2.1 模型

假设输入空间是 $x \subseteq R^n$，输出空间是$y = +1, -1$，x和y分属这两个空间，那么由输入空间到输出空间的如下函数：

$$f(x) = sign(w \cdot x + b)$$

称为**感知机**。其中，w和b称为感知机模型参数，$w \subseteq R^n$叫做权值或**权值向量**，$b \in R$叫做**偏置**(bias)，$w \cdot x$表示向量w和x的**内积**。sign是一个函数：

$$sign(x) = \begin{cases} 1 & x \geq 0 \\ -1 & x \leq 0 \end{cases}$$

感知机是定义在特征空间中的所有**线性分类模型**或线性分类器，，属于**判别模型**，即函数集合$\{f|f(x) = w \cdot x + b\}$，其**几何解释**是，线性方程

$$w \cdot x + b = 0$$

将特征空间划分为正负两个部分：

感知机模型.png

这个平面（2维时退化为直线）称为**分离超平面**(separating hyperplane，$n > 3$)。

2.2 学习策略

2.2.1 数据集的线性可分性

给定数据集

$$T = \{(x_1, y_1), (x_2, y_2), \cdots (x_N, y_N)\}$$

其中 $x_i \in X = R^n$ ，$y_i \in Y = +1, -1$，，$i = 1, 2, \cdots, N$，如果存在某个超平面S

$$w \cdot x + b = 0$$

能够**完全正确**地将正负实例点全部分割开来，则称**T线性可分**，否则称T线性不可分。

2.2.2 学习策略(损失函数)

假定数据集线性可分，我们希望找到一个合理的损失函数。

(1) 采用**误分类点的总数**

但是这样的损失函数不是参数w，b的连续可导函数，不可导自然不能把握函数的变化，也就不易优化（不知道什么时候该终止训练，或终止的时机不是最优的）。

(2) 选择**所有误分类点到超平面S的总距离**

点x_0到平面S的距离：

$$\frac{w \cdot x_0 + b}{||w||}$$

其中，$||w||$是w的L_2**范数**，$||w|| = \sqrt{w_1^2 + w_2^2 + \cdots + w_N^2}$。类比点到平面的距离，此处的点到超平面S的距离的几何意义就是上述距离在多维空间的推广。

$$d(x_i, y_i) = \frac{|ax_i + by_i + c|}{\sqrt{a^2 + b^2}}$$

若点i被**误分类**一定有

$$-y_i(w \cdot x_i + b) > 0$$

成立，所以我们去掉了绝对值符号，得到**误分类点到超平面S的距离**公式：

$$-\frac{y_i(w \cdot x_i + b)}{||w||}$$

假设所有误分类点构成集合M，那么**所有误分类点到超平面S的总距离**为

$$-\frac{1}{||w||} \sum_{x_i \in M} y_i(w \cdot x_i + b)$$

分母作用不大，反正一定是正的，不考虑分母，就得到了**感知机学习的损失函数**：

$$L(w, b) = -\frac{1}{||w||} \sum_{x_i \in M} y_i(w \cdot x_i + b)$$

2.3 学习算法

2.3.1 原始形式

感知机学习算法是对以下最优化问题的算法：

$$\min L(w,b) = -\sum_{x_i \in M} y_i(w \cdot x_i + b)$$

感知机学习算法是**误分类驱动**的，先随机选取一个超平面(误分类点)，然后用梯度下降法不断极小化上述损失函数。损失函数的梯度：

$$\nabla_w L(w,b) = -\sum_{x_i \in M} y_i x_i$$
$$\nabla_b L(w,b) = -\sum_{x_i \in M} y_i$$

随机选一个误分类点x_i，对参数w，b进行更新：

$$w \leftarrow w + \eta y_i x_i$$
$$b \leftarrow b + \eta y_i$$

上式$\eta(0 < \eta \leq 1)$是**学习率**。损失函数的参数加上**梯度上升的反方向**，于是就梯度下降了。所以，上述迭代可以使损失函数不断减小，直到为0。于是得到了**原始形式**的感知机学习算法：

算法1	感知机学习算法的原始形式
输入	$T = \{(x_1, y_1), (x_2, y_2), \cdots (x_N, y_N)\}$
输出	w , b ; 感知机模型 : $f(x) = sign(w \cdot x + b)$
(1)	选取初始值w_0, b_0
(2)	在训练集中选取数据(x_i, x_i)
(3)	若$y_i(w \cdot x_i + b) \leq 0$
	$w \leftarrow w + \eta y_i x_i$; $b \leftarrow b + \eta y_i$
(4)	转至 (2) ，直至训练集中没有误分类点

- **算法的收敛性**
  Novikoff定理 $\rightarrow$ 表明经过**有限次**搜索可以找到将训练数据完全正确分开的**分离超平面。**

2.3.2 对偶形式

- 基本思想
  (对偶) 将w和b表示为实例 x_i 和标记 y_i 的**线性组合形式**，通过求解系数得到w和b。具体说来，如果对误分类点 x_i 逐步修改wb修改了n次，则w，b关于(x_i, y_i) 的*增量*分别为 $\alpha_i y_i x_i$ 和 $\alpha_i y_i$，这里 $\alpha_i = n_i \eta$，则最终求解到的参数分别表示为：

$$w = \sum_{i=1}^{N} \alpha_i y_i x_i$$

$$b = \sum_{i=1}^{N} \alpha_i y_i$$

- 于是有算法2.2：

算法1 感知机学习算法的原始形式

输入	$T=\{(x_1,y_1),(x_2,y_2),\cdots(x_N,y_N)\}$
输出	α , b ; 感知机模型 : $f(x)=sign(\sum_{j=1}^{N}\alpha_j y_j x_j \cdot x+b)$
	其中 , $\alpha=(\alpha_1,\alpha_2,\cdots,\alpha_N)^T$
(1)	选取初始值$\alpha \leftarrow 0, b \leftarrow 0$
(2)	在训练集中选取数据(x_i,x_i)
(3)	若$y_i(\sum_{j=1}^{N}\alpha_j y_j x_j \cdot x_i+b) \leq 0$
	$\alpha_i \leftarrow \alpha+\eta$; $b \leftarrow b+\eta y_i$
(4)	转至(2),直至训练集中没有误分类点

由于训练实例仅以**内积**的形式出现，为方便，可预先将训练集中实例间的内积计算出来并以矩阵形式存储，这就是所谓的**Gram矩阵**。

$$G=[x_i \cdot x_j]_{N\times N}$$

2.4 算法实现

1. 原始形式的感知机算法

例 2.1 如图 2.2 所示的训练数据集，其正实例点是 $x_1=(3,3)^T$， $x_2=(4,3)^T$，负实例点是 $x_3=(1,1)^T$，试用感知机学习算法的原始形式求感知机模型 $f(x)=\mathrm{sign}(w\cdot x+b)$．这里， $w=(w^{(1)},w^{(2)})^T$， $x=(x^{(1)},x^{(2)})^T$．

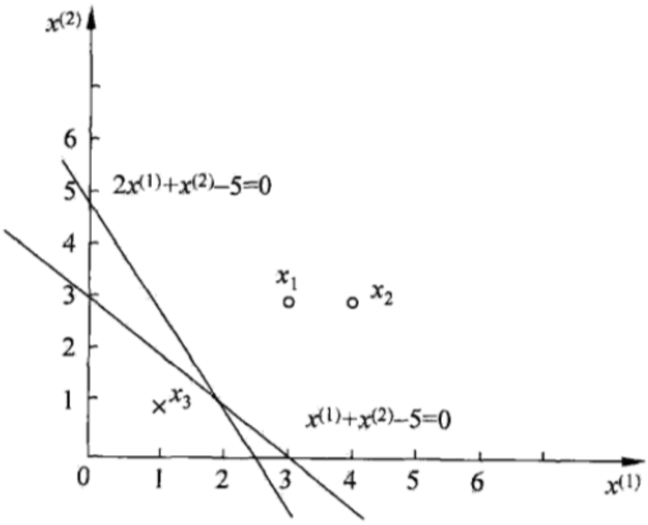

图 2.2 感知机示例

```
weights = []
b = 0
print_pattern = 'Iteration {}, 误分类点: x{}, w={}, b={}'
def update(item, learning_rate = 1):
    ''' 更新权值、偏置 '''
    global weights, b
    for i in range(len(item)):
        weights[i] += learning_rate * item[1] * item[0][i]
```

```python
        b += learning_rate * item[1]

def cal(item):
    ''' 计算 yi*(w*xi + b) <= 0 '''
    res = 0
    for i in range(len(item[0])):
        res += item[0][i] * weights[i]
    res += b
    res *= item[1]
    return res

def check(iteration, learning_rate = 1):
    # 标识是否存在误分类点
    flag = False
    for xi in range(len(data_set)):
        if cal(data_set[xi]) <= 0:
            flag = True
            update(data_set[xi], learning_rate=learning_rate)
            print(print_pattern.format(iteration, xi + 1, weights, b))
    # 误分类点不存在，迭代结束
    return flag

def train(data_set, max_iteration=1000, learning_rate=1):
    ''' 感知机学习算法的原始形式 '''
    for i in range(len(data_set)):
        weights.append(0)
    iteration = 0
    print(print_pattern.format(iteration, 1, weights, b))
    while True:
        if not check(iteration) or iteration>max_iteration:
            break
        iteration += 1

if __name__ == '__main__':
    # 训练集
    data_set = [[(3, 3), 1], [(4, 3), 1], [(1, 1), -1]]
    train(data_set)
```

1. 对偶形式的感知机算法

例 2.2 数据同例 2.1，正样本点是 $x_1 = (3,3)^T$， $x_2 = (4,3)^T$，负样本点是 $x_3 = (1,1)^T$，试用感知机学习算法对偶形式求感知机模型.

```python
import numpy as np
print_pattern = 'Iteration {}, 误分类点: x{}, alphas={}, b={}'

class Perceptron():
    def __init__(self, max_iteration=1000, learning_rate=1):
```

```python
        # 参数向量
        self.alphas = None
        # Gram 矩阵
        self.gram = None
        # 偏置
        self.b = 0
        # 迭代次数
        self.iteration = 0
        # 最大迭代次数
        self.max_iteration = max_iteration
        # 学习率
        self.learning_rate = learning_rate

    def __init(self, data_set):
        ''' 计算 Gram 矩阵、初始化 alphas :return: '''
        N = len(data_set)
        self.alphas = np.zeros(N, dtype=np.float)
        self.gram = np.zeros(shape=(N, N), dtype=int)
        for i in range(len(data_set)):
            for j in range(len(data_set)):
                # 求内积 -> np.dot 点乘或矩阵乘法
                self.gram[i][j] = np.dot(data_set[i][0], data_set[j][0])
        return self.gram

    def __cal(self, i, y):
        ''' 计算距离 '''
        res = np.dot(self.alphas * y, self.gram[i]) + self.b
        res *= y[i]
        return res

    def __update(self, i, y):
        ''' 更新权值、偏置 '''
        self.alphas[i] += self.learning_rate
        self.b += self.learning_rate * y[i]

    def __check(self, y):
        ''' 标识是否存在误分类点 '''
        flag = False
        for i in range(len(y)):
            if self.__cal(i, y) <= 0:
                flag = True
                self.__update(i, y)
                self.iteration += 1
                print(print_pattern.format(self.iteration, i+1, self.alphas, self.b))
        # 误分类点不存在，迭代结束
        return flag

    def train(self, data_set):
        ''' 感知机学习算法的对偶形式
```

```python
        '''
        self.__init(data_set)

        if type(data_set) is not np.ndarray:
            data_set = np.array(data_set)

        # 分类标签
        y = data_set[:, 1]

        while True:
            # 标识是否存在误分类点或是否大于最大迭代次数
            if not self.__check(y) or self.iteration > self.max_iteration:
                break

if __name__ == '__main__':
    perceptron = Perceptron()
    # 训练集
    data_set = [[(3, 3), 1], [(4, 3), 1], [(1, 1), -1]]
    perceptron.train(data_set)
```

2.5 引用

> http://www.hankcs.com/ml/the-perceptron.html
> https://www.cnblogs.com/naonaoling/p/5690219.html
> http://www.cnblogs.com/OldPanda/archive/2013/04/12/3017100.html