Regressão linear e logística

Revisões; algoritmos de treino; regularização

Implementação em python usando *numpy*

Representação do conjunto de dados

Nos próximos modelos iremos usar a seguinte representação para os dados:

- assumimos que todos os atributos são numéricos
- consideramos *m* exemplos e *n* atributos
- **Representação matricial**: X –matriz com atributos de entrada e seus valores (cada linha – exemplo; cada coluna - atributo); y – vector com valores do atributo de saída
- os valores $x_j^{(i)}, y_i$

representam respetivamente o valor do j-ésimo atributo para o i-ésimo exemplo (X_{ij}); e o valor do atributo de saída para o i-ésimo exemplo

Modelos clássicos de regressão

Representam relação entre variáveis de **entrada** $x_1,\ldots,x_n$ (variáveis independentes), e uma variável de **saída** *y* (variável dependente).

Previsão do modelo dado por (para i-ésimo exemplo):

$$\hat{y}^{(i)} = h_\theta(x_1^{(i)},\ldots,x_n^{(i)})$$

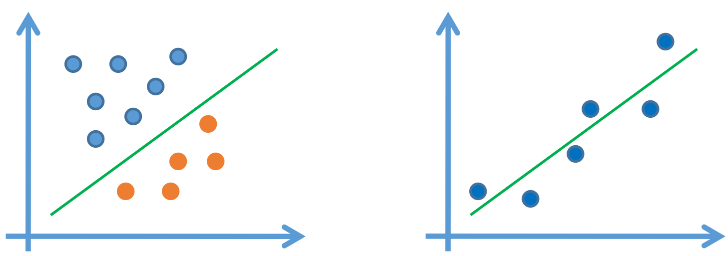
n – nº de entradas
θ - parâmetros do modelo

Modelos lineares

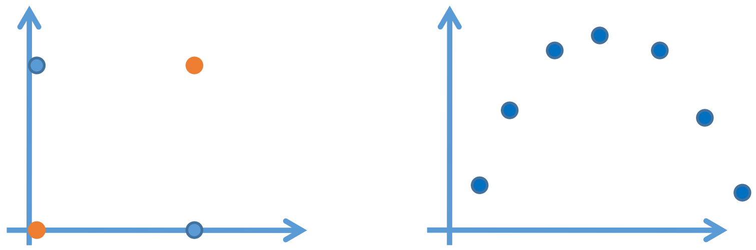
Atrativos, dada a simplicidade do cálculo e da análise

A linearidade é definida em termos de funções com as propriedades: *f(x + y) = f (x) + f (y)* e *f(ax)=af(x)*;

Podem ser usados para **classificação** (separação entre classes) ou **regressão**.



Problemas não lineares



Classificação
(e.g. XOR)

Regressão

Limitação:
Nestes casos, modelos lineares poderão ser insuficientes …

Modelos de regressão linear

Caso geral: modelos de **regressão**:

$$\hat{y}^{(i)} = h_\theta(x^{(i)}) = \theta_0 + \sum_{j=1}^{n} \theta_j x_j^{(i)}$$

Se *n* = 1: **regressão linear**

Se *n* >= 2: **regressão linear múltipla**

θ_i - parâmetros do modelo

Modelos de regressão linear

Formulação **vetorial / matricial**

$$h_\theta(x) = \theta^T x = \theta_0 x_0 + \ldots + \theta_n x_n$$

sendo *x* um vector de tamanho *n* representando um exemplo, onde $\boldsymbol{x_0}$ **é considerado igual a 1** por conveniência (este tem que ser adicionado a cada exemplo no conjunto de dados original)

θ – vector de tamanho *n+1* com os parâmetros do modelo

Modelos de regressão linear

Função de **erro** (*custo*): média dos quadrados dos erros - MQE

$$J_\theta = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Valor previsto pelo modelo: $\hat{y}^{(i)}$

Valor real

J é uma função dos parâmetros do modelo $\theta_1, \ldots, \theta_n$

Objetivo: identificar os parâmetros do modelo de forma a **minimizar o valor de J**

Regressão logística

Variável dependente discreta: problema de **classificação**

Regressão **logística**: usa modelos de regressão para classificação binária interpretando a saída do modelo de forma a extrair uma classe

$h_\theta(x) = g(\ \theta^T x\ )$ $\qquad 0 \leq h_\theta(x) \leq 1$

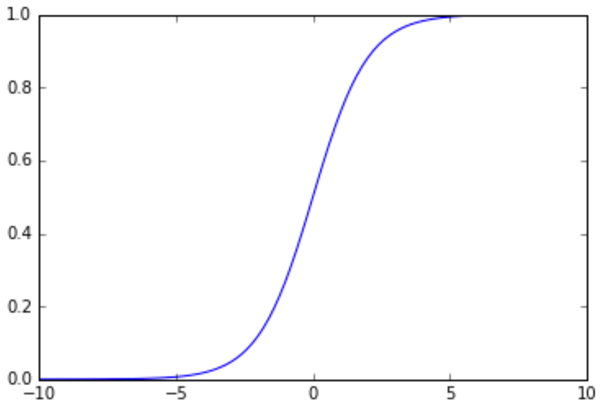
$$\boxed{h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}}$$

$\boxed{\frac{1}{1 + e^{-z}}}$ Função **sigmoid** (logística)

Modelo é dado pela aplicação da sigmoid à função da regressão linear

Interpretação: h estima a **probabilidade** de y (saída) ser igual a 1 para o exemplo *x*

Função sigmoid



Para z = 0 => sigmoid(z) = 0.5
Para z << 0 => sigmoid(z) aproxima-se de 0
Para z >> 0 => sigmoid(z) aproxima-se de 1

Regressão logística: múltiplas classes

Regressão logística pode ser aplicada a casos com mais do que duas classes

Neste caso, a estratégia é treinar um modelo "binário" para cada classe em separado (considerando as restantes como classe única)

Cada modelo estima a probabilidade do exemplo ser de uma dada classe

Ao prever novos exemplos, cada modelo é aplicado escolhendo-se a classe cujo valor previsto pelo modelo for maior

Regressão logística: função de erro

Função de erro (para cada exemplo *x*):

$$\begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

Se y = 1:
Se a previsão está correta: erro é zero
Senão, à medida que previsão fica mais próxima de 0, erro tende para infinito

Se y = 0:
Se previsão está correta: erro é zero
Senão, à medida que previsão fica mais próxima de 1, erro tende para infinito

Estimação dos parâmetros: otimização

Sabendo a estrutura do modelo -> estimação dos parâmetros é um problema de **optimização numérica** – minimização da função de erro

No caso dos modelos lineares, pode usar-se o método dos **mínimos quadrados**, que minimiza a função de erro (quadrado dos erros) ou métodos iterativos

Estimação dos parâmetros: método mínimos quadrados para regressão linear

Método analítico para determinar valores ótimos dos parâmetros que minimizam J

Método algébrico que envolve a resolução de um sistema de equações dadas por:

$$\frac{\partial}{\partial \theta_j} J(\theta) = 0, j = 1, \ldots, n$$

$$\theta = (X^T X)^{-1} X^T y \longleftarrow$$

Versão **matricial**
Matriz X inclui exemplos + 1ª coluna de 1's

Estimação dos parâmetros: gradiente descendente para regressão linear

Método que depende do facto da função de erro ser diferenciável

Método iterativo, que em cada iteração altera os valores de cada um dos parâmetros θ_j

Para cada θ_j a regra de atualização é a seguinte:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Atualizações simultâneas em todos os parâmetros

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Regressão logística: estimação dos parâmetros

Tal como na regressão linear, parâmetros estimados minimizando a função de erro J, dada pela soma dos erros para cada exemplo

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

Processo pode ser realizado usando método do gradiente descendente

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Algoritmo idêntico ao anterior, mas função h é diferente

Estimação dos parâmetros: gradiente descendente

- O parâmetro α denomina-se taxa de aprendizagem e controla a "velocidade" de atualização dos parâmetros
- Valores de α elevados podem levar a uma convergência mais rápida, mas trazem riscos de divergência
- Valores de α mais baixos garantem convergência mas esta pode ser mais lenta

Estimação dos parâmetros: gradiente descendente vs método analítico

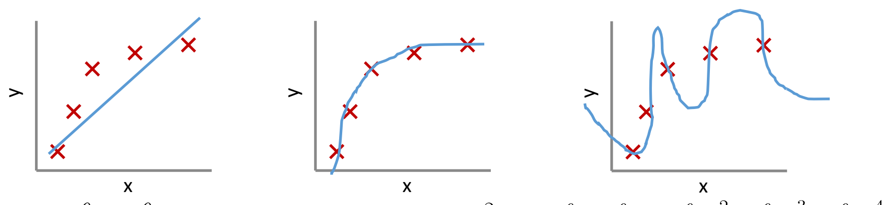
- Método analítico garante a solução ótima; GD pode não convergir
- No método analítico não há parâmetros; GD pode demorar a convergir
- Método analítico pode tornar-se lento com n muito grande (matrizes n x n podem tornar-se intratáveis para $n > 10^5$)
- GD mais genérico e aplicável a outros tipos de modelos

Estimação dos parâmetros: métodos avançados

- Em muitos casos, o gradiente descendente é demasiado lento na sua convergência para ser usado na prática
- Outros métodos de otimização numérica mais avançados podem ser usados
- Exemplo em Python com a função **fmin** do package **optimize** (neste caso, não necessita de derivadas)
- Outras alternativas disponíveis no mesmo package

Sobreajustamento em modelos funcionais

- Se tivermos um nº elevado de atributos, o modelo pode ajustar-se "demasiado bem" aos dados de treino e perder capacidade de generalização



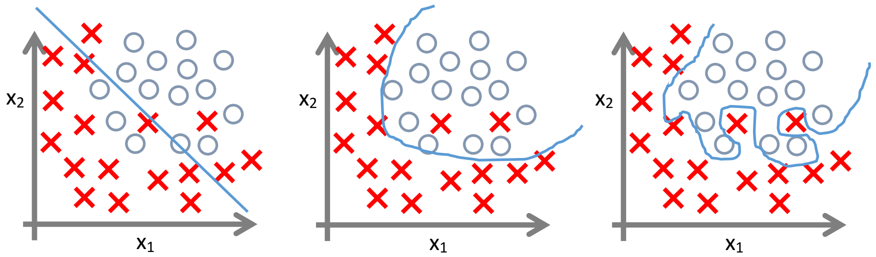
$\theta_0 + \theta_1 x$ | $\theta_0 + \theta_1 x + \theta_2 x^2$ | $\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

Sub-ajustamento: Complexidade insuficiente

Complexidade "adequada"

Sobre-ajustamento: Complexidade excessiva

Sobreajustamento em regressão logística: exemplo



$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$

g = função sigmoid

Sub-ajustamento: Complexidade insuficiente

$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2$
$+\theta_3 x_1^2 + \theta_4 x_2^2$
$+\theta_5 x_1 x_2)$

Complexidade "adequada"

$g(\theta_0 + \theta_1 x_1 + \theta_2 x_1^2$
$+\theta_3 x_1^2 x_2 + \theta_4 x_1^2 x_2^2$
$+\theta_5 x_1^2 x_2^3 + \theta_6 x_1^3 x_2 + \dots)$

Sobre-ajustamento: Complexidade excessiva

Soluções para sobreajustamento: modelos funcionais

Reduzir o número de atributos (coeficientes) usado

- Selecionar atributos "manualmente" por conhecimento do problema
- Algoritmos de **seleção de atributos**

Regularização

- Manter todos os atributos mas tentar reduzir magnitude dos valores dos parâmetros

Regularização: função de custo

Ideia: penalizar valores altos dos parâmetros na função de custo

Para regressão linear:

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

Parâmetro da regularização: valores mais altos penalizam mais os valores dos parâmetros
Se muito alto: risco de sub-ajustamento
Se muito baixo: risco de sobre-ajustamento

Regularização para regressão linear

Método analítico:

$$\theta = \left(X^T X + \lambda \begin{bmatrix} 0 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix}\right)^{-1} X^T y$$

Regularização para regressão linear

Gradiente descendente:

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)}$$

$$\theta_j := \theta_j (1 - \alpha \frac{\lambda}{m}) - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Termo imposto pela regularização
Note que valor entre () é sempre < 1

$$(j = \cancel{0}, 1, 2, 3, \ldots, n)$$

Regularização em regressão linear

Método anterior denominado de ***Ridge regression***: usa quadrado da norma L2 na penalização

Alternativa: usar soma dos valores absolutos (norma L1) dos parâmetros – ***Lasso regression***

Elastic nets – usam combinação de penalização por normas L2 e L1 de Ridge e Lasso com dois parâmetros de regularização

Regularização na regressão logística

Função de custo

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

Termo da regularização

Regularização na regressão logística

Gradiente

$$\frac{\partial}{\partial\theta_0} J(\theta) \qquad \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)}$$

$$\frac{\partial}{\partial\theta_1} J(\theta) \qquad \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_1^{(i)} - \frac{\lambda}{m}\theta_1$$

$$\frac{\partial}{\partial\theta_2} J(\theta) \qquad \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_2^{(i)} - \frac{\lambda}{m}\theta_2$$

(...)

Modelos não lineares (exemplos)

Modelos de **regressão linear generalizada**

$$\hat{y} = \beta_0 + \sum_{i=1}^{p} \beta_i f_i(x_i)$$

f_i – funções não lineares, contínuas

Modelos de **regressão polinomial**

$$\hat{y} = \beta_0 + \sum_{i=1}^{O_1} \beta_i x_1^i + \sum_{j=1}^{O_2} \beta_j x_2^j + \cdots$$

Podem obter-se usando regressão linear / logística, mas considerando transformações prévias nas variáveis de entrada

Preparação dos dados: tratamento de atributos nominais

Em modelos funcionais todos os atributos terão que ser **numéricos** – necessário converter valores nominais em numéricos

Hipótese 1: dividir intervalo numérico permitido às entradas pelo número de valores do atributo: 1 atributo nominal = 1 atributo numérico

Hipótese 2: binarizar o atributo: 1 atributo nominal com M valores possíveis = M atributos numéricos com valores possíveis 0 e 1 (codificação 1-of-C)

Normalização

Transformações nos dados muitas vezes necessárias para que o algoritmo de aprendizagem possa funcionar (melhor).

Algoritmos de gradiente descendente podem funcionar pior com variáveis com escalas muito diferentes

Vários métodos possíveis:

- Converter para média 0 e desvio padrão 1
- Converter para um intervalo [0,1] ou [-1, 1], definindo valores mínimo e máximo

IMPLEMENTAÇÃO EM PYTHON

Implementação com numpy

- Vamos implementar os métodos de regressão linear e logística em duas classes simples Python
- A base para a representação dos dados serão as estruturas de dados do package numpy
- A implementação do código será vetorizada, de forma a ser mais eficiente

IMPLEMENTAÇÃO EM PYTHON

Package *NumPy*

Inclui estruturas de dados e funções para facilitar o desenvolvimento de métodos algébricos e numéricos em python

Objeto ***array(ndarray)***: permite definir e manipular vetores, matrizes e arrays multidimensionais com uma gama alargada de funções disponíveis

Documentação: *www.numpy.org*

IMPLEMENTAÇÃO EM PYTHON

Package *NumPy*: tutorial

- https://docs.scipy.org/doc/numpy/user/quickstart.html
- Secções importantes a seguir:
  - The Basics: definições, 1º exemplo
  - Array creation: como criar objetos tipo array, funções array, zeros, ones, empty, random, reshape, fromfunction
  - Printing arrays: função print e opções
  - Basic operations: operações aritméticas elemento a elemento, aplicação de funções, multiplicação de matrizes, modificação dos arrays

IMPLEMENTAÇÃO EM PYTHON

Package *NumPy*: tutorial

- Secções importantes a seguir:
  - Funções sobre arrays na globalidade e especificando dimensão: *sum* , *min* , *max* , *cumsum* , *mean*, *std*, *median* ,…
  - Funções universais, aplicadas elemento a elemento*: sqrt*, *exp*, *sin* , *cos* , …
  - Funções sobre matrizes*: transpose (T)*, *dot*, *inv*
  - *Index, slicing and iterating:*
    - com 1 dimensão – semelhante a lists;
    - com 2+ dimensões: separadas por "," ; uso de …;
  - *Stacking together different arrays:* funções *hstack*, *vstack*

IMPLEMENTAÇÃO EM PYTHON

Classe *Dataset*

Classe que implementa os conjuntos de dados

Versão inicial muito simples considerando matriz de dados como array numpy

```
import numpy as np

def __init__(self, filename = None, X = None, Y = None):
    if filename is not None:
        self.readDataset(filename)
    elif X is not None and Y is not None:
        self.X = X
        self.Y = Y

def readDataset(self, filename, sep = ","):
        data = np.genfromtxt(filename, delimiter=sep)
        self.X = data[:,0:-1]
        self.Y = data[:,-1]

def getXy (self):
    return self.X, self.Y
```

IMPLEMENTAÇÃO EM PYTHON

Classe *LinearRegression*

Classe que implementa os modelos de regressão linear

```
import numpy as np

class LinearRegression:

  def __init__(self, dataset, normalize = False, regularization = False, lamda = 1):
    self.X, self.y = dataset.getXy()
    self.X = np.hstack ((np.ones([self.X.shape[0],1]), self.X ))
    self.theta = np.zeros(self.X.shape[1])
    self.regularization = regularization
    self.lamda = lamda
    if normalize:
        self.normalize()
    else:
        self.normalized = False
```

```
def normalize(self):
    self.mu = np.mean(self.X[:,1:], axis = 0)
    self.X[:,1:] = self.X[:,1:] - self.mu
    self.sigma = np.std(self.X[:,1:], axis = 0)
    self.X[:,1:] = self.X[:,1:] / self.sigma
    self.normalized = True
```

Variáveis da classe:
- *X, y* – dataset
- *theta* – parâmetros do modelo
- *regularization* – flag que define se se usa regularização
- *lamda* – parâmetro da regularização
- *normalized* – flag que indica se dados foram standardizados

IMPLEMENTAÇÃO EM PYTHON

Implementação – fase 1 (regressão linear sem regularização)

Criar métodos para:

- def ***predict*** (self, instance) – prever o valor para uma nova instância
- def ***costFunction*** (self) – calcula o valor da função de custo (para todos os exemplos do dataset)
- def ***buildModel*** (self, dataset) – cria o modelo usando o método analítico
- def **gradientDescent** (self, iterations = 1000, alpha = 0.001) – cria o modelo usando gradiente descendente

IMPLEMENTAÇÃO EM PYTHON

Implementação – fase 1 (regressão linear sem regularização)

Testar a implementação com exemplo lr-example1.data (2 variáveis):

```
import matplotlib.pyplot as plt

def plotData_2vars(self, xlab, ylab):
  plt.plot(self.X[:,1], self.y, 'rx', markersize=7)
  plt.ylabel(ylab)
  plt.xlabel(xlab)
  plt.show()
```

```
def test_2var():
   ds= Dataset("lr-example1.data")
   lrmodel = LinearRegression(ds)
   lrmodel.plotData_2vars("Population", "Profit")

   input("Press a key")
   print ("Cost function value for theta with zeros:”)
   print( lrmodel.costFunction())
```

IMPLEMENTAÇÃO EM PYTHON

Implementação – fase 1 (regressão linear sem regularização)

Testar a implementação com exemplo lr-example1.data (2 variáveis):

```
def plotDataAndModel(self, xlab, ylab):
  plt.plot(self.X[:,1], self.y, 'rx', markersize=7)
  plt.ylabel(ylab)
  plt.xlabel(xlab)
  plt.plot(self.X[:,1], np.dot(self.X, self.theta), '-')
  plt.legend(['Training data', 'Linear regression'])
  plt.show()

def printCoefs(self):
          print(self.theta)
```

```
def test_2var():
   (...)
   print("Model with analytical solution")
   lrmodel.buildModel(ds)
   print ("Cost value for theta from analytical solution:”)
   print (lrmodel.costFunction())
   print ("Coefficients from analytical solution”)
   lrmodel.printCoefs()
   lrmodel.plotDataAndModel("Population", "Profit")
```

IMPLEMENTAÇÃO EM PYTHON

Implementação – fase 1 (regressão linear sem regularização)

Testar a implementação com exemplo lr-example1.data (2 variáveis):

```
def test_2var():
    (...)
    print("Gradient descent:")
    lrmodel.gradientDescent(1500, 0.01)
    print("Cost function value for theta from gradient descent:")
    print(lrmodel.costFunction())
    print("Coefficients from gradient descent")
    lrmodel.printCoefs()
    lrmodel.plotDataAndModel("Population", "Profit")
    input("Press a key")
    ex = np.array([7.0, 0.0])
    print("Prediction for example:")
    print(lrmodel.predict(ex))
```

Exercício:

Aplicar o modelo criado anteriormente a um caso de mais do que duas variáveis – conjunto de dados *lr-example2*

IMPLEMENTAÇÃO EM PYTHON

Implementação – fase 1 (regressão linear sem regularização)

Possíveis soluções

```
def predict(self, instance):
        x = np.empty([self.X.shape[1]])
        x[0] = 1
        x[1:] = np.array(instance[:self.X.shape[1]-1])
        if self.normalized:
                x[1:] = (x[1:] - self.mu) / self.sigma
        return np.dot(self.theta, x)
```

```
def costFunction(self):
    m = self.X.shape[0]
    predictions = np.dot(self.X, self.theta)
    sqe = (predictions - self.y) ** 2
    res = np.sum(sqe) / (2*m)
    return res
```

```
def buildModel(self, dataset):
    from numpy.linalg import inv
    self.theta = inv(self.X.T.dot(self.X)).dot(self.X.T).dot(self.y)
```

```
def gradientDescent (self, iterations = 1000, alpha = 0.001):
    m = self.X.shape[0]
    n = self.X.shape[1]
    self.theta = np.zeros(n)
    for its in range(iterations):
        J = self.costFunction()
        if its%100 == 0: print(J)
        delta = self.X.T.dot(self.X.dot(self.theta) - self.y)
        self.theta -= (alpha /m * delta )
```

IMPLEMENTAÇÃO EM PYTHON

Implementação – fase 2 (regressão linear com regularização)

Criar métodos:

- def ***analyticalWithReg***(self) – método analítico para construção do modelo com regularização;
- adaptar o método ***buildModel*** para usar o anterior quando a flag de regularização for usada

- adaptar o método **gradientDescent** para usar regularização quando a flag de regularização for usada

Testar o código com os exemplos anteriores

IMPLEMENTAÇÃO EM PYTHON

Implementação – fase 2 (regressão linear com regularização)

Possíveis soluções

```
def analyticalWithReg (self):
    from numpy.linalg import inv
    matl = np.zeros([self.X.shape[1], self.X.shape[1]])
    for i in range(1,self.X.shape[1]): matl[i,i] = self.lamda
    mattemp = inv(self.X.T.dot(self.X) + matl)
    self.theta = mattemp.dot(self.X.T).dot(self.y)
```

```
def buildModel (self, dataset):
    from numpy.linalg import inv
    if self.regularization:
        self.analyticalWithReg()
    else:
        self.theta = inv(self.X.T.dot(self.X)).dot(self.X.T).dot(self.y)
```

IMPLEMENTAÇÃO EM PYTHON

Implementação – fase 2(regressão linear com regularização)

Possíveis soluções

```
def gradientDescent (self, iterations = 1000, alpha = 0.001):
   m = self.X.shape[0]
   n = self.X.shape[1]
   self.theta = np.zeros(n)
   if self.regularization:
         lamdas = np.zeros([self.X.shape[1]])
         for i in range(1,self.X.shape[1]):
            lamdas[i] = self.lamda
   for its in range(iterations):
      J = self.costFunction()
      if its%100 == 0: print(J)
      delta = self.X.T.dot(self.X.dot(self.theta) - self.y)
      if self.regularization:
            self.theta -= (alpha/m * (lamdas+delta))
      else:
            self.theta -= (alpha /m * delta )
```

IMPLEMENTAÇÃO EM PYTHON

Classe *LogisticRegression*

Classe que implementa os modelos de regressão logística

```
import numpy as np
from dataset import Dataset

class LogisticRegression:
   def __init__(self, dataset):
      self.X, self.y = dataset.getXy()
      self.X = np.hstack ( (np.ones([self.X.shape[0],1]), self.X ) )
      self.theta = np.zeros(self.X.shape[1])
      self.regularization = regularization
      self.lamda = lamda
      if normalize:
          self.normalize()
      else:
          self.normalized = False
```

IMPLEMENTAÇÃO EM PYTHON

Implementação – fase 3 (regressão logística)

Criar métodos para:

- def ***probability***(self, instance) – prever o valor de probabilidade para uma nova instância
- def ***predict*** (self, instance) – prever o valor (binário) para uma nova instância
- def ***costFunction*** (self) – calcula o valor da função de custo (para todos os exemplos do dataset)
- def ***gradientDescent*** (self, dataset, alpha = 0.01, iters = 1000) – cria o modelo usando gradiente descendente

IMPLEMENTAÇÃO EM PYTHON

Implementação – fase 3 (regressão logística)

Testar a implementação com exemplo log-ex1.data (2 variáveis):

```
def plotData(self):
   import matplotlib.pyplot as plt
   …
```

```
def plotModel(self):
   import matplotlib.pyplot as plt
   …
```

```
def test():
   ds= Dataset(”log-ex1.data")
   logmodel = LogisticRegression(ds)
   logmodel.plotData()
   logmodel. gradientDescent(ds, 0.001, 20000)
   logmodel.plotModel()
   print (logmodel.costFunction() )
   ex = np.array([45,85])
   print (logmodel.probability(ex))
   print (logmodel.predict(ex))
```

IMPLEMENTAÇÃO EM PYTHON

Implementação – fase 3 (regressão logística)

Possíveis soluções

```
def predict (self, instance):
   p = self.probability(instance)
   if p >= 0.5: res = 1
   else: res = 0
   return res

def probability (self, instance):
   x = np.empty([self.X.shape[1]])
   x[0] = 1
   x[1:] = np.array(instance[:self.X.shape[1]-1])
   if self.normalized:
         if np.all(self.sigma!= 0):
             x[1:] = (x[1:] - self.mu) / self.sigma
         else: x[1:] = (x[1:] - self.mu)
   return sigmoid ( np.dot(self.theta, x) )

def sigmoid(x):
   return 1 / (1 + np.exp(-x))
```

```
def costFunction (self, theta = None):
   if theta is None: theta= self.theta
   m = self.X.shape[0]
   p = sigmoid ( np.dot(self.X, theta) )
   cost = (-self.y * np.log(p) - (1-self.y) * np.log(1-p) )
   res = np.sum(cost) / m
   return res
```

```
def gradientDescent(self, dataset, alpha = 0.01, iters = 1000):
   m = self.X.shape[0]
   n = self.X.shape[1]
   self.theta = np.zeros(n)
   for its in range(iters):
      J = self.costFunction()
      if its%100 == 0: print J
      delta = self.X.T.dot(sigmoid(self.X.dot(self.theta)) - self.y)
      self.theta -= (alpha /m * delta )
```

IMPLEMENTAÇÃO EM PYTHON

Classe *LogisticRegression*

Gradiente descendente com métodos mais sofisticados de otimização

```
def optim_model(self):
   from scipy import optimize
   n = self.X.shape[1]
   options = {'full_output': True, 'maxiter': 400}
   initial_theta = np.zeros(n)
   self.theta, _, _, _, _ =
     optimize.fmin(lambda theta: self.costFunction(theta), initial_theta, **options)
```

Adaptar o código de teste para usar esta função no lugar do gradiente descendente e comparar resultados

LogisticRegression com regularização

IMPLEMENTAÇÃO EM PYTHON

Função custo com regularização

```
def costFunctionReg (self, theta = None, lamda = 1):
    if theta is None: theta= self.theta
    m = self.X.shape[0]
    p = sigmoid ( np.dot(self.X, theta) )
    cost = (-self.y * np.log(p) - (1-self.y) * np.log(1-p) )
    reg = np.dot(theta[1:], theta[1:]) * lamda / (2*m)
    return (np.sum(cost) / m) + reg
```

```
def optim_model_reg(self, lamda):
    from scipy import optimize
    n = self.X.shape[1]
    initial_theta = np.zeros(n)
    result = optimize.minimize(lambda theta: self.costFunctionReg(theta, lamda),
                initial_theta, method='BFGS',  options={"maxiter":500, "disp":True} )
    self.theta = result.x
```

Exercício

- Usando o conjunto de dados *log-ex2*:
  - Criar atributos auxiliares a partir das variáveis originais x_1 e x_2 com todos os termos polinomiais até grau 6

$$\begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \\ x_1^3 \\ \vdots \\ x_1 x_2^5 \\ x_2^6 \end{bmatrix}$$

LogisticRegression com regularização

IMPLEMENTAÇÃO EM PYTHON

Plot dos dados

```
def mapFeature (X1, X2, degrees = 6):
  out = np.ones( (np.shape(X1)[0], 1) )
  for i in range(1, degrees+1):
    for j in range(0, i+1):
      term1 = X1 ** (i-j)
      term2 = X2 ** (j)
      term = (term1 * term2).reshape( np.shape(term1)[0], 1 )
      out = np.hstack(( out, term ))
  return out
```

```
def mapX (self):
  self.origX = self.X.copy()
  mapX = mapFeature(self.X[:,1], self.X[:,2], 6)
  self.X = np.hstack((np.ones([self.X.shape[0],1]), mapX) )
  self.theta = np.zeros(self.X.shape[1])
```

```
def plotData(self):
  import matplotlib.pyplot as plt
  negatives = self.X[self.y == 0]
  positives = self.X[self.y == 1]
...
```

```
def plotModel2(self):
    import matplotlib.pyplot as plt
    negatives = self.origX[self.y == 0]
    positives = self.origX[self.y == 1]
    ...
```

Exercício

- Correr versão da regressão logística com regularização
- Criar um gráfico que mostre a discriminação gerada pelo algoritmo
- Analisar diversos valores do parâmetro de regularização (e.g. 0, 1, 10, 100, …)