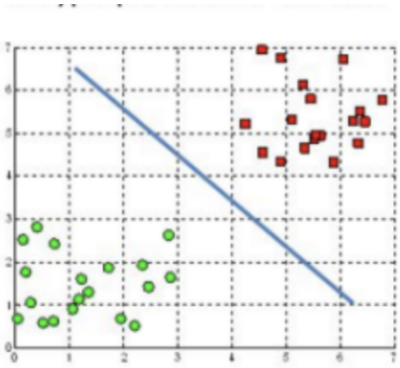
LIBSVM报告

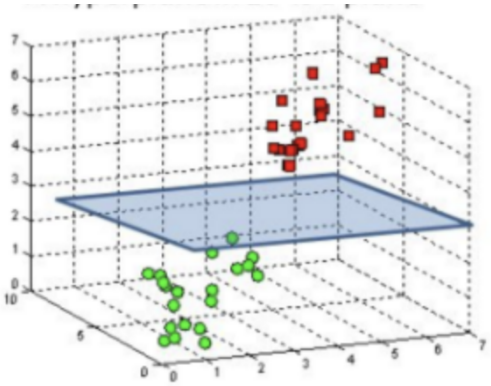
代码结构与思想

简介

支持向量机，英文全称“Support Vector Machines”（简称 SVM），是一种用于分类和回归的监督学习算法。这种线性分类器，建立在统计学习理论和结构风险最小原理的基础上，根据有限的训练集，在模型的复杂性和学习性之间寻求最佳的折中，以获得最好的泛化能力的经典分类方法。

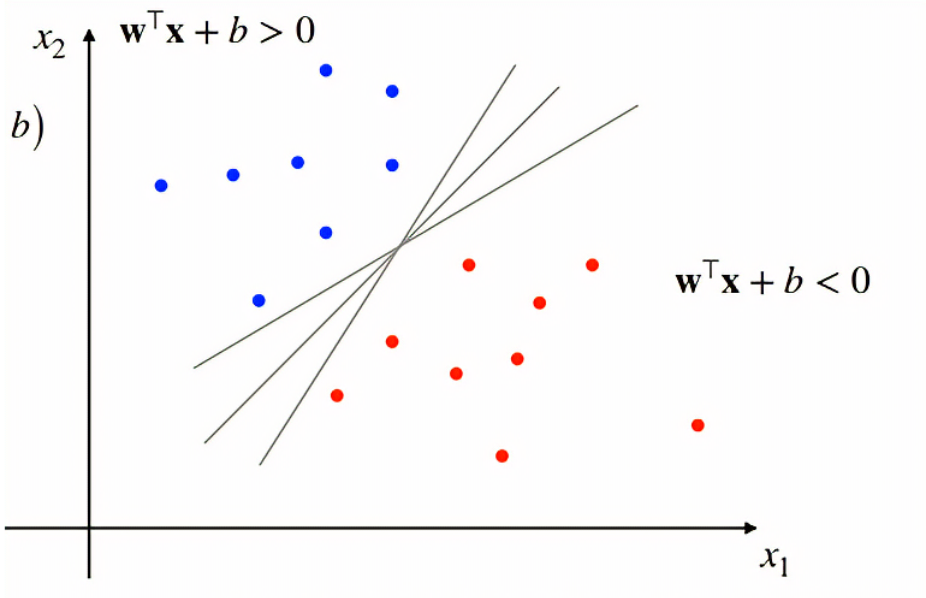
具体而言，SVM算法的目的是在**特征空间**中找到**间隔最大**的超平面。超平面指代的是将所有类别划分开的那条线，当数据特征是二维时，此超平面是一条线；当数据特征是多维时，它就变成了面，如下图所示。超平面与最近的样本点保持一定的距离。



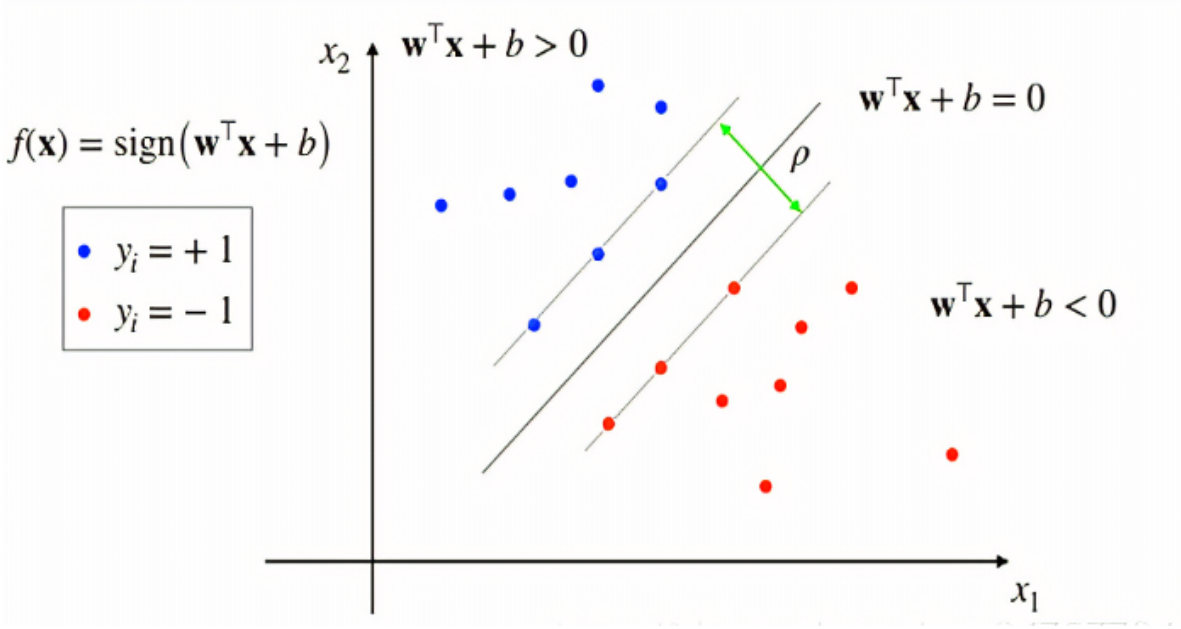


问题转换

一个数据集，可能存在多个不同的分类平面，SVM需要确定出选择哪个平面作为结果答案，从下图来讲，就是选择哪一条直线。



则定义，与分类平面距离最近的样本点称为支持向量，进而构成支持平面。而两个支持平面之间的距离称作分类器的分类间距ρ。而SVM的核心思想就是最大化分类间距ρ。



用数学语言概括即为：

$$\text{判别函数：}f(x)=sign(w^Tx+b)$$
$$\text{最大化分类间隔目标：}MaxMargin(w,b)$$
$$\text{约束条件：}s.t.\,y_i(w^Tx_i+b)>0$$

解法

如果将w作为超平面法向量，b 为位移项。由简单的解析几何得到空间中点x 到平面的距离：

$$r=\frac{|w^Tx+b|}{||w||}$$

将二分类问题中的数据集标签yi 映射到{−1, +1}，也就是正类yi = 1，负类yi = −1。而给定一个分离超平面，距离该超平面最近的正类样本和负类样本（也就是上述约束中的不等号为等号时的样本点，称作支持向量）到超平面的距离之和则有：

$$\gamma=\frac{2}{||w||}$$
$$1-y_i*(w^Tx_i+b)\le 0(i=0,\ 1,\ 2,\ 3.....)$$
$$\gamma\text{极大化，则为求}min\frac{1}{2}||w||^2$$

使用拉格朗日对偶求解：

$$\mathcal{L}(\boldsymbol{w},b,\boldsymbol{\lambda})=\frac{1}{2}\|\boldsymbol{w}\|^2+\sum_{i=1}^{m}\lambda_i(1-y_i(\boldsymbol{w}^\top\boldsymbol{x}_i+b))$$

其中 $\boldsymbol{\lambda}=(\lambda_1,\lambda_2,...,\lambda_m),\lambda_i\ge 0$. 我们先求 $\min_{\boldsymbol{w},b}\mathcal{L}(\boldsymbol{w},b,\boldsymbol{\lambda})$，对变量求偏导：

$$\begin{cases}\dfrac{\partial}{\partial\boldsymbol{w}}\mathcal{L}=\boldsymbol{w}-\sum_{i=1}^{m}\lambda_i y_i\boldsymbol{x}_i\\ \dfrac{\partial}{\partial b}\mathcal{L}=\sum_{i=1}^{m}\lambda_i y_i\end{cases}$$

将下式代入0归到上式，便可得到优化问题的对偶问题：

$$\max_{\boldsymbol{\lambda}} \quad \sum_{i=1}^{m} \lambda_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i^\top \boldsymbol{x}_j$$
$$\text{s.t.} \quad \sum_{i=1}^{m} \lambda_i y_i = 0$$
$$\lambda_i \geq 0, i = 1, 2, ..., m.$$

只要解出λ，就得到模型

$$\begin{aligned} f(\boldsymbol{x}) &= \boldsymbol{w}^\top \boldsymbol{x} + b \\ &= \sum_{i=1}^{m} \lambda_i y_i \boldsymbol{x}_i^\top \boldsymbol{x} + b \end{aligned}$$

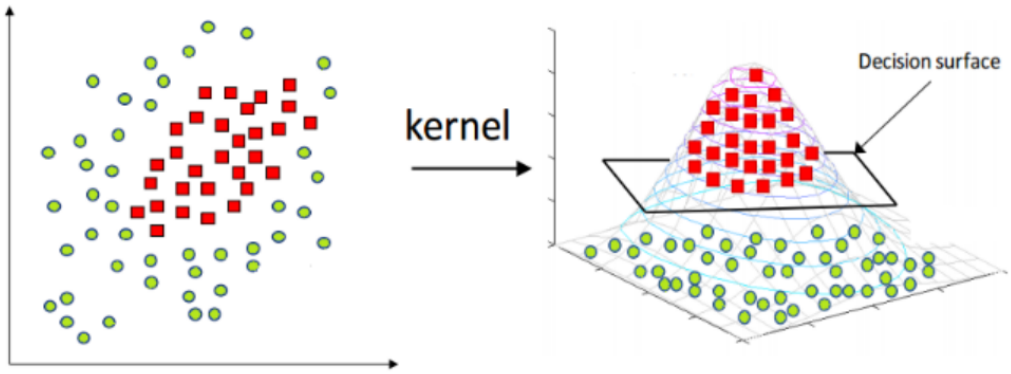
且满足条件：

$$\begin{cases} \lambda_i \geq 0; \\ y_i f(\boldsymbol{x}_i) - 1 \geq 0 \\ \lambda_i (y_i f(\boldsymbol{x}_i) - 1) = 0 \end{cases}$$

核函数

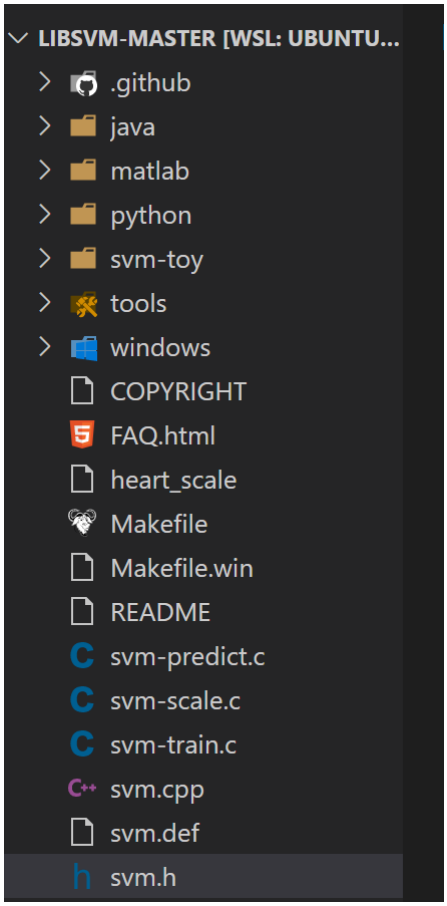
支持向量机通过某非线性变换φ(x)，将输入空间映射到高维特征空间。特征空间的维数可能非常高。如果支持向量机的求解只用到内积运算，而在低维输入空间又存在某个函数K(x,x')，它恰好等于在高维空间中这个内积，即K(x,x') = ⟨ϕ(x), ϕ(x')⟩。那么支持向量机就不用计算复杂的非线性变换，而由这个函数K(x,x')直接得到非线性变换的内积，使大大简化了计算。这样的函数K(x,x') 称为核函数。

简单而言，通过用K(x,x') 去替换简单的向量内积，使得决策边界不再是分离超平面，而是一个曲面。SVM核函数的引用主要将非线性分类问题转换为线性分类问题。因为存在一个假设，就是数据在低维空间不可分，但在高维空间可分



概览

Libsvm函数包的组织结构图如下图所示。主文件路径中包含了核心的C++程序，以及Java、Python版本的实现。Tool子文件路径中包含了一些检验数据格式以及选择SVM参数的tool，为辅助数据选择使用的作用。



数据结构表示

基本结构体

svm定义了4个结构体，分别是svm_node、svm_problem、svm_parameter、svm_model

而具体的解释如下：

- svm_node:

```
package libsvm;
public class svm_node implements java.io.Serializable
{
    public int index; //存储单一向量的索引
    public double value; //存储对应索引位置的值
}
```

这个结构来**存储数据节点**。index表示的是向量的第几维，value表示的是这一维度下的值。

index	1	2	3	4
value	0.1	0.2	0.3	0.4

- svm_problem

```
package libsvm;
public class svm_problem implements java.io.Serializable
{
    public int l; //记录样本总数
    public double[] y; //样本所属类别(标签，如+1，-1)的数组
    public svm_node[][] x; //存储样本内容出标签外的信息
}
```

这个结构用来存储**所有样本数据集**

例如l=2时，有两个样本，svm_node如下，存储两个样本的所有向量值：

v00	v01	v02	v03	.....
v10	v11	v12	v13	.....

- svm_parameter

```
package libsvm;
public class svm_parameter implements Cloneable,java.io.Serializable
{
    /* svm_type */
    public static final int C_SVC = 0;
    public static final int NU_SVC = 1;
    public static final int ONE_CLASS = 2;
    public static final int EPSILON_SVR = 3;
    public static final int NU_SVR = 4;

    /* kernel_type */
    //这里是各种可以使用的核
    public static final int LINEAR = 0;
    public static final int POLY = 1;
    public static final int RBF = 2;
    public static final int SIGMOID = 3;
    public static final int PRECOMPUTED = 4;

    public int svm_type;
    public int kernel_type;
    public int degree;  // for poly 即，多项式核
    public double gamma;    // for poly/rbf/sigmoid  （rbf为高斯核径向基）
    public double coef0;    // for poly/sigmoid

    // these are for training only
    //这里是有关于training的变量
    public double cache_size; // in MB
    public double eps;  // stopping criteria
    public double C;    // for C_SVC, EPSILON_SVR and NU_SVR
    public int nr_weight;       // for C_SVC
    public int[] weight_label;  // for C_SVC
    public double[] weight;     // for C_SVC
    public double nu;   // for NU_SVC, ONE_CLASS, and NU_SVR
    public double p;    // for EPSILON_SVR
    public int shrinking;   // use the shrinking heuristics
    public int probability; // do probability estimates
}
```

这里列举的是对svm**类型及参数的设置。**

- svm_model

```
package libsvm;
public class svm_model implements java.io.Serializable
{
    public svm_parameter param; // parameter
    public int nr_class;        // number of classes, = 2 in regression/one
class svm
    public int l;           // total #SV
    public svm_node[][] SV; // SVs (SV[l])
    public double[][] sv_coef;  // coefficients for SVs in decision functions
(sv_coef[k-1][l])
    public double[] rho;        // constants in decision functions (rho[k*(k-
1)/2])
    public double[] probA;         // pariwise probability information
    public double[] probB;
    public int[] sv_indices;       // sv_indices[0,...,nSV-1] are values in
[1,...,num_traning_data] to indicate SVs in the training set

    // for classification only

    public int[] label;     // label of each class (label[k])
    public int[] nSV;       // number of SVs for each class (nSV[k])
                // nSV[0] + nSV[1] + ... + nSV[k-1] = l
};
```

用于**保存训练后的SVM训练模型**，供predit使用

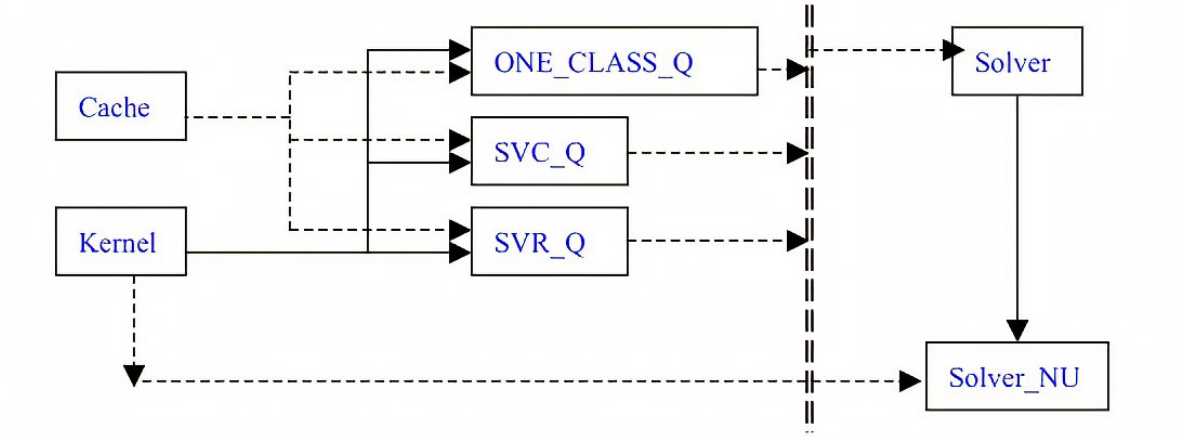
具体实现方式

核心代码

核心代码为svm.java。其总共有2000多行代码。其中，它实现了svm算法的核心功能。

核心代码里，总共有Cache、Kernel、ONE_CLASS_Q、QMatrix、Solver、Solver_NU、SVC_Q、SVR_Q 8个内部类。

类的关系如下图：



Cache

本类主要负责运算所涉及的**内存的管理**，包括申请、释放等。

实现过程中，首先通过Cache构造函数申请一块空间，这块空间的大小是：L个head_t大小的空间。然后get_data函数保证结构head_t中至少有len个float的内存，并且将可以使用的内存块的指针放在data指针中；而swap_index函数则是用于交换head[i]和head[j]。

```
class Cache
{
public:
    Cache(int l,long int size);
    ~Cache();
    // request data [0,len)
    // return some position p where [p,len) need to be filled
    // (p >= len if nothing needs to be filled)
    int get_data(const int index, Qfloat **data, int len);
    //该函数主要的就是每个head[i]具有len大小的内存空间。关于realloc函数是扩大内存用的
    void swap_index(int i, int j);
    //这个函数就是交换head_t[i]和head_t[j]的内容。首先将head_t[i]和head_t[j]从双向链表中脱离出去，然后交换它们的内容，最后重新把它们恢复到链表中。
private:
    int l;
    long int size;//所指定的全部内存，据说用Mb做单位
    //结构head_t用来记录所申请内存的指针，并记录长度，而且通过双向的指针，形成链表，增加寻址的速度
    struct head_t
    {
        head_t *prev, *next;    // a circular list是一个双向链表，非循环链表
        Qfloat *data;
        int len;        // data[0,len) is cached in this entry
    };

    head_t *head;//变量指针，该指针记录程序中所申请的内存。
    head_t lru_head;//双向链表的表头
    void lru_delete(head_t *h);//从双向链表中删去某个元素的链接，一般是删去当前所指向的元素
    void lru_insert(head_t *h);//在链表后面插入一个新的链接
};
```

这里再额外看一下构造函数，其目的是申请L个head_t的空间，初始化为0。而对于size的处理上，程序先将输入的size值（byte单位）转化为float的数目，然后再减去L个head_t所占的空间。

```
Cache::Cache(int l_,long int size_):l(l_),size(size_)
{
    //calloc函数的功能与malloc函数的功能相似，都是从堆分配内存该函数与malloc函数的一个显著不同时
    //是，calloc函数得到的内存空间是经过初始化的，其内容全为0。
    head = (head_t *)calloc(l,sizeof(head_t));  // initialized to 0
    size /= sizeof(Qfloat);//先将原来byte数目转化为float的数目。
    size -= l * sizeof(head_t) / sizeof(Qfloat);//扣除掉L个head_t的内存数目
    size = max(size, 2 * (long int) l); // cache must be large enough for two columns
    lru_head.next = lru_head.prev = &lru_head;
}
```

Kernel

$$1、K(x_i,x_j)=x_i^T x_j$$

$$2、K(x_i,x_j)=(\gamma x_i^T x_j+r)^d,\gamma>0$$

$$3、K(x_i,x_j)=\exp(-\gamma\left\|x_i-x_j\right\|^2),\gamma>0$$

$$4、K(x_i,x_j)=\tanh(\gamma x_i^T x_j+r)$$

本类主要包括核函数的定义、克隆，矩阵Q（ $Q=\sum y_i y_j K(x_i,x_j)$)的计算等。总结而言，Kernel是用来**存储和处理核函数矩阵**的类。

其中可选择的核分类为

线性核（linear）：主要用于线性可分的情况，其参数少速度快，对于线性可分数据，其分类效果很理想，因此我们通常首先尝试用线性核函数来做分类。不过一般效率不会特别高。

$$K(x,z)=<x,z>$$

多项式核（poly）：多项式核函数可以实现将低维的输入空间映射到高纬的特征空间，但是多项式核函数的参数多，当多项式的阶数比较高的时候，核矩阵的元素值将趋于无穷大或者无穷小，计算复杂度会大到无法计算

$$K(x,z)=(<x,z>+c)^d$$

高斯径向基核（RBF）：高斯径向基函数是一种局部性强的核函数，其可以将一个样本映射到一个更高维的空间内，该核函数是应用最广的一个，无论大样本还是小样本都有比较好的性能，而且其相对于多项式核函数参数要少，因此大多数情况下在不知道用什么核函数的时候，**优先使用高斯核函数**

多层感知器核（sigmoid）：采用sigmoid核函数，支持向量机实现的就是一种多层神经网络

在本kernel类中，复写了**快速幂powi，核运算结果求值kernel_function，构造器Kernel，索引相乘相加求值dot以及根据参数kernel_type，核运算结果k_function**

```
//
// Kernel evaluation
//
// the static method k_function is for doing single kernel evaluation
// the constructor of Kernel prepares to calculate the l*l kernel matrix
// the member function get_Q is for getting one column from the Q Matrix
//
abstract class QMatrix {
    abstract float[] get_Q(int column, int len);
    abstract double[] get_QD();
    abstract void swap_index(int i, int j);
};

abstract class Kernel extends QMatrix {
    private svm_node[][] x;
    private final double[] x_square;

    // svm_parameter
```

```
    private final int kernel_type;
    private final int degree;
    private final double gamma;
    private final double coef0;

    abstract float[] get_Q(int column, int len);
    abstract double[] get_QD();

    void swap_index(int i, int j)
    {
        do {svm_node[] tmp=x[i]; x[i]=x[j]; x[j]=tmp;} while(false);
        if(x_square != null) do {double tmp=x_square[i];
x_square[i]=x_square[j]; x_square[j]=tmp;} while(false);
    }

    //实现一个辅助运算的快速幂
    private static double powi(double base, int times)
    {
        double tmp = base, ret = 1.0;

        for(int t=times; t>0; t/=2)//对次数进行记录
        {
            if(t%2==1) ret*=tmp;
            tmp = tmp * tmp;
        }
        return ret;
    }

    /*
     * 根据不同的kernel_type分类。返回第i，j两行向量的核运算结果
     */
    double kernel_function(int i, int j) //i，j表示前向量第i行，后向量j列的输入数据，此
函数仅得到运算矩阵中的一个点
    {
        switch(kernel_type)   //kernel_type是int型的参数
        {
            case svm_parameter.LINEAR: //0 线性核
                // K(x,z)=<x,z>
                return dot(x[i],x[j]); //对应索引相乘再相加
            case svm_parameter.POLY:   //1 多项式核
                return powi(gamma*dot(x[i],x[j])+coef0,degree);
            case svm_parameter.RBF:    //2
                return Math.exp(-gamma*
(x_square[i]+x_square[j]-2*dot(x[i],x[j])));
            case svm_parameter.SIGMOID:
                return Math.tanh(gamma*dot(x[i],x[j])+coef0);
            case svm_parameter.PRECOMPUTED:
                return x[i][(int)(x[j][0].value)].value;
            default:
                return 0;   // java
        }
    }

     //构造函数
     //对成员变量中的SVM 参数赋值；
     //确定核函数；
     //复制训练集
    Kernel(int l, svm_node[][] x_, svm_parameter param)
```

```
    {
        this.kernel_type = param.kernel_type;
        this.degree = param.degree;
        this.gamma = param.gamma;
        this.coef0 = param.coef0;

        x = (svm_node[][])x_.clone();//将数据备份一次

        if(kernel_type == svm_parameter.RBF) //当高斯径向基核时
        {
            x_square = new double[l];
            for(int i=0;i<l;i++)
                x_square[i] = dot(x[i],x[i]);
        }
        else x_square = null;
    }

    /*
     * 将x，y对应索引相乘相加。实现向量乘法
     *  11 22 33
     *  1  3  5
     *  则结果11*1+22*3+33*5
     */
    static double dot(svm_node[] x, svm_node[] y) //参数为两行向量（单行单列向量）
    {
        double sum = 0;
        int xlen = x.length;
        int ylen = y.length;
        int i = 0;
        int j = 0;
        while(i < xlen && j < ylen)
        {
            if(x[i].index == y[j].index) //如果位置对应上
                sum += x[i++].value * y[j++].value;//进行乘后加
            else
            {
                if(x[i].index > y[j].index)
                    ++j;
                else
                    ++i;
            }
        }
        return sum;
    }
```

SVC_Q, ONE_CLASS_Q、SVR_Q

Kernel 类是抽象的核函数存储类，实际上不同的问题有不同的核函数矩阵，比如分类问题下

$$Qij \equiv yiyjK(xi, xj)$$

而在One-class SVM 中则是

$$Qij \equiv K(xi, xj)$$

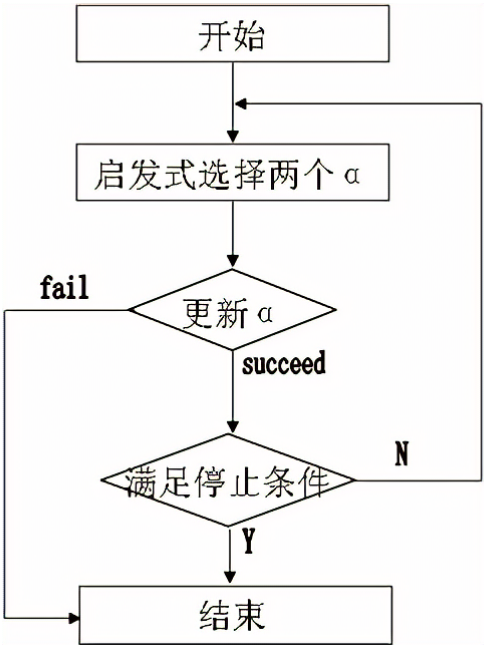
因此，根据不同问题，libSVM 以Kernel 为基类又派生出三个类：SVC_Q, ONE_CLASS_Q和SVR_Q，分别对应分类、区间估计和回归问题。这里以SVC_Q的代码为例，其余同理。

```
class SVC_Q extends Kernel
{
    private final byte[] y;
    private final Cache cache;//缓存机制的实现
    private final double[] QD;//Q的矩阵的对角线元素

    SVC_Q(svm_problem prob, svm_parameter param, byte[] y_)//构造函数
    {
        super(prob.l, prob.x, param);//参数赋值；
        y = (byte[])y_.clone();
        cache = new Cache(prob.l,(long)(param.cache_size*(1<<20)));
        //调用cache，数据集（包括特征向量和标签）的复制
        QD = new double[prob.l];
        for(int i=0;i<prob.l;i++)
            QD[i] = kernel_function(i,i);
        //初始化缓存和QD 数组；
    }

    float[] get_Q(int i, int len)//获取Q 矩阵指定行（考虑到对称性，列也可以）的多个元素
    {
        float[][] data = new float[1][];
        int start, j;
        if((start = cache.get_data(i,data,len)) < len)//对len以内的开始找
        {
            for(j=start;j<len;j++)
                data[0][j] = (float)(y[i]*y[j]*kernel_function(i,j));//调用
kernel_function开始计算
        }
        return data[0];
    }

    double[] get_QD()//获取对角线元素数组
    {
        return QD;
    }

    void swap_index(int i, int j)//将第i 个数据和第j个数据进行调换
    {
        cache.swap_index(i,j);
        super.swap_index(i,j);
        do {byte tmp=y[i]; y[i]=y[j]; y[j]=tmp;} while(false);//调整y的ij
        do {double tmp=QD[i]; QD[i]=QD[j]; QD[j]=tmp;} while(false);//调整QD的ij
    }
}
```

Solver

Solver类是一个SVM优化求解的实现技术：SMO（Sequential Minimal Optimization）即序列最小优化算法。Solve函数是这里的核心，我主要分析下对Solve函数的学习。

首先，**SMO**（Sequential Minimal Optimization）是求解SVM 问题的高效算法之一。SMO 算法是一种启发式算法：先选择两个变量αi 和αj，然后固定其他参数，从而将问题转化成一个二变量的二次规划问题。求出能使此时目标值最优的一对αi 和αj 后，将它们固定，再选择两个变量，直到目标值收敛。SMO算法就是采用**每次选取拉格朗日乘子中的2个来更新，直到收敛到最优解。**



由于这一块数学推导太过复杂，我从网上仅仅是进行了理论的学习，等后期有时间再好好理解一下这块的代码分析。

首先明确问题：需要对α进行优化

$$arg\max_{\alpha} \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j \alpha_i \alpha_j \langle x_i, x_j \rangle$$

$$\text{subject to } \alpha_i \geq 0 , \sum_{i=1}^{N} \alpha_i y_i = 0$$

如果固定α1和α2的话，可以合并其他的αi为常数项

$$W(\alpha_1, \alpha_2) = \alpha_1 + \alpha_2 - \frac{1}{2} K_{1,1} y_1^2 \alpha_1^2 - \frac{1}{2} K_{2,2} y_2^2 \alpha_2^2 - K_{1,2} y_1 y_2 \alpha_1 \alpha_2 - y_1 \alpha_1 \sum_{i=3}^{N} \alpha_i y_i K_{i,1} - y_2 \alpha_2 \sum_{i=3}^{N} \alpha_i y_i K_{i,2} + C$$

代入以下三式，就可以得到α1和α2

$$\eta = K_{1,1} + K_{2,2} - 2K_{1,2}$$

$$\alpha_2^{new} = \alpha_2^{old} + \frac{y_2 (E_1 - E_2)}{\eta}$$

在本式的基础上，还可以进行**修剪**

$$\alpha_2^{new} = \begin{cases} H & \alpha_2^{new,unclipped} > H \\ \alpha_2^{new,unclipped} & L \le \alpha_2^{new,unclipped} \le H \\ L & \alpha_2^{new,unclipped} < L \end{cases}$$

$$\alpha_1^{new} = \alpha_1^{old} + y_1 y_2 (\alpha_2^{old} - \alpha_2^{new})$$

另外，当我们更新了一对 αi,αj 之后都需要重新计算**阈值 b**

b的推导公式为：

$$b_1^{new} = y_1 - \sum_{i=3}^{N} \alpha_i y_i K_{i,1} - \alpha_1^{new} y_1 K_{1,1} - \alpha_2^{new} y_2 K_{2,1}$$

$$b_2^{new} = -E_2 - y_1 K_{1,2} (\alpha_1^{new} - \alpha_1^{old}) - y_2 K_{2,2} (\alpha_2^{new} - \alpha_2^{old}) + b^{old}$$

最终得到：

$$b^{new} = \frac{b_1^{new} + b_2^{new}}{2}$$

svm_train

svm_train的逻辑主要是：

- 统计类别总数，同时记录类别的标号，统计每个类的样本数目
- 将属于相同类的样本分组，连续存放并计算权重C
- 训练n(n-1)/2 个模型
- 输出模型，并填充svm_model供后面使用
- 清除内存

```
def svm_train(problem):
    if problem is distribution estimate or regression:
        Initialization
    if problem is regression and need estimating:
        call svm_svr_probability /* 噪声估计*/
        call svm_train_one /* 单次训练*/
        process the parameters
    else: /* 回归问题*/
        call svm_group_classes /* 整理多类别数据*/
        calculate parameter 'C'
    for i <- 1 to nr_class:
    for j <- i+1 to nr_class:
    /* 用一对一策略处理多分类问题*/
        if need estimating:
            call svm_binary_svc_probability
        call svm_train_one
    build output
free the sapce we allocated before
return model we trained
```

svm_scale、svm_predict

svm_scale是用来对原始样本进行缩放的，范围可以自己定，一般是[0,1]或[-1,1]。缩放的目的主要是

1）防止某个特征过大或过小，从而在训练中起的作用不平衡；

2）为了计算速度。因为在核计算中，会用到内积运算或exp运算，不平衡的数据可能造成计算困难。

svm_predict 是根据训练获得的模型，对数据集合进行预测。其的核心为svm_predict_values对测试数据进行预测

• 如果任务是回归，会直接计算拟合函数值并返回

$$\Sigma_j \alpha_j K(x_i, x) - \rho$$

• 如果任务是单类SVM，如果

$$\Sigma_j \alpha_j K(x_j, x) - \rho > 0,$$

则认为测试数据可以被归为训练数据集中，返回1，否则返回-1；
• 如果任务是回归，使用投票法选取票数最多的类作为输出

基本实验结果

使用分析

缩放样本

svm-scale [-l lower] [-u upper]

[-y y_lower y_upper]

[-s save_filename]

[-r restore_filename] filename

其中，[]中都是可选项：

-l：设定数据下限；lower：设定的数据下限值，缺省为-1

-u：设定数据上限；upper：设定的数据上限值，缺省为 1

-y：是否对目标值同时进行缩放；y_lower为下限值，y_upper为上限值；

-s save_filename：表示将缩放的规则保存为文件save_filename；

-r restore_filename：表示将按照已经存在的规则文件restore_filename进行缩放；

filename：待缩放的数据文件，文件格式按照libsvm格式。

举例为：svm-scale –l 0 –u 1 –s test.range test.txt > out.txt

数据范围[0,1]，并把规则保存为test.range，样本压缩后放入out.txt

训练样本

svm-train [options] training_set_file [model_file]

其中可选的主要参数为：

-s 设置svm类型：

0 – C-SVC 1 – v-SVC 2 – one-class-SVM 3 – ε-SVR 4 – n - SVR

-t 设置核函数类型，默认值为2

0 -- 线性核：u'*v* 1 -- *多项式核：* (*g*u'*v*+ *coef* 0)*degree* 2 -- *RBF 核：* *exp*(-γ||u-v||2) 3 -- sigmoid 核：tanh(γ*u*'v+ coef 0)

-d degree: 设置多项式核中degree的值，默认为3

其余的一些具体参数根据具体情况进行赋值

举例为：最基础的情况下应该是svmtrain test.txt test.model

预测数据

svm-predict [options] test_file model_file output_file

分别需要svm-train产生的test_file，符合数据格式要求的model_file，以及输出问卷output_file

实验测试

数据集

svmguide1

- Source: [CWH03a]
- Preprocessing: Original data: an astroparticle application from Jan Conrad of Uppsala University, Sweden.
- # of classes: 2
- # of data: 3,089 / 4,000 (testing)
- # of features: 4
- Files:
  - svmguide1
  - svmguide1.t (testing)

网址为：'LIBSVM Data: Classification (Binary Class) (ntu.edu.tw)'

使用的训练语句为：

```
svm-train svmguide1.txt
svm-predict svmguide1.t svmguide1.txt.model svmguide1.t.predict
#以上是不做scale处理的训练语句
svm-scale -l -1 -u 1 -s range1 svmguide1.txt > svmguide1.scale
svm-scale -r range1 svmguide1.t > svmguide1.t.scale
svm-train svmguide1.scale
svm-predict svmguide1.t.scale svmguide1.scale.model svmguide1.t.predict
#以上是进行scale处理后的结果
```

测试结果

通过测试发现，通过对数据做scale处理，可以大大提高精确度

```
D:\libsvm-master\svmguide1>svm-predict svmguide1.t svmguide1.txt.model svmguide1.t.predict
Accuracy = 66.925% (2677/4000) (classification)

D:\libsvm-master\svmguide1>svm-scale -l -1 -u 1 -s range1 svmguide1.txt > svmguide1.scale

D:\libsvm-master\svmguide1>svm-scale -r range1 svmguide1.t > svmguide1.t.scale

D:\libsvm-master\svmguide1>svm-train svmguide1.scale
*
optimization finished, #iter = 496
nu = 0.202599
obj = -507.307046, rho = 2.627039
nSV = 630, nBSV = 621
Total nSV = 630

D:\libsvm-master\svmguide1>svm-predict svmguide1.t.scale svmguide1.scale.model svmguide1.t.predict
Accuracy = 96.15% (3846/4000) (classification)
```

不过有时候还需要对参数有更多的调整，才能得到好的实验结果。

```
D:\>cd D:\libsvm-master\svmguide3

D:\libsvm-master\svmguide3>svm-scale -l -1 -u 1 -s range1 svmguide3.txt > svmguide3.scale

D:\libsvm-master\svmguide3>svm-scale -r range1 svmguide3.t > svmguide3.t.scale

D:\libsvm-master\svmguide3>svm-train svmguide3.scale
*
optimization finished, #iter = 809
nu = 0.464225
obj = -551.002527, rho = 0.334382
nSV = 602, nBSV = 556
Total nSV = 602

D:\libsvm-master\svmguide3>svm-predict svmguide3.t.scale svmguide3.scale.model svmguide3.t.predict
Accuracy = 78.3588% (974/1243) (classification)

D:\libsvm-master\svmguide3>
```

总结

总结而言,Libsvm是一个可以简单快速使用SVM模式识别与回归的开源包。它提供了很多的可以用来调节的参数，在svm-scale数据，svm-train训练样本，最后svm-predict预测结果中可以简便使用，在本次对数据的训练之中，我尝试了对svmguide提供的数据样本做了测试，在进行了svm-scale后得到了精确率很大的改善；此外，我在阅读代码指导的过程中，也曾查阅到用libsvm对minst的数据可以进行实验，我略微修改了它的代码进行了实验，得到的精确率大约在95%左右。

在使用上，我觉得碰到过的一个问题是，如果要使用python来进行一个自动化执行的方式，在windows上的配置有些复杂。因此在对minst进行实验的时候，我是在虚拟机上重新下载了libsvm配了环境去使用的，会方便许多。

在代码方面，在阅读了这次代码后，我对于SVM的框架，比如Cache、Kernel模块，对于机器学习都有了更深刻的了解。并且，在查阅资料中，我还了解到，libsvm可以选择各种核，可以解决C-SVM、v-SVM、ε-SVR和v-SVR等问题，以及基于一对一算法的多类模式识别问题。对于LibSVM，确实是一个十分方便的软件包。

但是在理论方面，对于Solver的包我确还是有一些阅读困难，可能是这一块的原理比较复杂，目前只能是跟着网上的专门分析大致了解这一块的数学原理和思想是什么，希望等到后面再次碰到SMO算法的时候对这一块的基础有一个更好的补全。

作者

From 胡若凡 3200102312