

LibSVM-2.6 程序代码注释

我不经心地，服下你调好的毒
我知道今后我将万劫不复
但是你的红唇仍让我屈服
四月的樱花火红满天
我和我的梦，却要一以何处去继续？
虽然人间的情爱万万千千
世上已有太多崩毁的誓言
七个黑夜，七个白天
我为你写下的歌，彩绘的纸笺
却只能随着晚风
飘在大海的岸边
我仍愿服下你精心为我调好的毒
从你那深情的吻
吞下我与你在人间
最后的流光万千辗转朱颜……

第一节： SVM.h 文件

struct svm_node

```
{
    int index;
    double value;
};
```

struct svm_node 用来存储单一向量中的单个特征，例如：

向量 x1={ 0.002, 0.345, 4, 5.677};

那么用 struct svm_node 来存储时就使用一个包含 5 个 svm_node 的数组来存储此 4 维向量，内存映象如下：

1	2	3	4	-1
0.002	0.345	4. 000	5.677	空

其中如果 value 为 0.00,该特征将不会被存储，其中(特征 3)被跳过：

1	2	4	5	-1
0.002	0.345	4. 000	5.677	空

0.00 不保留的好处在于，做点乘的时候，可以加快计算速度，对于稀疏矩阵，更能充分体现这种数据结构的优势。但做归一化时，操作就比较麻烦了。

(类型转换不再说明)

struct svm_problem

```
{
    int l;
    double *y;
    struct svm_node **x;
};
```

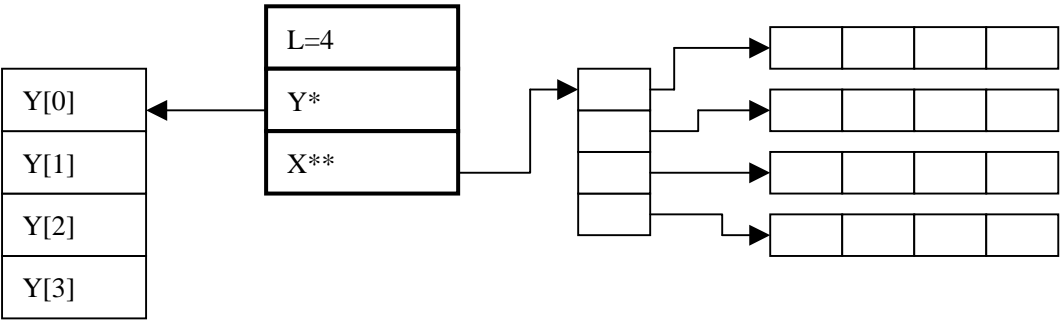
struct svm_problem存储本次参加运算的所有样本（数据集），及其所属类别。在某些数据挖掘实现中，常用DataSet来实现。

int l;记录样本总数

double *y;指向样本所属类别的数组。在多类问题中，因为使用了one-against-one方法，可能原始样本中y[i]的内容是1.0, 2.0, 3.0, ...,但参与多类计算时，参加分类的两类所对应的y[i]内容是+1, 和-1。

Struct svm_node **x;指向一个存储内容为指针的数组；

如下图,最右边的四个长条格同上表，存储三维数据。(黑边框的是最主要的部分)



这样的数据结构有一个直接的好处，可以用 $x[i][j]$ 来访问其中的某一元素（如果value为0.00的也全部保留的话）

私下认为其中有一个败笔，就是把 $\text{svm_node}^* \text{x_space}$ 放到结构外面去了。

```
enum { C_SVC, NU_SVC, ONE_CLASS, EPSILON_SVR, NU_SVR }; /* svm_type */
enum { LINEAR, POLY, RBF, SIGMOID }; /* kernel_type */

struct svm_parameter
{
    int svm_type; /* SVM类型，见前enum */
    int kernel_type; /* 核函数 */
    double degree; /* for poly */
    double gamma; /* for poly/rbf/sigmoid */
    double coef0; /* for poly/sigmoid */

    /* these are for training only */
    double cache_size; /* in MB */
    double eps; /* stopping criteria */
    double C; /* for C_SVC, EPSILON_SVR and NU_SVR */
    int nr_weight; /* for C_SVC */
    int *weight_label; /* for C_SVC */
    double *weight; /* for C_SVC */
    double nu; /* for NU_SVC, ONE_CLASS, and NU_SVR */
    double p; /* for EPSILON_SVR */
    int shrinking; /* use the shrinking heuristics */
    int probability; /* do probability estimates */
};
```

部分参数解释,(附核函数)

- 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、 $K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2), \gamma > 0$
- 4、 $K(x_i, x_j) = \tanh(\gamma x_i^T x_j + r)$

```
double degree; /* 就是2式中的d */
double gamma; /* 就是2,3,4式中的gamma */
double coef0; /* 就是2,4式中的r */
```

$\text{double cache_size};$ /* in MB */ 制定训练所需要的内存，默认是40M，LibSVM2.5中是4M,所以自己做开发选LibSVM2.5还是不错的！

$\text{double eps};$ 见参考文献[1]中式3.13

$\text{double C};$ 没什么好说的，惩罚因子，越大训练的模型越那个...,当然耗的时间越多

`int nr_weight;` //权重的数目,目前在实例代码中只有两个值,一个是默认0,另外一个是在 `svm_binary_svc_probability` 函数中使用数值2。

`int *weight_label;` //权重,元素个数由 `nr_weight` 决定。

`double nu;` // 没什么好说的,too

`double p;` // 没什么好说的,three

`int shrinking;` //指明训练过程是否使用压缩。

`int probability;` //新增,指明是否要做概率估计

`struct svm_model`

```
{
    svm_parameter param;    // parameter
    int nr_class;            // number of classes, = 2 in regression/one class svm
    int l;                   // total #SV
    svm_node **SV;           // SVs (SV[l])
    double **sv_coef;        // coefficients for SVs in decision functions (sv_coef[n-1][l])
    double *rho;             // constants in decision functions (rho[n*(n-1)/2])
    double *probA;           // pariwise probability information
    double *probB;

    // for classification only
    int *label;              // label of each class (label[n])
    int *nSV;                // number of SVs for each class (nSV[n])
                             // nSV[0] + nSV[1] + ... + nSV[n-1] = l
    // XXX
    int free_sv;             // 1 if svm_model is created by svm_load_model
                             // 0 if svm_model is created by svm_train
};
```

结构体 `svm_model` 用于保存训练后的训练模型,当然原来的训练参数也必须保留。

`svm_parameter param;` // 训练参数

`int nr_class;` // 类别数

`int l;` // 支持向量数

`svm_node **SV;` // 保存支持向量的指针,至于支持向量的内容,如果是从文件中读取,内容会额外保留;如果是直接训练得来,则保留在原来的训练集中。如果训练完成后需要预报,原来的训练集内存不可以释放。

`double **sv_coef;` //相当于判别函数中的 α

`double *rho;` //相当于判别函数中的 b

`double *probA;` // pariwise probability information

`double *probB;` //均为新增函数

`int *label;` // label of each class (label[n])

`int *nSV;` // number of SVs for each class (nSV[n])

`int free_sv;` //见 `svm_node **SV` 的注释

//以下接口函数设计得非常合理，最后一节详细说明

//最主要的驱动函数，训练数据

```
struct svm_model *svm_train(const struct svm_problem *prob, const struct svm_parameter *param);
```

//用SVM做交叉验证

```
void svm_cross_validation(const struct svm_problem *prob, const struct svm_parameter *param, int  
nr_fold, double *target);
```

//保存训练好的模型到文件

```
int svm_save_model(const char *model_file_name, const struct svm_model *model);
```

//从文件中把训练好的模型读到内存中

```
struct svm_model *svm_load_model(const char *model_file_name);
```

//

```
int svm_get_svm_type(const struct svm_model *model);
```

//得到数据集的类别数（必须经过训练得到模型后才可以）

```
int svm_get_nr_class(const struct svm_model *model);
```

//得到数据集的类别标号（必须经过训练得到模型后才可以）

```
void svm_get_labels(const struct svm_model *model, int *label);
```

//LibSvm2.6新增函数

```
double svm_get_svr_probability(const struct svm_model *model);
```

//用训练好的模型预报样本的值，输出结果保留到数组中。（并非接口函数）

```
void svm_predict_values(const struct svm_model *model, const struct svm_node *x, double*  
dec_values);
```

//预报某一样本的值

```
double svm_predict(const struct svm_model *model, const struct svm_node *x);
```

// LibSvm2.6新增函数

```
double svm_predict_probability(const struct svm_model *model, const struct svm_node *x, double*  
prob_estimates);
```

//消除训练的模型，释放资源

```
void svm_destroy_model(struct svm_model *model);
```

// LibSvm2.6新增函数

```
void svm_destroy_param(struct svm_parameter *param);
```

//检查输入的参数，保证后面的训练能正常进行。

```
const char *svm_check_parameter(const struct svm_problem *prob, const struct svm_parameter  
*param);
```

```
// LibSvm2.6新增函数
```

```
int svm_check_probability_model(const struct svm_model *model);
```

第二节： SVM.cpp 文件

.头文件：

从整个.cpp 文件来看，感觉有些头文件是多余的，不知何故，反正多包含头文件不会犯错。

后面的 typedef,特别是 typedef float Qfloat,是为了方便控制内存存储的精度。

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <float.h>
#include <string.h>
#include <stdarg.h>
#include "svm.h"
typedef float Qfloat;
typedef signed char schar;
```

//以下是定义的几个主要的模板，主要是为了比较大小，交换数据和完全复制数据。

Min()和 Max()在<math.h>中提供了相应的函数，这里的处理，估计是为了使函数内联，执行速度会相对快一些，而且不同的数据类型，存储方式不同，使用模板会更有针对性，也从另外一方面提高程序性能。

```
#ifndef min
template <class T> inline T min(T x,T y) { return (x<y)?x:y; }
#endif

#ifndef max
template <class T> inline T max(T x,T y) { return (x>y)?x:y; }
#endif

template <class T> inline void swap(T& x, T& y) { T t=x; x=y; y=t; }
```

//这里的克隆函数是完全克隆，不同于一般的复制。操作结束后，内部的所有数据和指针完全一样。

```
template <class S, class T> inline void clone(T*& dst, S* src, int n)
{
    dst = new T[n];
    memcpy((void *)dst,(void *)src,sizeof(T)*n);
}
```

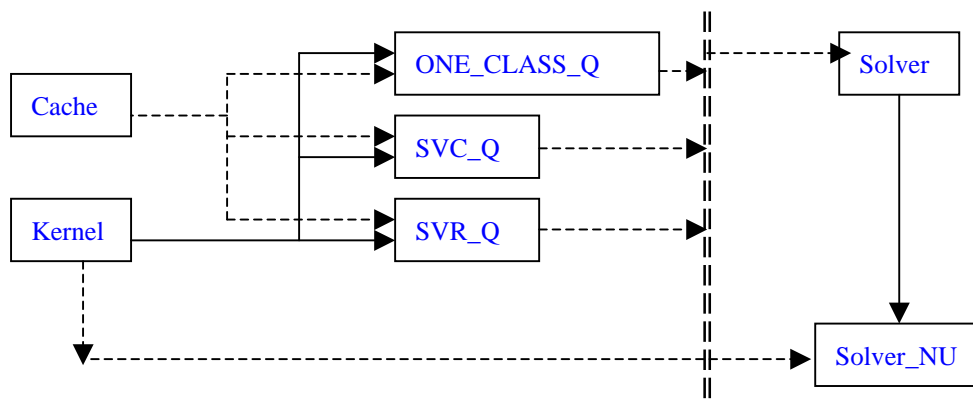
//这里使用了 define，非内联函数

```
#define INF HUGE_VAL
#define Malloc(type,n) (type *)malloc((n)*sizeof(type))
```


//以下的函数用作调试。跳过～

```
#if 1
void info(char *fmt,...)
{
    va_list ap;
    va_start(ap,fmt);
    vprintf(fmt,ap);
    va_end(ap);
}
void info_flush()
{
    fflush(stdout);
}
#else void info(char *fmt,...) {}
void info_flush() {}
#endif
```

//以下部分为 svm.cpp 中的类继承和组合图: (实线表示继承关系, 虚线表示组合关系)



2.1 类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。

类定义:

```
class Cache
{
public:
    Cache(int l,int size);
    ~Cache();
    int get_data(const int index, Qfloat **data, int len);
    void swap_index(int i, int j); // future_option
private:
    int l;
    int size;
    struct head_t
    {
```

```

    head_t *prev, *next; // a circular list
    Qfloat *data;
    int len;           // data[0,len) is cached in this entry
};

head_t * head;
head_t lru_head;
void lru_delete(head_t *h);
void lru_insert(head_t *h);
};

```

成员变量:

`head_t * head;` //变量指针, 该指针用来记录程序所申请的内存, 单块申请到的内存用 `struct head_t` 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。记录所有申请到的内存, 一方面便于释放内存, 另外方便在内存不够时适当释放一部分已经申请到的内存。

`head_t lru_head;` //双向链表的头。

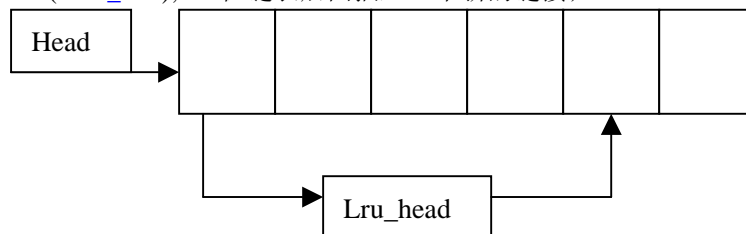
`int l;` //样本总数。

`int size;` //所指定的全部内存, 据说用Mb做单位。

成员函数:

`void lru_delete(head_t *h);` //从双向链表中删除某个元素的链接, 不删除、不释放该元素所涉及的内存。一般是删除当前所指向的元素。

`void lru_insert(head_t *h);` //在链表后面插入一个新的链接;



`Cache(int l, int size);`

构造函数。该函数根据样本数 `L`, 申请 `L` 个 `head_t` 的空间。根据说明, 该区域会初始化为 0, (表示怀疑)。 `Lru_head` 因为尚没有 `head_t` 中申请到内存, 故双向链表指向自己。至于 `size` 的处理, 先将原来的 `byte` 数目转化为 `float` 的数目, 然后扣除 `L` 个 `head_t` 的内存数目。 `size` 为程序指定的内存大小 4M/40M。 `size` 不要设得太小。

`int get_data(const int index, Qfloat **data, int len);`

该函数保证 `head_t[index]` 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。返回值为申请到的内存。

函数首先将 `head_t[index]` 从链表中断开, 如果 `head_t[index]` 原来没有分配内存, 则跳过断开这一步。计算当前 `head_t[index]` 已经申请到的内存, 如果不够, 释放部分内存 (怀疑这样做的动机: 老数据为什么就可以释放, 而不真的另外申请一块? 老数据没用了?), 等内存足够后, 重新分配内存。重新使 `head_t[index]` 进入双向链表。并返回申请到的内存的长度。

//返回值不为申请到的内存的长度, 为 `head_t[index]` 原来的数据长度 `h->len`。

调用该函数后，程序会计算 $Q = \sum y_i y_j K(x_i, x_j)$ 的值，并将其填入 data 所指向的内存区域，如果下次 index 不变，正常情况下，不用重新计算该区域的值。若 index 不变，则 get_data() 返回值 len 与本次传入的 len 一致，从 Kernel::get_Q() 中可以看到，程序不会重新计算。从而提高运算速度。

While 循环内的部分基本上难得用到一次。

```
void swap_index(int i, int j);
```

交换 head_t[i] 和 head_t[j] 的内容，先从双向链表中断开，交换后重新进入双向链表中。对后面的处理不理解，可能是防止中 head_t[i] 和 head_t[j] 可能有一方并未申请内存。但 $h \rightarrow len > i$ 和 $h \rightarrow len > j$ 无法解释。

```
for(head_t *h = lru_head.next; h!=&lru_head; h=h->next)
{
    if(h->len > i)
    {
        if(h->len > j)
            swap(h->data[i],h->data[j]);
        else
        {
            // give up
            lru_delete(h);
            free(h->data);
            size += h->len;
            h->data = 0;
            h->len = 0;
        }
    }
}
```

2.2 类 Kernel

```
class Kernel {
public:
    Kernel(int l, svm_node * const * x, const svm_parameter& param);
    virtual ~Kernel();

    static double k_function(const svm_node *x, const svm_node *y, const svm_parameter& param);
    virtual Qfloat *get_Q(int column, int len) const = 0;
    virtual void swap_index(int i, int j) const // no so const...
    {
        swap(x[i],x[j]);
        if(x_square) swap(x_square[i],x_square[j]);
    }
protected:
    double (Kernel::*kernel_function)(int i, int j) const;
```

private:

```
const svm_node **x;
double *x_square;

// svm_parameter
const int kernel_type;
const double degree;
const double gamma;
const double coef0;

static double dot(const svm_node *px, const svm_node *py);
double kernel_linear(int i, int j) const(skipped)
double kernel_poly(int i, int j) const(skipped)
double kernel_rbf(int i, int j) const(skipped)
double kernel_sigmoid(int i, int j) const(skipped)
};
```

成员变量:

`const svm_node **x;` //用来指向样本数据，每次数据传入时通过克隆函数来实现，完全重新分配内存，主要是为处理多类着想。

`double *x_square;` //使用 RBF 核才使用。

`const int kernel_type;` //核函数类型。

`const double degree;` // kernel_function

`const double gamma;` // kernel_function

`const double coef0;` // kernel_function

成员函数:

`Kernel(int l, svm_node * const * x, const svm_parameter& param);`

构造函数。初始化类中的部分常量、指定核函数、克隆样本数据。如果使用 RBF 核函数，则计算 `x_square[i]`。

`static double dot(const svm_node *px, const svm_node *py);`

点乘两个样本数据，按 `svm_node` 中 `index` (一般为特征)进行运算，一般来说，`index` 中 1, 2, ... 直到 -1。返回点乘总和。

例如：`x1 = { 1,2,3}` , `x2 = {4, 5, 6}` 总和为 `sum = 1*4 + 2*5 + 3*6` ;在 `svm_node[3]`中存储 `index = -1` 时，停止计算。

`static double k_function(const svm_node *x, const svm_node *y, const svm_parameter& param);`

核函数。但只有在预报时才用到。

其中 RBF 部分很有讲究。因为存储时，0 值不保留。如果所有 0 值都保留，第一个 `while` 就可以都做完了；如果第一个 `while` 做不完，在 `x`, `y` 中任意一个出现 `index = -1`，第一个 `while` 就停止，剩下的代码中两个 `while` 只会有一个工作，该循环直接把剩下的计算做完。

```
virtual Qfloat *get_Q(int column, int len) const = 0;
```

纯虚函数，将来在子类中实现。相当重要的函数。

```
virtual void swap_index(int i, int j)
```

虚函数，x[i]和 x[j]中所存储指针的内容。如果 x_square 不为空，则交换相应的内容。

```
double (Kernel::*kernel_function)(int i, int j) const;
```

函数指针，根据相应的核函数类型，来决定所使用的函数。在计算矩阵 Q 时使用。

$$Q = \sum y_i y_j K(x_i, x_j)$$

$$1、K(x_i, x_j) = x_i^T x_j$$

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

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

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

2.2 类 Solver

```
class Solver {
public:
    Solver() {};
    virtual ~Solver() {};

    struct SolutionInfo {
        double obj;
        double rho;
        double upper_bound_p;
        double upper_bound_n;
        double r; // for Solver_NU
    };

    void Solve(int l, const Kernel& Q, const double *b_, const schar *y_,
               double *alpha_, double Cp, double Cn, double eps,
               SolutionInfo* si, int shrinking);
protected:
    int active_size;
    schar *y;
    double *G;           // gradient of objective function
    enum { LOWER_BOUND, UPPER_BOUND, FREE };
    char *alpha_status; // LOWER_BOUND, UPPER_BOUND, FREE
    double *alpha;
```

```

const Kernel *Q;
double eps;
double Cp,Cn;
double *b;
int *active_set;
double *G_bar;    // gradient, if we treat free variables as 0
int l;
bool unshrunked;   // XXX

double get_C(int i) {    }
void update_alpha_status(int i) {    }
bool is_upper_bound(int i) { return alpha_status[i] == UPPER_BOUND; }
bool is_lower_bound(int i) { return alpha_status[i] == LOWER_BOUND; }
bool is_free(int i) { return alpha_status[i] == FREE; }
void swap_index(int i, int j);
void reconstruct_gradient();
virtual int select_working_set(int &i, int &j);
virtual double calculate_rho();
virtual void do_shrinking();
};

```

成员变量:

`int active_size;` // 计算时实际参加运算的样本数目，经过 `shrink` 处理后，该数目会小于全部样本总数。

`schar *y;` // 样本所属类别，该值只取+1/-1。虽然可以处理多类，最终是用两类 SVM 完成的。

`double *G;` // 梯度，计算公式如下(公式 3.5)[1]:

$$(Q\alpha + p)_t = \nabla f(\alpha)_t = G_t$$

在代码实现中，用 `b[i]` 来代替公式中的 p 。

`char *alpha_status;` // $\alpha[i]$ 的状态，根据情况分为 $\alpha[i] \leq 0$ ， $\alpha[i] \geq c$ 和 $0 < \alpha[i] < c$ ，分别对应内部点(非 SV)，错分点(BSV)和支持向量(SV)。

`double *alpha;` // α_i

`const Kernel *Q;` // 指定核。核函数和 Solver 相互结合，可以产生多种 SVC,SVR

`double eps;` // 误差限

`double *b;` // 见 `double *G` 的说明。

`int *active_set;` //

`double *G_bar;` // \bar{G} ，(这名字取的)。计算公式如下:

$$\bar{G} = C \sum_{\alpha_j = C} Q_{ij}, i = 1, \dots, l$$

该值可以在对样本集做 `shrink` 时，减小重建梯度的计算量。

$$G = \bar{G} + \sum_{0 < \alpha < C} Q_{ij} \alpha_j = \sum_{j=1}^l Q_{ij} \alpha_j$$

```
int l; //样本总数
bool unshrunked; //
```

成员函数:

```
double get_C(int i)
```

返回对应于样本的 C 。设置不同的 C_p 和 C_n 是为了处理数据的不平衡。见《6 Unbalanced data》[1], 有时 $C_p = C_n$ 。

```
void swap_index(int i, int j);
```

完全交换样本 i 和样本 j 的内容, 包括所申请的内存的地址。

```
void reconstruct_gradient();
```

重新计算梯度。 $G_bar[i]$ 在初始化时并未加入 $b[i]$, 所以程序首先增加 $b[i]$ 。Shrink 后依然参加运算的样本位于 $active_size$ 和 $L-1$ 位置上。在 $0 \sim active_size$ 之间的 $alpha[i]$ 如果在区间 $(0, c)$ 上, 才有必要更新相应的 $active_size$ 和 $L-1$ 位置上的样本的梯度。

```
virtual int select_working_set(int &i, int &j)
```

选择工作集。公式如下:

$$i \equiv \arg \max(\{-\nabla f(\alpha)_t \mid y_t = 1, \alpha_t < C\}, \{\nabla f(\alpha)_t \mid y_t = -1, \alpha_t > 0\})$$

$$j \equiv \arg \min(\{\nabla f(\alpha)_t \mid y_t = -1, \alpha_t < C\}, \{-\nabla f(\alpha)_t \mid y_t = 1, \alpha_t > 0\})$$

```
virtual void do_shrinking();
```

对样本集做缩减。大致是当 $0 < \alpha < C$ 时, (还有两种情况)程序认为该样本可以不参加下次迭代。($0 < \alpha < C$ 时, 为内部点) 程序会减小 $active_size$, 为 (内部点) 增加位置。 $active_size$ 表明了不可以参加下次迭代的样本的最小标号, 在 $active_size$ 与 L 之间的元素都对分类没有贡献。

程序中 $k--$ 是为了消除交换后的影响, 使重新换来的样本也被检查一次。

如果程序在缩减一次后没有达到结束条件, 就重新构造梯度矢量, 并再缩减一次 (总觉得这里不太严密)。

```
virtual double calculate_rho();
```

计算 ρ 值。见 3.7[1] 节, *The calculation of b or ρ*

$$r_1 = \frac{\sum_{0 < \alpha < C, y_i = 1} \nabla f(\alpha)_i}{\sum_{0 < \alpha < C, y_i = 1} 1}$$

$$\rho = \frac{r_1 + r_2}{2}$$

```
void Solve(int l, const Kernel& Q, const double *b_, const schar *y_,  
           double *alpha_, double Cp, double Cn, double eps,  
           SolutionInfo* si, int shrinking);
```

//程序较大，逐步分解

part 1

// initialize alpha_status

```
{  
    alpha_status = new char[l];  
    for(int i=0;i<l;i++)  
        update_alpha_status(i);  
}
```

更新一下 alpha 的状态

part 2

// initialize active set (for shrinking)

```
{  
    active_set = new int[l];  
    for(int i=0;i<l;i++)  
        active_set[i] = i;  
    active_size = l;  
}
```

为缩减做准备，将来要做交换

part 3

// initialize gradient

```
{  
    G = new double[l];  
    G_bar = new double[l];  
    int i;  
    for(i=0;i<l;i++)  
    {  
        G[i] = b[i];  
        G_bar[i] = 0;  
    }  
    for(i=0;i<l;i++)  
        if(!is_lower_bound(i))  
        {  
            Qfloat *Q_i = Q.get_Q(i,l);  
            double alpha_i = alpha[i];  
            int j;  
            for(j=0;j<l;j++)  
                G[j] += alpha_i*Q_i[j];  
            if(is_upper_bound(i))  
                for(j=0;j<l;j++)
```



```

        G_bar[j] += get_C(i) * Q_i[j];
    }
}

```

$G_bar[j]$ 的生成公式如下：（注意，其中不包含 $b[i]$ 的值）

$$\bar{G} = C \sum_{\alpha_j=C} Q_{ij}, i=1,...,l$$

因为第一次建立 $G(i)$, 所以没有判断 α 的状态。而是按公式，全部计算了一遍。

$get_Q(i,l)$ 返回的值是 Q_{ij} 矩阵中的第 i 列，而不是第 i 行，这是需要注意的地方。

再往下是大循环：

如果有必要，先进行筛选，使部分数据不再参加运算；选择工作集；更新 α_i, α_j , 其更新的思路是保证： $\alpha_i^{new} y_i + \alpha_j^{new} y_j = \alpha_i^{old} y_i + \alpha_j^{old} y_j$; 对于边界情况，有特殊处理，主要是考虑

$0 \leq \alpha_i \leq C_i$ 的要求。当某一 α 小于 0 时，做适当调整，调整的结果是 α_i, α_j 仍然在

$0 \leq \alpha_i \leq C_i$ 范围内，同时其和同原来一样。对于推导过程，可以参考 *Sequential Minimal Optimization for SVM*

part 4

更新 $G(i)$, 根据 α_i, α_j 的变化更新；

```

// update G
double delta_alpha_i = alpha[i] - old_alpha_i;
double delta_alpha_j = alpha[j] - old_alpha_j;

for(int k=0; k<active_size; k++)
{
    G[k] += Q_i[k]*delta_alpha_i + Q_j[k]*delta_alpha_j;
}

```

part 5

以下是更新 α_status 和 \bar{G} , α 状态更新较简单，根据 α 状态前后是否有变化，适

当更新，更新的内容参考公式 $\bar{G} = C \sum_{\alpha_j=C} Q_{ij}, i=1,...,l$

```

// update alpha_status and G_bar
{
    bool ui = is_upper_bound(i);
    bool uj = is_upper_bound(j);
    update_alpha_status(i);
}

```

```
    update_alpha_status(j);
    int k;
    if(ui != is_upper_bound(i))//更新alpha_i的影响
    {
        Q_i = Q.get_Q(i,l);
        if(ui)
            for(k=0;k<l;k++)
                G_bar[k] -= C_i * Q_i[k];
        else
            for(k=0;k<l;k++)
                G_bar[k] += C_i * Q_i[k];
    }
    if(uj != is_upper_bound(j)) //更新alpha_j的影响
    {
        Q_j = Q.get_Q(j,l);
        if(uj)
            for(k=0;k<l;k++)
                G_bar[k] -= C_j * Q_j[k];
        else
            for(k=0;k<l;k++)
                G_bar[k] += C_j * Q_j[k];
    }
}
```

part 6

以下计算目标函数值，因为 $G_t = (Q\alpha + p)_t$ ，而目标值为 $\frac{1}{2}\alpha^T Q\alpha + p^T \alpha$ ，故：

```
// calculate objective value
{
    double v = 0;
    int i;
    for(i=0;i<l;i++)
        v += alpha[i] * (G[i] + b[i]);

    si->obj = v/2;
}
```

part 7

回送结果。

```
// put back the solution
{
    for(int i=0;i<l;i++)
        alpha_[active_set[i]] = alpha[i];
}
```

2.3 类 Solver_NU

```
class Solver_NU : public Solver
{
public:
    Solver_NU() {}
    void Solve(int l, const Kernel& Q, const double *b, const schar *y,
               double *alpha, double Cp, double Cn, double eps,
               SolutionInfo* si, int shrinking)
    {
        this->si = si;
        Solver::Solve(l,Q,b,y,alpha,Cp,Cn,eps,si,shrinking);
    }
private:
    SolutionInfo *si;
    int select_working_set(int &i, int &j);
    double calculate_rho();
    void do_shrinking();
};
```

其中函数 `void Solve()` 完全调用了 `Solver::Solve()`, `this->si = si;` 一句是因为 C++ 内部变量访问的限制而添加。

成员函数:

```
int select_working_set(int &i, int &j);
```

选择工作集, 参考[1],[4],[5],同时可以参考 `Solver::select_working_set`。

```
double calculate_rho();
```

计算 ρ 值, 参考[1],[4],[5] (对应 libsvm 论文[1], 其实返回值是 b , 这可以从后面预测目标值可以看出。与 `Solver::calculate_rho` 相比, 增加了另外一个返回值, r , 该值才是真正的 ρ 值。

```
void do_shrinking();
```

对样本进行剪裁, 参考[1],[4],[5],同时可以参考 `Solver::do_shrinking()`。

2.4 类 SVC_Q

```
class SVC_Q: public Kernel
{
public:
    SVC_Q(const svm_problem& prob, const svm_parameter& param, const schar *y_)
    :Kernel(prob.l, prob.x, param)
    {
```

```
        clone(y,y_,prob.l);
        cache = new Cache(prob.l,(int)(param.cache_size*(1<<20)));
    }

    Qfloat *get_Q(int i, int len) const
    {
        Qfloat *data;
        int start;
        if((start = cache->get_data(i,&data,len)) < len)
        {
            for(int j=start;j<len;j++)
                data[j] = (Qfloat)(y[i]*y[j]*(this->*kernel_function)(i,j));
        }
        return data;
    }

    void swap_index(int i, int j) const
    {
        cache->swap_index(i,j);
        Kernel::swap_index(i,j);
        swap(y[i],y[j]);
    }

    ~SVC_Q()
    {
        delete[] y;
        delete cache;
    }
private:
    schar *y;
    Cache *cache;
};
```

说明:

```
SVC_Q(const svm_problem& prob, const svm_parameter& param, const schar *y_)
:Kernel(prob.l, prob.x, param)
```

该构造函数利用初始化列表`Kernel(prob.l, prob.x, param)`将样本数据和参数传入(非常简洁)。

`get_Q(int i, int len)`函数与其他同类相比, 在于核函数不同。

`swap_index(int i, int j)` //交换的东西太多了点

2.5 类 ONE_CLASS_Q

```
class ONE_CLASS_Q: public Kernel
{
```

```
public:
    ONE_CLASS_Q(const svm_problem& prob, const svm_parameter& param)
        :Kernel(prob.l, prob.x, param)
    {
        cache = new Cache(prob.l,(int)(param.cache_size*(1<<20)));
    }

    Qfloat *get_Q(int i, int len) const
    {
        Qfloat *data;
        int start;
        if((start = cache->get_data(i,&data,len)) < len)
        {
            for(int j=start;j<len;j++)
                data[j] = (Qfloat)(this->*kernel_function)(i,j);
        }
        return data;
    }

    void swap_index(int i, int j) const
    {
        cache->swap_index(i,j);
        Kernel::swap_index(i,j);
    }

    ~ONE_CLASS_Q()
    {
        delete cache;
    }

private:
    Cache *cache;
};
```

ONE_CLASS_Q 只处理 1 类分类问题(?), 故不保留 $y[i]$ 。编号只有 1 类。
`get_Q(int i, int len)`函数中缺少了 $y[i], y[j]$, 这与 One_Class 本身特点有关, 只处理一类。
`swap_index(int i, int j)`少 `swap(y[i], y[j]);`这句, 因为根本没有 $y[i]$ 可供交换。

2.5 类 SVR_Q

```
class SVR_Q: public Kernel
{
public:
    SVR_Q(const svm_problem& prob, const svm_parameter& param)
        :Kernel(prob.l, prob.x, param)
```

```
{
    //skipped
}

void swap_index(int i, int j) const
{
    swap(sign[i],sign[j]);
    swap(index[i],index[j]);
}

Qfloat *get_Q(int i, int len) const
{
    //skipped
}

~SVR_Q()
{
    //skipped
}
private:
    int l;
    Cache *cache;
    schar *sign;
    int *index;
    mutable int next_buffer;
    Qfloat* buffer[2];
};
```

本类主要是用于做回归，同分类有许多不同之处。参考[1],[5]

//以下的函数全为静态函数，只能在本文件范围内被访问。对照[1]中公式查看。

2.6 函数 solve_c_svc

```
static void solve_c_svc(const svm_problem *prob, const svm_parameter* param,
    double *alpha, Solver::SolutionInfo* si, double Cp, double Cn)
```

在公式 $\frac{1}{2}\alpha^T Q \alpha + p^T \alpha$ 中， p^T 为全-1,另外 $\alpha[i]=0$,保证 $y^T \alpha = 0$ 的限制条件，在将来选择工作集后更新 α 时，仍能保证该限制条件。

2.7 函数 solve_nu_svc

```
static void solve_nu_svc( const svm_problem *prob, const svm_parameter *param,
    double *alpha, Solver::SolutionInfo* si)
```

p^T 为全 0, $\alpha[i]$ 能保证 $e^T \alpha = 0, y^T \alpha = 0$.

2.8 函数 solve_one_class

```
static void solve_one_class(const svm_problem *prob, const svm_parameter *param,
    double *alpha, Solver::SolutionInfo* si)
```

限制条件 $e^T \alpha = vl$, 前 vl 个 α 为 1, 此后的 α 全为 0, 初始条件满足限制条件 $e^T \alpha = vl$

p^T 为全 0, y 为全 1

2.9 函数 solve_epsilon_svr

```
static void solve_epsilon_svr(const svm_problem *prob, const svm_parameter *param,
    double *alpha, Solver::SolutionInfo* si)
```

2.10 函数 solve_nu_svr

```
static void solve_nu_svr(const svm_problem *prob, const svm_parameter *param,
    double *alpha, Solver::SolutionInfo* si)
```

第三节： 接口函数、流程

```
decision_function svm_train_one(const svm_problem *prob, const svm_parameter *param,
    double Cp, double Cn)
```

训练一组样本集，通常参加训练的样本集只有两类。

程序根据相应的参数，选择所使用的训练或者拟合算法。(这个地方的代码居然如此少)，最后统计SV和BSV，最后输出决策函数。

```
void sigmoid_train( int l, const double *dec_values, const double *labels,
    double& A, double& B)
```

LibSVM2.6新增函数

根据预报值来确定 A,B $r_{ij} \approx \frac{1}{1 + e^{A\hat{f} + B}}$ 见第 8 节[1],其中 A,B 的确定就由本函数确定。

```
double sigmoid_predict(double decision_value, double A, double B)
```

LibSVM2.6新增函数

可以看看，里面的公式很简单。

```
void multiclass_probability(int k, double **r, double *p)
```

LibSVM2.6新增函数

(好像比较复杂哦☺)

```
void svm_binary_svc_probability(const svm_problem *prob, const svm_parameter *param,
    double Cp, double Cn, double& probA, double& probB)
```

LibSVM2.6新增函数

先做交叉验证，然后用决策值来做概率估计。需要调用 `sigmoid_train` 函数。

`double svm_svr_probability(const svm_problem *prob, const svm_parameter *param)`

LibSVM2.6新增函数

先做交叉验证，然后函数经过计算后，输出概率值。

`svm_model *svm_train(const svm_problem *prob, const svm_parameter *param)`

根据选择的算法，来组织参加训练的分样本，以及进行训练结果的保存。其中会对样本进行初步的统计。

一、分类部分：

- 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
- 计算权重C
- 训练 $n(n-1)/2$ 个模型
 - 初始化nozero数组，便于统计SV
 - //初始化概率数组
 - 训练过程中，需要重建子数据集，样本的特征不变，但样本的类别要改为+1/-1
 - //如果有必要，先调用`svm_binary_svc_probability`
 - 训练子数据集`svm_train_one`
 - 统计一下nozero, 如果nozero已经是真，就不变，如果为假，则改为真
- 输出模型
 - 主要是填充`svm_model`,
- 清除内存

二、回归部分：

- 类别数固定为2
- //选择性地做`svm_svr_probability`, one-class不做概率估计
- 训练
- 输出模型
- 清除内存

训练过程函数调用：

`svm_train`→`svm_train_one`→`solve_c_svc`(for example)→

- `Solver s`;//这里调用构造函数，但啥也没有做。
- `s.Solve(l, SVC_Q(*prob,*param,y), minus_ones, y, alpha, Cp, Cn, param->eps, si,`

`param->shrinking);`

→调用`SVC_Q(Kernel)` 类的构造函数，同时也会调用`Kernel`类的构造函数。在`SVC_Q`类的构造函数中复制目标值(y),同时申请内存，此时激发`Cache`类,申请内存，构造双向列表等。

→`Solve`函数做完其他部分工作，主要是算法的实现。

`void svm_cross_validation(const svm_problem *prob, const svm_parameter *param, int nr_fold, double *target)`

LibSVM2.6新增函数,LibSVM2.5中为示例函数。

先随机打乱次序，然后根据n折的数目，留一份作为测试集，其他的作为训练集，做n次。
随机打乱次序使用的非标准的扑克洗牌的算法。(LibSVM2.5里面随机排序的结果很乱)

For example:

样本集被分为10份；第一次，将样本集的第2~10部分作为整体进行训练，得到一个模型，然后对样本集的第1部分进行预报，得到一个精度；第二次，将样本集的第1, 3~10作为整体训练，对第二部分进行预报，得到又一个精度，…。最后对10个精度做一下处理（方法很多，不逐一列出）。

```
int svm_get_nr_class(const svm_model *model)
```

获得样本类别数；本函数为典型的马后炮。

```
void svm_get_labels(const svm_model *model, int* label)
```

某类样本的标号（样本并不按编号排列，通过标号，可以循序访问样本集）。

```
double svm_get_svr_probability(const svm_model *model)
```

访问训练好的模型中的概率值。

```
void svm_predict_values(const svm_model *model, const svm_node *x, double* dec_values)
```

预测样本数据目标值；

如果是做分类问题，返回一大堆值，供后续的函数做决策；如果是回归问题，返回一个值。

其中 one-v-one 方法需要做 $n(n-1)/2$ 次，产生 $n(n-1)/2$ 个预报值。

```
double svm_predict(const svm_model *model, const svm_node *x)
```

预测，分类问题主要使用了One-to-One方法组织 $n*(n-1)/2$ 种方法。

如果是分类问题，对预测的 $n*(n-1)/2$ 个值，做投票处理，票数最高的是预报的类。

如果是 One-Class,根据预报值的符号，返回+1/-1

如果是回归问题，直接返回该 double 类型的值。

```
double svm_predict_probability(
```

```
    const svm_model *model, const svm_node *x, double *prob_estimates)
```

LibSVM2.6 新增函数

跳过。

```
int svm_save_model(const char *model_file_name, const svm_model *model)
```

```
svm_model *svm_load_model(const char *model_file_name)
```

```
void svm_destroy_model(svm_model* model)
```

以上 3 个函数均为 LibSVM2.5 示例程序中的函数，现成为 LibSVM2.6 的一部分。

看看名字就知道是干什么的了，不介绍了。

```
void svm_destroy_param(svm_parameter* param)
```

LibSVM2.6 新增函数

释放权重系数数组的内存。

```
//检查数据
```

```
const char *svm_check_parameter(const struct svm_problem *prob, const struct svm_parameter
*param);
```

该段代码检查参数的合理性。凡对 LibSVM 进行增加 SVC 类型和核函数，都必须修改该文件。

LibSVM2.5 在该部分代码会存在内存泄漏，LibSVM2.6 中已经修正。

其中需要注意的是， ν 的取值的范围，

$$\nu < \frac{n_{Min} \times 2}{n_{Max} + n_{Min}}$$

其中 n_{Max} 为样本数最多的类的样本数， n_{Min} 为样本数最少的类的样本数。

```
int svm_check_probability_model(const svm_model *model)
```

LibSVM2.6 新增函数

检查概率模型，主要是检查一些限制条件。

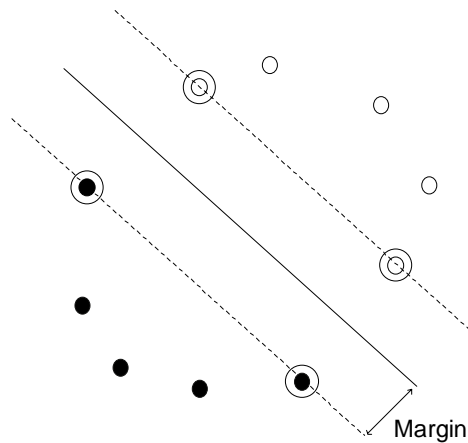


Figure 1: SVM separation of two data classes - SV points circled.

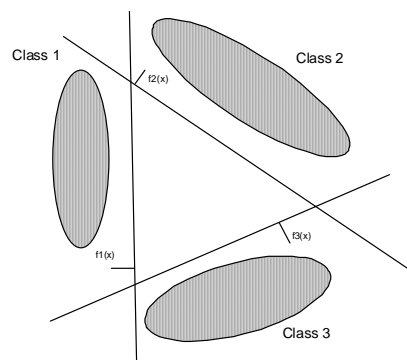


Figure 2: One-against-rest SVM separation of three data classes

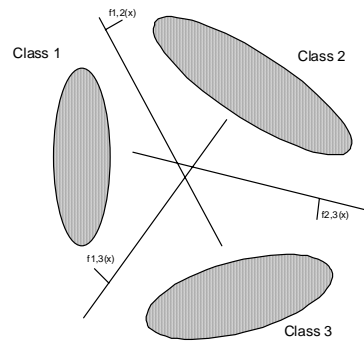


Figure 3: One-against-one SVM separation of three data classes

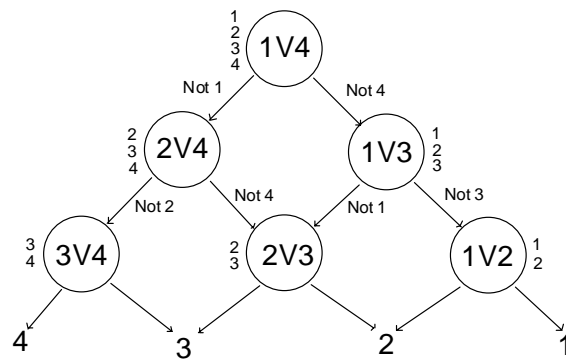


Figure 4: Decision DAG SVM

其他:

一、One-v-Rest 多类方法

<http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/1vsall/>

二、DDAG 多类方法

<http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/libsvm-2.3dag.zip>

参考文献:

[1]Chih-Chung Chang and Chih-Jen Lin, LIBSVM : a library for support vector machines, 2001.
Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>

[2]J. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Scholkopf, C. Burges, and A. Smola, editors, Advances in kernel methods: support vector learning. MIT Press, 1998.

[3] Sequential Minimal Optimization for SVM
<http://www.datalab.uci.edu/people/xge/svm/smo.pdf>

[4]Chang, C.-C. and C.-J. Lin (2001). Training ν -support vector classifiers: Theory and algorithms. Neural Computation 13 (9), 2119–2147.

[5]Chang, C.-C. and C.-J. Lin (2002). Training ν -support vector regression: Theory and algorithms. Neural Computation 14 (8), 1959–1977.