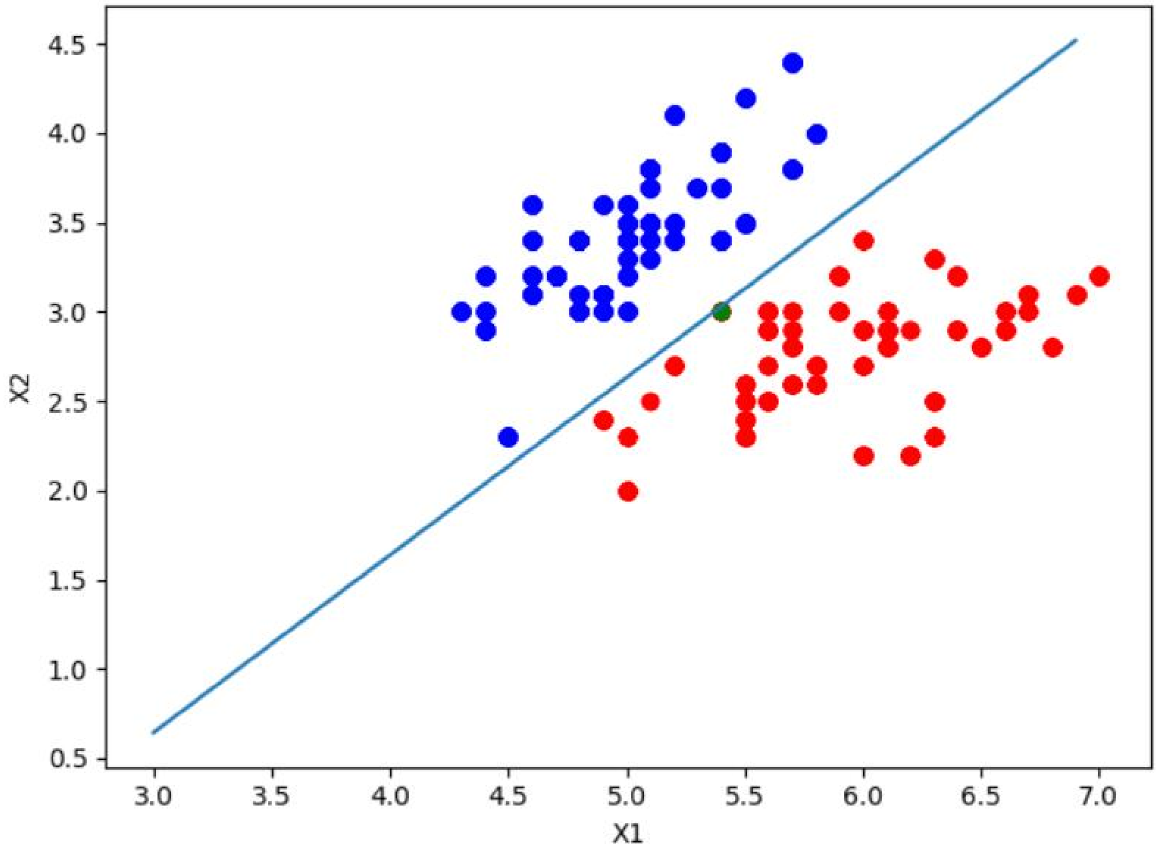
# SVM鸢尾花分类问题

能达到较好的分类效果，图片如下所示




```
bias:       -2.481
w:      [ 1.05517241 -1.06206897]
accuracy:    1.000
```

最终可以达到100%的识别率，weight与bias如上图所示

图中红蓝两色表示两种不同的鸢尾花种类，**绿色点为支持向量**

代码中注释确为**本人手写**

`main`函数如下

```python
if __name__ == "__main__":
    trainset = create_data()

    # # save data as csv when first run
    # with open("iris_data.csv", "w") as csvdata:
    #     writer = csv.writer(csvdata, delimiter="\n")
    #     writer.writerows(trainset)

    features, labels = trainset[:, :2], trainset[:, -1]


    model = SVM()

```

```
    # Train model
    model.train(features, labels)

    # calculate accuracy
    y_hat = model.predict(features)
    acc = calc_acc(labels, y_hat)

    print("bias:\t\t%.3f" % (model.b))
    print("w:\t\t" + str(model.w))
    print("accuracy:\t%.3f" % (acc))

    showpoints(trainset, model.w, model.b)
```

# 1.数据获取

使用以下python库，其中svm为自定义SVM算法文件

```
"""
* @author   孟子喻
* @time     2021.4.30
* @file     classify_iris.py
*           svm.py
"""
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from svm import SVM
import matplotlib.pyplot as plt
```

```
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]])
    for i in range(len(data)):
        if data[i, -1] == 0:
            data[i, -1] = -1
    # print(data)
    return data

# 初次使用时将数据以csv格式存储
# with open("iris_data.csv", "w") as csvdata:
#     writer = csv.writer(csvdata, delimiter="\n")
#     writer.writerows(trainset)
```

`create_data` 即为老师提供的数据生成函数，将其存入csv文件中供后续调用

# 2.使用SVM进行训练

我使用的svm代码结构参照了MIT的Lasse Regin Nielsen2015年所写的代码，其中有很多巧妙的设计，每一段代码都会有具体到课本公式编号的单独注释和解析

**__init__**

```python
class SVM():
    def __init__(self, max_iter=10000, kernel_type='linear', C=1.0, epsilon=0.001):
        self.kernels = {
            'linear': self.kernel_linear,
            'quadratic': self.kernel_quadratic
        }                                   # 根据不同的核函数实现线性与非线性支持向量机
        self.max_iter = max_iter            # 最大迭代次数，超过将自动退出
        self.kernel_type = kernel_type      # 选择核函数
        self.C = C                          # C为惩罚参数，C越大对误分类的惩罚越大
        self.epsilon = epsilon              # 设置允许差错的范围
```

把核函数`kernel`分为`linear`和`quadratic`两种

1. `linear`：线性核函数，对输入直接求点积
2. `quadratic`：平方核函数，对输入求点积的平方

其余参数意义见上述注释

**kernel_linear&kernel_quadratic**

```python
# 定义核函数
def kernel_linear(self, x1, x2):
    # 线性核函数
    return np.dot(x1, x2.T)
def kernel_quadratic(self, x1, x2):
    # 二次核函数
    return (np.dot(x1, x2.T) ** 2)
```

**train解读**

```python
# 初始化
n, d  = X.shape[0], X.shape[1]
alpha = np.zeros((n))                      # 取拉格朗日乘子初值alpha全为0
kernel= self.kernels[self.kernel_type]     # 设置核函数
count = 0                                  # 计算迭代次数
```

参考课本P149，7.4.3 SMO算法，第一步取拉格朗日乘子初值alpha全为0，并设置核函数为线性（由数据点分布得）

```
while True:
    count += 1
    alpha_prev = np.copy(alpha)         # 将alpha深拷贝
    # print(alpha.shape)
    # print(alpha)

    for j in range(0, n):
        i = self.get_rnd_int(0, n-1, j)                   # 随机获取不同的i与j，得到两个优化变量
        x_1, x_2, y_1, y_2 = X[i, :], X[j, :], y[i], y[j]   # 储存实例的特征和标签
        k_ij = kernel(x_1, x_1) + kernel(x_2, x_2) - 2 * kernel(x_1, x_2)
        if k_ij == 0:                                     # k_ij
                    continue                                      # 保证两个实例不同
                alpha_prime_2, alpha_prime_1 = alpha[j], alpha[i]
```

随机获取两个变量的特征和标签并保证他们不同，选取alpha1和alpha2来做优化变量

```
                (L, H) = self.compute_L_H(self.C, alpha_prime_2, alpha_prime_1, y_2, y_1)
                                # print(L, H)
                                # 计算weight和bias
                                self.w = self.calc_w(alpha, y, X)
                                self.b = self.calc_b(X, y, self.w)
```

这里`compute_L_H`是用来计算alpha2的上下边界，这个边界对应两个拉格朗日乘子alpha1和alpha2的取值范围，即[0, C] x [0, C]，其中C为惩罚系数，alpha1和alpha2即被限制在这样的一个正方形内，我们在一开始先忽略这个取值范围求最优解，求得最优情况下的取值后再与边界比较

其中求新的alpha2的上下边界的函数`compute_L_H`实现如下：

```
def compute_L_H(self, C, alpha_prime_j, alpha_prime_i, y_j, y_i):
        # 求alpha2所在对角线端点的边界，即alpha2的取值范围
        # print(C, alpha_prime_j, alpha_prime_i, y_j, y_i)
        if(y_i != y_j):         # 若非同类
            return (max(0, alpha_prime_j - alpha_prime_i), min(C, C - alpha_prime_i + alpha_prime_j))
        else:                   # 若为同类
            return (max(0, alpha_prime_i + alpha_prime_j - C), min(C, alpha_prime_i + alpha_prime_j))
```

由最优化问题的对$\alpha_1$和$\alpha_2$的约束推得，即

$\alpha_1 y_1 + \alpha_2 y_2 = \text{常数}$

$y_1$和$y_2$是否相同，便对应P144 图7.8的两种情况，$\alpha_1$与$\alpha_2$的等式约束在平行于正方形的直线上，原因如以下3公式所示，其中k也是常数，通过约束把双变量下的最优化转换为单变量下的最优化

$$y_1 \neq y_2 \text{时}，\alpha_1 - \alpha_2 = k$$

$$y_1 = y_2 \text{时}，\alpha_1 + \alpha_2 = k$$

$$L \leq \alpha_2^{new} \leq H$$

如果$y_1 \neq y_2$，则

$L = max(0, \alpha_2^{old} - \alpha_1^{old})$

$H = min(C, C + \alpha_2^{old} - \alpha_1^{old})$

如果$y_1 = y_2$，则

$L = max(0, \alpha_2^{old} + \alpha_1^{old} - C)$

$H = min(C, \alpha_2^{old} + \alpha_1^{old})$

这两个$L$与$H$的取值由以上4式决定，这里与0和C取max和min是为了裁剪保证不超出$\alpha$的取值范围

接`train`函数，然后是计算weight和bias

```
                        self.w = self.calc_w(alpha, y, X)
                        self.b = self.calc_b(X, y, self.w)

                        # 计算x_i，x_j的预测值与真实值的误差
                        E_i = self.calc_E(x_1, y_1, self.w, self.b)
                        E_j = self.calc_E(x_2, y_2, self.w, self.b)
```

`calc_b`和`calc_w`实现如下

```
def calc_b(self, X, y, w):
    b_tmp = y - np.dot(w.T, X.T)
    return np.mean(b_tmp)

def calc_w(self, alpha, y, X):
    return np.dot(X.T, np.multiply(alpha,y))
```

然后是计算预测值与真实值之间的误差

```
                E_i = self.calc_E(x_1, y_1, self.w, self.b)
                E_j = self.calc_E(x_2, y_2, self.w, self.b)
```

`calc_E`实现如下

```
def calc_E(self, x_k, y_k, w, b):
    # 求E，即g(x)对输入x_k的预测值y_k与真实值之差
    return self.decision_f(x_k, w, b) - y_k
def decision_f(self, X, w, b):
    # 决策函数，即对输入进行预测
    return np.sign(np.dot(w.T, X.T) + b).astype(int)
```

对应课本P145公式7.105，这里课本上写的复杂一点，但是里边一些部分已经存为weight和bias了，课本这里直接带换一下较好，`decision_f`决策函数其实就是$y = w \cdot x + b$

```
                # 求出alpha2未经剪辑的解
                alpha[j] = alpha_prime_2 + float(y_2 * (E_i - E_j))/k_ij
                # 利用求出的L,H对alpha2进行剪辑
                alpha[j] = max(alpha[j], L)
                alpha[j] = min(alpha[j], H)

                # 用alpha2反推alpha1
                alpha[i] = alpha_prime_1 + y_1*y_2 * (alpha_prime_2 - alpha[j])
```

这一步是正式计算$\alpha_2^{new}$，求出取值后与前面求出的L和H边界点比较，取极大或极小值，**这里与L比较求大，与H比较求小的思路比较巧妙，能省去判断**（正常思路是如果大于H则取H，如果小于L则取L）

最后利用$\alpha 1$和$\alpha 2$的等式关系求出$\alpha_1$，这样优化变量的整个迭代思路就很明确了，主体部分完成

完整代码

```
from __future__ import division, print_function
import os
import numpy as np
import random as rnd
filepath = os.path.dirname(os.path.abspath(__file__))

class SVM():
    def __init__(self, max_iter=10000, kernel_type='linear', C=1.0, epsilon=0.001):
        self.kernels = {
            'linear': self.kernel_linear,
            'quadratic': self.kernel_quadratic
        }                                        # 根据不同的核函数实现线性与非线性支持向量机
        self.max_iter = max_iter                 # 最大迭代次数，超过将自动退出
        self.kernel_type = kernel_type           # 选择核函数
        self.C = C                               # C为惩罚参数，C越大对误分类的惩罚越大
        self.epsilon = epsilon                   # 设置允许差错的范围
    def train(self, X, y):
        # 初始化
        n, d = X.shape[0], X.shape[1]
        alpha = np.zeros((n))                    # 取初值拉格朗日乘子alpha全为0
        kernel = self.kernels[self.kernel_type] # 设置核函数
        count = 0                                # 计算迭代次数
        while True:
            count += 1
            alpha_prev = np.copy(alpha)          # 将alpha深拷贝
            # print(alpha.shape)
            # print(alpha)

            for j in range(0, n):
                i = self.get_rnd_int(0, n-1, j)                    # 随机获取不同的i与j，得到两个优化变量
                x_1, x_2, y_1, y_2 = X[i, :], X[j, :], y[i], y[j]    # 储存实例的特征和标签
                k_ij = kernel(x_1, x_1) + kernel(x_2, x_2) - 2 * kernel(x_1, x_2)
```

```
                if k_ij == 0:                                       # k_ij
                    continue                                        # 保证两个实例不同
                alpha_prime_2, alpha_prime_1 = alpha[j], alpha[i]
                # 求alpha2所在对角线端点的边界，即alpha2的取值范围
                (L, H) = self.compute_L_H(self.C, alpha_prime_2, alpha_prime_1, y_2, y_1)
                # print(L, H)
                # 计算weight和bias
                self.w = self.calc_w(alpha, y, X)
                self.b = self.calc_b(X, y, self.w)

                # 计算x_i，x_j的预测值与真实值的误差
                E_i = self.calc_E(x_1, y_1, self.w, self.b)
                E_j = self.calc_E(x_2, y_2, self.w, self.b)

                # 求出alpha2未经剪辑的解
                alpha[j] = alpha_prime_2 + float(y_2 * (E_i - E_j))/k_ij
                # 利用求出的L,H对alpha2进行剪辑
                alpha[j] = max(alpha[j], L)
                alpha[j] = min(alpha[j], H)

                # 用alpha2反推alpha1
                alpha[i] = alpha_prime_1 + y_1*y_2 * (alpha_prime_2 - alpha[j])

            # 检查是否超出误差允许范围
            diff = np.linalg.norm(alpha - alpha_prev)
            if diff < self.epsilon:
                break

            # 如果超出设定的最大迭代次数仍未求出最优解，则返回
            if count >= self.max_iter:
                print("Iteration number exceeded the max of %d iterations" % (self.max_iter))
                return
        # 临输出前计算最终的weight和bias
        self.b = self.calc_b(X, y, self.w)
        if self.kernel_type == 'linear':
            self.w = self.calc_w(alpha, y, X)
        return count

    def predict(self, X):
        return self.decision_f(X, self.w, self.b)
    def calc_b(self, X, y, w):
        b_tmp = y - np.dot(w.T, X.T)
        return np.mean(b_tmp)

    def calc_w(self, alpha, y, X):
        return np.dot(X.T, np.multiply(alpha,y))

    def decision_f(self, X, w, b):
        # 决策函数，即对输入进行预测
        return np.sign(np.dot(w.T, X.T) + b).astype(int)

    def calc_E(self, x_k, y_k, w, b):
        # 求E，即g(x)对输入x_k的预测值y_k与真实值之差
        return self.decision_f(x_k, w, b) - y_k
```

```python

    def compute_L_H(self, C, alpha_prime_j, alpha_prime_i, y_j, y_i):
        # 求alpha2所在对角线端点的边界，即alpha2的取值范围
        # print(C, alpha_prime_j, alpha_prime_i, y_j, y_i)
        if(y_i != y_j):         # 若非同类
            return (max(0, alpha_prime_j - alpha_prime_i), min(C, C - alpha_prime_i + alpha_prime_j))
        else:                   # 若为同类
            return (max(0, alpha_prime_i + alpha_prime_j - C), min(C, alpha_prime_i + alpha_prime_j))
    def get_rnd_int(self, a,b,z):
        i = z
        cnt=0
        while i == z and cnt<1000:
            i = rnd.randint(a,b)
            cnt=cnt+1
        return i
    # 定义核函数
    def kernel_linear(self, x1, x2):
        # 线性核函数
        return np.dot(x1, x2.T)
    def kernel_quadratic(self, x1, x2):
        # 二次核函数
        return (np.dot(x1, x2.T) ** 2)
```

## 3.可视化显示

最后的绘图函数也很简单了

```python
def showpoints(data, w, b):
    positive_x = []
    positive_y = []
    negative_x = []
    negative_y = []
    fig = plt.figure()
    ax = fig.add_subplot(111)
    for item in data:
        if item[-1] == 1:
            positive_x.append(item[0])
            positive_y.append(item[1])
        if item[-1] == -1:
            negative_x.append(item[0])
            negative_y.append(item[1])
        ax.scatter(positive_x, positive_y, color="r")
        ax.scatter(negative_x, negative_y, color="b")
    x = np.arange(3, 7, 0.1)
    y = []
    for item in x:
        y.append(-w[0]/w[1]*item - b/w[1])
    ax.plot(x, y)
    sup_vc = get_sup_vc(data[:, 0:2], w, b)

    ax.scatter(sup_vc[0], sup_vc[1], color="g")
    plt.xlabel('X1')
    plt.ylabel('X2')

    plt.show()
```