0 - custo1) + reg )
```

Conforme solicitado, executei testes para analisar o comportamento da função de custo regularizado quando executada com diferentes valores para λ.
O que pude observar foi que utilizando um valor muito baixo, o custo calculado é mais baixo.
Quando o valor é muito alto, a função de custo retorna valores mais altos.
O programa **testes.py** pode ser executado para realizar esses testes. Seu resultado é reproduzido a seguir:

```
VARIACAO DO CUSTO OBTIDO COM DIFERENTES VALORES DE LAMBDA:
Custo com lambda = 0:   2.02044153500484
Custo com lambda = 1:   2.1348483146658572
Custo com lambda = 110:   13.461119501106534
```

IV.3 Esboço da fronteira de decisão

Conforme solicitado, escrevi o código para plotar a fronteira de decisão e o disponibilizo no arquivo plotDecisionBoundary.py.
É possível modificar os valores de lambda antes de executar o script de forma a que se possa visualizar como esse hiperparâmetro influencia no viés e na variância do modelo. Abaixo reproduzo o trecho do código onde ele pode ser encontrado com o nome lambd:

```
#Inicializa parametros da funcao de custo
Xmap = mapFeature(X[:,0],X[:,1])
thetaInicial = np.zeros(28)
lambd = 1
```

Capítulo V Considerações Finais

Infelizmente, não foi possível concluir as partes 5 a 7 do trabalho dentro do prazo fornecidos, Nesse início de mestrad o. Nesse início (acabei de ingressar no PPPCIC) há muitas coisas importantes a aprender e muito pouco tempo para tal. Peço desculpas. No futuro, à medida que for me acostumando (principalmente com o Python), estou certo de que conseguirei entregar os trabalhos completos