第6章 逻辑斯谛回归

逻辑斯谛回归(LR)是经典的分类方法

1．逻辑斯谛回归模型是由以下条件概率分布表示的分类模型。逻辑斯谛回归模型可以用于二类或多类分类。

$P(Y=k|x)=\frac{\exp(w_k\cdot x)}{1+\sum_{k=1}^{K-1}\exp(w_k\cdot x)},\quad k=1,2,\cdots,K-1$

$P(Y=K|x)=\frac{1}{1+\sum_{k=1}^{K-1}\exp(w_k\cdot x)}$ 这里，x为输入特征，w为特征的权值。

逻辑斯谛回归模型源自逻辑斯谛分布，其分布函数$F(x)$是S形函数。逻辑斯谛回归模型是由输入的线性函数表示的输出的对数几率模型。

2．最大熵模型是由以下条件概率分布表示的分类模型。最大熵模型也可以用于二类或多类分类。

$P_w(y|x)=\frac{1}{Z_w(x)}\exp(\sum_{i=1}^{n}w_i f_i(x,y))\ Z_w(x)=\sum_y\exp(\sum_{i=1}^{n}w_i f_i(x,y))$

其中，$Z_w(x)$是规范化因子，f_i为特征函数，w_i为特征的权值。

3．最大熵模型可以由最大熵原理推导得出。最大熵原理是概率模型学习或估计的一个准则。最大熵原理认为在所有可能的概率模型（分布）的集合中，熵最大的模型是最好的模型。

最大熵原理应用到分类模型的学习中，有以下约束最优化问题：

$\min -H(P)=\sum_{x,y}\tilde{P}(x)P(y|x)\log P(y|x)$

$s.t.\quad P(f_i)-\tilde{P}(f_i)=0,\quad i=1,2,\cdots,n$

$\sum_y P(y|x)=1$

求解此最优化问题的对偶问题得到最大熵模型。

4．逻辑斯谛回归模型与最大熵模型都属于对数线性模型。

5．逻辑斯谛回归模型及最大熵模型学习一般采用极大似然估计，或正则化的极大似然估计。逻辑斯谛回归模型及最大熵模型学习可以形式化为无约束最优化问题。求解该最优化问题的算法有改进的迭代尺度法、梯度下降法、拟牛顿法。

回归模型：$f(x)=\frac{1}{1+e^{-wx}}$

其中wx线性函数：$wx=w_0\cdot x_0+w_1\cdot x_1+w_2\cdot x_2+\ldots+w_n\cdot x_n,(x_0=1)$

```
from math import exp
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
```

```
# data
def create_data():
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['label'] = iris.target
    df.columns = ['sepal length', 'sepal width', 'petal length', 'petal
width', 'label']
    data = np.array(df.iloc[:100, [0,1,-1]])
    # print(data)
    return data[:,:2], data[:,-1]
```

```
X, y = create_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

```
class LogisticReressionClassifier:
    def __init__(self, max_iter=200, learning_rate=0.01):
        self.max_iter = max_iter
        self.learning_rate = learning_rate

    def sigmoid(self, x):
        return 1 / (1 + exp(-x))

    def data_matrix(self, X):
        data_mat = []
        for d in X:
            data_mat.append([1.0, *d])
        return data_mat

    def fit(self, X, y):
        # label = np.mat(y)
        data_mat = self.data_matrix(X)  # m*n
        self.weights = np.zeros((len(data_mat[0]), 1), dtype=np.float32)

        for iter_ in range(self.max_iter):
            for i in range(len(X)):
                result = self.sigmoid(np.dot(data_mat[i], self.weights))
                error = y[i] - result
                self.weights += self.learning_rate * error * np.transpose(
```

```
                    [data_mat[i]])
        print('LogisticRegression Model(learning_rate={},max_iter={})'.format(
            self.learning_rate, self.max_iter))

    # def f(self, x):
    #     return -(self.weights[0] + self.weights[1] * x) / self.weights[2]

    def score(self, X_test, y_test):
        right = 0
        X_test = self.data_matrix(X_test)
        for x, y in zip(X_test, y_test):
            result = np.dot(x, self.weights)
            if (result > 0 and y == 1) or (result < 0 and y == 0):
                right += 1
        return right / len(X_test)
```

```
lr_clf = LogisticReressionClassifier()
lr_clf.fit(X_train, y_train)
```

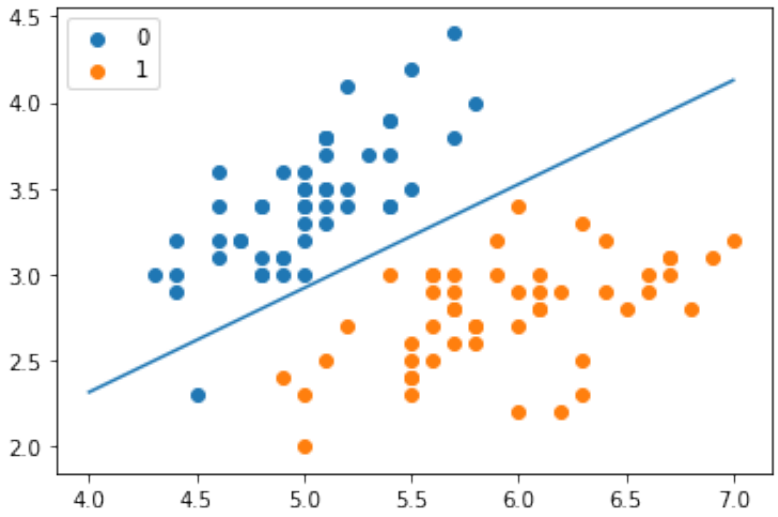
```
LogisticRegression Model(learning_rate=0.01,max_iter=200)
```

```
lr_clf.score(X_test, y_test)
```

```
0.9666666666666667
```

```
x_ponits = np.arange(4, 8)
y_ = -(lr_clf.weights[1]*x_ponits + lr_clf.weights[0])/lr_clf.weights[2]
plt.plot(x_ponits, y_)

#lr_clf.show_graph()
plt.scatter(X[:50,0],X[:50,1], label='0')
plt.scatter(X[50:,0],X[50:,1], label='1')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1b6e7cc16c8>
```



scikit-learn实例

sklearn.linear_model.LogisticRegression

solver参数决定了我们对逻辑回归损失函数的优化方法，有四种算法可以选择，分别是：

- a) liblinear：使用了开源的liblinear库实现，内部使用了坐标轴下降法来迭代优化损失函数。
- b) lbfgs：拟牛顿法的一种，利用损失函数二阶导数矩阵即海森矩阵来迭代优化损失函数。
- c) newton-cg：也是牛顿法家族的一种，利用损失函数二阶导数矩阵即海森矩阵来迭代优化损失函数。
- d) sag：即随机平均梯度下降，是梯度下降法的变种，和普通梯度下降法的区别是每次迭代仅仅用一部分的样本来计算梯度，适合于样本数据多的时候。

```
from sklearn.linear_model import LogisticRegression
```

```
clf = LogisticRegression(max_iter=200)
```

```
clf.fit(X_train, y_train)
```

```
LogisticRegression(max_iter=200)
```

```
clf.score(X_test, y_test)
```

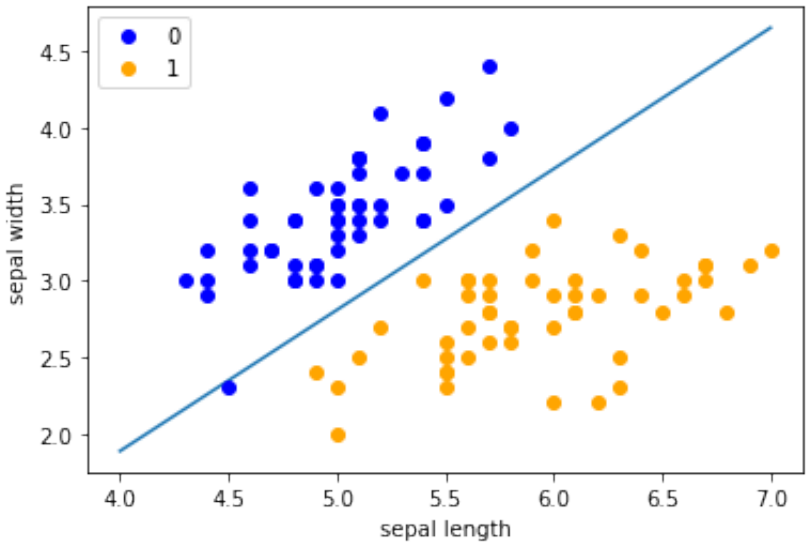
```
0.9666666666666667
```

```
print(clf.coef_, clf.intercept_)
```

```
[[ 2.59546005 -2.81261232]] [-5.08164524]
```

```
x_ponits = np.arange(4, 8)
y_ = -(clf.coef_[0][0]*x_ponits + clf.intercept_)/clf.coef_[0][1]
plt.plot(x_ponits, y_)

plt.plot(X[:50, 0], X[:50, 1], 'bo', color='blue', label='0')
plt.plot(X[50:, 0], X[50:, 1], 'bo', color='orange', label='1')
plt.xlabel('sepal length')
plt.ylabel('sepal width')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1b6e8248fc8>
```



最大熵模型

```
import math
from copy import deepcopy
```

```
class MaxEntropy:
    def __init__(self, EPS=0.005):
        self._samples = []
        self._Y = set()  # 标签集合，相当去去重后的y
```

```
        self._numXY = {}  # key为(x,y)，value为出现次数
        self._N = 0  # 样本数
        self._Ep_ = []  # 样本分布的特征期望值
        self._xyID = {}  # key记录(x,y),value记录id号
        self._n = 0  # 特征键值(x,y)的个数
        self._C = 0  # 最大特征数
        self._IDxy = {}  # key为(x,y)，value为对应的id号
        self._w = []
        self._EPS = EPS  # 收敛条件
        self._lastw = []  # 上一次w参数值

    def loadData(self, dataset):
        self._samples = deepcopy(dataset)
        for items in self._samples:
            y = items[0]
            X = items[1:]
            self._Y.add(y)  # 集合中y若已存在则会自动忽略
            for x in X:
                if (x, y) in self._numXY:
                    self._numXY[(x, y)] += 1
                else:
                    self._numXY[(x, y)] = 1

        self._N = len(self._samples)
        self._n = len(self._numXY)
        self._C = max([len(sample) - 1 for sample in self._samples])
        self._w = [0] * self._n
        self._lastw = self._w[:]

        self._Ep_ = [0] * self._n
        for i, xy in enumerate(self._numXY):  # 计算特征函数fi关于经验分布的期望
            self._Ep_[i] = self._numXY[xy] / self._N
            self._xyID[xy] = i
            self._IDxy[i] = xy

    def _Zx(self, X):  # 计算每个Z(x)值
        zx = 0
        for y in self._Y:
            ss = 0
            for x in X:
                if (x, y) in self._numXY:
                    ss += self._w[self._xyID[(x, y)]]
            zx += math.exp(ss)
        return zx

    def _model_pyx(self, y, X):  # 计算每个P(y|x)
        zx = self._Zx(X)
        ss = 0
        for x in X:
```

```
            if (x, y) in self._numXY:
                ss += self._w[self._xyID[(x, y)]]
        pyx = math.exp(ss) / zx
        return pyx

    def _model_ep(self, index):  # 计算特征函数fi关于模型的期望
        x, y = self._IDxy[index]
        ep = 0
        for sample in self._samples:
            if x not in sample:
                continue
            pyx = self._model_pyx(y, sample)
            ep += pyx / self._N
        return ep

    def _convergence(self):  # 判断是否全部收敛
        for last, now in zip(self._lastw, self._w):
            if abs(last - now) >= self._EPS:
                return False
        return True

    def predict(self, X):  # 计算预测概率
        Z = self._Zx(X)
        result = {}
        for y in self._Y:
            ss = 0
            for x in X:
                if (x, y) in self._numXY:
                    ss += self._w[self._xyID[(x, y)]]
            pyx = math.exp(ss) / Z
            result[y] = pyx
        return result

    def train(self, maxiter=1000):  # 训练数据
        for loop in range(maxiter):  # 最大训练次数
            print("iter:%d" % loop)
            self._lastw = self._w[:]
            for i in range(self._n):
                ep = self._model_ep(i)  # 计算第i个特征的模型期望
                self._w[i] += math.log(self._Ep_[i] / ep) / self._C  # 更新参数
            print("w:", self._w)
            if self._convergence():  # 判断是否收敛
                break
```

```
dataset = [['no', 'sunny', 'hot', 'high', 'FALSE'],
           ['no', 'sunny', 'hot', 'high', 'TRUE'],
           ['yes', 'overcast', 'hot', 'high', 'FALSE'],
           ['yes', 'rainy', 'mild', 'high', 'FALSE'],
           ['yes', 'rainy', 'cool', 'normal', 'FALSE'],
```

```
            ['no', 'rainy', 'cool', 'normal', 'TRUE'],
            ['yes', 'overcast', 'cool', 'normal', 'TRUE'],
            ['no', 'sunny', 'mild', 'high', 'FALSE'],
            ['yes', 'sunny', 'cool', 'normal', 'FALSE'],
            ['yes', 'rainy', 'mild', 'normal', 'FALSE'],
            ['yes', 'sunny', 'mild', 'normal', 'TRUE'],
            ['yes', 'overcast', 'mild', 'high', 'TRUE'],
            ['yes', 'overcast', 'hot', 'normal', 'FALSE'],
            ['no', 'rainy', 'mild', 'high', 'TRUE']]
```

```
maxent = MaxEntropy()
x = ['overcast', 'mild', 'high', 'FALSE']
```

```
maxent.loadData(dataset)
maxent.train()
```

```
iter:0
w: [0.0455803891984887, -0.002832177999673058, 0.031103560672370825,
-0.1772024616282862, -0.0037548445453157455, 0.16394435955437575,
-0.02051493923938058, -0.049675901430111545, 0.08288783767234777,
0.030474400362443962, 0.05913652210443954, 0.08028783103573349,
0.1047516055195683, -0.017733409097415182, -0.12279936099838235,
-0.2525211841208849, -0.033080678592754015, -0.06511302013721994,
-0.08720030253991244]
iter:1
w: [0.11525071899801315, 0.019484939219927316, 0.07502777039579785,
-0.29094979172869884, 0.023544184009850026, 0.2833018051925922,
-0.04928887087664562, -0.101950931659509, 0.12655289130431963,
0.016078718904129236, 0.09710585487843026, 0.10327329399123442,
0.16183727320804359, 0.013224083490515591, -0.17018583153306513,
-0.44038644519804815, -0.07026660158873668, -0.11606564516054546,
-0.1711390483931799]
iter:2
w: [0.18178907332733973, 0.04233703122822168, 0.11301330241050131,
-0.37456674484068975, 0.0559976427099043, 0.38356978711239126,
-0.07488546168160945, -0.14671211613144097, 0.15633348706002106,
-0.011836411721359321, 0.12895826039781944, 0.10572969681821211,
0.19953102749655352, 0.06399991656546679, -0.17475388854415905,
-0.5893308194447993, -0.10405912653008922, -0.16350962040062977,
-0.24701967386590512]
iter:3
```

```
iter:663
w: [3.806361507565719, 0.0348973837073587, 1.6391762776402004,
-4.46082036700038, 1.7872898160522181, 5.305910631880809,
-0.13401635325297073, -2.2528324581617647, 1.4833115301839292,
-1.8899383652170454, 1.9323695880561387, -1.2622764904730739,
1.7249196963071136, 2.966398532640618, 3.904166955381073, -9.515244625579237,
-1.8726512915652174, -3.4821197858946427, -5.634828605832783]
iter:664
w: [3.8083642640626554, 0.03486819339595951, 1.6400224976589866,
-4.463151671894514, 1.7883062251202617, 5.308526768308639,
-0.13398764643967714, -2.2539799445450406, 1.4840784189709668,
-1.890906591367886, 1.933249316738729, -1.2629454476069037,
1.7257519419059324, 2.967849703391228, 3.9061632698216244, -9.520241584621713,
-1.8736788731126397, -3.483844660866203, -5.637874599559359]
```

```
print('predict:', maxent.predict(x))
```

```
predict: {'yes': 0.9999971802186581, 'no': 2.819781341881656e-06}
```

第6章Logistic回归与最大熵模型-习题

习题6.1

确认Logistic分布属于指数分布族。

解答：
第1步：
首先给出指数分布族的定义：
对于随机变量x，在给定参数η下，其概率分别满足如下形式：$p(x|\eta)=h(x)g(\eta)\exp(\eta^T u(x))$我们称之为**指数分布族**。
其中：
x：可以是标量或者向量，可以是离散值也可以是连续值
η：自然参数
$g(\eta)$：归一化系数
$h(x),u(x)$：x的某个函数

第2步：证明伯努利分布属于指数分布族

伯努利分布：φ是$y=1$的概率，即$P(Y=1)=\varphi$

$$\begin{aligned}P(y|\varphi)&=\varphi^y(1-\varphi)^{(1-y)}\\&=(1-\varphi)\varphi^y(1-\varphi)^{(-y)}\\&=(1-\varphi)(\frac{\varphi}{1-\varphi})^y\\&=(1-\varphi)\exp\left(y\ln\frac{\varphi}{1-\varphi}\right)\\&=\frac{1}{1+e^{\eta}}\exp(\eta y)\end{aligned}$$

其中，$\eta=\ln\frac{\varphi}{1-\varphi}\Leftrightarrow\varphi=\frac{1}{1+e^{-\eta}}$

将y替换成x，可得$P(x|\eta)=\frac{1}{1+e^{\eta}}\exp(\eta x)$ 对比可知，伯努利分布属于指数分布族，其中

$h(x)=1,g(\eta)=\frac{1}{1+e^{\eta}},u(x)=x$

第3步：

广义线性模型（GLM）必须满足三个假设：

1. $y|x;\theta\sim ExponentialFamily(\eta)$，即假设预测变量$y$在给定$x$，以$\theta$为参数的条件概率下，属于以$\eta$作为自然参数的指数分布族；
2. 给定x，求解出以x为条件的$T(y)$的期望$E[T(y)|x]$，即算法输出为$h(x)=E[T(y)|x]$
3. 满足$\eta=\theta^Tx$，即自然参数和输入特征向量x之间线性相关，关系由θ决定，仅当η是实数时才有意义，若η是一个向量，则$\eta_i=\theta_i^Tx$

第4步：推导伯努利分布的GLM

已知伯努利分布属于指数分布族，对给定的x,η，求解期望：

$$\begin{aligned}h_\theta(x)&=E[y|x;\theta]\\&=1\cdot p(y=1)+0\cdot p(y=0)\\&=\varphi\\&=\frac{1}{1+e^{-\eta}}\\&=\frac{1}{1+e^{-\theta^Tx}}\end{aligned}$$

可得到Logistic回归算法，故Logistic分布属于指数分布族，得证。

习题6.2

写出Logistic回归模型学习的梯度下降算法。

解答：

对于Logistic模型：$P(Y=1|x)=\frac{\exp(w\cdot x+b)}{1+\exp(w\cdot x+b)}$对数似然函数为：

$$P(Y=0|x)=\frac{1}{1+\exp(w\cdot x+b)}$$

$$L(w)=\sum_{i=1}^{N}\left[y_i(w\cdot x_i)-\log(1+\exp(w\cdot x_i))\right]$$

似然函数求偏导，可得$\frac{\partial L(w)}{\partial w^{(j)}}=\sum_{i=1}^{N}\left[x_i^{(j)}\cdot y_i-\frac{\exp(w\cdot x_i)\cdot x_i^{(j)}}{1+\exp(w\cdot x_i)}\right]$

梯度函数为：$\nabla L(w)=\left[\frac{\partial L(w)}{\partial w^{(0)}},\cdots,\frac{\partial L(w)}{\partial w^{(m)}}\right]$

Logistic回归模型学习的梯度下降算法：

(1) 取初始值$x^{(0)}\in R$，置$k=0$

(2) 计算$f(x^{(k)})$

(3) 计算梯度$g_k=g(x^{(k)})$，当$\|g_k\|<\varepsilon$时，停止迭代，令$x^*=x^{(k)}$；否则，求λ_k，使得$f(x^{(k)}+\lambda_k g_k)=\max\limits_{\lambda\geqslant 0} f(x^{(k)}+\lambda g_k)$

(4) 置$x^{(k+1)}=x^{(k)}+\lambda_k g_k$，计算$f(x^{(k+1)})$，当$\|f(x^{(k+1)})-f(x^{(k)})\|<\varepsilon$或$\|x^{(k+1)}-x^{(k)}\|<\varepsilon$时，停止迭代，令$x^*=x^{(k+1)}$

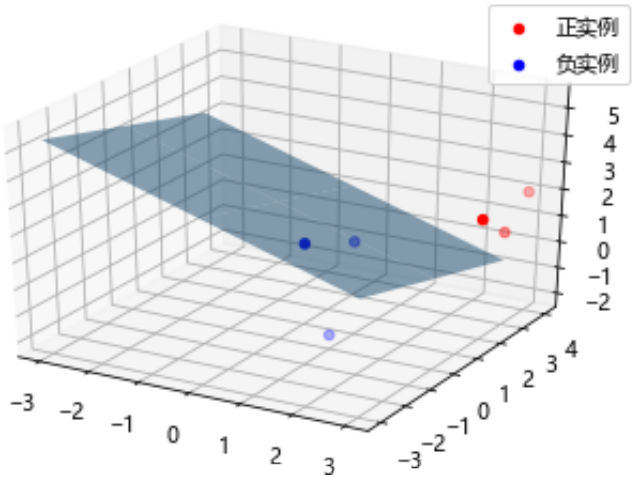
(5) 否则，置$k=k+1$，转(3)

```
%matplotlib inline
import numpy as np
import time
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from pylab import mpl

# 图像显示中文
mpl.rcParams['font.sans-serif'] = ['Microsoft YaHei']


class LogisticRegression:
    def __init__(self, learn_rate=0.1, max_iter=10000, tol=1e-2):
        self.learn_rate = learn_rate  # 学习率
        self.max_iter = max_iter  # 迭代次数
        self.tol = tol  # 迭代停止阈值
        self.w = None  # 权重

    def preprocessing(self, X):
        """将原始x末尾加上一列，该列数值全部为1"""
        row = X.shape[0]
        y = np.ones(row).reshape(row, 1)
        X_prepro = np.hstack((X, y))
        return X_prepro

    def sigmod(self, x):
        return 1 / (1 + np.exp(-x))

    def fit(self, X_train, y_train):
        X = self.preprocessing(X_train)
        y = y_train.T
        # 初始化权重w
        self.w = np.array([[0] * X.shape[1]], dtype=np.float)
        k = 0
        for loop in range(self.max_iter):
            # 计算梯度
            z = np.dot(X, self.w.T)
```

```
            grad = X * (y - self.sigmod(z))
            grad = grad.sum(axis=0)
            # 利用梯度的绝对值作为迭代中止的条件
            if (np.abs(grad) <= self.tol).all():
                break
            else:
                # 更新权重w 梯度上升——求极大值
                self.w += self.learn_rate * grad
                k += 1
        print("迭代次数：{}次".format(k))
        print("最终梯度：{}".format(grad))
        print("最终权重：{}".format(self.w[0]))

    def predict(self, x):
        p = self.sigmod(np.dot(self.preprocessing(x), self.w.T))
        print("Y=1的概率被估计为：{:.2%}".format(p[0][0]))  # 调用score时，注释掉
        p[np.where(p > 0.5)] = 1
        p[np.where(p < 0.5)] = 0
        return p

    def score(self, X, y):
        y_c = self.predict(X)
        error_rate = np.sum(np.abs(y_c - y.T)) / y_c.shape[0]
        return 1 - error_rate

    def draw(self, X, y):
        # 分离正负实例点
        y = y[0]
        X_po = X[np.where(y == 1)]
        X_ne = X[np.where(y == 0)]
        # 绘制数据集散点图
        ax = plt.axes(projection='3d')
        x_1 = X_po[0, :]
        y_1 = X_po[1, :]
        z_1 = X_po[2, :]
        x_2 = X_ne[0, :]
        y_2 = X_ne[1, :]
        z_2 = X_ne[2, :]
        ax.scatter(x_1, y_1, z_1, c="r", label="正实例")
        ax.scatter(x_2, y_2, z_2, c="b", label="负实例")
        ax.legend(loc='best')
        # 绘制p=0.5的区分平面
        x = np.linspace(-3, 3, 3)
        y = np.linspace(-3, 3, 3)
        x_3, y_3 = np.meshgrid(x, y)
        a, b, c, d = self.w[0]
        z_3 = -(a * x_3 + b * y_3 + d) / c
        ax.plot_surface(x_3, y_3, z_3, alpha=0.5)  # 调节透明度
        plt.show()
```

```
# 训练数据集
X_train = np.array([[3, 3, 3], [4, 3, 2], [2, 1, 2], [1, 1, 1], [-1, 0, 1],
                    [2, -2, 1]])
y_train = np.array([[1, 1, 1, 0, 0, 0]])
# 构建实例，进行训练
clf = LogisticRegression()
clf.fit(X_train, y_train)
clf.draw(X_train, y_train)
```

```
迭代次数：3232次
最终梯度：[ 0.00144779  0.00046133  0.00490279 -0.00999848]
最终权重：[  2.96908597   1.60115396   5.04477438 -13.43744079]
```



习题6.3

写出最大熵模型学习的DFP算法。（关于一般的DFP算法参见附录B）

解答：

第1步：

最大熵模型为：$\max \quad H(p)=-\sum_{x,y} P(x)P(y|x)\log P(y|x)$

$\text{st.} \quad E_p(f_i)-E_{\hat{p}}(f_i)=0, \quad i=1,2,\cdots,n$ 引入拉格朗日乘子，定义拉格朗日函数：$\sum_y P(y|x)=1$

$L(P,w)=\sum_{xy} P(x)P(y|x)\log P(y|x)+w_0\left(1-\sum_y P(y|x)\right)$ 最优化原始问题为：

$+\sum_{i=1} w_i\left(\sum_{xy} P(x,y)f_i(x,y)-\sum_{xy} P(x,y)P(y|x)f_i(x,y)\right)$

$\min_{P\in C}\max_w L(P,w)$对偶问题为：$\max_w \min_{P\in C} L(P,w)$令

$\Psi(w)=\min_{P\in C} L(P,w)=L(P_w,w)\Psi(w)$称为对偶函数，同时，其解记作

$P_w=\arg\min_{P\in C} L(P,w)=P_w(y|x)$求$L(P,w)$对$P(y|x)$的偏导数，并令偏导数等于0，解得：

$P_w(y|x)=\frac{1}{Z_w(x)}\exp(\sum_{i=1}^n w_i f_i(x,y))$其中：$Z_w(x)=\sum_y \exp(\sum_{i=1}^n w_i f_i(x,y))$则最大熵模型目标函数表示为

$\varphi(w)=\min_{w\in R_n}\Psi(w)=\sum_x P(x)\log\sum_y\exp(\sum_{i=1}^n w_i f_i(x,y))-\sum_{x,y}P(x,y)\sum_{i=1}^n w_i f_i(x,y)$

第2步：

DFP的G_{k+1}的迭代公式为：$G_{k+1}=G_k+\frac{\delta_k\delta_k^T}{\delta_k^T y_k}-\frac{G_k y_k y_k^T G_k}{y_k^T G_k y_k}$

最大熵模型的DFP算法：

输入：目标函数$\varphi(w)$，梯度$g(w)=\nabla g(w)$，精度要求ε；

输出：$\varphi(w)$的极小值点w^*

(1)选定初始点$w^{(0)}$，取G_0为正定对称矩阵，置$k=0$

(2)计算$g_k=g(w^{(k)})$，若$\|g_k\|<\varepsilon$，则停止计算，得近似解$w^*=w^{(k)}$，否则转(3)

(3)置$p_k=-G_k g_k$

(4)一维搜索：求λ_k使得$\varphi\left(w^{(k)}+\lambda_k P_k\right)=\min_{\lambda\geqslant 0}\varphi\left(w^{(k)}+\lambda P_k\right)$(5)置$w^{(k+1)}=w^{(k)}+\lambda_k p_k$

(6)计算$g_{k+1}=g(w^{(k+1)})$，若$\|g_{k+1}\|<\varepsilon$，则停止计算，得近似解$w^*=w^{(k+1)}$；否则，按照迭代式算出G_{k+1}

(7)置$k=k+1$，转(3)

参考代码：https://github.com/wzyonggege/statistical-learning-method

本文代码更新地址：https://github.com/fengdu78/lihang-code

习题解答：https://github.com/datawhalechina/statistical-learning-method-solutions-manual

中文注释制作：机器学习初学者公众号：ID:ai-start-com

配置环境：python 3.5+

代码全部测试通过。