第二章: 感知机　理论与编码实现

田雅夫

December 15, 2016

摘　要

李航老师”统计学习方法”的读书笔记. 同时提供了一份我自己实现的感知机代码. 若文中有任何错误, 还请大家联系我, 我会第一时间改正并更新本文的版本. 本文可以在我的 Github 上免费下载, 地址是:https://github.com/SweetYafu/Blog

1　感知机模型

- 最简单的模型
- 感知机是二类, 线性分类模型. 要求给定数据线性可分.
- 感知机算法本质: 求一个超平面, 使得预定义的损失函数最小化
- 模型, 策略, 算法:
  - 模型: 超平面 (对二维空间就是直线), 线性模型. 若样本维数为 n, 假设空间即 $\mathbb{R}^{n+1}$[1]
  - 策略: 损失函数
  - 算法: 梯度下降法
- 超平面方程: $\boldsymbol{w}\cdot\boldsymbol{x}+b=0$, $\boldsymbol{w},\boldsymbol{x}$ 是与样本 $\boldsymbol{x}$ 相同维数的向量
- 损失函数: $-\sum\frac{1}{\|\boldsymbol{w}\|}y_i(\boldsymbol{w}\cdot\boldsymbol{x}_i+b)$, 只考虑所有被错误分类的点. 感知机算法即最优化这样一个函数.
- 感知机的形式: $sign(\boldsymbol{w}\cdot\boldsymbol{x}+b)$, $sign(x)$ 是符号函数
- 与 SVM 不同, 感知机只能做到产生一个分割, 但并不能产生一个” 很好的” 分割. 对于分类问题, 感知机是可用的. 但如果用来预测新样本的属性, 最好还是用 SVM.
- 对于线性不可分的数据, SVM 算法不能收敛, 会产生震荡现象. 解决方法 (两种要同时使用):
  - 规定最大迭代次数
  - 每次更新算法的参数当且仅当在该参数下误分割的样本数量减少了.

2　感知机的编码实现

对于任意分类器的编码实现, 主要要实现的就是如下四个函数:

- ___init___(*args, **kwargs) : 分类器的初始化
- ___str___ (*args, **kwargs): 显示分类器的状态 (重载 print 运算符)
- train(dataSet) : 用数据集训练分类器
- predict(sample) : 对于一个新的样本, 预测该样本的归属
- (非必须) 对于增量式学习, 还要增加一个”incremental(sample)” 函数.

知道了分类器的代码结构, 剩下的工作就是往函数定义里面填空了. 感知机有两种相互对偶的训练方式. 在李航老师的书中已经很详细的叙述了 (P29/P31). 本文后面附加的代码中对这两种形式都进行了实现.

[1]一个 n 阶超平面用一个 n 维多项式刻画, 而这个 n 维多项式可以唯一的用一个 $n+1$ 维的向量表示 (别忘了常数项:))

需要注意的是, 在感知机算法的对偶形式一节中, 式

$$y_i(\sum_{j=1}^{N} a_j y_j x_i \cdot x_j + b) \leq 0 \qquad (1)$$

中的 $x_i \cdot x_j$ 就是前文提到的格拉姆矩阵的第 ij 个元素. 考虑到格拉姆矩阵是对称阵, 所以只算一半就行. (事实上应该用一个一维数组来表示格拉姆矩阵, 这里为方便调试还使用矩阵形式)

在感知机对偶形式中, 求取了 α_i 后还要通过式 2.14 求 w.

用两个例子来对该算法的正确性进行测试. 第一个例子就是原书中的例题. 第二个例子是给定平面上两个圆, 数据的正例与反例随机分布在这两个圆中. 考虑到感知机算法要求数据线性可分, 所以要保证两圆心间距离大于两圆半径之和. 读者可以更改本文附录中代码关于生成数据集部分的参数, 然后观察现象.

3 习题

1. 异或函数是线性不可分的, 所以无法用感知机进行分类. 考虑所有样本的维数都是 2(即平面上) 的情况, 异或函数的结果就是在 $y = x$ 直线上为正例, 在平面其他地方为反例. 明显是线性不可分的.

2. 见代码. 运行结果如图

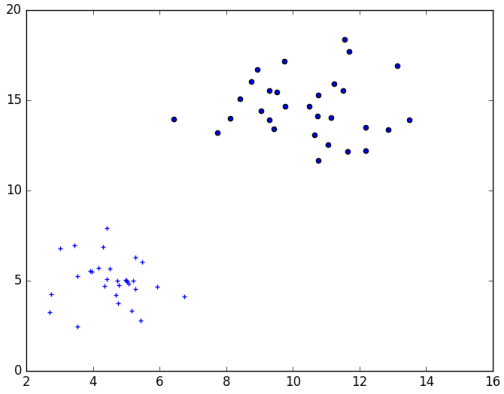

图 1: 算法运行结果

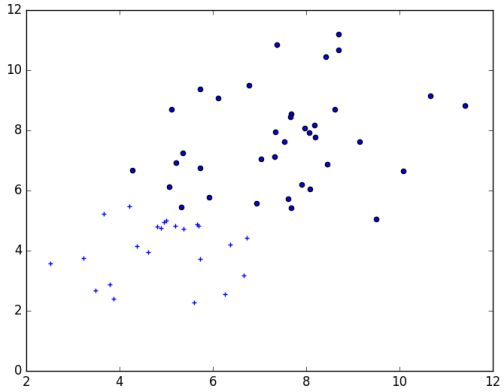

图 2: 算法运行结果 (两类线性不可分的情况)

2.3 待续, 有空补上

```
## main.py
__author__ = 'tian'
from matplotlib import pyplot
import numpy as np
import pylab as pl
import math
import time

import tools

def countTime_deco(func):
    '''
    Decorator to count the running time of a function
    :param func: Don't use this function directly
    :return: God Know That (not determined, depends on the function to decorate)
    '''
    def tempFunc(*args, **kwargs):
        start = time.clock()
        ret = func(*args, **kwargs)
        end = time.clock()
        print "Running function '%s', running time = %s ms" %(func.__name__,
        %%%str((end-start)*1000))
        return ret
    return tempFunc

class Perceptron:
    '''
    Perceptron Algo
    '''
    def __init__(self, _w = np.array([0,0]), _b = 0, _eta = 1):
        '''
        init modal : f(x) = sign(w * x + b)
        :param _w: par
        :param _b: par
        :param _eta : par -> step length
        :return: None
        '''
        self.w = _w.T
        self.b = _b
        self.eta = _eta

    def __str__(self):
        '''
        Get the status of perception
        :return: str
        '''
        return "Algo Perceptron, w = %s, b = %s \n" %(str(self.w), str(self.b))

    @countTime_deco
    def train(self, data, maxIteration = 30):
        '''
        Train the algo
        :param data: np.array sample: testData = np.array([(3, 3, 1), (4, 3, 1), (1, 1, 0)])
        ## x[0], x[1] : data; x[2] : label
        :param maxIteration : maximum amount of data iteration
        :return: None
        '''
```

```
        allPassFlag = False
        count = 0
        while(allPassFlag == False and count < maxIteration):
            #print "New turn"
            for sample in data:
                #print self
                if((np.dot(self.w, sample[0]) + self.b) * sample[1] <= 0):
                    ## wx+b <= 0
                    self.w = self.w + self.eta * (sample[0] * sample[1])
                    self.b = self.b + self.eta * sample[1]
                    allPassFlag = False
                    count += 1
                    break
            else:
                ## no mistake data exists
                allPassFlag = True
        if(allPassFlag == False):
            print "Reached Max Step Number!"
        else:
            print "Iteration %s steps" %str(count)
        print self

    @countTime_deco
    def train_anotherForm(self, data, maxIteration = 30):
        '''
        train the classifier with another form in page 33-35
        :param data: np.array sample: testData = np.array([(3, 3, 1), (4, 3, 1), (1, 1, 0)])
        :param maxIteration: maximum amount of data iteration
        :return: None
        '''
        dim = len(data)
        GramMatrix = [[0 for i in range(dim)] for i in range(dim)]
        for i in range(dim):
            for j in range(i, dim):
                tempValue = np.dot(data[i][0], data[j][0])
                GramMatrix[i][j] = tempValue
                GramMatrix[j][i] = tempValue

        # for i in GramMatrix:
        # print i
        alpha_i = [0 for i in range(dim)]
        allPassFlag = False
        count = 0
        while(allPassFlag == False and count <= maxIteration):
            #print "new iteration"
            for index in range(dim):
                if(data[index][1] * (sum([alpha_i[i] * data[i][1] * GramMatrix[index][i] for i in
                range(dim)]) + self.b) <= 0):
                    #print self, "alpha_i = ", alpha_i
                    alpha_i[index] += self.eta
                    self.b += data[index][1]
                    count += 1
                    break
            else:
                allPassFlag = True
        if(allPassFlag == False):
            print "Reached Max Step Number!"
```

```
        else:
            print "Iteration %s steps" %str(count)
        self.w = sum([alpha_i[i] * data[i][1] * data[i][0] for i in range(dim)])
        print self

    def predict(self, sample):
        '''
        predict a sample
        :param sample: x:(x0, x1, ..., xn)
        :return: int (1 or -1)
        '''
        return tools.sign(np.dot(self.w, sample) + self.b)

def showPreceptronResults(preceptronInstance, testData):
    fig = pyplot.figure()
    ax = fig.add_subplot("111")

    #positiveSample = [i[0] for i in testData if i[1] == 1]
    #negativeSample = [i[0] for i in testData if i[1] == -1]
    positiveSample = [i[0] for i in testData if preceptronInstance.predict(i[0]) == 1]
    negativeSample = [i[0] for i in testData if preceptronInstance.predict(i[0]) == -1]
    print "positive", [i[0] for i in positiveSample], [i[1] for i in positiveSample]
    print "negative", [i[0] for i in negativeSample], [i[1] for i in negativeSample]
    ax.scatter([i[0] for i in positiveSample], [i[1] for i in positiveSample], marker = '+')
    ax.scatter([i[0] for i in negativeSample], [i[1] for i in negativeSample], marker = 'o')
    pyplot.show()


if(__name__ == "__main__"):
    ## Test
    ## x[x1, x2, ..., xn] : data; x[2] : label

    # testData = [(np.array(i[0]), i[1]) for i in [([3, 3], 1), ([4, 3], 1), ([1, 1], -1)]]
    # algo = Perceptron()
    # algo.train_anotherForm(testData)
    # showPreceptronResults(algo, testData)


    testData = [(np.array(i), 1) for i in tools.createCircleDataSet(5, 5, 3, 30)] + [(np.array(i),
    -1) for i in tools.createCircleDataSet(10, 15, 4, 30)]
    algo = Perceptron()
    algo.train_anotherForm(testData, maxIteration=1000)
    showPreceptronResults(algo, testData)

    ## 288ms with all console output
    ## 13.942ms if no output, not very quick but enough
    ## 74ms in another form......

## tools.py
__author__ = 'tian'
import math
import random

def sign(x):
    '''
    sign function
    :param x:
    :return: int (1 or -1)
```

```
    '''
    if(x >= 0):
        return 1
    else:
        return -1

def dot(vector1, vector2):
    '''
    the inner product of two vectors
    :param vector1:
    :param vector2:
    :return: int
    '''
    dim = len(vector1)
    count = 0
    if(dim != len(vector2)):
        raise ValueError
    for i in range(dim):
        count += vector1[i] * vector2[i]
    return count

def createCircleDataSet(_x, _y, _r, amount):
    '''
    Create a 2D dataset [(x1, y1), (x2, y2), ...], all points are in a circle
    :param _x: X coord of the center of circle
    :param _y: Y coord of the center of circle
    :param _r: radius of circle
    :param amount: length of dataSets
    :return: list : [(x1, y1), (x2, y2), ...]
    '''
    polarPointList = [(random.uniform(0, _r), math.radians(random.uniform(0, 360))) for i in
    range(amount)]
    return [(i[0] * math.cos(i[1]) + _x, (i[0] * math.sin(i[1]) + _y) ) for i in polarPointList]
```