Disciplina: Métodos Numéricos</p>

Semestre: 2020.2</p>

Aluno: Jardel Brandon de Araujo Regis</p>

Mátricula: 201621250014</p>

2ª Unidade: Álgebra Linear Computacional</p>

Projeto - 2

Predição de dados de saúde usando regressão linear

Regressão linear

A análise de regressão estuda a relação entre uma variável chamada a variável dependente e outras variáveis chamadas variáveis independentes.

A relação entre elas é representada por um modelo matemático, que associa a variável dependente com as variáveis independentes.

Este modelo é designado por modelo de regressão linear simples (MRLS) se define uma relação linear entre a variável dependente e uma variável independente.

Se em vez de uma, forem incorporadas várias variáveis independentes, o modelo passa a denominar-se modelo de regressão linear múltipla (MRLM).

ESTIMAÇÃO DOS PARÂMETROS DO MODELO

Suponha que temos n observações $(n > p)$ da variável resposta e das p variáveis explicativas. Assim, yi é o valor da variável resposta na i-ésima observação enquanto que xij é o valor da variável xj na i-ésima observação, j=1,...,p. Os dados de um MRLM podem ser representados da seguinte forma:

y	x_1	x_2	$\dots$	x_p
y_1	x_{11}	x_{12}	$\dots$	x_{1p}
y_2	x_{21}	x_{22}	$\dots$	x_{2p}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
y_n	x_{n1}	x_{n2}	$\dots$	x_{np}

Tabela 2.2.1: Representação dos dados

$$(1)$$

em que cada observação satisfaz

$$Y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip} + \varepsilon_i, \quad i = 1, \ldots, n \tag{2}$$

Método dos Mínimos Quadrados

O objetivo é minimizar a função, para isso deve-se encontrar os valores de α e β que minimizam a soma dos quadrados dos erros (ou desvios ou resíduos), dados por

$$L = \sum_{i=1}^{n} \varepsilon_i^2 = \sum_{i=1}^{n} \left(Y_i - \beta_0 - \beta_1 x_{i1} - \beta_2 x_{i2} - \ldots - \beta_p x_{ip}\right)^2 \tag{3}$$

Obtemos então, a quantidade de informação perdida pelo modelo ou soma dos quadrados dos resíduos

$$\begin{gathered} \frac{\partial L}{\partial \beta_0} = -2 \sum_{i=1}^{n} \left[Y_i - \beta_0 - \beta_1 x_{i1} - \beta_2 x_{i2} - \cdots - \beta_p x_{ip}\right] \\ \frac{\partial L}{\partial \beta_j} = -2 \sum_{i=1}^{n} \left[Y_i - \beta_0 - \beta_1 x_{i1} - \beta_2 x_{i2} - \cdots - \beta_p x_{ip}\right] x_{ji}, \quad j = 1, 2, \ldots, p \end{gathered} \tag{4}$$

Igualando as derivadas parciais a zero e substituindo $\beta_0, \beta_1, \ldots, \beta_p$ por $\widehat{\beta}_0, \widehat{\beta}_1, \ldots, \widehat{\beta}_p$, temos o sistema de equações

$$\begin{cases} n\widehat{\beta}_0 + \widehat{\beta}_1 \sum_{i=1}^{n} x_{i1} + \widehat{\beta}_2 \sum_{i=1}^{n} x_{i2} + \ldots + \widehat{\beta}_p \sum_{i=1}^{n} x_{ip} = \sum_{i=1}^{n} Y_i \\ \widehat{\beta}_0 \sum_{i=1}^{n} x_{i1} + \widehat{\beta}_1 \sum_{i=1}^{n} x_{i1}^2 + \widehat{\beta}_2 \sum_{i=1}^{n} x_{i1} x_{i2} + \ldots + \widehat{\beta}_p \sum_{i=1}^{n} x_{i1} x_{ip} = \sum_{i=1}^{n} x_{i1} Y_i \\ \vdots \\ \widehat{\beta}_0 \sum_{i=1}^{n} x_{ip} + \widehat{\beta}_1 \sum_{i=1}^{n} x_{ip} x_{i1} + \widehat{\beta}_2 \sum_{i=1}^{n} x_{ip} x_{i2} + \ldots + \widehat{\beta}_p \sum_{i=1}^{n} \frac{2}{ip} = \sum_{i=1}^{n} x_{ip} Y_i \end{cases} \tag{5}$$

Resolvendo este sistema, obtemos os estimadores de mínimos quadrados $\widehat{\beta}_0, \widehat{\beta}_1, \ldots, \widehat{\beta}_p$, dos parâmetros do modelo em questão.

Representação matricial do MRLM

pode-se notar que os estimadores de mínimos quadrados dos parâmetros podem ser facilmente encontrados considerando a notação matricial dos dados, que é de fácil manipulação. Desta forma, o modelo de Regressão Linear Múltipla pode ser escrito como

$$Y = X\beta + \varepsilon,$$

Com

$$Y = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix}, \quad X = \begin{bmatrix} 1 & x_{11} & x_{12} & \ldots & x_{1p} \\ 1 & x_{21} & x_{22} & \ldots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots & \\ 1 & x_{n1} & x_{n2} & \ldots & x_{np} \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix} \quad \text{e } \varepsilon = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix}, \tag{6}$$

em que

- Y é um vetor n×1 cujos componentes corresponde às n respostas;
- X é uma matriz de dimensão n×(p+1) denominada matriz do modelo;
- ε é um vetor de dimensão n×1 cujos componentes são os erros e
- β é um vetor (p+1)×1 cujos elementos são os coeficientes de regressão.

O método de mínimos quadrados tem como objetivo encontrar o vetor $\widehat{\beta}$ que minimiza

$$\begin{gathered} L = \sum_{i=1}^{n} \varepsilon_i^2 = \varepsilon'\varepsilon = (Y - X\beta)'(Y - X\beta) = \\ = Y'Y - Y'X\beta - \beta'X'Y + \beta'X'X\beta = Y'Y - 2\beta'X'Y + \beta'X'X\beta \end{gathered} \tag{7}$$

sendo que $Y'X\beta = \beta'X'Y$ pois o produto resulta em um escalar. A notação X' representa o transposto da matriz X enquanto que Y' e β' representam os transpostos dos vetores Y e β, respectivamente. Usando a técnica de derivação (em termos matriciais) obtemos

$$\frac{\partial L}{\partial \beta} = -2X'Y + 2X'X\beta \tag{8}$$

Igualando a zero e substituindo o vetor β pelo vetor $\widehat{\beta}$, temos

$$(X'X)\widehat{\beta} = X'Y$$

Em geral, a matriz $(X'X)$ é não singular, ou seja, tem determinante diferente de zero, e portanto é invertível. Desta forma, conclui-se que os estimadores para os parâmetros $\beta j,\ j = 0, \ldots, p$ são dados pelo vetor

$$\widehat{\beta} = (X'X)^{-1}X'Y$$

Portanto, o modelo de regressão linear ajustado e o vetor de resíduos são respectivamente

$$\widehat{Y} = X\widehat{\beta} \quad e \quad e = Y - Y\hat{} = Y - Y\hat{}$$

Assim, ao substituir os estimadores de mínimos quadrados, obtêm-se que $\widehat{Y} = HY$ no qual $H = X(X'X)^{-1}X'$ é a matriz chapéu, ou matriz de projeção do vetor de respostas Y no vetor de respostas ajustadas $\widehat{Y}$.

In [1]:
```
from sklearn import datasets, linear_model, metrics
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
import math, scipy, numpy as np
from scipy import linalg
```

Conjunto de dados retirados de exames de diabetes

Todos os conjuntos de dados foram retirados de pacientes com diabetes. Os dados consistem em 442 amostras e 10 variáveis (todas são características fisiológicas), por isso todo o grupo é composto de pessoas altas e magras. A variável dependente é uma medida quantitativa da progressão da doença um ano após o início do estudo.

Este é um conjunto de dados clássico, conhecido por Efron, Hastie, Johnstone e Tibshirani em seu artigo Least Angle Regression e um dos muitos conjuntos de dados incluído no scikit-learn.

In [2]:
```
data = datasets.load_diabetes()
```

In [3]:
```
feature_names=['age', 'sex', 'bmi', 'bp', 's1', 's2', 's3', 's4', 's5', 's6']
```

In [4]:
```
trn,test,y_trn,y_test = train_test_split(data.data, data.target, test_size=0.2)
```

In [5]:
```
trn.shape, test.shape
```

Out[5]:
```
((353, 10), (89, 10))
```

Regressão linear no Scikit Learn

Considere um sistema $X\beta = y$, onde X tem mais linhas do que colunas. Isso ocorre quando você tem mais amostras de dados do que variáveis. Queremos encontrar $\hat{\beta}$ que minimize:

$$||X\beta - y||_2$$

Let's start by using the sklearn implementation:

In [6]:
```
regr = linear_model.LinearRegression()
%timeit regr.fit(trn, y_trn)
```

```
773 µs ± 171 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [7]:
```
pred = regr.predict(test)
```

It will be helpful to have some metrics on how good our prediciton is. We will look at the mean squared norm (L2) and mean absolute error (L1).

In [8]:
```
def regr_metrics(act, pred):
    return (math.sqrt(metrics.mean_squared_error(act, pred)),
     metrics.mean_absolute_error(act, pred))
```

In [9]:
```
regr_metrics(y_test, regr.predict(test))
```

Out[9]:
```
(54.40683110341174, 44.6858312920976)
```

Recursos polinomiais

A regressão linear encontra os melhores coeficientes β_i para:

$$x_0\beta_0 + x_1\beta_1 + x_2\beta_2 = y$$

Adicionando os termos polinomiais, ainda se trata de um problema de regressão linear, apenas com mais termos:

$$x_0\beta_0 + x_1\beta_1 + x_2\beta_2 + x_0^2\beta_3 + x_0x_1\beta_4 + x_0x_2\beta_5 + x_1^2\beta_6 + x_1x_2\beta_7 + x_2^2\beta_8 = y$$

Utilizando os dados originais X para calcular as características polinomiais adicionais.

In [10]:
```
trn.shape
```

Out[10]:
```
(353, 10)
```

Agora, para tentar melhorar o desempenho do modelo foi adicionado mais alguns recursos. Atualmente, o modelo é linear em cada variável, mas pode-se adicionar recursos polinomiais para alterar isso.

In [11]:
```
poly = PolynomialFeatures(include_bias=False)
```

In [12]:
```
trn_feat = poly.fit_transform(trn)
```

In [13]:
```
', '.join(poly.get_feature_names(feature_names))
```

Out[13]:
```
'age, sex, bmi, bp, s1, s2, s3, s4, s5, s6, age^2, age sex, age bmi, age bp, age s1, age s2, age s
3, age s4, age s5, age s6, sex^2, sex bmi, sex bp, sex s1, sex s2, sex s3, sex s4, sex s5, sex s6,
bmi^2, bmi bp, bmi s1, bmi s2, bmi s3, bmi s4, bmi s5, bmi s6, bp^2, bp s1, bp s2, bp s3, bp s4, b
p s5, bp s6, s1^2, s1 s2, s1 s3, s1 s4, s1 s5, s1 s6, s2^2, s2 s3, s2 s4, s2 s5, s2 s6, s3^2, s3 s
4, s3 s5, s3 s6, s4^2, s4 s5, s4 s6, s5^2, s5 s6, s6^2'
```

In [14]:
```
trn_feat.shape
```

Out[14]:
```
(353, 65)
```

In [15]:
```
regr.fit(trn_feat, y_trn)
```

Out[15]:
```
LinearRegression()
```

In [16]:
```
regr_metrics(y_test, regr.predict(poly.fit_transform(test)))
```

Out[16]:
```
(65.11138664286574, 52.13794990801871)
```

O tempo é elevado ao quadrado em #features e linear em #points, dessa forma, a tendência é de lentidão computacional!

In [17]:
```
%timeit poly.fit_transform(trn)
```

```
560 µs ± 36.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Acelerando a geração dos recursos

Para acelerar isso. pode-se utilizar a Numba, uma biblioteca Python que compila o código diretamente para C.

Recursos

[Este tutorial] (https://jakevdp.github.io/blog/2012/08/24/numba-vs-cython/) de Jake VanderPlas é uma boa introdução. Aqui Jake implementa um algoritmo não trivial (transformação rápida de Fourier não uniforme) com Numba. Cython é outra alternativa. Descobri que o Cython exige mais conhecimento para ser usado do que o Numba (é mais próximo do C), mas fornece acelerações semelhantes ao Numba.

Aqui está uma resposta completa sobre as diferenças entre um Ahead Of Time (AOT), um compilador Just In Time (JIT) e um interpretador.

Experimentos com vetorização e com código nativo

Primeiro, para se familiarizar com o Numba e, em seguida, retornamos ao problema de características polinomiais para regressão no conjunto de dados de diabates.

In [18]:
```
%matplotlib inline
```

In [19]:
```
import math, numpy as np, matplotlib.pyplot as plt
from pandas_summary import DataFrameSummary
from scipy import ndimage
```

In [20]:
```
from numba import jit, vectorize, guvectorize, cuda, float32, void, float64
```

Será possível observar o impacto de:

- Evitar alocações de memória e cópias (mais lento do que cálculos de CPU)
- Melhor localizações
- Vetorização

Se for utilizado o numpy em arrays inteiros de uma vez, isso criará muitos temporários e será possível usar o cache. Se for utilizado o loop numba por meio de um item do array por vez, não precisamos alocar grandes arrays temporários e pode-se reutilizar os dados armazenados em cache, pois está sendo realizado vários cálculos em cada item do array.

In [21]:
```
# Untype and Unvectorized
def proc_python(xx,yy):
    zz = np.zeros(nobs, dtype='float32')
    for j in range(nobs):
        x, y = xx[j], yy[j]
        x = x*2 - ( y * 55 )
        y = x + y*2
        z = x + y + 99
        z = z * ( z - .88 )
        zz[j] = z
    return zz
```

In [22]:
```
nobs = 10000
x = np.random.randn(nobs).astype('float32')
y = np.random.randn(nobs).astype('float32')
```

In [23]:
```
%timeit proc_python(x,y)   # Untyped and unvectorized
```

```
158 ms ± 23.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Numpy

Utilizando Numpy é possível realizar a vetorizarização:

In [24]:
```
def proc_numpy(x,y):
    z = np.zeros(nobs, dtype='float32')
    x = x*2 - ( y * 55 )
    y = x + y*2
    z = x + y + 99
    z = z * ( z - .88 )
    return z
```

In [25]:
```
np.allclose( proc_numpy(x,y), proc_python(x,y), atol=1e-4 )
```

Out[25]:
```
True
```

In [26]:
```
%timeit proc_numpy(x,y)    # Typed and vectorized
```

```
89.3 µs ± 10.6 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Numba

Numba oferece vários #decorators diferentes. Tentaremos dois diferentes:

- `@ jit` : muito geral
- `@ vectorize` : não precisa escrever um loop for. útil ao operar em vetores do mesmo tamanho

Primeiro, será utilizado o decorator de compilador `jit` (just-in-time) do Numba, sem vetorizar explicitamente. Isso evita grandes alocações de memória, então temos melhor locação:

In [27]:
```
@jit()
def proc_numba(xx,yy,zz):
    for j in range(nobs):
        x, y = xx[j], yy[j]
        x = x*2 - ( y * 55 )
        y = x + y*2
        z = x + y + 99
        z = z * ( z - .88 )
        zz[j] = z
    return zz
```

In [28]:
```
z = np.zeros(nobs).astype('float32')
np.allclose( proc_numpy(x,y), proc_numba(x,y,z), atol=1e-4 )
```

Out[28]: True

In [29]:
```
%timeit proc_numba(x,y,z)
```

```
12.4 µs ± 1.34 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Agora, será utilizado o decorator `vectorize` do Numba. O compilador do Numba otimiza isso de uma maneira mais inteligente do que é possível com Python e Numpy simples.

In [30]:
```
@vectorize
def vec_numba(x,y):
    x = x*2 - ( y * 55 )
    y = x + y*2
    z = x + y + 99
    return z * ( z - .88 )
```

In [31]:
```
np.allclose(vec_numba(x,y), proc_numba(x,y,z), atol=1e-4 )
```

Out[31]: True

In [32]:
```
%timeit vec_numba(x,y)
```

```
16.5 µs ± 4.38 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Assim, observa-se o poder do **Numba** que é impressionante!

Recursos polinomiais do Numba

In [33]:
```
@jit(nopython=True)
def vec_poly(x, res):
```

```
    m,n=x.shape
    feat_idx=0
    for i in range(n):
        v1=x[:,i]
        for k in range(m): res[k,feat_idx] = v1[k]
        feat_idx+=1
        for j in range(i,n):
            for k in range(m): res[k,feat_idx] = v1[k]*x[k,j]
            feat_idx+=1
```

Row-Major vs Column-Major Storage

Desta [postagem no blog de Eli Bendersky] (http://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays/): "O layout da linha principal de uma matriz coloca a primeira linha na memória contígua, depois a segunda linha logo após, a terceira e assim por diante. O layout da coluna principal coloca a primeira coluna na memória contígua, depois a segunda, etc. .... Embora saber qual layout um determinado conjunto de dados está usando seja crítico para um bom desempenho, não há uma resposta única para a questão de qual layout 'é melhor' em geral.

"Acontece que combinar a maneira como seu algoritmo funciona com o layout de dados pode melhorar ou prejudicar o desempenho de um aplicativo.

"O resumo é: **sempre cruze os dados na ordem em que foram dispostos**."

Layout da coluna principal: Fortran, Matlab, R e Julia

Layout de linha principal: C, C ++, Python, Pascal, Mathematica

In [34]:
```
trn = np.asfortranarray(trn)
test = np.asfortranarray(test)
```

In [35]:
```
m,n=trn.shape
n_feat = n*(n+1)//2 + n
trn_feat = np.zeros((m,n_feat), order='F')
test_feat = np.zeros((len(y_test), n_feat), order='F')
```

In [36]:
```
vec_poly(trn, trn_feat)
vec_poly(test, test_feat)
```

In [37]:
```
regr.fit(trn_feat, y_trn)
```

Out[37]:
```
LinearRegression()
```

In [38]:
```
regr_metrics(y_test, regr.predict(test_feat))
```

Out[38]:
```
(65.1113866428662, 52.137949908019266)
```

In [39]:
```
%timeit vec_poly(trn, trn_feat)
```

```
17.5 µs ± 1.26 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Lembrete de que essa foi a época da implementação do scikit learn PolynomialFeatures, que foi criada por

especialistas:

In [40]:
```
%timeit poly.fit_transform(trn)
```

```
662 µs ± 102 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Como é possível observar o Numba apresenta um ganho de performace incrível! Com uma única linha de código, geralmente em média dependendo do poder computacional se é obtido uma aceleração de 78 vezes em relação ao scikit learn (que foi otimizado por especialistas).

Regularização e ruído

A regularização é uma forma de reduzir o sobreajuste e criar modelos que generalizem melhor para obtenção de novos dados.

Regularização

A regressão Lasso usa uma penalidade L1, que empurra para coeficientes esparsos. O parâmetro α é usado para ponderar o termo de penalidade. O LassoCV do Scikit Learn realiza validação cruzada com vários valores diferentes para α. Assista a este vídeo do Coursera sobre regressão Lasso para obter mais informações.

In [41]:
```
reg_regr = linear_model.LassoCV(n_alphas=10)
```

In [42]:
```
reg_regr.fit(trn_feat, y_trn)
```

```
C:\Users\jarde\Anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:527: Conver
genceWarning: Objective did not converge. You might want to increase the number of iterations. Dua
lity gap: 14633.597475241055, tolerance: 167.76957021276598
  tol, rng, random, positive)
C:\Users\jarde\Anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:527: Conver
genceWarning: Objective did not converge. You might want to increase the number of iterations. Dua
lity gap: 37450.013853975805, tolerance: 169.5592570921986
  tol, rng, random, positive)
C:\Users\jarde\Anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:527: Conver
genceWarning: Objective did not converge. You might want to increase the number of iterations. Dua
lity gap: 6187.253244669642, tolerance: 166.78198657243817
  tol, rng, random, positive)
```

Out[42]:
```
LassoCV(n_alphas=10)
```

In [43]:
```
reg_regr.alpha_
```

Out[43]:
```
0.009656131375463322
```

In [44]:
```
regr_metrics(y_test, reg_regr.predict(test_feat))
```

Out[44]:
```
(53.22433795374341, 44.09299567937261)
```

ruído

Agora será adicionado algum ruído aos dados

In [45]:
```
idxs = np.random.randint(0, len(trn), 10)
```

In [46]:
```
y_trn2 = np.copy(y_trn)
y_trn2[idxs] *= 10 # label noise
```

In [47]:
```
regr = linear_model.LinearRegression()
regr.fit(trn, y_trn)
regr_metrics(y_test, regr.predict(test))
```

Out[47]: (54.406831103411726, 44.685831292097596)

In [48]:
```
regr.fit(trn, y_trn2)
regr_metrics(y_test, regr.predict(test))
```

Out[48]: (74.08903254608788, 58.642846024731185)

A perda de Huber é uma função de perda menos sensível a outliers do que a perda de erro quadrático. É quadrático para pequenos valores de erro e linear para grandes valores.

$$L(x) = \begin{cases} \frac{1}{2}x^2, & \text{para } |x| \leq \delta \\ \delta(|x| - \frac{1}{2}\delta), & \text{Caso contrário} \end{cases}$$

In [49]:
```
hregr = linear_model.HuberRegressor()
hregr.fit(trn, y_trn2)
regr_metrics(y_test, hregr.predict(test))
```

```
C:\Users\jarde\Anaconda3\lib\site-packages\sklearn\linear_model\_huber.py:296: ConvergenceWarning:
lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
  self.n_iter_ = _check_optimize_result("lbfgs", opt_res, self.max_iter)
```

Out[49]: (54.70857501798143, 45.301044643818095)