Regressão Logística

O problema

Temos um conjunto de dados anônimos de aproximadamente 200 usuários, contendo o **salário** de cada usuário, seus anos de **experiência** como cientista de dados e se ele **pagou** uma conta premium. Como é comum em variáveis categóricas, representaremos a variável dependente (variável resposta) como 0 para sem conta premium ou 1 para com conta premium.

Como de costume, nossos dados estão em uma matriz onde cada linha é uma lista **[experiência, salário, conta_paga]** ([experience, salary, paid_account]). Vamos transformá-lo no formato que precisamos:

In [7]:

```
from collections import Counter
import matplotlib.pyplot as plt
import random
import math
import numpy as np
from linear_algebra import *
import gradient_descent as gd
from working_with_data import rescale
from machine_learning import train_test_split
```

In [32]:

```
def predict(x_i, beta):
    """assumes that the first element of each x_i is 1"""
    return dot(x_i, beta)

def error(x_i, y_i, beta):
    return y_i - predict(x_i, beta)

def squared_error(x_i, y_i, beta):
    return error(x_i, y_i, beta) ** 2

def squared_error_gradient(x_i, y_i, beta):
    """the gradient (with respect to beta) corresponding to the ith square
    return [-2 * x_ij * error(x_i, y_i, beta) for x_ij in x_i]

def in_random_order(data):
    """generator that returns the elements of data in random order"""
    indexes = [i for i, _ in enumerate(data)] # create a list of indexes
    random.shuffle(indexes) # shuffle them
    for i in indexes: # return the data in that order
        yield data[i]

def minimize_stochastic(target_fn, gradient_fn, x, y, theta_0, alpha_0=0.0
    data = zip(x, y)
    theta = theta_0 # initial guess
    alpha = alpha_0 # initial step size
    min_theta, min_value = None, float("inf") # the minimum so far
    iterations_with_no_improvement = 0
    # if we ever go 100 iterations with no improvement, stop
    while iterations_with_no_improvement < 1000:
        value = sum( target_fn(x_i, y_i, theta) for x_i, y_i in data )
        if value < min_value:
            # if we've found a new minimum, remember it
            # and go back to the original step size
            min_theta, min_value = theta, value
            iterations_with_no_improvement = 0
            alpha = alpha_0
        else:
            # otherwise we're not improving, so try shrinking the step size
            iterations_with_no_improvement += 1
            alpha *= 0.9

        # and take a gradient step for each of the data points
        for x_i, y_i in in_random_order(data):
            gradient_i = gradient_fn(x_i, y_i, theta)
            theta = vector_subtract(theta, scalar_multiply(alpha, gradient_

    return min_theta

def estimate_beta(x, y):
    beta_initial = [random.random() for x_i in x[0]]
    return minimize_stochastic(squared_error,
                               squared_error_gradient,
                               x, y,
                               beta_initial,
                               0.001)
```

In [33]:

```
data = [(0.7,48000,1),(1.9,48000,0),(2.5,60000,1),(4.2,63000,0),(6,76000,0
data = list(map(list, data)) # change tuples to lists
```

In [34]:

```
#data
```

In [35]:

```
x = [[1] + row[:2] for row in data] # each element is [1, experience, sala
y = [row[2] for row in data] # each element is paid_account
```

In [36]:

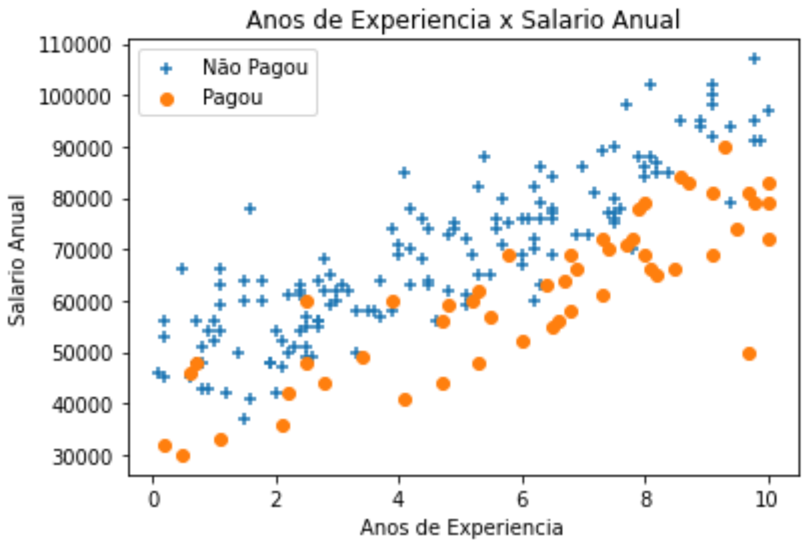
```
#x
#y
```

Uma primeira tentativa óbvia é usar a regressão linear e encontrar o melhor modelo:

$$\text{paid account } = \beta_0 + \beta_1 \text{experience } + \beta_2 \text{salary } + \varepsilon$$

In [13]:

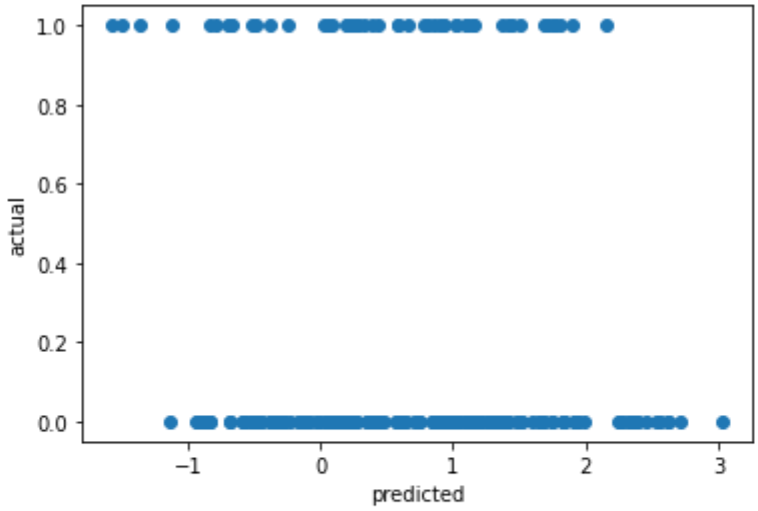
```
plt.scatter([xi[1] for xi,yi in zip(x,y) if yi == 0],
            [xi[2] for xi,yi in zip(x,y) if yi == 0],
            marker="+", label="Não Pagou")
plt.scatter([xi[1] for xi,yi in zip(x,y) if yi == 1],
            [xi[2] for xi,yi in zip(x,y) if yi == 1],
            marker="o", label="Pagou")
plt.legend(loc='upper left');
plt.title("Anos de Experiencia x Salario Anual")
plt.ylabel("Salario Anual")
plt.xlabel("Anos de Experiencia")
plt.show()
```



E certamente não há nada que nos impeça de modelar o problema dessa maneira. o resultados são mostrados abaixo:

In [37]:

```
rescaled_x = rescale(x)
beta = estimate_beta(rescaled_x, y) # [0.26, 0.43, -0.43]
#beta = estimate_beta(x, y)
predictions = [predict(x_i, beta) for x_i in rescaled_x]
plt.scatter(predictions, y)
plt.xlabel("predicted")
plt.ylabel("actual")
plt.show()
```



Mas essa abordagem leva a alguns problemas, o principal deles é:

- Gostaríamos que nossos resultados previstos fossem 0 ou 1, para indicar a qual classe pertence. Tudo bem se eles estiverem entre 0 e 1, já que podemos interpretá-los como probabilidades onde uma saída de 0,25 poderia significar 25% de chance de ser um membro pago. Mas as saídas do modelo linear podem ser números positivos enormes ou até números negativos, o que não fica claro como interpretar. De fato, muitas de nossas previsões foram negativas.

O que gostaríamos é que valores positivos grandes de ponto (x_i, beta) correspondam a probabilidades próximas a 1 e que valores negativos grandes correspondam a probabilidades próximas a 0. Podemos conseguir isso aplicando outra função ao resultado.

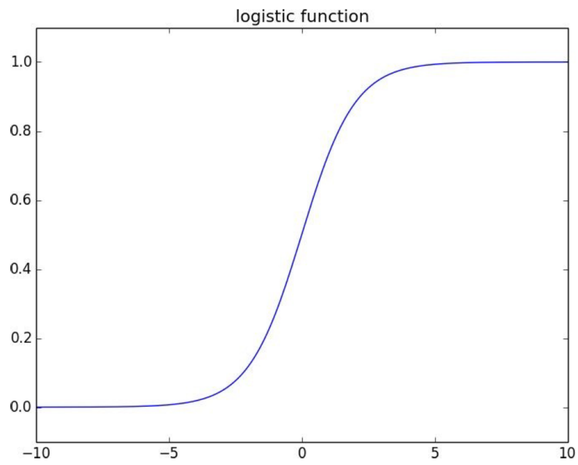
A função logística

No caso da regressão logística, usamos a função logística ou uma curva logística é uma função cuja curva tem o formato de "S" comum (curva sigmoide), com equação:

$$f(x) = \frac{L}{1 + \mathrm{e}^{-k(x-x_0)}}$$

Onde:

- e = a base dos logaritmos naturais (também conhecido como número de Euler)
- x0 = o valor de x no ponto médio da curva sigmoide
- L = o valor máximo da curva, e
- k = a declividade da curva.



A função foi nomeada em 1844–1845 por Pierre François Verhulst, que estudou isso relacionando ao crescimento populacional. O estágio inicial de crescimento é aproximadamente exponencial;então, conforme a saturação se inicia, o crescimento diminui, e na maturidade,o crescimento para.

A **função logística padrão** é a função logística com parâmetros (k = 1, x0 = 0, L = 1) que produz:

$$f(x) = \frac{1}{1 + e^{-x}}$$

In [15]:

```
def logistic(x):
    return 1.0 / (1 + math.exp(-x))
```

À medida que sua entrada se torna grande e positiva, ela se aproxima e se aproxima de 1. À medida que sua entrada se torna grande e negativa, ela se aproxima e se aproxima de 0.

Lembre-se de que, para a regressão linear, ajustamos o modelo minimizando a soma dos erros quadrados, o que resultou na escolha do "beta" que maximizou a probabilidade dos dados.

Não conseguimos usar um calculo baseado na distância (erro) como o R2 com Regressão Logística, por isso usaremos gradiente descendente para maximizar a probabilidade diretamente. Isso significa que precisamos calcular a função de probabilidade e seu gradiente.

Dado algum "beta", nosso modelo diz que cada "yi" deve ser igual a 1 com probabilidade "f(xi*beta)" e 0 com probabilidade "1 - f(xi*beta)".

In [16]:

```
def logistic_log_likelihood_i(x_i, y_i, beta):
    if y_i == 1:
        return math.log(logistic(dot(x_i, beta)))
    else:
        return math.log(1 - logistic(dot(x_i, beta)))
```

Se assumirmos que pontos de dados diferentes são independentes um do outro, a probabilidade geral é apenas o produto das probabilidades individuais. O que significa que a probabilidade geral do log é a soma das probabilidades do log individual:

In [17]:
```
def logistic_log_likelihood(x, y, beta):
    return sum(logistic_log_likelihood_i(x_i, y_i, beta)
               for x_i, y_i in zip(x, y))
```

Um pouco de cálculo nos dá o gradiente:

In [18]:
```
def logistic_log_partial_ij(x_i, y_i, beta, j):
    """here i is the index of the data point, j the index of the derivativ
    return (y_i - logistic(dot(x_i, beta))) * x_i[j]

def logistic_log_gradient_i(x_i, y_i, beta):
    """the gradient of the log likelihood corresponding to the ith data po
    return [logistic_log_partial_ij(x_i, y_i, beta, j)
            for j, _ in enumerate(beta)]

def logistic_log_gradient(x, y, beta):
    return reduce(vector_add,
                  [logistic_log_gradient_i(x_i, y_i, beta)
                   for x_i, y_i in zip(x,y)])
```

Aplicando o modelo

Vamos dividir nossos dados em um conjunto de treinamento e um conjunto de testes:

In [25]:
```
random.seed(0)
x_train, x_test, y_train, y_test = train_test_split(rescaled_x, y, 0.33)

# want to maximize log likelihood on the training data
fn = partial(logistic_log_likelihood, x_train, y_train)
gradient_fn = partial(logistic_log_gradient, x_train, y_train)

# pick a random starting point
beta_0 = [random.random() for _ in range(3)]

# and maximize using gradient descent
beta_hat = gd.maximize_batch(fn, gradient_fn, beta_0)
```

Alternativamente, você pode usar a descida de gradiente estocástica:

In [26]:
```
beta_hat = gd.maximize_stochastic(logistic_log_likelihood_i,
                                  logistic_log_gradient_i,
                                  x_train, y_train, beta_0)
```

De qualquer maneira, encontramos aproximadamente:

In [27]:
```
beta_hat
```

Out[27]:
```
[-1.9042766078873332, 4.047489956690998, -3.8751362233682034]
```

In [28]:
```
#beta_hat_unscaled = [7.61, 1.42, -0.000249]
```

Infelizmente, estes valores não são tão fáceis de interpretar como os coeficientes de regressão linear. Se os demais valores forem iguais, um ano extra de experiência acrescenta 1,42 à entrada da logística. Se os demais valores forem iguais, um valor extra de $ 10.000 de salário subtrai 2.49 da entrada de logística.

O impacto na saída, no entanto, também depende dos outros insumos. Se "dot(beta, x_i)" já for grande (correspondendo a uma probabilidade próxima de 1), aumentá-lo mesmo por muito não pode afetar muito a probabilidade. Se for próximo de 0, aumentá-lo um pouco pode aumentar bastante a probabilidade.

O que podemos dizer é que - sendo tudo o mais igual - as pessoas com mais experiência têm maior probabilidade de pagar pelas contas. E isso - sendo tudo o mais igual - as pessoas com salários mais altos têm menor probabilidade de pagar pelas contas. (Isso também ficou aparente quando plotamos os dados.)

Qualidade do ajuste

Ainda não usamos os dados de teste que divulgamos. Vejamos o que acontece se prevermos uma conta paga sempre que a probabilidade for superior a 0,5:

In [29]:

```
true_positives = false_positives = true_negatives = false_negatives = 0
for x_i, y_i in zip(x_test, y_test):
    predict = logistic(dot(beta_hat, x_i))
    if y_i == 1 and predict >= 0.5: # TP: paid and we predict paid
        true_positives += 1
    elif y_i == 1: # FN: paid and we predict unpaid
        false_negatives += 1
    elif predict >= 0.5: # FP: unpaid and we predict paid
        false_positives += 1
    else: # TN: unpaid and we predict unpaid
        true_negatives += 1

precision = true_positives / (true_positives + false_positives)
recall = true_positives / (true_positives + false_negatives)
```

In [30]:

```
print("Precision =", precision)
print("Recall =", recall)
```

```
Precision = 0.9333333333333333
Recall = 0.8235294117647058
```

Isso dá uma precisão de 93% ("quando prevemos que a conta paga está certa 93% do tempo") e um recall de 82% ("quando um usuário tem uma conta paga, prevemos uma conta paga 82% do tempo")), ambos são números bastante respeitáveis.

In [ ]: