

```

-----classifier.h-----
#ifndef _CLASSIFIER_H_
#define _CLASSIFIER_H_

#include <vector>
#include <iostream>

class classifier {
public:
    classifier();
    virtual ~classifier();

    virtual classifier *clone() const = 0;

    virtual int classify(const std::vector<double> &x) const = 0;

    virtual void train(const std::vector<std::vector<double> > &x,
        const std::vector<int> &y, const std::vector<double> &weight) = 0;

    // just calls the above with the weights all set to 1.0
    virtual void train(const std::vector<std::vector<double> > &x,
        const std::vector<int> &y) {
        return train(x,y,std::vector<double>(x.size(),1.0));
    }

    // returns fraction of x's that the classify method gets correct
    virtual double test(const std::vector<std::vector<double> > &x,
        const std::vector<int> &y, const std::vector<double> &w) const
;

    // same as above, but without the weighting
    inline virtual double test(const std::vector<std::vector<double> > &x,
        const std::vector<int> &y) const {
        return test(x,y,std::vector<double>(x.size(),1.0));
    }

    // assumes that the classes are in the range of 0-(|c|-1)
    // scores in confmatrix[i][j] the number of times i was the
    // true class and j was predicted
    virtual void test(const std::vector<std::vector<double> > &x,
        const std::vector<int> &y,
        std::vector<std::vector<int> > &confmatrix) const;

    virtual void display(std::ostream &os) const {}
};

#endif

```

```

-----classifier.cpp-----
#include "classifier.h"

using namespace std;

classifier::classifier() {
}

classifier::~classifier() {
}

double classifier::test(const vector<vector<double> > &x,
    const vector<int> &y, const vector<double> &w) const {
    double ncorrect = 0;
    double sum = 0;

    for(int i=0;i<x.size();i++) {
        if (y[i] == classify(x[i])) ncorrect += w[i];
        sum += w[i];
    }
    return ncorrect/sum;
}

```

```

void classifier::test(const vector<vector<double> > &x,
    const vector<int> &y, vector<vector<int> > &confmatrix) const {

    int nc = 0;
    for(int i=0;i<y.size();i++)
        if (y[i]>nc) nc = y[i]+1;
    confmatrix.resize(nc);
    for(int i=0;i<nc;i++) {
        confmatrix[i].resize(nc);
        for(int j=0;j<nc;j++)
            confmatrix[i][j] = 0;
    }
    for(int i=0;i<x.size();i++)
        confmatrix[y[i]][classify(x[i])]++;
}

```

```

-----simpleclass.h-----
#ifndef _SIMPLECLASS_H_
#define _SIMPLECLASS_H_

#include "classifier.h"
#include <vector>

class simpleclass : public classifier {
public:
    simpleclass();
    virtual ~simpleclass();

    virtual simpleclass *clone() const { return new simpleclass(*this); }

    virtual int classify(const std::vector<double> &x) const;
    virtual void train(const std::vector<std::vector<double> > &x,
                      const std::vector<int> &y, const std::vector<double> &weights)
;

    virtual void train(const std::vector<std::vector<double> > &x,
                      const std::vector<int> &y) {
        return train(x,y,std::vector<double>(x.size(),1.0));
    }

private:
    void evalfeature(const std::vector<std::vector<double> > &x,
                    const std::vector<int> &y, const std::vector<double> &w,
                    int fnum, double &split, bool &dir, double &loss);

    int featurenum;
    double splitvalue;
    bool positiveup;
};

#endif

-----simpleclass.cpp-----
#include "simpleclass.h"

using namespace std;

simpleclass::simpleclass() : classifier() {
    featurenum = 0; splitvalue = 0; positiveup = true;
}

simpleclass::~simpleclass() {
}

int simpleclass::classify(const vector<double> &x) const {
    return ((x[featurenum]>=splitvalue)==positiveup) ? 1 : -1;
}

void simpleclass::train(const vector<vector<double> > &x,
                       const vector<int> &y, const vector<double> &weights) {
    featurenum = 0;
    double bestloss;
    evalfeature(x,y,weights,0,splitvalue,positiveup,bestloss);
    for(int f=1;f<x[0].size();f++) {
        double currloss,currsplit;
        bool currpup;
        evalfeature(x,y,weights,f,currsplit,currpup,currloss);
        if (currloss<bestloss) {
            bestloss = currloss;
            featurenum = f;
            positiveup = currpup;
            splitvalue = currsplit;
        }
    }
}

```

```

class extype {
public:
    double x;
    double w;
    int y;

    static int comp(const void *a, const void *b) {
        const extype *aa = (const extype *)a;
        const extype *bb = (const extype *)b;
        if (aa->x<bb->x) return -1;
        if (aa->x>bb->x) return 1;
        return 0;
    }
};

void simpleclass::evalfeature(const vector<vector<double> > &x,
                             const vector<int> &y, const vector<double> &w,
                             int fnum, double &split, bool &dir, double &loss) {
    extype *data = new extype[x.size()];
    for(int i=0;i<x.size();i++) {
        data[i].x = x[i][fnum];
        data[i].w = w[i];
        data[i].y = y[i];
    }
    qsort(data,x.size(),sizeof(extype),extype::comp);

    vector<double> splitpts;
    vector<double> npos,nneg;
    double ttlneg=0,ttlpos=0;

    for(int i=0;i<x.size();i++) {
        if (splitpts.size()==0
            || splitpts.back()!=data[i].x) {
            splitpts.push_back(data[i].x);
            npos.push_back(0.0);
            nneg.push_back(0.0);
        }
        if (data[i].y < 0) {
            ttlneg += data[i].w;
            nneg.back() += data[i].w;
        }
        else {
            ttlpos += data[i].w;
            npos.back() += data[i].w;
        }
    }

    split = splitpts[0]-1.0;
    if (ttlneg>ttlpos) {
        loss = ttlpos; dir = false;
    }
    else {
        loss = ttlneg; dir = true;
    }

    double pleft=0,pright=ttlpos,nleft=0,nright=ttlneg;
    for(int i=0;i<splitpts.size();i++) {
        pleft += npos[i]; pright -= npos[i];
        nleft += nneg[i]; nright -= nneg[i];
        if (pleft+nright > nleft+pright) {
            if (nleft+pright < loss) {
                loss = nleft+pright;
                dir = false;
                if (i<splitpts.size()-1)
                    split = (splitpts[i]+splitpts[i+1])/2;
                else split = splitpts[i]+1.0;
            }
        }
        else {
            if (pleft+nright < loss) {
                loss = pleft+nright;
                dir = true;
            }
        }
    }
}

```

```

        if (i<splitpts.size()-1)
            split = (splitpts[i]+splitpts[i+1])/2;
        else split = splitpts[i]+1.0;
    }
}
delete []data;
}
}

```

```

-----multiclass.cpp-----
#ifndef _MULTICLASS_H_
#define _MULTICLASS_H_

#include "classifier.h"
#include <vector>

class multiclass : public classifier {
public:
    multiclass(const classifier *base);
    multiclass(const multiclass &mc);
    virtual ~multiclass();

    virtual multiclass *clone() const { return new multiclass(*this); }

    virtual int classify(const std::vector<double> &x) const;
    virtual void train(const std::vector<std::vector<double> > &x,
        const std::vector<int> &y, const std::vector<double> &weights);
;
    virtual void train(const std::vector<std::vector<double> > &x,
        const std::vector<int> &y) {
        return train(x,y,std::vector<double>(x.size(),1.0));
    }

private:
    void dealloc();

    classifier *b;
    std::vector<int> classes;
    classifier **c;
};

#endif

-----multiclass.h-----
#include "multiclass.h"
#include "dataset.h"

using namespace std;

multiclass::multiclass(const classifier *base) : classifier() {
    b = base->clone();
    c = NULL;
}

multiclass::multiclass(const multiclass &mc) : classifier(mc) {
    b = mc.b->clone();
    if (mc.c==NULL) c = NULL;
    else {
        classes = mc.classes;
        int ncl = classes.size()*(classes.size()-1)/2;
        c = new classifier*[ncl];
        for(int i=0;i<ncl;i++)
            c[i] = mc.c[i]->clone();
    }
}

multiclass::~multiclass() {
    delete b;
    if (c!=NULL) dealloc();
}

void multiclass::dealloc() {
    int ncl = classes.size()*(classes.size()-1)/2;
    for(int i=0;i<ncl;i++)
        delete c[i];
    delete []c;
}

```

```

int multiclass::classify(const vector<double> &x) const {
    vector<int> votes(classes.size(),0);

    for(int i=0,ii=0;i<classes.size();i++)
        for(int j=i+1;j<classes.size();j++,ii++) {
            int y = c[ii]->classify(x);
            if (y<0) votes[classes[j]]++;
            else votes[classes[i]]++;
        }
    int mv = votes[0];
    int mc = 0;
    for(int i=1;i<votes.size();i++)
        if (votes[i]>mv) { mv = votes[i]; mc = i; }
    return mc;
}

static int findint(const vector<int> &v, int tofind) {
    for(int i=0;i<v.size();i++)
        if (v[i]==tofind) return i;
    return -1;
}

void multiclass::train(const vector<vector<double> > &x,
    const vector<int> &y, const vector<double> &weights) {

    if (c!=NULL) dealloc();
    classes.resize(0);

    for(int i=0;i<y.size();i++)
        if (findint(classes,y[i])==-1) classes.push_back(y[i]);
    int ncl = classes.size()*(classes.size()-1)/2;
    c = new classifier*[ncl];
    for(int i=0,ii=0;i<classes.size();i++)
        for(int j=i+1;j<classes.size();j++,ii++) {
            vector<double> w2;
            vector<int> y2;
            vector<vector<double> > x2;
            // like to use splitdata here, but I have to split the
            // weights too... humph.
            for(int k=0;k<y.size();k++) {
                if (y[k]==classes[i]) {
                    x2.push_back(x[k]);
                    y2.push_back(1);
                    w2.push_back(weights[k]);
                } else if (y[k]==classes[j]) {
                    x2.push_back(x[k]);
                    y2.push_back(-1);
                    w2.push_back(weights[k]);
                }
            }
            c[ii] = b->clone();
            c[ii]->train(x2,y2,w2);
        }
}

```

```

-----boostclass.h-----
#ifndef _BOOSTCLASS_H_
#define _BOOSTCLASS_H_

#include "classifier.h"

class boostclass : public classifier {
public:
    boostclass(const classifier *base);
    boostclass(const boostclass &bc);
    virtual ~boostclass();

    virtual boostclass *clone() const { return new boostclass(*this); }

    virtual int classify(const std::vector<double> &x) const;
    virtual void train(const std::vector<std::vector<double> > &x,
        const std::vector<int> &y, const std::vector<double> &weights)
    ;

    virtual void train(const std::vector<std::vector<double> > &x,
        const std::vector<int> &y) {
        return train(x,y,std::vector<double>(x.size(),1.0));
    }

private:
    classifier *b;

    std::vector<classifier *> c;
    std::vector<double> w;
};

-----boostclass.cpp-----
#include "boostclass.h"
#include <stdlib.h>
#include <iostream>
#include <math.h>

using namespace std;

boostclass::boostclass(const classifier *base) : classifier() {
    b = base->clone();
}

boostclass::boostclass(const boostclass &bc) : classifier(bc) {
    b = bc.b->clone();
    w = bc.w;
    for(int i=0;i<bc.c.size();i++)
        c.push_back(bc.c[i]->clone());
}

boostclass::~boostclass() {
    delete b;
    for(int i=0;i<c.size();i++) delete c[i];
}

int boostclass::classify(const vector<double> &x) const {
    double wpos=0.0,wneg=0.0;
    for(int i=0;i<c.size();i++)
        if (c[i]->classify(x)<0) wneg += w[i];
        else wpos += w[i];
    return wpos>=wneg ? 1 : -1;
}

void boostclass::train(const vector<vector<double> > &x,
    const vector<int> &y, const vector<double> &weights) {
    for(int i=0;i<c.size();i++) delete c[i];
    c.clear();
    w.clear();
}

```

```

vector<double> expm(weights);
for(int i=0;i<100;i++) {
    classifier *nextc = b->clone();
    nextc->train(x,y,expm);
    double v = nextc->test(x,y,expm);
    if (v<=0.5) {
        delete nextc;
        break;
    }
    if (v>=1.0) {
        if (i==0) {
            c.push_back(nextc);
            w.push_back(1.0);
        } else {
            // something a little weird -- we should have
            // gotten this before
            delete nextc;
        }
        return;
    }
    c.push_back(nextc);
    double beta = v/(1.0-v);
    w.push_back(log(beta));
    double expmsum = 0.0;
    for(int i=0;i<x.size();i++) {
        if (y[i]==nextc->classify(x[i])) expm[i] /= beta;
        expmsum += expm[i];
    }
    // good to keep it normalized
    for(int i=0;i<x.size();i++)
        expm[i] /= expmsum;
}
}

```

```

-----runexperiments.cpp-----
#include "dataset.h"
#include <stdio.h>
#include <stdlib.h>
#include "simpleclass.h"
#include "multiclass.h"
#include "boostclass.h"
#include <iomanip>

using namespace std;

int main(int argc, char **argv) {
    vector<vector<double> > x,xx,xx2,x2;
    vector<int> y,yy,yy2,y2;

    loaddata(argv[1],xx,yy);
    loaddata(argv[2],xx2,yy2);

    simpleclass basec;

    // swap the comments on the next two lines to go
    // from simple to complex or back
    boostclass *c = new boostclass(&basec);
    //simpleclass *c = new simpleclass();
    for(int i=0;i<10;i++) {
        //cout << i << ": ";
        for(int j=0;j<10;j++) {
            if (i==j) {
                cout << "          ";
                continue;
            }
            x.clear();
            y.clear();
            x2.clear();
            y2.clear();
            splitdata(xx,yy,x,y,i,j);
            splitdata(xx2,yy2,x2,y2,i,j);
            c->train(x,y);
            cout << setw(4);
            cout << setiosflags(ios::fixed) << setprecision(2);
            cout << c->test(x2,y2) << ' ';
            cout << "(" << c->test(x,y) << ") ";
        }
        cout << endl;
    }

    cout << "total: " << endl;
    // swap the comments on the next two lines to go
    // from simple to complex or back
    multiclass mc(new boostclass(&basec));
    //multiclass mc(&basec);
    mc.train(xx,yy);
    cout << mc.test(xx,yy) << endl;
    cout << mc.test(xx2,yy2) << endl;
    vector<vector<int> > conf;
    mc.test(xx2,yy2,conf);
    for(int i=0;i<conf.size();i++) {
        for(int j=0;j<conf[i].size();j++)
            cout << conf[i][j] << ' ';
        cout << endl;
    }
}

```