**SOURCE CODE**

**TABLE OF CONTENTS**

[User Interface 3](#_Toc508133407)

[MyForm.h 3](#_Toc508133408)

[ColorTrap.cpp 8](#_Toc508133409)

[Segmentation.h 9](#_Toc508133410)

[svm.h 10](#_Toc508133411)

[Source Files 12](#_Toc508133412)

[DetermineConnected.cpp 12](#_Toc508133413)

[FindPeakForNormalNumber.cpp 24](#_Toc508133414)

[ImagePreprocess.cpp 31](#_Toc508133415)

[MyForm.cpp 35](#_Toc508133416)

[OvOrecognition.cpp 36](#_Toc508133417)

[Segmentation.cpp 39](#_Toc508133418)

[SplitConnected\_Dropfall.cpp 89](#_Toc508133419)

[svm-predict.cpp 99](#_Toc508133420)

[svm.cpp 105](#_Toc508133421)

[toCSV.cpp 209](#_Toc508133422)

[toSVMformat.cpp 210](#_Toc508133423)

# User Interface

## MyForm.h

#pragma once

#include <stdio.h>

#include <iostream>

#include <Windows.h>

#include <string.h>

#include <msclr\marshal\_cppstd.h>

#include <sstream>

#include "Segmentation.h"

std::string filePath = "";

namespace Sample {

using namespace System;

using namespace System::ComponentModel;

using namespace System::Collections;

using namespace System::Windows::Forms;

using namespace System::Data;

using namespace System::Drawing;

/// <summary>

/// Summary for MyForm

/// </summary>

public ref class MyForm : public System::Windows::Forms::Form{

public:

MyForm(void) {

InitializeComponent();

}

protected:

/// <summary>

/// Clean up any resources being used.

/// </summary>

~MyForm(){

if (components){

delete components;

}

}

private: System::Windows::Forms::PictureBox^ pictureBox1;

protected:

private: System::Windows::Forms::TextBox^ textBox1;

private: System::Windows::Forms::Button^ button1;

private: System::Windows::Forms::Button^ button2;

private: System::Windows::Forms::Button^ button3;

private:

/// <summary>

/// Required designer variable.

/// </summary>

System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code

/// <summary>

/// Required method for Designer support - do not modify

/// the contents of this method with the code editor.

/// </summary>

void InitializeComponent(void){

this->pictureBox1 = (gcnew System::Windows::Forms::PictureBox());

this->textBox1 = (gcnew System::Windows::Forms::TextBox());

this->button1 = (gcnew System::Windows::Forms::Button());

this->button2 = (gcnew System::Windows::Forms::Button());

this->button3 = (gcnew System::Windows::Forms::Button());

(cli::safe\_cast<System::ComponentModel::ISupportInitialize^ >

(this->pictureBox1))->BeginInit();

this->SuspendLayout();

//

// pictureBox1

//

this->pictureBox1->Location = System::Drawing::Point(49, 24);

this->pictureBox1->Name = L"pictureBox1";

this->pictureBox1->Size = System::Drawing::Size(610, 172);

this->pictureBox1->TabIndex = 0;

this->pictureBox1->TabStop = false;

//

// textBox1

//

this->textBox1->Location = System::Drawing::Point(49, 240);

this->textBox1->Name = L"textBox1";

this->textBox1->Size = System::Drawing::Size(100, 20);

this->textBox1->TabIndex = 1;

this->textBox1->TextChanged += gcnew

System::EventHandler(this, &MyForm::textBox1\_TextChanged);

//

// button1

//

this->button1->Location = System::Drawing::Point(168, 237);

this->button1->Name = L"button1";

this->button1->Size = System::Drawing::Size(75, 23);

this->button1->TabIndex = 2;

this->button1->Text = L"Browse...";

this->button1->UseVisualStyleBackColor = true;

this->button1->Click += gcnew

System::EventHandler(this, &MyForm::button1\_Click);

//

// button2

//

this->button2->Location = System::Drawing::Point(49, 277);

this->button2->Name = L"button2";

this->button2->Size = System::Drawing::Size(75, 23);

this->button2->TabIndex = 3;

this->button2->Text = L"OK";

this->button2->UseVisualStyleBackColor = true;

this->button2->Click += gcnew

System::EventHandler(this, &MyForm::button2\_Click);

//

// button3

//

this->button3->Location = System::Drawing::Point(168, 277);

this->button3->Name = L"button3";

this->button3->Size = System::Drawing::Size(75, 23);

this->button3->TabIndex = 4;

this->button3->Text = L"Cancel";

this->button3->UseVisualStyleBackColor = true;

this->button3->Click += gcnew

System::EventHandler(this, &MyForm::button3\_Click);

//

// MyForm

//

this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);

this->AutoScaleMode =

System::Windows::Forms::AutoScaleMode::Font;

this->ClientSize = System::Drawing::Size(839, 364);

this->Controls->Add(this->button3);

this->Controls->Add(this->button2);

this->Controls->Add(this->button1);

this->Controls->Add(this->textBox1);

this->Controls->Add(this->pictureBox1);

this->Name = L"MyForm";

this->Text = L"MyForm";

this->Load += gcnew System::EventHandler(this,

&MyForm::MyForm\_Load);

(cli::safe\_cast<System::ComponentModel::ISupportInitialize^

>(this->pictureBox1))->EndInit();

this->ResumeLayout(false);

this->PerformLayout();

}

private: System::Void button1\_Click(System::Object^

sender, System::EventArgs^ e) {

OpenFileDialog^ openFileDialog1 = gcnew OpenFileDialog;

openFileDialog1 -> InitialDirectory = "C:\\";

openFileDialog1 -> Filter = "Picture Files (\*.jpg, \*.png)\0|\*.jpg;\*.png\0";

openFileDialog1 -> FilterIndex = 1;

openFileDialog1 -> RestoreDirectory = true;

// Dialog box will appear for the selection of photos

if (openFileDialog1 -> ShowDialog() ==

System::Windows::Forms::DialogResult::OK){

pictureBox1 -> Image = Image::FromFile (openFileDialog1-

>FileName); // Image will be shown in the picture box

// The file path will be shown in the text box.

textBox1 -> Text = openFileDialog1 -> FileName;

// Conversion from System String to Standard String

msclr::interop::marshal\_context context;

std::string standardString = context.marshal\_as<std::string>

( textBox1 -> Text);

filePath = standardString;

}

}

private: System::Void button2\_Click(System::Object^

sender, System::EventArgs^ e) {

int operand1;

int reg = segmentation(filePath);

MyForm::Close();

}

private: System::Void button3\_Click(System::Object^

sender, System::EventArgs^ e) {

MyForm::Close();

}

};

}

## ColorTrap.cpp

#include<opencv2/core/core.hpp>

#include<opencv2/highgui/highgui.hpp>

#include<opencv2/imgproc/imgproc.hpp>

using namespace std;

using namespace cv;

Mat colorTrap(Mat image) {

int darkBackground = 0;

int lightBackground = 0;

for (int x = 0; x < image.cols; x++) {

for (int y = 0; y < image.rows; y++) {

uchar color = image.at<uchar>(Point(x, y));

if (color > 150) {

lightBackground++;

}

else {

darkBackground++;

}

}

}

if (lightBackground > darkBackground) {

threshold(image, image, 70, 255, CV\_THRESH\_BINARY\_INV);

}

return image;

}

## Segmentation.h

#include<opencv2/core/core.hpp>

#include<opencv2/highgui/highgui.hpp>

#include<opencv2/imgproc/imgproc.hpp>

using namespace cv;

#define SEGMENTATION\_H

#define DETERMINECONNECTED\_H

#define FINDPEAKFORNORMALNUMBER\_H

#define SPLITCONNECTED\_DROPFALL\_H

#define TOCSV\_H

#define TOSVMFORMAT\_H

#define SVMPREDICT\_H

#define colorTrap\_H

#define ImageProcess\_H

#define OVORECOGNITION\_H

#define RECOGNITION\_H

Mat SplitImage(Mat); //drop fall from bottom

int segmentation(std::string);

bool determineConnected(Mat);

int topPeak(Mat);

int bottomPeak(Mat);

int getCSV(Mat);

int toSVMformat();

int svmPredict();

Mat colorTrap(Mat);

Mat ImageTrap(Mat);

char ovo(Mat);

Mat splitConnected(Mat); //drop fall from top

## svm.h

#ifndef \_LIBSVM\_H

#define \_LIBSVM\_H

#define LIBSVM\_VERSION 312

#ifdef \_\_cplusplus

extern "C" {

#endif

extern int libsvm\_version;

struct svm\_node

{

int index;

double value;

};

struct svm\_problem

{

int l;

double \*y;

struct svm\_node \*\*x;

};

enum { C\_SVC, NU\_SVC, ONE\_CLASS, EPSILON\_SVR, NU\_SVR };

/\* svm\_type \*/

enum { LINEAR, POLY, RBF, SIGMOID, PRECOMPUTED }; /\* kernel\_type \*/

struct svm\_parameter

{

int svm\_type;

int kernel\_type;

int 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 \*/

};

//

// svm\_model

//

struct svm\_model

{

struct svm\_parameter param; /\* parameter \*/

int nr\_class; /\* number of classes, = 2 in regression/one class svm \*/

int l; /\* total #SV \*/

struct svm\_node \*\*SV; /\* SVs (SV[l]) \*/

double \*\*sv\_coef; /\* coefficients for SVs in decision functions (sv\_coef[k-1][l]) \*/

double \*rho; /\* constants in decision functions (rho[k\*(k-1)/2]) \*/

double \*probA; /\* pariwise probability information \*/

double \*probB;

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 \*/

int \*label; /\* label of each class (label[k]) \*/

int \*nSV; /\* number of SVs for each class (nSV[k]) \*/

/\* nSV[0] + nSV[1] + ... + nSV[k-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 \*/

};

struct svm\_model \*svm\_train(const struct svm\_problem \*prob, const struct svm\_parameter \*param);

void svm\_cross\_validation(const struct svm\_problem \*prob, const struct svm\_parameter \*param, int nr\_fold, double \*target);

int svm\_save\_model(const char \*model\_file\_name, const struct svm\_model \*model);

struct svm\_model \*svm\_load\_model(const char \*model\_file\_name);

int svm\_get\_svm\_type(const struct svm\_model \*model);

int svm\_get\_nr\_class(const struct svm\_model \*model);

void svm\_get\_labels(const struct svm\_model \*model, int \*label);

double svm\_get\_svr\_probability(const struct svm\_model \*model);

double svm\_predict\_values(const struct svm\_model \*model, const struct svm\_node \*x, double\* dec\_values);

double svm\_predict(const struct svm\_model \*model, const struct svm\_node \*x);

double svm\_predict\_probability(const struct svm\_model \*model, const struct svm\_node \*x, double\* prob\_estimates);

void svm\_free\_model\_content(struct svm\_model \*model\_ptr);

void svm\_free\_and\_destroy\_model(struct svm\_model \*\*model\_ptr\_ptr);

void svm\_destroy\_param(struct svm\_parameter \*param);

const char \*svm\_check\_parameter(const struct svm\_problem \*prob, const struct svm\_parameter \*param);

int svm\_check\_probability\_model(const struct svm\_model \*model);

void svm\_set\_print\_string\_function(void(\*print\_func)(const char \*));

#ifdef \_\_cplusplus

}

#endif

#endif /\* \_LIBSVM\_H \*/

# Source Files

## DetermineConnected.cpp

#include<opencv2/core/core.hpp>

#include<opencv2/highgui/highgui.hpp>

#include<opencv2/imgproc/imgproc.hpp>

#include<opencv2/ml/ml.hpp>

#include"Segmentation.h"

#include<iostream>

#include<vector>

using namespace std;

using namespace cv;

const int MIN\_CONTOUR\_AREA = 100;

const int RESIZED\_IMAGE\_WIDTH = 20;

const int RESIZED\_IMAGE\_HEIGHT = 20;

Mat peakHistogram(Mat& img, int t)

{

//col or row histogram

int sz = (t) ? img.rows : img.cols;

Mat mhist = Mat::zeros(1, sz, CV\_8U);

//count nonzero value and check max V

int max = -100;

for (int j = 0; j < sz; ++j)

{

Mat data = (t) ? img.row(j) : img.col(j);

int v = countNonZero(data);

mhist.at< unsigned char >(j) = v;

if (v > max)

max = v;

}

Mat histo;

int width, height;

if (t)

{

width = max;

height = sz;

histo = Mat::zeros(Size(width, height), CV\_8U);

for (int i = 0; i < height; ++i)

{

for (int j = 0; j < mhist.at< unsigned char >(i); ++j)

histo.at< unsigned char >(i, j) = 255;

}

transpose(histo, histo);

flip(histo, histo, 0);

}

else {

width = sz;

height = max;

histo = Mat::zeros(Size(width, height), CV\_8U);

for (int i = 0; i< width; ++i)

{

for (int j = 0; j< mhist.at< unsigned char >(i); ++j)

histo.at< unsigned char >(max - j - 1, i) = 255;

}

}

return histo;

}

// PEAK INFO

class peakInfo {

public:

int locationX;

int locationY;

void cleanInfo() {

locationX = 0;

locationY = 0;

}

};

void findPeaks(Mat image, vector<peakInfo>& info) {

peakInfo peak;

int pointer = 0;

int greatestVal = 0;

int prevX = 0;

int prevY = 0;

bool uphill = false, downhill = false;

for (int x = 0; x < image.cols; x++) {

for (int y = 0; y < image.rows; y++) {

uchar color = image.at<uchar>(Point(x, y));

if (color == 0) {

if (pointer < y) {

pointer = y;

}

}

}

if (greatestVal > pointer) {

if (uphill == true) {

peak.locationX = prevX;

peak.locationY = prevY;

info.push\_back(peak);

uphill = false;

}

else {

downhill = true;

uphill = false;

}

}

else if (greatestVal < pointer) {

uphill = true;

downhill = false;

}

prevX = x;

prevY = pointer;

greatestVal = pointer;

pointer = 0;

}

prevX = 0;

prevY = 0;

uphill = false;

downhill = false;

}

int countNoOfPixels(Mat img, int& allPixels) {

int numberOfPixels = 0;

allPixels = 0;;

for (int x = 0; x < img.cols; x++) {

for (int y = 0; y < img.rows; y++) {

allPixels++;

uchar color = img.at<uchar>(Point(x, y));

if (color == 0) {

numberOfPixels++;

}

}

}

return numberOfPixels;

}

void findSecondAndLargestPeak(vector<peakInfo>& info, Mat& hHisto, int& largest, int& largestX, int& secondLargest, int& secondLargestX) {

largest = 0, secondLargest = 0,

largestX = 0; secondLargestX = 0;

for (int x = 0; x < info.size(); x++) {

if (info[x].locationY > secondLargest) {

secondLargest = info[x].locationY;

secondLargestX = info[x].locationX;

}

if (info[x].locationY > largest) {

secondLargest = largest;

secondLargestX = largestX;

largest = info[x].locationY;

largestX = info[x].locationX;

}

}

circle(hHisto, Point(largestX, largest), 3, Scalar(100, 30, 20), CV\_FILLED, 8, 0);

circle(hHisto, Point(secondLargestX, secondLargest), 2, Scalar(100, 2, 0),

CV\_FILLED, 8, 0);

}

int findConnectedThroughPeaksVertical(Mat matROI) {

std::vector<peakInfo> infoVertical;

int numOfPeaks = 0;

Mat img = matROI.clone();

Mat vHisto = peakHistogram(img, 1);

blur(vHisto, vHisto, Size(9, 9));

threshold(vHisto, vHisto, 50, 255, CV\_THRESH\_BINARY);

int allPixels = 0;

int numberOfPixels = countNoOfPixels(vHisto, allPixels);

int percent = (allPixels \* 2) / 100;

if (numberOfPixels > percent) {

findPeaks(vHisto, infoVertical);

int largest = 0, secondLargest = 0;

int largestX = 0; int secondLargestX = 0;

findSecondAndLargestPeak(infoVertical, vHisto, largest, largestX, secondLargest,

secondLargestX);

if (secondLargest != 0) {

if (secondLargest < (matROI.cols \* 5) / 100) {

numOfPeaks = 1;

}

else {

numOfPeaks = 2;

}

}

else {

numOfPeaks = 1;

}

}

else {

numOfPeaks = 0;

}

for (int i = 0; i < infoVertical.size(); i++) {

infoVertical[i].cleanInfo();

}

return numOfPeaks;

}

int findConnectedThroughPeaksHorizontal(Mat matROI) {

std::vector<peakInfo> info;

int numOfPeaks = 0;

Mat img = matROI.clone();

Mat hHisto = peakHistogram(img, 0);

blur(hHisto, hHisto, Size(9, 9));

threshold(hHisto, hHisto, 50, 255, CV\_THRESH\_BINARY);

int allPixels = 0;

int numberOfPixels = countNoOfPixels(hHisto, allPixels);

int percent = (allPixels \* 3) / 100;

if (numberOfPixels > percent) {

findPeaks(hHisto, info);

int largest = 0, secondLargest = 0;

int largestX = 0; int secondLargestX = 0;

findSecondAndLargestPeak(info, hHisto, largest, largestX, secondLargest,

secondLargestX);

if (secondLargest != 0) {

if (secondLargest < (matROI.cols / 2 \* 5) / 100) {

numOfPeaks = 1;

}

else {

numOfPeaks = 2;

}

}

else {

numOfPeaks = 1;

}

}

else {

numOfPeaks = 0;

}

for (int i = 0; i < info.size(); i++) {

info[i].cleanInfo();

}

return numOfPeaks;

}

bool findConnectedThoughSize(Mat matROI) {

bool connected = false;

int count = 0;

int half = matROI.cols / 2;

int NoPixelsLeft = 0, NoPixelsRight = 0;

for (int x = 0; x < matROI.cols; x++) {

for (int y = 0; y < matROI.rows; y++) {

uchar color = matROI.at<uchar>(Point(x, y));

if (color == 255) {

if (count <= half) {

NoPixelsLeft++;

}

else {

NoPixelsRight++;

}

}

}

count++;

}

int Percent = (NoPixelsLeft + NoPixelsRight) \* 50 / 100;

if (NoPixelsLeft > NoPixelsRight) {

if (NoPixelsLeft-NoPixelsRight<=Percent && matROI.cols > matROI.rows+5) {

if (NoPixelsLeft - NoPixelsRight <= Percent) {

connected = true;

}

else {

connected = false;

}

}

else {

connected = false;

}

}

else {

if (NoPixelsRight - NoPixelsLeft <= Percent && matROI.cols > matROI.rows+5) {

if (NoPixelsRight - NoPixelsLeft <= Percent) {

connected = true;

}

else {

connected = false;

}

}

else {

connected = false;

}

}

return connected;

}

void findPeakNormalNumber(Mat matROI, int& top, int& bottom) {

int b = bottomPeak(matROI);

int t = topPeak(matROI);

top = t;

bottom = b;

}

bool determineConnected(Mat matROI) {

bool connected = false;

int top = 0, bottom = 0;

findPeakNormalNumber(matROI, top, bottom);

int numOfPeaksH = findConnectedThroughPeaksHorizontal(matROI);

int numOfPeaksV = findConnectedThroughPeaksVertical(matROI);

bool bySize = findConnectedThoughSize(matROI);

if ((top + bottom) == 4 && (numOfPeaksH + numOfPeaksV) >= 3 && bySize) {

connected = true;

}

else {

connected = false;

}

return connected;

}

## FindPeakForNormalNumber.cpp

#include<opencv2/core/core.hpp>

#include<opencv2/highgui/highgui.hpp>

#include<opencv2/imgproc/imgproc.hpp>

#include<opencv2/ml/ml.hpp>

#include<iostream>

#include<vector>

class labels{

public:

int predict;

};

using namespace std;

using namespace cv;

class infoPeaks {

public:

int locationX;

int locationY;

void cleanInfo() {

locationX = 0;

locationY = 0;

}

};

int topPeak(Mat matROI) {

int top = 0;

vector<infoPeaks> topinfo;

infoPeaks infopeaks;

int pointer = matROI.rows;

int leastVal = matROI.rows;

int prevX = 0;

int prevY = 0;

bool uphill = false, downhill = false;

for (int x = 0; x < matROI.cols; x++) {

for (int y = 0; y < matROI.rows; y++) {

uchar color = matROI.at<uchar>(Point(x, y));

if (color == 0) {

if (pointer > y) {

pointer = y;

}

}

}

if (leastVal < pointer) {

if (uphill == true) {

infopeaks.locationX = prevX;

infopeaks.locationY = prevY;

topinfo.push\_back(infopeaks);

uphill = false;

}

else {

downhill = true;

uphill = false;

}

}

else if (leastVal > pointer) {

uphill = true;

downhill = false;

}

prevX = x;

prevY = pointer;

leastVal = pointer;

pointer = matROI.rows;

}

prevX = 0;

prevY = 0;

uphill = false;

downhill = false;

int largest = matROI.rows, secondLargest = matROI.rows;

int largestX = 0, secondLargestX = 0;

for (int x = 0; x < topinfo.size(); x++) {

if (topinfo[x].locationY < secondLargest) {

secondLargest = topinfo[x].locationY;

secondLargestX = topinfo[x].locationX;

}

if (topinfo[x].locationY < largest) {

secondLargest = largest;

secondLargestX = largestX;

largest = topinfo[x].locationY;

largestX = topinfo[x].locationX;

}

}

if (secondLargest >(matROI.rows / 2) - 10) {

secondLargest = matROI.rows;

secondLargestX = matROI.rows;

}

if (secondLargest == matROI.rows) {

top = 1;

}

else {

top = 2;

}

circle(matROI, Point(largestX, largest), 3, Scalar(100, 30, 20), CV\_FILLED, 8, 0);

circle(matROI, Point(secondLargestX, secondLargest), 2, Scalar(100, 2, 0), CV\_FILLED, 8, 0);

return top;

}

int bottomPeak(Mat matROI) {

int bottom = 0;

vector<infoPeaks> info;

infoPeaks peaks;

int pointer = 0;

int greatestVal = 0;

int prevX = 0;

int prevY = 0;

bool uphill = false, downhill = false;

for (int x = 0; x < matROI.cols; x++) {

for (int y = 0; y < matROI.rows; y++) {

uchar color = matROI.at<uchar>(Point(x, y));

if (color == 0) {

if (pointer < y) {

pointer = y;

}

}

}

if (greatestVal > pointer) {

if (uphill == true) {

peaks.locationX = prevX;

peaks.locationY = prevY;

info.push\_back(peaks);

uphill = false;

}

else {

downhill = true;

uphill = false;

}

}

else if (greatestVal < pointer) {

uphill = true;

downhill = false;

}

prevX = x;

prevY = pointer;

greatestVal = pointer;

pointer = 0;

}

prevX = 0;

prevY = 0;

uphill = false;

downhill = false;

int largest = 0, secondLargest = 0;

int largestX = 0, secondLargestX = 0;

for (int x = 0; x < info.size(); x++) {

if (info[x].locationY > secondLargest) {

secondLargest = info[x].locationY;

secondLargestX = info[x].locationX;

}

if (info[x].locationY > largest) {

secondLargest = largest;

secondLargestX = largestX;

largest = info[x].locationY;

largestX = info[x].locationX;

}

}

if (secondLargest < (matROI.rows / 2) + 10) {

secondLargest = 0;

secondLargestX = 0;

}

if (secondLargest == 0) {

bottom = 1;

}

else {

bottom = 2;

}

circle(matROI, Point(largestX, largest), 3, Scalar(100, 30, 20), CV\_FILLED, 8, 0);

circle(matROI, Point(secondLargestX, secondLargest), 2, Scalar(100, 2, 0), CV\_FILLED, 8, 0);

return bottom;

}

## ImagePreprocess.cpp

#include<opencv2/core/core.hpp>

#include<opencv2/highgui/highgui.hpp>

#include<opencv2/imgproc/imgproc.hpp>

#include<opencv2/ml/ml.hpp>

#include"Segmentation.h"

#include<iostream>

#include<vector>

using namespace std;

using namespace cv;

const int MIN\_CONTOUR\_AREA = 40;

const int RESIZED\_IMAGE\_WIDTH = 20;

const int RESIZED\_IMAGE\_HEIGHT = 20;

Mat ImageTrap(Mat image) {

Mat matThreshCopy; // create a copy of the image before finding contours

std::vector<std::vector<Point> > ptContours; // declare contours vector

std::vector<Vec4i> v4iHierarchy; // declare contours hierarchy

Mat matTrainingImages; // we have to declare a single Mat

if (image.cols > 1000 && (image.cols < 2000 )) {

resize(image, image, Size(image.cols / 2, image.rows / 2));

}

if (image.cols > 2000 && (image.cols < 3000)) {

resize(image, image, Size(image.cols / 3, image.rows / 2));

}

if (image.cols > 3000 && (image.cols < 4000)) {

resize(image, image, Size(image.cols / 5, image.rows / 5));

}

// Grayscale

cvtColor(image, image, CV\_BGR2GRAY);

int rows = image.rows;

int cols = image.cols;

Size s = image.size();

rows = s.height;

cols = s.width;

// OTSU METHOD

//convert into binary image using Otsu method

Mat left = image(Rect(0, 0, cols, rows));

Mat right = image(Rect(cols, 0, image.cols - cols, rows));

threshold(left, left, 0, 255, CV\_THRESH\_OTSU);

threshold(right, right, 0, 255, CV\_THRESH\_OTSU);

image = colorTrap(image);

// make a copy of the thresh image, this in necessary because findContours modifies the image

matThreshCopy = image.clone();

// input image, make sure to use a copy since the function will modify this image in the course of finding contours

findContours (matThreshCopy,

ptContours,

v4iHierarchy,

RETR\_EXTERNAL,

CHAIN\_APPROX\_SIMPLE);

int largestNumY = 0;

int largestNumX = 0;

int smallestNumY = image.rows;

int smallestNumX = image.cols;

int lastContoursize = 0;

int lastContourh =0;

int largestSize = 0;

for (int i = 0; i < ptContours.size(); i++) {

if (contourArea(ptContours[i]) > MIN\_CONTOUR\_AREA) {

Rect boundingRect = cv::boundingRect(ptContours[i]);

Mat matROI = image(boundingRect);

int x = boundingRect.x;

int y = boundingRect.y;

if (largestNumX < x) {

largestNumX = x;

largestSize = boundingRect.width;

}

if (largestNumY < y) {

largestNumY = y;

}

if (smallestNumY > y) {

smallestNumY = y;

}

if (smallestNumX > x) {

smallestNumX = x;

}

if (lastContourh < boundingRect.height) {

lastContourh = boundingRect.height;

}

}

}

largestNumX = largestNumX + largestSize + 10;

largestNumY = smallestNumY + lastContourh;

int w = (largestNumX + 10) - (smallestNumX - 10);

int h = (largestNumY + 20) - (smallestNumY - 10);

if (w+20 < image.cols && h+20 < image.rows &&((smallestNumX - 10)> 0) && ((smallestNumY - 10)> 0) && ((largestNumX + 10)< image.cols) && ((largestNumY + 20)< image.rows)) {

Rect roi(smallestNumX - 10, smallestNumY - 10,w,h);

Mat image\_roi = image(roi);

image = image\_roi;

}

waitKey(0);

return image;

}

## MyForm.cpp

#include "MyForm.h"

using namespace System;

using namespace System::Windows::Forms;

[STAThread]

int main(array<System::String^>^ args){

Application::EnableVisualStyles();

Application::SetCompatibleTextRenderingDefault(false);

Sample::MyForm form;

Application::Run(%form);

}

## OvOrecognition.cpp

#include<opencv2/core/core.hpp>

#include<opencv2/highgui/highgui.hpp>

#include<opencv2/imgproc/imgproc.hpp>

#include"Segmentation.h"

#include<iostream>

#include <stdio.h>

using namespace std;

using namespace cv;

char ovo(Mat imageROIResized){

int label = getCSV(imageROIResized);

char input;

if(label == 0){

input = '0';

}

if(label == 1){

input = '1';

}

if(label == 2){

input = '2';

}

if(label == 3){

input = '3';

}

if(label == 4){

input = '4';

}

if(label == 5){

input = '5';

}

if(label == 6){

input = '6';

}

if(label == 7){

input = '7';

}

if(label == 8){

input = '8';

}

if(label == 9){

input = '9';

}

if(label == 10){

input = 'a';

}

if(label == 11){

input = 'b';

}

if(label == 12){

input = 'c';

}

if(label == 13){

input = 'x';

}

if(label == 14){

input = 'y';

}

if(label == 15){

input = 'z';

}

if(label == 16){

input = '+';

}

if(label == 17){

input = '-';

}

if(label == 18){

input = '/';

}

if(label == 19){

input = '(';

}

if(label == 20){

input = ')';

}

if(label == 21){

input = '=';

}

cout << "prediction: " << input << endl;

return input;

}

## Segmentation.cpp

// This file involves the image pre-processing

// (greyscale and binarization) and image

// processing (segmentation)

#include<opencv2/core/core.hpp>

#include <opencv2\opencv.hpp>

#include<opencv2/highgui/highgui.hpp>

#include<opencv2/imgproc/imgproc.hpp>

#include<iostream>

#include"Segmentation.h"

#include<vector>

using namespace std;

using namespace cv;

struct node {

char x;

node \*next;

};

string equation="";

const int MIN\_CONTOUR\_AREA = 10;

const int RESIZED\_WIDTH = 28;

const int RESIZED\_HEIGHT = 28;

double MAX\_CONTOUR\_AREA = 0;

int MAX\_CONTOUR\_INDEX = 0;

// CONTOUR WITH DATA

class contourWithInfoData {

public:

vector< Point> contours; // contour

Rect boundingRect; // bounding rect for contour

double fltArea; // area of contour

bool checkIfContourIsValid() {

if (fltArea < MIN\_CONTOUR\_AREA)

return false;

return true;

}

// this function allows us to sort and returns the contours from left to right

static bool sortByBoundingRectXPosition(const contourWithInfoData& cwdLeft, const contourWithInfoData& cwdRight) {

return(cwdLeft.boundingRect.x < cwdRight.boundingRect.x);

}

};

int segmentation(string seg) {

vector<contourWithInfoData> contourInfoData; // declare empty vectors

vector<contourWithInfoData> validContourInfoData;

Mat inputImageEquation; // input image handwritten equation

Mat imageGrayscale; // for greyscaling

Mat imageThresh; // for otsu method

Mat imageThreshCopy; // create a copy of the image before finding contours

vector< vector< Point> > contours; // declare contours vector

vector< Vec4i> v4iHierarchy; // declare contours hierarchy

node \*root;

root = new node;

root->next = 0;

node \*conductor;

conductor = root;

inputImageEquation = imread(seg); // read input image with handwritten equation

if (inputImageEquation.empty()) { // if unable to open image

cout << "Error: Image not read from file\n\n"; return(0);

}

// IMAGE PRE-PROCESS

imageThresh = ImageTrap(inputImageEquation.clone());

imageThreshCopy = imageThresh.clone();

threshold(imageThresh, inputImageEquation, 70, 255,

CV\_THRESH\_BINARY\_INV);

findContours(imageThreshCopy,

contours,

v4iHierarchy,

RETR\_EXTERNAL,

CHAIN\_APPROX\_SIMPLE); // compress horizontal, vertical, and diagonal segments and leave only their end points

for (int i = 0; i < contours.size(); i++) { // for each contour

contourWithInfoData contourWithInfoData; // instantiate a contour with data object

contourWithInfoData.contours = contours[i]; // assign contour to contour with data

contourWithInfoData.boundingRect = boundingRect(contourWithInfoData.contours); // get the bounding rect

contourWithInfoData.fltArea = contourArea(contourWithInfoData.contours); // calculate the contour area

contourInfoData.push\_back(contourWithInfoData); // add contour with data object to list of all contours with data

}

for (int i = 0; i < contourInfoData.size(); i++) { // for all contours

if (contourInfoData[i].checkIfContourIsValid()) {

validContourInfoData.push\_back(contourInfoData[i]); // if so, append to valid contour list

}

}

// SORT CONTOURS FROM LEFT TO RIGHT

sort(validContourInfoData.begin(),

validContourInfoData.end(),

contourWithInfoData::sortByBoundingRectXPosition);

bool equals = false;

for (int i = 0; i < validContourInfoData.size(); i++) { // for each contour

cout << "Contour # : " << i << " ";

// draw a red rect around the current char

rectangle(inputImageEquation, // draw rectangle on original image

validContourInfoData[i].boundingRect, // rect to draw

Scalar(0, 0, 255), // red

2); // thickness

Mat imageROI = imageThresh(validContourInfoData[i].boundingRect); // get ROI image of bounding rect

Mat img = imageROI.clone();

threshold(img, img, 70, 255, CV\_THRESH\_BINARY\_INV);

bool connected = determineConnected(img);

Mat imageROIResized;

if (connected == false) {

int width = validContourInfoData[i].boundingRect.width;

int height = validContourInfoData[i].boundingRect.height;

// for small contours like minus and equals

if (validContourInfoData[i].boundingRect.height < (width/2)+10 /\*&&

validContourInfoData[i].boundingRect.width >=(width/2)+10\*/) {

Mat image = Mat::zeros(28, 28, CV\_8UC1); // create a black image

int j = i + 1;

int x = i - 1;

// FOR EQUAL SIGN

if (validContourInfoData[j].boundingRect.height <(width/2)+10 &&

validContourInfoData[i].boundingRect.height < (width/2)+10 &&

validContourInfoData[x].boundingRect.height >(width/2)+10) {

resize(imageROI, imageROI, Size(15, 4));

imageROI.copyTo(image(Rect(5, 6, imageROI.cols,

imageROI.rows)));

imageROI =imageThresh(validContourInfoData[j].boundingRect);

resize(imageROI, imageROI, Size(15,4));

imageROI.copyTo(image(Rect(5, 15, imageROI.cols,

imageROI.rows)));

imshow("imageROIResized", image);

imageROIResized = image.clone();

char input = ovo(imageROIResized);

conductor->x = input;

conductor->next = new node;

conductor = conductor->next;

conductor->next = 0;

equals = true;

string str(1, input);

equation += str + " ";

}

//FOR MINUS SIGNs

else if (validContourInfoData[x].boundingRect.height > (width/2)+10

&&validContourInfoData[j].boundingRect.height>(width/2)+10

&&validContourInfoData[i].boundingRect.height<(width/2)+10) {

resize(imageROI, imageROI, Size(15, 4));

imageROI.copyTo(image(Rect(5, 12, imageROI.cols,

imageROI.rows)));

imshow("imageROIResized", image);

imageROIResized = image.clone();

char input = ovo(imageROIResized);

conductor->x = input;

conductor->next = new node;

conductor = conductor->next;

conductor->next = 0;

string str(1, input);

equation += str + " ";

}

else if(equals == true){

equals = false;

}

//if the width is too small

else {

resize(imageROI, imageROI, Size(15, 4));

imageROI.copyTo(image(Rect(5, 12, imageROI.cols,

imageROI.rows)));

imshow("imageROIResized", image);

imageROIResized = image.clone();

char input = ovo(imageROIResized);

conductor->x = input;

conductor->next = new node;

conductor = conductor->next;

conductor->next = 0;

string str(1, input);

equation += str + " ";

}

}

else if (validContourInfoData[i].boundingRect.height > width &&

validContourInfoData[i].boundingRect.width < height/2-20) {

Mat img = Mat::zeros(28, 28, CV\_8UC1);

resize(imageROI, imageROI, Size(4,20));

imageROI.copyTo(img(Rect(12, 4, imageROI.cols,

imageROI.rows)));

imageROIResized = img.clone();

imshow("imageROIResized", imageROIResized);

char input = ovo(imageROIResized);

conductor->x = input;

conductor->next = new node;

conductor = conductor->next;

conductor->next = 0;

string str(1, input);

equation += str + " ";

}

else if(equals == true){

equals = false;

cout<<equals;

}

else {

Mat img = Mat::zeros(28, 28, CV\_8UC1);

resize(imageROI, imageROI, Size(20,20));

imageROI.copyTo(img(Rect(4, 4, imageROI.cols, imageROI.rows)));

imageROIResized = img.clone();

imshow("imageROIResized", imageROIResized);

char input = ovo(imageROIResized);

conductor->x = input;

conductor->next = new node;

conductor = conductor->next;

conductor->next = 0;

string str(1, input);

equation += str + " ";

}

}

// FOR CONNECTED CHARACTERS

else {

vector<contourWithInfoData> connectedInfoData;

vector< vector< Point> > ptContours;

vector< Vec4i> hierarchy;

Mat image;

cout << "Connected" << endl;

Mat connected = imageROI.clone();

image = SplitImage(imageROI);

imshow("connected ",image);

imwrite("connected.png",image);

Mat clone = image.clone();

// input image, make sure to use a copy since the function will modify this

// image in the course of finding contours

findContours(image.clone(),

ptContours, // output contours

hierarchy, // output hierarchy

RETR\_EXTERNAL, // retrieve the outermost contours only

CHAIN\_APPROX\_SIMPLE);

int numberOfConnected = 0;

for(int j = 0; j < ptContours.size(); j++){

numberOfConnected++;

}

if (numberOfConnected>2){

image = splitConnected(connected);

imshow("connected ",image);

findContours(image.clone(),

ptContours, // output contours

hierarchy, // output hierarchy

RETR\_EXTERNAL, // retrieve the outermost contours only

CHAIN\_APPROX\_SIMPLE);

}

for(int x = 0; x < ptContours.size(); x++){

contourWithInfoData connectedWithInfoData;

connectedWithInfoData.contours = ptContours[x];

connectedWithInfoData.boundingRect =

boundingRect(connectedWithInfoData.contours);

connectedWithInfoData.fltArea =

contourArea(connectedWithInfoData.contours);

connectedInfoData.push\_back(connectedWithInfoData);

}

sort(connectedInfoData.begin(),

connectedInfoData.end(),

contourWithInfoData::sortByBoundingRectXPosition);

for(int j = 0; j < connectedInfoData.size(); j++){

if(connectedInfoData[j].fltArea > 50){

Mat roi = clone(connectedInfoData[j].boundingRect);

Mat img = Mat::zeros(28, 28, CV\_8UC1);

resize(roi, roi, Size(20,20));

roi.copyTo(img(Rect(4, 4, 20, 20)));

imageROIResized = img.clone();

imshow("imageROIResized", imageROIResized);

char input = ovo(imageROIResized);

conductor->x = input;

conductor->next = new node;

conductor = conductor->next;

conductor->next = 0;

string str(1, input);

equation += str + " ";

}

}

}

imshow("inputImageEquation", inputImageEquation);

int intChar = waitKey(0); // get key press

if (intChar == 27) { // if esc key was pressed

return(0);

}

}

cout << "\n" << equation << endl;

node \*temp;

temp = root;

node \*t;

//DELETING OF LAST NODE

conductor = root;

temp = root;

while (temp->next != 0) {

temp = temp->next;

if (temp->next != 0) {

conductor = temp;

}

}

conductor->next = 0;

free(temp);

conductor = root;

while (conductor != 0) {

cout << conductor->x;

conductor = conductor->next;

}

// DETERMINING OF OPERATOR AND OPERAND

//Node for operand

node \*rootOperand;

rootOperand = new node;

rootOperand->next = 0;

node \*operandPtr;

operandPtr = rootOperand;

//Node for operator

node \*rootOperator;

rootOperator = new node;

rootOperator->next = 0;

node \*operatorPtr;

operatorPtr = rootOperator;

//THE PROCESS

node \*ptrX;

node \*ptrY;

ptrX = root;

ptrY = ptrX->next;

int counter = 0;

temp = root;

while (temp != 0) {

temp = temp->next;

counter++;

}

if (counter <= 2) { // if there are less than 3 characters

cout << "\n\nNOT AN EQUATION";

}

else {

conductor = root;

while (conductor != 0) {

if ((isdigit(ptrX->x)) && ((isdigit(ptrY->x)) && (ptrY->next == 0))) {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

else if ((isdigit(ptrX->x)) && (isdigit(ptrY->x))) {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

// digit and alpha

else if ((isdigit(ptrX->x)) && ((isalpha(ptrY->x)))) {

if (ptrY->next == 0) {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

operandPtr->x = ',';

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

else {

// if y is normal alpha

if ((ptrY->x == 'a') || (ptrY->x == 'b') || (ptrY->x == 'c') || (ptrY->x

== 'y') || (ptrY->x == 'z') ) {

temp = ptrY->next;

if ( (isdigit(temp->x)) || (temp->x == 'a') || (temp->x == 'b') ||

(temp->x == 'c') || (temp->x == 'y') || (temp->x == 'z')) {

if ( isdigit(temp->x) ) {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

operandPtr->x = ',';

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

else if ( (temp->x == 'a') || (temp->x == 'b') || (temp->x ==

'c') || (temp->x == 'y') || (temp->x == 'z') ) {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

operandPtr->x = ',';

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

}

else if ( (temp->x == '(') || (temp->x == ')') || (temp->x == '+') ||

(temp->x == '-') || (temp->x == '/') || (temp->x == '=') ) {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

operandPtr->x = ',';

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

cout << ptrX->x;

cout << ptrY->x;

}

else if ( (temp->x == 'x') && (temp->next == 0) ){

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

else if (temp->x == 'x') {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

operandPtr->x = ',';

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operatorPtr->x = ptrX->x;

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

operatorPtr->x = ',';

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

}

// if y is x

else if (ptrY->x == 'x') {

temp = ptrY->next;

if ( (isdigit(temp->x)) || (temp->x == 'a') || (temp->x == 'b') ||

(temp->x == 'c') || (temp->x == 'y') || (temp->x == 'z') ) {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

operandPtr->x = ',';

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operatorPtr->x = ptrX->x;

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

operatorPtr->x = ',';

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

else if ( (temp->x == '+') || (temp->x == '-') || (temp->x == '/') ||

(temp->x == '(') || (temp->x == ')') || (temp->x == '=')) {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

operandPtr->x = ',';

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

else if (temp->x == 'x') {

t = temp->next;

if (t != 0) {

if ( (isdigit(t->x)) || (isalpha(t->x)) || (t->x == '(') ) {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

operandPtr->x = ',';

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operatorPtr->x = ptrX->x;

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

operatorPtr->x = ',';

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

else if ( (t->x == '+') || (t->x == '-') || (t->x == '/') || (t-

>x == ')') || (t->x == '=') ) {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

operandPtr->x = ',';

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operatorPtr->x = ptrX->x;

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

operatorPtr->x = ',';

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

}

else if (t == 0) {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

operandPtr->x = ',';

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operatorPtr->x = ptrX->x;

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

operatorPtr->x = ',';

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

}

}

}

}

// alpha and alpha

else if ( (isalpha(ptrX->x)) && (isalpha(ptrY->x)) ) {

if (ptrY->next == 0) {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

operandPtr->x = ',';

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

else {

if ( (ptrY->x == 'a') || (ptrY->x == 'b') || (ptrY->x == 'c') || (ptrY->x

== 'y') || (ptrY->x == 'z') ) {

cout << "\n";

cout << ptrX->x;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

cout << ptrX->x;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

cout << " is not a variable...";

}

else if ( ptrY->x == 'x' ) {

temp = ptrY->next;

if ((isdigit(temp->x)) ||(isalpha(temp->x)) || (temp->x == '(')) {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

operandPtr->x = ',';

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operatorPtr->x = ptrX->x;

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

operatorPtr->x = ',';

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

else if ( (temp->x == '+') || (temp->x == '-') || (temp->x == '/') ||

(temp->x == ')') || (temp->x == '=') ) {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

operandPtr->x = ',';

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operatorPtr->x = ptrX->x;

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

operatorPtr->x = ',';

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

}

}

}

// alpha and operator || alpha and close parenthesis

else if ((isalpha(ptrX->x)) && ((ptrY->x == '+') || (ptrY->x == '-') || (ptrY-

>x == 'x') || (ptrY->x == '/') || (ptrY->x == ')') || (ptrY->x == '='))) {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

operandPtr->x = ',';

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

// digit and operator || digit and close parenthesis

else if ((isdigit(ptrX->x)) && ((ptrY->x == '+') || (ptrY->x == '-') || (ptrY->x

== 'x') || (ptrY->x == '/') || (ptrY->x == ')') || (ptrY->x == '=')/\* || (ptrY-

>next == 0)\*/)) {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

operandPtr->x = ',';

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

// operator and digit

else if (((ptrX->x == '+') || (ptrX->x == '-') || (ptrX->x == 'x') || (ptrX->x ==

'/') || (ptrX->x == '=')) && (isdigit(ptrY->x)) ) {

operatorPtr->x = ptrX->x;

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

operatorPtr->x = ',';

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

// operator and alpha

else if ( ((ptrX->x == '+') || (ptrX->x == '-') || (ptrX->x == 'x') || (ptrX->x ==

'/') || (ptrX->x == '=')) && (isalpha(ptrY->x)) ) {

operatorPtr->x = ptrX->x;

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

operatorPtr->x = ',';

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

// operator ending

else if ( ((ptrX->x == '+') || (ptrX->x == '-') || (ptrX->x == 'x') || (ptrX->x ==

'/')) && (ptrY->next == 0) ) {

operatorPtr->x = ptrX->x;

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

// operator and equals

else if ( ((ptrX->x == '+') || (ptrX->x == '-') || (ptrX->x == 'x') || (ptrX->x ==

'/')) && (ptrY->x == '=') ) {

operatorPtr->x = ptrX->x;

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operatorPtr->x = ptrX->x;

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

operatorPtr->x = ',';

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

// operator and operator

else if ( ((ptrX->x == '+') || (ptrX->x == '-') || (ptrX->x == 'x') || (ptrX->x ==

'/') || (ptrX->x == '=') ) && ((ptrY->x == '+') || (ptrY->x == '-') || (ptrY-

>x == 'x') || (ptrY->x == '/') || (ptrY->x == '=')) ) {

cout << "here";

operatorPtr->x = ptrX->x;

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operatorPtr->x = ptrX->x;

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

operatorPtr->x = ',';

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

// operator and open parenthesis

else if (((ptrX->x == '+') || (ptrX->x == '-') || (ptrX->x == 'x') || (ptrX->x ==

'/') || (ptrY->x == '=')) && ((ptrY->x == '(')/\* || (ptrY->next == 0)\*/)) {

operatorPtr->x = ptrX->x;

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

operatorPtr->x = ',';

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

ptrX = ptrX->next;

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

// open parenthesis and digit || open parenthesis and alpha || open parenthesis

// and open parenthesis

else if ((ptrX->x == '(') && ((isdigit(ptrY->x)) || (isalpha(ptrY->x)) || (ptrY-

>x == '('))) {

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

// close parenthesis and operator

else if ((ptrX->x == ')') && ((ptrY->x == '+') || (ptrY->x == '-') || (ptrY->x

== 'x') || (ptrY->x == '/') || (ptrY->x == '='))) {

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

else if ((ptrX->x == ')') && ((ptrY->x == ')') /\*|| (ptrY->next == 0)\*/)) {

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

//Not acceptable / Errors

// digit/alpha and close/open parenthesis and open parenthesis

else if ( ((isdigit(ptrX->x)) || (isalpha(ptrX->x))) && (ptrY->x == '(') ) {

operandPtr->x = ptrX->x;

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

operandPtr->x = ',';

operandPtr->next = new node;

operandPtr = operandPtr->next;

operandPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

// alpha and digit

else if ( (isalpha(ptrX->x)) && (isdigit(ptrY->x)) ) {

cout << "\n";

cout << ptrX->x;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

cout << " not a part of the operand...";

}

// open parenthesis and operator

else if ( (ptrX->x == '(') && ( (ptrY->x == '+') || (ptrY->x == '-') || (ptrY->x

== '/') || (ptrY->x == '=') || (ptrY->x == ')')) ) {

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

conductor = conductor->next;

operatorPtr->x = ptrX->x;

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

operatorPtr->x = ',';

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

// close parenthesis and digit || alpha || close parenthesis

else if ( (ptrX->x == ')') && ((isdigit(ptrY->x)) || (isalpha(ptrY->x)) ||

(ptrY->x = ')')) ) {

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

//operator and close parenthesis

else if ( ((ptrX->x == '+') || (ptrX->x == '-') || (ptrX->x == '/') || (ptrX->x ==

'=') || (ptrX->x == ')')) && (ptrY->x == ')') ) {

operatorPtr->x = ptrX->x;

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

operatorPtr->x = ',';

operatorPtr->next = new node;

operatorPtr = operatorPtr->next;

operatorPtr->next = 0;

if (ptrX->next != 0) {

ptrX = ptrX->next;

}

if (ptrY->next != 0) {

ptrY = ptrY->next;

}

}

conductor = conductor->next;

}

// Cleaning of Linked List

node \*c;

node \*ptr;

// Operands

if (rootOperand->next == 0) {

cout << "\n\nNO OPERANDS DETECTED... NOT AN EQUATION";

}

else {

conductor = rootOperand;

c = conductor->next;

ptr = c->next;

if (ptr == 0) {

conductor->next = 0;

free(c);

free(ptr);

}

else {

while (ptr->next != 0) {

conductor = conductor->next;

c = conductor->next;

ptr = c->next;

}

if ((ptr->next == 0) && ((isdigit(c->x)) || (isalpha(c->x)) || (c->x ==

','))) {

if (c->x == ',') {

conductor->next = 0;

free(c);

free(ptr);

}

else if ((isdigit(c->x)) || (isdigit(c->x))) {

c->next = 0;

free(ptr);

}

}

}

cout << "\nOPERANDS: ";

operandPtr = rootOperand;

while (operandPtr != 0) {

cout << operandPtr->x;

operandPtr = operandPtr->next;

}

}

// Operator

if (rootOperator->next == 0) {

cout << "\n\nNO OPERATORS DETECTED... NOT AN EQUATION";

}

else {

conductor = rootOperator;

c = conductor->next;

ptr = c->next;

if (ptr == 0) {

conductor->next = 0;

free(c);

free(ptr);

}

else {

while (ptr->next != 0) {

conductor = conductor->next;

c = conductor->next;

ptr = c->next;

}

if ((ptr->next == 0) && (c->x == ',')) {

conductor->next = 0;

free(c);

free(ptr);

}

else {

c->next = 0;

free(ptr);

}

}

cout << "\nOPERATORS: ";

operatorPtr = rootOperator;

while (operatorPtr != 0) {

cout << operatorPtr->x;

operatorPtr = operatorPtr->next;

}

}

}

cin.get();

cin.get();

return 0;

}

## SplitConnected\_Dropfall.cpp

#include<opencv2/core/core.hpp>

#include<opencv2/highgui/highgui.hpp>

#include<opencv2/imgproc/imgproc.hpp>

#include<opencv2/ml/ml.hpp>

#include<iostream>

#include<vector>

using namespace std;

using namespace cv;

const int MIN\_CONTOUR\_AREA = 100;

const int RESIZED\_IMAGE\_WIDTH = 30;

const int RESIZED\_IMAGE\_HEIGHT = 30;

Mat mareHistogram(Mat& img, int t){

int sz = (t) ? img.rows : img.cols; // 1 == rows // 0 == columns

Mat mhist = Mat::zeros(1, sz, CV\_8U);

//count nonzero value and check max V

int max = -100;

for (int j = 0; j < sz; ++j) {

Mat data = (t) ? img.row(j) : img.col(j);

int v = countNonZero(data);

mhist.at< unsigned char >(j) = v;

if (v > max)

max = v;

}

Mat histo;

int width, height;

if (t){

width = max;

height = sz;

histo = Mat::zeros(Size(width, height), CV\_8U);

for (int i = 0; i < height; ++i){

for (int j = 0; j < mhist.at< unsigned char >(i); ++j)

histo.at< unsigned char >(i, j) = 255;

}

}

else {

width = sz;

height = max;

histo = Mat::zeros(Size(width, height), CV\_8U);

for (int i = 0; i< width; ++i){

for (int j = 0; j< mhist.at< unsigned char >(i); ++j)

histo.at< unsigned char >(max - j - 1, i) = 255;

}

}

return histo;

}

Mat SplitImage(Mat matROI) {

//Drop fall from bottom

Mat img = matROI.clone();

cv::threshold(img, img, 70, 255, CV\_THRESH\_BINARY\_INV);

Mat hHisto = mareHistogram(img, 0);

Mat vHisto = mareHistogram(img, 1);

cv::imwrite("matROIconnected1.jpg", matROI);

cv::Point center(matROI.cols / 2, matROI.rows / 2);

int cx = center.x;

int cy = center.y;

int width = matROI.cols;

int height = matROI.rows;

// Initial Drop Fall Algorithm

int x = matROI.cols / 2.5;

int y = matROI.rows;

int tempX = 0;

int prevX = 0, prevY = 0;

int yCoor = 0, xCoor = 0;

int up, upright, upleft, right, left, current;

while (y > 0 && x < matROI.cols) {

xCoor = x;

yCoor = y;

up = matROI.at<uchar>(Point(xCoor, --yCoor));

xCoor = x;

yCoor = y;

upright = matROI.at<uchar>(Point(++xCoor, --yCoor));

xCoor = x;

yCoor = y;

upleft = matROI.at<uchar>(Point(--xCoor, --yCoor));

xCoor = x;

yCoor = y;

right = matROI.at<uchar>(Point(++xCoor, yCoor));

xCoor = x;

yCoor = y;

left = matROI.at<uchar>(Point(--xCoor, yCoor));

current = matROI.at<uchar>(Point(x, y));

if (current == 0) { //on black

if (up != 255) { //go up

prevX = x;

prevY = y;

y--;

}

else if (upleft != 255) { //go upleft

prevX = x;

prevY = y;

x--;

y--;

}

else if (upright != 255) { //go upright

prevX = x;

prevY = y;

x++;

y--;

}

else if (right != 255) { //right

tempX = x + 1;

if (prevX >= tempX) {

prevX = x;

prevY = y;

x--;

}

else {

prevX = x;

prevY = y;

x++;

}

}

else if (left != 255) { //left

tempX = x - 1;

if (prevX <= tempX) {

prevX = x;

prevY = y;

x++;

}

else {

prevX = x;

prevY = y;

x--;

}

}

else { //up

y--;

}

}

else if (current == 255) { //on white

if (matROI.at<uchar>(Point(x, yCoor++)) == 255) {//go up

matROI.at<uchar>(Point(x, y)) = 0;

y--;

}

}

}

return matROI;

}

Mat splitConnected(Mat matROI){

//Drop fall from top

cv::Point center(matROI.cols / 2, matROI.rows / 2);

int x = matROI.cols/2;

uchar charX;

uchar charY;

// initial drop fall algorithm

int y = 0;

int prev = 0, prevy =0;

int yCoor = 0, xCoor = 0;

int down, downright, downleft, right, left, current;

while (y < matROI.rows && x < matROI.cols) {

xCoor = x;

yCoor = y;

down = matROI.at<uchar>(Point(x, ++yCoor));

xCoor = x;

yCoor = y;

downright = matROI.at<uchar>(Point(++xCoor, ++yCoor));

xCoor = x;

yCoor = y;

downleft = matROI.at<uchar>(Point(--xCoor, ++yCoor));

xCoor = x;

yCoor = y;

right = matROI.at<uchar>(Point(++xCoor, yCoor));

xCoor = x - 1;

yCoor = y;

left = matROI.at<uchar>(Point(xCoor, yCoor));

current = matROI.at<uchar>(Point(x, y));

if (current == 0) { //on black

if (down != 255) { //go down

y++;

}

else if (downright != 255) {//go downright

x++;

y++;

}

else if (downleft != 255) { //go downleft

x--;

y++;

}

else if (left != 255) { //left

int xC = x + 1;

if (prev <= xC) {

prev = x;

x--;

}

else {

y++;

}

prevy = y;

}

else if (right != 255) { //right

int xCy = x + 1;

if (prev <= xCy) {

prev = x;

y++;

}

else{

x++;

}

}

else { //down

y++;

}

}

else if (current == 255){ //on white

if (matROI.at<uchar>(Point(x, yCoor++)) == 255) { //go down

matROI.at<uchar>(Point(x, y)) = 0;

y++;

}

}

}

return matROI;

}

## svm-predict.cpp

#include <stdio.h>

#include <ctype.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include "svm.h"

int print\_null(const char \*s, ...) {

return 0;

}

static int(\*info)(const char \*fmt, ...) = &printf;

struct svm\_node \*x;

int max\_nr\_attr = 64;

struct svm\_model\* model;

int predict\_probability = 0;

static char \*line = NULL;

static int max\_line\_len;

static char\* readline(FILE \*input) {

int len;

if (fgets(line, max\_line\_len, input) == NULL)

return NULL;

while (strrchr(line, '\n') == NULL) {

max\_line\_len \*= 2;

line = (char \*)realloc(line, max\_line\_len);

len = (int)strlen(line);

if (fgets(line + len, max\_line\_len - len, input) == NULL)

break;

}

return line;

}

void exit\_input\_error(int line\_num) {

fprintf(stderr, "Wrong input format at line %d\n", line\_num);

exit(1);

}

int predict(FILE \*input, FILE \*output) {

int correct = 0;

int total = 0;

double error = 0;

double sump = 0, sumt = 0, sumpp = 0, sumtt = 0, sumpt = 0;

double target\_label, predict\_label;

int svm\_type = svm\_get\_svm\_type(model);

int nr\_class = svm\_get\_nr\_class(model);

double \*prob\_estimates = NULL;

int j;

if (predict\_probability) {

if (svm\_type == NU\_SVR || svm\_type == EPSILON\_SVR)

info("Prob. model for test data: target value = predicted value + z,\nz:

Laplace distribution e^(-|z|/sigma)/(2sigma),sigma=%g\n",

svm\_get\_svr\_probability(model));

else {

int \*labels = (int \*)malloc(nr\_class\*sizeof(int));

svm\_get\_labels(model, labels);

prob\_estimates = (double \*)malloc(nr\_class\*sizeof(double));

fprintf(output, "labels");

for (j = 0; j<nr\_class; j++)

fprintf(output, " %d", labels[j]);

fprintf(output, "\n");

free(labels);

}

}

max\_line\_len = 1024;

line = (char \*)malloc(max\_line\_len\*sizeof(char));

while (readline(input) != NULL) {

int i = 0;

char \*idx, \*val, \*label, \*endptr;

// strtol gives 0 if wrong format, and precomputed kernel has <index>

// start from 0

int inst\_max\_index = -1;

label = strtok(line, " \t\n");

if (label == NULL) // empty line

exit\_input\_error(total + 1);

while (1) {

if (i >= max\_nr\_attr - 1) { // need one more for index = -1

max\_nr\_attr \*= 2;

x = (struct svm\_node \*) realloc(x, max\_nr\_attr\*sizeof(struct

svm\_node));

}

idx = strtok(NULL, ":");

val = strtok(NULL, " \t");

if (val == NULL)

break;

errno = 0;

x[i].index = (int)strtol(idx, &endptr, 10);

if (endptr==idx||errno != 0 || \*endptr != '\0' || x[i].index <= inst\_max\_index)

exit\_input\_error(total + 1);

else

inst\_max\_index = x[i].index;

errno = 0;

x[i].value = strtod(val, &endptr);

if (endptr == val || errno != 0 || (\*endptr != '\0' && !isspace(\*endptr)))

exit\_input\_error(total + 1);

++i;

}

x[i].index = -1;

if (predict\_probability && (svm\_type == C\_SVC || svm\_type == NU\_SVC)) {

predict\_label = svm\_predict\_probability(model, x, prob\_estimates);

fprintf(output, "%g", predict\_label);

for (j = 0; j<nr\_class; j++){

fprintf(output, " %g", prob\_estimates[j]);

}

fprintf(output, "\n");

printf("labels :%g \n", predict\_label);

}

else {

predict\_label = svm\_predict(model, x);

fprintf(output, "%g\n", predict\_label);

}

}

return predict\_label;

}

int svmPredict() {

FILE \*input, \*output;

int i;

char \*\*argv;

input = fopen("exmpleData.data", "r");

if (input == NULL) {

fprintf(stderr, "can't open input file %s\n", argv[i]);

exit(1);

}

output = fopen("mnist.outputguesses", "w");

if (output == NULL) {

fprintf(stderr, "can't open output file %s\n", argv[i + 2]);

exit(1);

}

if ((model = svm\_load\_model("model1.model")) == 0) {

fprintf(stderr, "can't open model file %s\n", argv[i + 1]);

exit(1);

}

x = (struct svm\_node \*) malloc(max\_nr\_attr\*sizeof(struct svm\_node));

if (predict\_probability) {

if (svm\_check\_probability\_model(model) == 0) {

fprintf(stderr, "Model does not support probabiliy estimates\n");

exit(1);

}

}

else {

if (svm\_check\_probability\_model(model) != 0) {

info("Model supports probability estimates, but disabled in prediction.\n");

}

}

int label= predict(input, output);

svm\_free\_and\_destroy\_model(&model);

free(x);

free(line);

fclose(input);

fclose(output);

return label;

}

## svm.cpp

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <float.h>

#include <string.h>

#include <stdarg.h>

#include <limits.h>

#include <locale.h>

#include "svm.h"

int libsvm\_version = LIBSVM\_VERSION;

typedef float Qfloat;

typedef signed char schar;

#ifndef min

template <class T> static inline T min(T x, T y) { return (x<y) ? x : y; }

#endif

#ifndef max

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

#endif

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

template <class S, class T> static inline void clone(T\*& dst, S\* src, int n){

dst = new T[n];

memcpy((void \*)dst, (void \*)src, sizeof(T)\*n);

}

static inline double powi(double base, int times){

double tmp = base, ret = 1.0;

for (int t = times; t>0; t /= 2){

if (t % 2 == 1) ret \*= tmp;

tmp = tmp \* tmp;

}

return ret;

}

#define INF HUGE\_VAL

#define TAU 1e-12

#define Malloc(type,n) (type \*)malloc((n)\*sizeof(type))

static void print\_string\_stdout(const char \*s) {

fputs(s, stdout);

fflush(stdout);

}

static void(\*svm\_print\_string) (const char \*) = &print\_string\_stdout;

#if 1

static void info(const char \*fmt, ...) {

char buf[BUFSIZ];

va\_list ap;

va\_start(ap, fmt);

vsprintf(buf, fmt, ap);

va\_end(ap);

(\*svm\_print\_string)(buf);

}

#else

static void info(const char \*fmt, ...) {}

#endif

// Kernel Cache

// l is the number of total data items

// size is the cache size limit in bytes

class Cache {

public:

Cache(int l, long int size);

~Cache();

int get\_data(const int index, Qfloat \*\*data, int len);

void swap\_index(int i, int j);

private:

int l;

long 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);

};

Cache::Cache(int l\_, long int size\_) :l(l\_), size(size\_) {

head = (head\_t \*)calloc(l, sizeof(head\_t)); // initialized to 0

size /= sizeof(Qfloat);

size -= l \* sizeof(head\_t) / sizeof(Qfloat);

size = max(size, 2 \* (long int)l); // cache must be large enough for two columns

lru\_head.next = lru\_head.prev = &lru\_head;

}

Cache::~Cache() {

for (head\_t \*h = lru\_head.next; h != &lru\_head; h = h->next)

free(h->data);

free(head);

}

void Cache::lru\_delete(head\_t \*h) {

h->prev->next = h->next;

h->next->prev = h->prev;

}

void Cache::lru\_insert(head\_t \*h) {

// insert to last position

h->next = &lru\_head;

h->prev = lru\_head.prev;

h->prev->next = h;

h->next->prev = h;

}

int Cache::get\_data(const int index, Qfloat \*\*data, int len) {

head\_t \*h = &head[index];

if (h->len) lru\_delete(h);

int more = len - h->len;

if (more > 0) {

// free old space

while (size < more) {

head\_t \*old = lru\_head.next;

lru\_delete(old);

free(old->data);

size += old->len;

old->data = 0;

old->len = 0;

}

// allocate new space

h->data = (Qfloat \*)realloc(h->data, sizeof(Qfloat)\*len);

size -= more;

swap(h->len, len);

}

lru\_insert(h);

\*data = h->data;

return len;

}

void Cache::swap\_index(int i, int j) {

if (i == j) return;

if (head[i].len) lru\_delete(&head[i]);

if (head[j].len) lru\_delete(&head[j]);

swap(head[i].data, head[j].data);

swap(head[i].len, head[j].len);

if (head[i].len) lru\_insert(&head[i]);

if (head[j].len) lru\_insert(&head[j]);

if (i>j) swap(i, j);

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;

}

}

}

}

// 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

class QMatrix {

public:

virtual Qfloat \*get\_Q(int column, int len) const = 0;

virtual double \*get\_QD() const = 0;

virtual void swap\_index(int i, int j) const = 0;

virtual ~QMatrix() {}

};

class Kernel : public QMatrix {

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 double \*get\_QD() 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 int 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 {

return dot(x[i], x[j]);

}

double kernel\_poly(int i, int j) const {

return powi(gamma\*dot(x[i], x[j]) + coef0, degree);

}

double kernel\_rbf(int i, int j) const {

return exp(-gamma\*(x\_square[i] + x\_square[j] - 2 \* dot(x[i], x[j])));

}

double kernel\_sigmoid(int i, int j) const {

return tanh(gamma\*dot(x[i], x[j]) + coef0);

}

double kernel\_precomputed(int i, int j) const {

return x[i][(int)(x[j][0].value)].value;

}

};

Kernel::Kernel(int l, svm\_node \* const \* x\_, const svm\_parameter& param)

:kernel\_type(param.kernel\_type), degree(param.degree),

gamma(param.gamma), coef0(param.coef0) {

switch (kernel\_type) {

case LINEAR:

kernel\_function = &Kernel::kernel\_linear;

break;

case POLY:

kernel\_function = &Kernel::kernel\_poly;

break;

case RBF:

kernel\_function = &Kernel::kernel\_rbf;

break;

case SIGMOID:

kernel\_function = &Kernel::kernel\_sigmoid;

break;

case PRECOMPUTED:

kernel\_function = &Kernel::kernel\_precomputed;

break;

}

clone(x, x\_, l);

if (kernel\_type == RBF) {

x\_square = new double[l];

for (int i = 0; i<l; i++)

x\_square[i] = dot(x[i], x[i]);

}

else

x\_square = 0;

}

Kernel::~Kernel() {

delete[] x;

delete[] x\_square;

}

double Kernel::dot(const svm\_node \*px, const svm\_node \*py) {

double sum = 0;

while (px->index != -1 && py->index != -1) {

if (px->index == py->index) {

sum += px->value \* py->value;

++px;

++py;

}

else {

if (px->index > py->index)

++py;

else

++px;

}

}

return sum;

}

double Kernel::k\_function(const svm\_node \*x, const svm\_node \*y,

const svm\_parameter& param) {

switch (param.kernel\_type) {

case LINEAR:

return dot(x, y);

case POLY:

return powi(param.gamma\*dot(x, y) + param.coef0, param.degree);

case RBF:

{

double sum = 0;

while (x->index != -1 && y->index != -1) {

if (x->index == y->index) {

double d = x->value - y->value;

sum += d\*d;

++x;

++y;

}

else {

if (x->index > y->index) {

sum += y->value \* y->value;

++y;

}

else {

sum += x->value \* x->value;

++x;

}

}

}

while (x->index != -1){

sum += x->value \* x->value;

++x;

}

while (y->index != -1){

sum += y->value \* y->value;

++y;

}

return exp(-param.gamma\*sum);

}

case SIGMOID:

return tanh(param.gamma\*dot(x, y) + param.coef0);

case PRECOMPUTED: //x: test (validation), y: SV

return x[(int)(y->value)].value;

default:

return 0; // Unreachable

}

}

// An SMO algorithm in Fan et al., JMLR 6(2005), p. 1889--1918

// Solves:

// min 0.5(\alpha^T Q \alpha) + p^T \alpha

//

// y^T \alpha = \delta

// y\_i = +1 or -1

// 0 <= alpha\_i <= Cp for y\_i = 1

// 0 <= alpha\_i <= Cn for y\_i = -1

// Given:

// Q, p, y, Cp, Cn, and an initial feasible point \alpha

// l is the size of vectors and matrices

// eps is the stopping tolerance

// solution will be put in \alpha, objective value will be put in obj

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 QMatrix& Q, const double \*p\_, 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 QMatrix \*Q;

const double \*QD;

double eps;

double Cp, Cn;

double \*p;

int \*active\_set;

double \*G\_bar; // gradient, if we treat free variables as 0

int l;

bool unshrink; // XXX

double get\_C(int i) {

return (y[i] > 0) ? Cp : Cn;

}

void update\_alpha\_status(int i) {

if (alpha[i] >= get\_C(i))

alpha\_status[i] = UPPER\_BOUND;

else if (alpha[i] <= 0)

alpha\_status[i] = LOWER\_BOUND;

else alpha\_status[i] = FREE;

}

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();

private:

bool be\_shrunk(int i, double Gmax1, double Gmax2);

};

void Solver::swap\_index(int i, int j) {

Q->swap\_index(i, j);

swap(y[i], y[j]);

swap(G[i], G[j]);

swap(alpha\_status[i], alpha\_status[j]);

swap(alpha[i], alpha[j]);

swap(p[i], p[j]);

swap(active\_set[i], active\_set[j]);

swap(G\_bar[i], G\_bar[j]);

}

void Solver::reconstruct\_gradient() {

// reconstruct inactive elements of G from G\_bar and free variables

if (active\_size == l) return;

int i, j;

int nr\_free = 0;

for (j = active\_size; j<l; j++)

G[j] = G\_bar[j] + p[j];

for (j = 0; j<active\_size; j++)

if (is\_free(j))

nr\_free++;

if (2 \* nr\_free < active\_size)

info("\nWARNING: using -h 0 may be faster\n");

if (nr\_free\*l > 2 \* active\_size\*(l - active\_size)) {

for (i = active\_size; i<l; i++) {

const Qfloat \*Q\_i = Q->get\_Q(i, active\_size);

for (j = 0; j<active\_size; j++)

if (is\_free(j))

G[i] += alpha[j] \* Q\_i[j];

}

}

else {

for (i = 0; i<active\_size; i++)

if (is\_free(i)) {

const Qfloat \*Q\_i = Q->get\_Q(i, l);

double alpha\_i = alpha[i];

for (j = active\_size; j<l; j++)

G[j] += alpha\_i \* Q\_i[j];

}

}

}

void Solver::Solve(int l, const QMatrix& Q, const double \*p\_, const schar \*y\_,

double \*alpha\_, double Cp, double Cn, double eps,

SolutionInfo\* si, int shrinking) {

this->l = l;

this->Q = &Q;

QD = Q.get\_QD();

clone(p, p\_, l);

clone(y, y\_, l);

clone(alpha, alpha\_, l);

this->Cp = Cp;

this->Cn = Cn;

this->eps = eps;

unshrink = false;

// initialize alpha\_status

{

alpha\_status = new char[l];

for (int i = 0; i<l; i++)

update\_alpha\_status(i);

}

// initialize active set (for shrinking)

{

active\_set = new int[l];

for (int i = 0; i<l; i++)

active\_set[i] = i;

active\_size = l;

}

// initialize gradient

{

G = new double[l];

G\_bar = new double[l];

int i;

for (i = 0; i<l; i++) {

G[i] = p[i];

G\_bar[i] = 0;

}

for (i = 0; i<l; i++)

if (!is\_lower\_bound(i)) {

const 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];

}

}

// optimization step

int iter = 0;

int max\_iter = max(10000000, l>INT\_MAX / 100 ? INT\_MAX : 100 \* l);

int counter = min(l, 1000) + 1;

while (iter < max\_iter) {

// show progress and do shrinking

if (--counter == 0) {

counter = min(l, 1000);

if (shrinking) do\_shrinking();

info(".");

}

int i, j;

if (select\_working\_set(i, j) != 0) {

// reconstruct the whole gradient

reconstruct\_gradient();

// reset active set size and check

active\_size = l;

info("\*");

if (select\_working\_set(i, j) != 0)

break;

else

counter = 1; // do shrinking next iteration

}

++iter;

// update alpha[i] and alpha[j], handle bounds carefully

const Qfloat \*Q\_i = Q.get\_Q(i, active\_size);

const Qfloat \*Q\_j = Q.get\_Q(j, active\_size);

double C\_i = get\_C(i);

double C\_j = get\_C(j);

double old\_alpha\_i = alpha[i];

double old\_alpha\_j = alpha[j];

if (y[i] != y[j]) {

double quad\_coef = QD[i] + QD[j] + 2 \* Q\_i[j];

if (quad\_coef <= 0)

quad\_coef = TAU;

double delta = (-G[i] - G[j]) / quad\_coef;

double diff = alpha[i] - alpha[j];

alpha[i] += delta;

alpha[j] += delta;

if (diff > 0) {

if (alpha[j] < 0) {

alpha[j] = 0;

alpha[i] = diff;

}

}

else {

if (alpha[i] < 0) {

alpha[i] = 0;

alpha[j] = -diff;

}

}

if (diff > C\_i - C\_j) {

if (alpha[i] > C\_i) {

alpha[i] = C\_i;

alpha[j] = C\_i - diff;

}

}

else {

if (alpha[j] > C\_j) {

alpha[j] = C\_j;

alpha[i] = C\_j + diff;

}

}

}

else {

double quad\_coef = QD[i] + QD[j] - 2 \* Q\_i[j];

if (quad\_coef <= 0)

quad\_coef = TAU;

double delta = (G[i] - G[j]) / quad\_coef;

double sum = alpha[i] + alpha[j];

alpha[i] -= delta;

alpha[j] += delta;

if (sum > C\_i) {

if (alpha[i] > C\_i) {

alpha[i] = C\_i;

alpha[j] = sum - C\_i;

}

}

else{

if (alpha[j] < 0){

alpha[j] = 0;

alpha[i] = sum;

}

}

if (sum > C\_j) {

if (alpha[j] > C\_j) {

alpha[j] = C\_j;

alpha[i] = sum - C\_j;

}

}

else {

if (alpha[i] < 0){

alpha[i] = 0;

alpha[j] = sum;

}

}

}

// 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;

}

// 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)) {

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)){

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];

}

}

}

if (iter >= max\_iter){

if (active\_size < l) {

// reconstruct the whole gradient to calculate objective value

reconstruct\_gradient();

active\_size = l;

info("\*");

}

fprintf(stderr, "\nWARNING: reaching max number of iterations\n");

}

// calculate rho

si->rho = calculate\_rho();

// calculate objective value

{

double v = 0;

int i;

for (i = 0; i<l; i++)

v += alpha[i] \* (G[i] + p[i]);

si->obj = v / 2;

}

// put back the solution

{

for (int i = 0; i<l; i++)

alpha\_[active\_set[i]] = alpha[i];

}

// juggle everything back

si->upper\_bound\_p = Cp;

si->upper\_bound\_n = Cn;

info("\noptimization finished, #iter = %d\n", iter);

delete[] p;

delete[] y;

delete[] alpha;

delete[] alpha\_status;

delete[] active\_set;

delete[] G;

delete[] G\_bar;

}

// return 1 if already optimal, return 0 otherwise

int Solver::select\_working\_set(int &out\_i, int &out\_j) {

// return i,j such that

// i: maximizes -y\_i \* grad(f)\_i, i in I\_up(\alpha)

// j: minimizes the decrease of obj value

// (if quadratic coefficeint <= 0, replace it with tau)

// -y\_j\*grad(f)\_j < -y\_i\*grad(f)\_i, j in I\_low(\alpha)

double Gmax = -INF;

double Gmax2 = -INF;

int Gmax\_idx = -1;

int Gmin\_idx = -1;

double obj\_diff\_min = INF;

for (int t = 0; t<active\_size; t++)

if (y[t] == +1) {

if (!is\_upper\_bound(t))

if (-G[t] >= Gmax) {

Gmax = -G[t];

Gmax\_idx = t;

}

}

else {

if (!is\_lower\_bound(t))

if (G[t] >= Gmax){

Gmax = G[t];

Gmax\_idx = t;

}

}

int i = Gmax\_idx;

const Qfloat \*Q\_i = NULL;

if (i != -1) // NULL Q\_i not accessed: Gmax=-INF if i=-1

Q\_i = Q->get\_Q(i, active\_size);

for (int j = 0; j<active\_size; j++) {

if (y[j] == +1){

if (!is\_lower\_bound(j)){

double grad\_diff = Gmax + G[j];

if (G[j] >= Gmax2)

Gmax2 = G[j];

if (grad\_diff > 0){

double obj\_diff;

double quad\_coef = QD[i] + QD[j] - 2.0\*y[i] \* Q\_i[j];

if (quad\_coef > 0)

obj\_diff = -(grad\_diff\*grad\_diff) / quad\_coef;

else

obj\_diff = -(grad\_diff\*grad\_diff) / TAU;

if (obj\_diff <= obj\_diff\_min){

Gmin\_idx = j;

obj\_diff\_min = obj\_diff;

}

}

}

}

else{

if (!is\_upper\_bound(j)){

double grad\_diff = Gmax - G[j];

if (-G[j] >= Gmax2)

Gmax2 = -G[j];

if (grad\_diff > 0){

double obj\_diff;

double quad\_coef = QD[i] + QD[j] + 2.0\*y[i] \* Q\_i[j];

if (quad\_coef > 0)

obj\_diff = -(grad\_diff\*grad\_diff) / quad\_coef;

else

obj\_diff = -(grad\_diff\*grad\_diff) / TAU;

if (obj\_diff <= obj\_diff\_min){

Gmin\_idx = j;

obj\_diff\_min = obj\_diff;

}

}

}

}

}

if (Gmax + Gmax2 < eps || Gmin\_idx == -1)

return 1;

out\_i = Gmax\_idx;

out\_j = Gmin\_idx;

return 0;

}

bool Solver::be\_shrunk(int i, double Gmax1, double Gmax2){

if (is\_upper\_bound(i)){

if (y[i] == +1)

return(-G[i] > Gmax1);

else

return(-G[i] > Gmax2);

}

else if (is\_lower\_bound(i)){

if (y[i] == +1)

return(G[i] > Gmax2);

else

return(G[i] > Gmax1);

}

else

return(false);

}

void Solver::do\_shrinking(){

int i;

double Gmax1 = -INF; // max { -y\_i \* grad(f)\_i | i in I\_up(\alpha) }

double Gmax2 = -INF; // max { y\_i \* grad(f)\_i | i in I\_low(\alpha) }

// find maximal violating pair first

for (i = 0; i<active\_size; i++) {

if (y[i] == +1){

if (!is\_upper\_bound(i)){

if (-G[i] >= Gmax1)

Gmax1 = -G[i];

}

if (!is\_lower\_bound(i)){

if (G[i] >= Gmax2)

Gmax2 = G[i];

}

}

else{

if (!is\_upper\_bound(i)){

if (-G[i] >= Gmax2)

Gmax2 = -G[i];

}

if (!is\_lower\_bound(i)){

if (G[i] >= Gmax1)

Gmax1 = G[i];

}

}

}

if (unshrink == false && Gmax1 + Gmax2 <= eps \* 10){

unshrink = true;

reconstruct\_gradient();

active\_size = l;

info("\*");

}

for (i = 0; i<active\_size; i++)

if (be\_shrunk(i, Gmax1, Gmax2)){

active\_size--;

while (active\_size > i){

if (!be\_shrunk(active\_size, Gmax1, Gmax2)){

swap\_index(i, active\_size);

break;

}

active\_size--;

}

}

}

double Solver::calculate\_rho(){

double r;

int nr\_free = 0;

double ub = INF, lb = -INF, sum\_free = 0;

for (int i = 0; i<active\_size; i++){

double yG = y[i] \* G[i];

if (is\_upper\_bound(i)){

if (y[i] == -1)

ub = min(ub, yG);

else

lb = max(lb, yG);

}

else if (is\_lower\_bound(i)){

if (y[i] == +1)

ub = min(ub, yG);

else

lb = max(lb, yG);

}

else{

++nr\_free;

sum\_free += yG;

}

}

if (nr\_free>0)

r = sum\_free / nr\_free;

else

r = (ub + lb) / 2;

return r;

}

// Solver for nu-svm classification and regression

// additional constraint: e^T \alpha = constant

class Solver\_NU : public Solver{

public:

Solver\_NU() {}

void Solve(int l, const QMatrix& Q, const double \*p, const schar \*y,

double \*alpha, double Cp, double Cn, double eps,

SolutionInfo\* si, int shrinking){

this->si = si;

Solver::Solve(l, Q, p, y, alpha, Cp, Cn, eps, si, shrinking);

}

private:

SolutionInfo \*si;

int select\_working\_set(int &i, int &j);

double calculate\_rho();

bool be\_shrunk(int i, double Gmax1, double Gmax2, double Gmax3, double Gmax4);

void do\_shrinking();

};

// return 1 if already optimal, return 0 otherwise

int Solver\_NU::select\_working\_set(int &out\_i, int &out\_j){

// return i,j such that y\_i = y\_j and

// i: maximizes -y\_i \* grad(f)\_i, i in I\_up(\alpha)

// j: minimizes the decrease of obj value

// (if quadratic coefficeint <= 0, replace it with tau)

// -y\_j\*grad(f)\_j < -y\_i\*grad(f)\_i, j in I\_low(\alpha)

double Gmaxp = -INF;

double Gmaxp2 = -INF;

int Gmaxp\_idx = -1;

double Gmaxn = -INF;

double Gmaxn2 = -INF;

int Gmaxn\_idx = -1;

int Gmin\_idx = -1;

double obj\_diff\_min = INF;

for (int t = 0; t<active\_size; t++)

if (y[t] == +1){

if (!is\_upper\_bound(t))

if (-G[t] >= Gmaxp){

Gmaxp = -G[t];

Gmaxp\_idx = t;

}

}

else{

if (!is\_lower\_bound(t))

if (G[t] >= Gmaxn){

Gmaxn = G[t];

Gmaxn\_idx = t;

}

}

int ip = Gmaxp\_idx;

int in = Gmaxn\_idx;

const Qfloat \*Q\_ip = NULL;

const Qfloat \*Q\_in = NULL;

if (ip != -1) // NULL Q\_ip not accessed: Gmaxp=-INF if ip=-1

Q\_ip = Q->get\_Q(ip, active\_size);

if (in != -1)

Q\_in = Q->get\_Q(in, active\_size);

for (int j = 0; j<active\_size; j++){

if (y[j] == +1){

if (!is\_lower\_bound(j)){

double grad\_diff = Gmaxp + G[j];

if (G[j] >= Gmaxp2)

Gmaxp2 = G[j];

if (grad\_diff > 0){

double obj\_diff;

double quad\_coef = QD[ip] + QD[j] - 2 \* Q\_ip[j];

if (quad\_coef > 0)

obj\_diff = -(grad\_diff\*grad\_diff) / quad\_coef;

else

obj\_diff = -(grad\_diff\*grad\_diff) / TAU;

if (obj\_diff <= obj\_diff\_min){

Gmin\_idx = j;

obj\_diff\_min = obj\_diff;

}

}

}

}

else{

if (!is\_upper\_bound(j)){

double grad\_diff = Gmaxn - G[j];

if (-G[j] >= Gmaxn2)

Gmaxn2 = -G[j];

if (grad\_diff > 0){

double obj\_diff;

double quad\_coef = QD[in] + QD[j] - 2 \* Q\_in[j];

if (quad\_coef > 0)

obj\_diff = -(grad\_diff\*grad\_diff) / quad\_coef;

else

obj\_diff = -(grad\_diff\*grad\_diff) / TAU;

if (obj\_diff <= obj\_diff\_min){

Gmin\_idx = j;

obj\_diff\_min = obj\_diff;

}

}

}

}

}

if (max(Gmaxp + Gmaxp2, Gmaxn + Gmaxn2) < eps || Gmin\_idx == -1)

return 1;

if (y[Gmin\_idx] == +1)

out\_i = Gmaxp\_idx;

else

out\_i = Gmaxn\_idx;

out\_j = Gmin\_idx;

return 0;

}

bool Solver\_NU::be\_shrunk(int i, double Gmax1, double Gmax2, double Gmax3, double Gmax4){

if (is\_upper\_bound(i)){

if (y[i] == +1)

return(-G[i] > Gmax1);

else

return(-G[i] > Gmax4);

}

else if (is\_lower\_bound(i)){

if (y[i] == +1)

return(G[i] > Gmax2);

else

return(G[i] > Gmax3);

}

else

return(false);

}

void Solver\_NU::do\_shrinking(){

double Gmax1 = -INF; // max { -y\_i \* grad(f)\_i | y\_i = +1, i in I\_up(\alpha) }

double Gmax2 = -INF; // max { y\_i \* grad(f)\_i | y\_i = +1, i in I\_low(\alpha) }

double Gmax3 = -INF; // max { -y\_i \* grad(f)\_i | y\_i = -1, i in I\_up(\alpha) }

double Gmax4 = -INF; // max { y\_i \* grad(f)\_i | y\_i = -1, i in I\_low(\alpha) }

// find maximal violating pair first

int i;

for (i = 0; i<active\_size; i++){

if (!is\_upper\_bound(i)){

if (y[i] == +1){

if (-G[i] > Gmax1) Gmax1 = -G[i];

}

else if (-G[i] > Gmax4) Gmax4 = -G[i];

}

if (!is\_lower\_bound(i)){

if (y[i] == +1){

if (G[i] > Gmax2) Gmax2 = G[i];

}

else if (G[i] > Gmax3) Gmax3 = G[i];

}

}

if (unshrink == false && max(Gmax1 + Gmax2, Gmax3 + Gmax4) <= eps \* 10){

unshrink = true;

reconstruct\_gradient();

active\_size = l;

}

for (i = 0; i<active\_size; i++)

if (be\_shrunk(i, Gmax1, Gmax2, Gmax3, Gmax4)){

active\_size--;

while (active\_size > i){

if (!be\_shrunk(active\_size, Gmax1, Gmax2, Gmax3, Gmax4)){

swap\_index(i, active\_size);

break;

}

active\_size--;

}

}

}

double Solver\_NU::calculate\_rho(){

int nr\_free1 = 0, nr\_free2 = 0;

double ub1 = INF, ub2 = INF;

double lb1 = -INF, lb2 = -INF;

double sum\_free1 = 0, sum\_free2 = 0;

for (int i = 0; i<active\_size; i++){

if (y[i] == +1){

if (is\_upper\_bound(i))

lb1 = max(lb1, G[i]);

else if (is\_lower\_bound(i))

ub1 = min(ub1, G[i]);

else{

++nr\_free1;

sum\_free1 += G[i];

}

}

else{

if (is\_upper\_bound(i))

lb2 = max(lb2, G[i]);

else if (is\_lower\_bound(i))

ub2 = min(ub2, G[i]);

else{

++nr\_free2;

sum\_free2 += G[i];

}

}

}

double r1, r2;

if (nr\_free1 > 0)

r1 = sum\_free1 / nr\_free1;

else

r1 = (ub1 + lb1) / 2;

if (nr\_free2 > 0)

r2 = sum\_free2 / nr\_free2;

else

r2 = (ub2 + lb2) / 2;

si->r = (r1 + r2) / 2;

return (r1 - r2) / 2;

}

// Q matrices for various formulations

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, (long int)(param.cache\_size\*(1 << 20)));

QD = new double[prob.l];

for (int i = 0; i<prob.l; i++)

QD[i] = (this->\*kernel\_function)(i, i);

}

Qfloat \*get\_Q(int i, int len) const{

Qfloat \*data;

int start, j;

if ((start = cache->get\_data(i, &data, len)) < len){

for (j = start; j<len; j++)

data[j] = (Qfloat)(y[i] \* y[j] \* (this->\*kernel\_function)(i, j));

}

return data;

}

double \*get\_QD() const{

return QD;

}

void swap\_index(int i, int j) const{

cache->swap\_index(i, j);

Kernel::swap\_index(i, j);

swap(y[i], y[j]);

swap(QD[i], QD[j]);

}

~SVC\_Q(){

delete[] y;

delete cache;

delete[] QD;

}

private:

schar \*y;

Cache \*cache;

double \*QD;

};

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, (long int)(param.cache\_size\*(1 << 20)));

QD = new double[prob.l];

for (int i = 0; i<prob.l; i++)

QD[i] = (this->\*kernel\_function)(i, i);

}

Qfloat \*get\_Q(int i, int len) const{

Qfloat \*data;

int start, j;

if ((start = cache->get\_data(i, &data, len)) < len){

for (j = start; j<len; j++)

data[j] = (Qfloat)(this->\*kernel\_function)(i, j);

}

return data;

}

double \*get\_QD() const{

return QD;

}

void swap\_index(int i, int j) const{

cache->swap\_index(i, j);

Kernel::swap\_index(i, j);

swap(QD[i], QD[j]);

}

~ONE\_CLASS\_Q(){

delete cache;

delete[] QD;

}

private:

Cache \*cache;

double \*QD;

};

class SVR\_Q : public Kernel{

public:

SVR\_Q(const svm\_problem& prob, const svm\_parameter& param)

:Kernel(prob.l, prob.x, param){

l = prob.l;

cache = new Cache(l, (long int)(param.cache\_size\*(1 << 20)));

QD = new double[2 \* l];

sign = new schar[2 \* l];

index = new int[2 \* l];

for (int k = 0; k<l; k++){

sign[k] = 1;

sign[k + l] = -1;

index[k] = k;

index[k + l] = k;

QD[k] = (this->\*kernel\_function)(k, k);

QD[k + l] = QD[k];

}

buffer[0] = new Qfloat[2 \* l];

buffer[1] = new Qfloat[2 \* l];

next\_buffer = 0;

}

void swap\_index(int i, int j) const{

swap(sign[i], sign[j]);

swap(index[i], index[j]);

swap(QD[i], QD[j]);

}

Qfloat \*get\_Q(int i, int len) const{

Qfloat \*data;

int j, real\_i = index[i];

if (cache->get\_data(real\_i, &data, l) < l){

for (j = 0; j<l; j++)

data[j] = (Qfloat)(this->\*kernel\_function)(real\_i, j);

}

// reorder and copy

Qfloat \*buf = buffer[next\_buffer];

next\_buffer = 1 - next\_buffer;

schar si = sign[i];

for (j = 0; j<len; j++)

buf[j] = (Qfloat)si \* (Qfloat)sign[j] \* data[index[j]];

return buf;

}

double \*get\_QD() const{

return QD;

}

~SVR\_Q(){

delete cache;

delete[] sign;

delete[] index;

delete[] buffer[0];

delete[] buffer[1];

delete[] QD;

}

private:

int l;

Cache \*cache;

schar \*sign;

int \*index;

mutable int next\_buffer;

Qfloat \*buffer[2];

double \*QD;

};

// construct and solve various formulations

static void solve\_c\_svc(

const svm\_problem \*prob, const svm\_parameter\* param,

double \*alpha, Solver::SolutionInfo\* si, double Cp, double Cn){

int l = prob->l;

double \*minus\_ones = new double[l];

schar \*y = new schar[l];

int i;

for (i = 0; i<l; i++){

alpha[i] = 0;

minus\_ones[i] = -1;

if (prob->y[i] > 0) y[i] = +1; else y[i] = -1;

}

Solver s;

s.Solve(l, SVC\_Q(\*prob, \*param, y), minus\_ones, y,

alpha, Cp, Cn, param->eps, si, param->shrinking);

double sum\_alpha = 0;

for (i = 0; i<l; i++)

sum\_alpha += alpha[i];

if (Cp == Cn)

info("nu = %f\n", sum\_alpha / (Cp\*prob->l));

for (i = 0; i<l; i++)

alpha[i] \*= y[i];

delete[] minus\_ones;

delete[] y;

}

static void solve\_nu\_svc(

const svm\_problem \*prob, const svm\_parameter \*param,

double \*alpha, Solver::SolutionInfo\* si){

int i;

int l = prob->l;

double nu = param->nu;

schar \*y = new schar[l];

for (i = 0; i<l; i++)

if (prob->y[i]>0)

y[i] = +1;

else

y[i] = -1;

double sum\_pos = nu\*l / 2;

double sum\_neg = nu\*l / 2;

for (i = 0; i<l; i++)

if (y[i] == +1){

alpha[i] = min(1.0, sum\_pos);

sum\_pos -= alpha[i];

}

else{

alpha[i] = min(1.0, sum\_neg);

sum\_neg -= alpha[i];

}

double \*zeros = new double[l];

for (i = 0; i<l; i++)

zeros[i] = 0;

Solver\_NU s;

s.Solve(l, SVC\_Q(\*prob, \*param, y), zeros, y,

alpha, 1.0, 1.0, param->eps, si, param->shrinking);

double r = si->r;

info("C = %f\n", 1 / r);

for (i = 0; i<l; i++)

alpha[i] \*= y[i] / r;

si->rho /= r;

si->obj /= (r\*r);

si->upper\_bound\_p = 1 / r;

si->upper\_bound\_n = 1 / r;

delete[] y;

delete[] zeros;

}

static void solve\_one\_class(

const svm\_problem \*prob, const svm\_parameter \*param,

double \*alpha, Solver::SolutionInfo\* si){

int l = prob->l;

double \*zeros = new double[l];

schar \*ones = new schar[l];

int i;

int n = (int)(param->nu\*prob->l); // # of alpha's at upper bound

for (i = 0; i<n; i++)

alpha[i] = 1;

if (n<prob->l)

alpha[n] = param->nu \* prob->l - n;

for (i = n + 1; i<l; i++)

alpha[i] = 0;

for (i = 0; i<l; i++){

zeros[i] = 0;

ones[i] = 1;

}

Solver s;

s.Solve(l, ONE\_CLASS\_Q(\*prob, \*param), zeros, ones,

alpha, 1.0, 1.0, param->eps, si, param->shrinking);

delete[] zeros;

delete[] ones;

}

static void solve\_epsilon\_svr(

const svm\_problem \*prob, const svm\_parameter \*param,

double \*alpha, Solver::SolutionInfo\* si){

int l = prob->l;

double \*alpha2 = new double[2 \* l];

double \*linear\_term = new double[2 \* l];

schar \*y = new schar[2 \* l];

int i;

for (i = 0; i<l; i++){

alpha2[i] = 0;

linear\_term[i] = param->p - prob->y[i];

y[i] = 1;

alpha2[i + l] = 0;

linear\_term[i + l] = param->p + prob->y[i];

y[i + l] = -1;

}

Solver s;

s.Solve(2 \* l, SVR\_Q(\*prob, \*param), linear\_term, y,

alpha2, param->C, param->C, param->eps, si, param->shrinking);

double sum\_alpha = 0;

for (i = 0; i<l; i++){

alpha[i] = alpha2[i] - alpha2[i + l];

sum\_alpha += fabs(alpha[i]);

}

info("nu = %f\n", sum\_alpha / (param->C\*l));

delete[] alpha2;

delete[] linear\_term;

delete[] y;

}

static void solve\_nu\_svr(

const svm\_problem \*prob, const svm\_parameter \*param,

double \*alpha, Solver::SolutionInfo\* si){

int l = prob->l;

double C = param->C;

double \*alpha2 = new double[2 \* l];

double \*linear\_term = new double[2 \* l];

schar \*y = new schar[2 \* l];

int i;

double sum = C \* param->nu \* l / 2;

for (i = 0; i<l; i++){

alpha2[i] = alpha2[i + l] = min(sum, C);

sum -= alpha2[i];

linear\_term[i] = -prob->y[i];

y[i] = 1;

linear\_term[i + l] = prob->y[i];

y[i + l] = -1;

}

Solver\_NU s;

s.Solve(2 \* l, SVR\_Q(\*prob, \*param), linear\_term, y,

alpha2, C, C, param->eps, si, param->shrinking);

info("epsilon = %f\n", -si->r);

for (i = 0; i<l; i++)

alpha[i] = alpha2[i] - alpha2[i + l];

delete[] alpha2;

delete[] linear\_term;

delete[] y;

}

// decision\_function

struct decision\_function{

double \*alpha;

double rho;

};

static decision\_function svm\_train\_one(

const svm\_problem \*prob, const svm\_parameter \*param,

double Cp, double Cn){

double \*alpha = Malloc(double, prob->l);

Solver::SolutionInfo si;

switch (param->svm\_type){

case C\_SVC:

solve\_c\_svc(prob, param, alpha, &si, Cp, Cn);

break;

case NU\_SVC:

solve\_nu\_svc(prob, param, alpha, &si);

break;

case ONE\_CLASS:

solve\_one\_class(prob, param, alpha, &si);

break;

case EPSILON\_SVR:

solve\_epsilon\_svr(prob, param, alpha, &si);

break;

case NU\_SVR:

solve\_nu\_svr(prob, param, alpha, &si);

break;

}

info("obj = %f, rho = %f\n", si.obj, si.rho);

// output SVs

int nSV = 0;

int nBSV = 0;

for (int i = 0; i<prob->l; i++){

if (fabs(alpha[i]) > 0){

++nSV;

if (prob->y[i] > 0){

if (fabs(alpha[i]) >= si.upper\_bound\_p)

++nBSV;

}

else{

if (fabs(alpha[i]) >= si.upper\_bound\_n)

++nBSV;

}

}

}

info("nSV = %d, nBSV = %d\n", nSV, nBSV);

decision\_function f;

f.alpha = alpha;

f.rho = si.rho;

return f;

}

// Platt's binary SVM Probablistic Output: an improvement from Lin et al.

static void sigmoid\_train(

int l, const double \*dec\_values, const double \*labels,

double& A, double& B){

double prior1 = 0, prior0 = 0;

int i;

for (i = 0; i<l; i++)

if (labels[i] > 0) prior1 += 1;

else prior0 += 1;

int max\_iter = 100; // Maximal number of iterations

double min\_step = 1e-10; // Minimal step taken in line search

double sigma = 1e-12; // For numerically strict PD of Hessian

double eps = 1e-5;

double hiTarget = (prior1 + 1.0) / (prior1 + 2.0);

double loTarget = 1 / (prior0 + 2.0);

double \*t = Malloc(double, l);

double fApB, p, q, h11, h22, h21, g1, g2, det, dA, dB, gd, stepsize;

double newA, newB, newf, d1, d2;

int iter;

// Initial Point and Initial Fun Value

A = 0.0; B = log((prior0 + 1.0) / (prior1 + 1.0));

double fval = 0.0;

for (i = 0; i<l; i++){

if (labels[i]>0) t[i] = hiTarget;

else t[i] = loTarget;

fApB = dec\_values[i] \* A + B;

if (fApB >= 0)

fval += t[i] \* fApB + log(1 + exp(-fApB));

else

fval += (t[i] - 1)\*fApB + log(1 + exp(fApB));

}

for (iter = 0; iter<max\_iter; iter++){

// Update Gradient and Hessian (use H' = H + sigma I)

h11 = sigma; // numerically ensures strict PD

h22 = sigma;

h21 = 0.0; g1 = 0.0; g2 = 0.0;

for (i = 0; i<l; i++){

fApB = dec\_values[i] \* A + B;

if (fApB >= 0){

p = exp(-fApB) / (1.0 + exp(-fApB));

q = 1.0 / (1.0 + exp(-fApB));

}

else{

p = 1.0 / (1.0 + exp(fApB));

q = exp(fApB) / (1.0 + exp(fApB));

}

d2 = p\*q;

h11 += dec\_values[i] \* dec\_values[i] \* d2;

h22 += d2;

h21 += dec\_values[i] \* d2;

d1 = t[i] - p;

g1 += dec\_values[i] \* d1;

g2 += d1;

}

// Stopping Criteria

if (fabs(g1)<eps && fabs(g2)<eps)

break;

// Finding Newton direction: -inv(H') \* g

det = h11\*h22 - h21\*h21;

dA = -(h22\*g1 - h21 \* g2) / det;

dB = -(-h21\*g1 + h11 \* g2) / det;

gd = g1\*dA + g2\*dB;

stepsize = 1; // Line Search

while (stepsize >= min\_step){

newA = A + stepsize \* dA;

newB = B + stepsize \* dB;

// New function value

newf = 0.0;

for (i = 0; i<l; i++){

fApB = dec\_values[i] \* newA + newB;

if (fApB >= 0)

newf += t[i] \* fApB + log(1 + exp(-fApB));

else

newf += (t[i] - 1)\*fApB + log(1 + exp(fApB));

}

// Check sufficient decrease

if (newf<fval + 0.0001\*stepsize\*gd){

A = newA; B = newB; fval = newf;

break;

}

else

stepsize = stepsize / 2.0;

}

if (stepsize < min\_step){

info("Line search fails in two-class probability estimates\n");

break;

}

}

if (iter >= max\_iter)

info("Reaching maximal iterations in two-class probability estimates\n");

free(t);

}

static double sigmoid\_predict(double decision\_value, double A, double B){

double fApB = decision\_value\*A + B;

// 1-p used later; avoid catastrophic cancellation

if (fApB >= 0)

return exp(-fApB) / (1.0 + exp(-fApB));

else

return 1.0 / (1 + exp(fApB));

}

// Method 2 from the multiclass\_prob paper by Wu, Lin, and Weng

static void multiclass\_probability(int k, double \*\*r, double \*p){

int t, j;

int iter = 0, max\_iter = max(100, k);

double \*\*Q = Malloc(double \*, k);

double \*Qp = Malloc(double, k);

double pQp, eps = 0.005 / k;

for (t = 0; t<k; t++){

p[t] = 1.0 / k; // Valid if k = 1

Q[t] = Malloc(double, k);

Q[t][t] = 0;

for (j = 0; j<t; j++){

Q[t][t] += r[j][t] \* r[j][t];

Q[t][j] = Q[j][t];

}

for (j = t + 1; j<k; j++){

Q[t][t] += r[j][t] \* r[j][t];

Q[t][j] = -r[j][t] \* r[t][j];

}

}

for (iter = 0; iter<max\_iter; iter++){

// stopping condition, recalculate QP,pQP for numerical accuracy

pQp = 0;

for (t = 0; t<k; t++){

Qp[t] = 0;

for (j = 0; j<k; j++)

Qp[t] += Q[t][j] \* p[j];

pQp += p[t] \* Qp[t];

}

double max\_error = 0;

for (t = 0; t<k; t++){

double error = fabs(Qp[t] - pQp);

if (error>max\_error)

max\_error = error;

}

if (max\_error<eps) break;

for (t = 0; t<k; t++){

double diff = (-Qp[t] + pQp) / Q[t][t];

p[t] += diff;

pQp = (pQp + diff\*(diff\*Q[t][t] + 2 \* Qp[t])) / (1 + diff) / (1 + diff);

for (j = 0; j<k; j++){

Qp[j] = (Qp[j] + diff\*Q[t][j]) / (1 + diff);

p[j] /= (1 + diff);

}

}

}

if (iter >= max\_iter)

info("Exceeds max\_iter in multiclass\_prob\n");

for (t = 0; t<k; t++) free(Q[t]);

free(Q);

free(Qp);

}

// Cross-validation decision values for probability estimates

static void svm\_binary\_svc\_probability(

const svm\_problem \*prob, const svm\_parameter \*param,

double Cp, double Cn, double& probA, double& probB){

int i;

int nr\_fold = 5;

int \*perm = Malloc(int, prob->l);

double \*dec\_values = Malloc(double, prob->l);

// random shuffle

for (i = 0; i<prob->l; i++) perm[i] = i;

for (i = 0; i<prob->l; i++){

int j = i + rand() % (prob->l - i);

swap(perm[i], perm[j]);

}

for (i = 0; i<nr\_fold; i++){

int begin = i\*prob->l / nr\_fold;

int end = (i + 1)\*prob->l / nr\_fold;

int j, k;

struct svm\_problem subprob;

subprob.l = prob->l - (end - begin);

subprob.x = Malloc(struct svm\_node\*, subprob.l);

subprob.y = Malloc(double, subprob.l);

k = 0;

for (j = 0; j<begin; j++){

subprob.x[k] = prob->x[perm[j]];

subprob.y[k] = prob->y[perm[j]];

++k;

}

for (j = end; j<prob->l; j++){

subprob.x[k] = prob->x[perm[j]];

subprob.y[k] = prob->y[perm[j]];

++k;

}

int p\_count = 0, n\_count = 0;

for (j = 0; j<k; j++)

if (subprob.y[j]>0)

p\_count++;

else

n\_count++;

if (p\_count == 0 && n\_count == 0)

for (j = begin; j<end; j++)

dec\_values[perm[j]] = 0;

else if (p\_count > 0 && n\_count == 0)

for (j = begin; j<end; j++)

dec\_values[perm[j]] = 1;

else if (p\_count == 0 && n\_count > 0)

for (j = begin; j<end; j++)

dec\_values[perm[j]] = -1;

else{

svm\_parameter subparam = \*param;

subparam.probability = 0;

subparam.C = 1.0;

subparam.nr\_weight = 2;

subparam.weight\_label = Malloc(int, 2);

subparam.weight = Malloc(double, 2);

subparam.weight\_label[0] = +1;

subparam.weight\_label[1] = -1;

subparam.weight[0] = Cp;

subparam.weight[1] = Cn;

struct svm\_model \*submodel = svm\_train(&subprob, &subparam);

for (j = begin; j<end; j++){

svm\_predict\_values(submodel, prob->x[perm[j]],

&(dec\_values[perm[j]]));

// ensure +1 -1 order; reason not using CV subroutine

dec\_values[perm[j]] \*= submodel->label[0];

}

svm\_free\_and\_destroy\_model(&submodel);

svm\_destroy\_param(&subparam);

}

free(subprob.x);

free(subprob.y);

}

sigmoid\_train(prob->l, dec\_values, prob->y, probA, probB);

free(dec\_values);

free(perm);

}

// Return parameter of a Laplace distribution

static double svm\_svr\_probability(

const svm\_problem \*prob, const svm\_parameter \*param){

int i;

int nr\_fold = 5;

double \*ymv = Malloc(double, prob->l);

double mae = 0;

svm\_parameter newparam = \*param;

newparam.probability = 0;

svm\_cross\_validation(prob, &newparam, nr\_fold, ymv);

for (i = 0; i<prob->l; i++){

ymv[i] = prob->y[i] - ymv[i];

mae += fabs(ymv[i]);

}

mae /= prob->l;

double std = sqrt(2 \* mae\*mae);

int count = 0;

mae = 0;

for (i = 0; i<prob->l; i++)

if (fabs(ymv[i]) > 5 \* std)

count = count + 1;

else

mae += fabs(ymv[i]);

mae /= (prob->l - count);

info("Prob. model for test data: target value = predicted value + z,\nz: Laplace distribution e^(-|z|/sigma)/(2sigma),sigma= %g\n", mae);

free(ymv);

return mae;

}

// label: label name, start: begin of each class, count: #data of classes, perm: indices to the original data

// perm, length l, must be allocated before calling this subroutine

static void svm\_group\_classes(const svm\_problem \*prob, int \*nr\_class\_ret, int \*\*label\_ret, int \*\*start\_ret, int \*\*count\_ret, int \*perm){

int l = prob->l;

int max\_nr\_class = 16;

int nr\_class = 0;

int \*label = Malloc(int, max\_nr\_class);

int \*count = Malloc(int, max\_nr\_class);

int \*data\_label = Malloc(int, l);

int i;

for (i = 0; i<l; i++){

int this\_label = (int)prob->y[i];

int j;

for (j = 0; j<nr\_class; j++){

if (this\_label == label[j]){

++count[j];

break;

}

}

data\_label[i] = j;

if (j == nr\_class){

if (nr\_class == max\_nr\_class){

max\_nr\_class \*= 2;

label = (int \*)realloc(label, max\_nr\_class\*sizeof(int));

count = (int \*)realloc(count, max\_nr\_class\*sizeof(int));

}

label[nr\_class] = this\_label;

count[nr\_class] = 1;

++nr\_class;

}

}

// Labels are ordered by their first occurrence in the training set.

// However, for two-class sets with -1/+1 labels and -1 appears first,

// we swap labels to ensure that internally the binary SVM has positive data

// corresponding to the +1 instances.

if (nr\_class == 2 && label[0] == -1 && label[1] == 1){

swap(label[0], label[1]);

swap(count[0], count[1]);

for (i = 0; i<l; i++){

if (data\_label[i] == 0)

data\_label[i] = 1;

else

data\_label[i] = 0;

}

}

int \*start = Malloc(int, nr\_class);

start[0] = 0;

for (i = 1; i<nr\_class; i++)

start[i] = start[i - 1] + count[i - 1];

for (i = 0; i<l; i++){

perm[start[data\_label[i]]] = i;

++start[data\_label[i]];

}

start[0] = 0;

for (i = 1; i<nr\_class; i++)

start[i] = start[i - 1] + count[i - 1];

\*nr\_class\_ret = nr\_class;

\*label\_ret = label;

\*start\_ret = start;

\*count\_ret = count;

free(data\_label);

}

// Interface functions

svm\_model \*svm\_train(const svm\_problem \*prob, const svm\_parameter \*param){

svm\_model \*model = Malloc(svm\_model, 1);

model->param = \*param;

model->free\_sv = 0; // XXX

if (param->svm\_type == ONE\_CLASS ||

param->svm\_type == EPSILON\_SVR ||

param->svm\_type == NU\_SVR){

// regression or one-class-svm

model->nr\_class = 2;

model->label = NULL;

model->nSV = NULL;

model->probA = NULL; model->probB = NULL;

model->sv\_coef = Malloc(double \*, 1);

if (param->probability &&

(param->svm\_type == EPSILON\_SVR ||

param->svm\_type == NU\_SVR)) {

model->probA = Malloc(double, 1);

model->probA[0] = svm\_svr\_probability(prob, param);

}

decision\_function f = svm\_train\_one(prob, param, 0, 0);

model->rho = Malloc(double, 1);

model->rho[0] = f.rho;

int nSV = 0;

int i;

for (i = 0; i<prob->l; i++)

if (fabs(f.alpha[i]) > 0) ++nSV;

model->l = nSV;

model->SV = Malloc(svm\_node \*, nSV);

model->sv\_coef[0] = Malloc(double, nSV);

model->sv\_indices = Malloc(int, nSV);

int j = 0;

for (i = 0; i<prob->l; i++)

if (fabs(f.alpha[i]) > 0){

model->SV[j] = prob->x[i];

model->sv\_coef[0][j] = f.alpha[i];

model->sv\_indices[j] = i + 1;

++j;

}

free(f.alpha);

}

else{

// classification

int l = prob->l;

int nr\_class;

int \*label = NULL;

int \*start = NULL;

int \*count = NULL;

int \*perm = Malloc(int, l);

// group training data of the same class

svm\_group\_classes(prob, &nr\_class, &label, &start, &count, perm);

if (nr\_class == 1)

info("WARNING: training data in only one class. See README for

details.\n");

svm\_node \*\*x = Malloc(svm\_node \*, l);

int i;

for (i = 0; i<l; i++)

x[i] = prob->x[perm[i]];

// calculate weighted C

double \*weighted\_C = Malloc(double, nr\_class);

for (i = 0; i<nr\_class; i++)

weighted\_C[i] = param->C;

for (i = 0; i<param->nr\_weight; i++){

int j;

for (j = 0; j<nr\_class; j++)

if (param->weight\_label[i] == label[j])

break;

if (j == nr\_class)

fprintf(stderr, "WARNING: class label %d specified in weight is not

found\n", param->weight\_label[i]);

else

weighted\_C[j] \*= param->weight[i];

}

// train k\*(k-1)/2 models

bool \*nonzero = Malloc(bool, l);

for (i = 0; i<l; i++)

nonzero[i] = false;

decision\_function \*f = Malloc(decision\_function, nr\_class\*(nr\_class - 1) / 2);

double \*probA = NULL, \*probB = NULL;

if (param->probability){

probA = Malloc(double, nr\_class\*(nr\_class - 1) / 2);

probB = Malloc(double, nr\_class\*(nr\_class - 1) / 2);

}

int p = 0;

for (i = 0; i<nr\_class; i++)

for (int j = i + 1; j<nr\_class; j++){

svm\_problem sub\_prob;

int si = start[i], sj = start[j];

int ci = count[i], cj = count[j];

sub\_prob.l = ci + cj;

sub\_prob.x = Malloc(svm\_node \*, sub\_prob.l);

sub\_prob.y = Malloc(double, sub\_prob.l);

int k;

for (k = 0; k<ci; k++){

sub\_prob.x[k] = x[si + k];

sub\_prob.y[k] = +1;

}

for (k = 0; k<cj; k++){

sub\_prob.x[ci + k] = x[sj + k];

sub\_prob.y[ci + k] = -1;

}

if (param->probability)

svm\_binary\_svc\_probability(&sub\_prob, param, weighted\_C[i],

weighted\_C[j], probA[p], probB[p]);

f[p] = svm\_train\_one(&sub\_prob, param, weighted\_C[i],

weighted\_C[j]);

for (k = 0; k<ci; k++)

if (!nonzero[si + k] && fabs(f[p].alpha[k]) > 0)

nonzero[si + k] = true;

for (k = 0; k<cj; k++)

if (!nonzero[sj + k] && fabs(f[p].alpha[ci + k]) > 0)

nonzero[sj + k] = true;

free(sub\_prob.x);

free(sub\_prob.y);

++p;

}

// build output

model->nr\_class = nr\_class;

model->label = Malloc(int, nr\_class);

for (i = 0; i<nr\_class; i++)

model->label[i] = label[i];

model->rho = Malloc(double, nr\_class\*(nr\_class - 1) / 2);

for (i = 0; i<nr\_class\*(nr\_class - 1) / 2; i++)

model->rho[i] = f[i].rho;

if (param->probability){

model->probA = Malloc(double, nr\_class\*(nr\_class - 1) / 2);

model->probB = Malloc(double, nr\_class\*(nr\_class - 1) / 2);

for (i = 0; i<nr\_class\*(nr\_class - 1) / 2; i++){

model->probA[i] = probA[i];

model->probB[i] = probB[i];

}

}

else{

model->probA = NULL;

model->probB = NULL;

}

int total\_sv = 0;

int \*nz\_count = Malloc(int, nr\_class);

model->nSV = Malloc(int, nr\_class);

for (i = 0; i<nr\_class; i++){

int nSV = 0;

for (int j = 0; j<count[i]; j++)

if (nonzero[start[i] + j]){

++nSV;

++total\_sv;

}

model->nSV[i] = nSV;

nz\_count[i] = nSV;

}

info("Total nSV = %d\n", total\_sv);

model->l = total\_sv;

model->SV = Malloc(svm\_node \*, total\_sv);

model->sv\_indices = Malloc(int, total\_sv);

p = 0;

for (i = 0; i<l; i++)

if (nonzero[i]){

model->SV[p] = x[i];

model->sv\_indices[p++] = perm[i] + 1;

}

int \*nz\_start = Malloc(int, nr\_class);

nz\_start[0] = 0;

for (i = 1; i<nr\_class; i++)

nz\_start[i] = nz\_start[i - 1] + nz\_count[i - 1];

model->sv\_coef = Malloc(double \*, nr\_class - 1);

for (i = 0; i<nr\_class - 1; i++)

model->sv\_coef[i] = Malloc(double, total\_sv);

p = 0;

for (i = 0; i<nr\_class; i++)

for (int j = i + 1; j<nr\_class; j++){

// classifier (i,j): coefficients with

// i are in sv\_coef[j-1][nz\_start[i]...],

// j are in sv\_coef[i][nz\_start[j]...]

int si = start[i];

int sj = start[j];

int ci = count[i];

int cj = count[j];

int q = nz\_start[i];

int k;

for (k = 0; k<ci; k++)

if (nonzero[si + k])

model->sv\_coef[j - 1][q++] = f[p].alpha[k];

q = nz\_start[j];

for (k = 0; k<cj; k++)

if (nonzero[sj + k])

model->sv\_coef[i][q++] = f[p].alpha[ci + k];

++p;

}

free(label);

free(probA);

free(probB);

free(count);

free(perm);

free(start);

free(x);

free(weighted\_C);

free(nonzero);

for (i = 0; i<nr\_class\*(nr\_class - 1) / 2; i++)

free(f[i].alpha);

free(f);

free(nz\_count);

free(nz\_start);

}

return model;

}

// Stratified cross validation

void svm\_cross\_validation(const svm\_problem \*prob, const svm\_parameter \*param, int nr\_fold, double \*target){

int i;

int \*fold\_start;

int l = prob->l;

int \*perm = Malloc(int, l);

int nr\_class;

if (nr\_fold > l){

nr\_fold = l;

fprintf(stderr, "WARNING: # folds > # data. Will use # folds = # data instead

(i.e., leave-one-out cross validation)\n");

}

fold\_start = Malloc(int, nr\_fold + 1);

// stratified cv may not give leave-one-out rate

// Each class to l folds -> some folds may have zero elements

if ((param->svm\_type == C\_SVC ||

param->svm\_type == NU\_SVC) && nr\_fold < l) {

int \*start = NULL;

int \*label = NULL;

int \*count = NULL;

svm\_group\_classes(prob, &nr\_class, &label, &start, &count, perm);

// random shuffle and then data grouped by fold using the array perm

int \*fold\_count = Malloc(int, nr\_fold);

int c;

int \*index = Malloc(int, l);

for (i = 0; i<l; i++)

index[i] = perm[i];

for (c = 0; c<nr\_class; c++)

for (i = 0; i<count[c]; i++){

int j = i + rand() % (count[c] - i);

swap(index[start[c] + j], index[start[c] + i]);

}

for (i = 0; i<nr\_fold; i++){

fold\_count[i] = 0;

for (c = 0; c<nr\_class; c++)

fold\_count[i] += (i + 1)\*count[c] / nr\_fold - i\*count[c] / nr\_fold;

}

fold\_start[0] = 0;

for (i = 1; i <= nr\_fold; i++)

fold\_start[i] = fold\_start[i - 1] + fold\_count[i - 1];

for (c = 0; c<nr\_class; c++)

for (i = 0; i<nr\_fold; i++){

int begin = start[c] + i\*count[c] / nr\_fold;

int end = start[c] + (i + 1)\*count[c] / nr\_fold;

for (int j = begin; j<end; j++) {

perm[fold\_start[i]] = index[j];

fold\_start[i]++;

}

}

fold\_start[0] = 0;

for (i = 1; i <= nr\_fold; i++)

fold\_start[i] = fold\_start[i - 1] + fold\_count[i - 1];

free(start);

free(label);

free(count);

free(index);

free(fold\_count);

}

else{

for (i = 0; i<l; i++) perm[i] = i;

for (i = 0; i<l; i++){

int j = i + rand() % (l - i);

swap(perm[i], perm[j]);

}

for (i = 0; i <= nr\_fold; i++)

fold\_start[i] = i\*l / nr\_fold;

}

for (i = 0; i<nr\_fold; i++){

int begin = fold\_start[i];

int end = fold\_start[i + 1];

int j, k;

struct svm\_problem subprob;

subprob.l = l - (end - begin);

subprob.x = Malloc(struct svm\_node\*, subprob.l);

subprob.y = Malloc(double, subprob.l);

k = 0;

for (j = 0; j<begin; j++){

subprob.x[k] = prob->x[perm[j]];

subprob.y[k] = prob->y[perm[j]];

++k;

}

for (j = end; j<l; j++){

subprob.x[k] = prob->x[perm[j]];

subprob.y[k] = prob->y[perm[j]];

++k;

}

struct svm\_model \*submodel = svm\_train(&subprob, param);

if (param->probability &&

(param->svm\_type == C\_SVC || param->svm\_type == NU\_SVC)) {

double \*prob\_estimates = Malloc(double, svm\_get\_nr\_class(submodel));

for (j = begin; j<end; j++)

target[perm[j]] = svm\_predict\_probability(submodel, prob->x[perm[j]],

prob\_estimates);

free(prob\_estimates);

}

else

for (j = begin; j<end; j++)

target[perm[j]] = svm\_predict(submodel, prob->x[perm[j]]);

svm\_free\_and\_destroy\_model(&submodel);

free(subprob.x);

free(subprob.y);

}

free(fold\_start);

free(perm);

}

int svm\_get\_svm\_type(const svm\_model \*model){

return model->param.svm\_type;

}

int svm\_get\_nr\_class(const svm\_model \*model){

return model->nr\_class;

}

void svm\_get\_labels(const svm\_model \*model, int\* label){

if (model->label != NULL)

for (int i = 0; i<model->nr\_class; i++)

label[i] = model->label[i];

}

void svm\_get\_sv\_indices(const svm\_model \*model, int\* indices){

if (model->sv\_indices != NULL)

for (int i = 0; i<model->l; i++)

indices[i] = model->sv\_indices[i];

}

int svm\_get\_nr\_sv(const svm\_model \*model){

return model->l;

}

double svm\_get\_svr\_probability(const svm\_model \*model){

if ((model->param.svm\_type == EPSILON\_SVR || model->param.svm\_type ==

NU\_SVR) && model->probA != NULL)

return model->probA[0];

else {

fprintf(stderr, "Model doesn't contain information for SVR probability

inference\n");

return 0;

}

}

double svm\_predict\_values(const svm\_model \*model, const svm\_node \*x, double\* dec\_values){

int i;

if (model->param.svm\_type == ONE\_CLASS ||

model->param.svm\_type == EPSILON\_SVR ||

model->param.svm\_type == NU\_SVR) {

double \*sv\_coef = model->sv\_coef[0];

double sum = 0;

for (i = 0; i<model->l; i++)

sum += sv\_coef[i] \* Kernel::k\_function(x, model->SV[i], model->param);

sum -= model->rho[0];

\*dec\_values = sum;

if (model->param.svm\_type == ONE\_CLASS)

return (sum>0) ? 1 : -1;

else

return sum;

}

else{

int nr\_class = model->nr\_class;

int l = model->l;

double \*kvalue = Malloc(double, l);

for (i = 0; i<l; i++)

kvalue[i] = Kernel::k\_function(x, model->SV[i], model->param);

int \*start = Malloc(int, nr\_class);

start[0] = 0;

for (i = 1; i<nr\_class; i++)

start[i] = start[i - 1] + model->nSV[i - 1];

int \*vote = Malloc(int, nr\_class);

for (i = 0; i<nr\_class; i++)

vote[i] = 0;

int p = 0;

for (i = 0; i<nr\_class; i++)

for (int j = i + 1; j<nr\_class; j++){

double sum = 0;

int si = start[i];

int sj = start[j];

int ci = model->nSV[i];

int cj = model->nSV[j];

int k;

double \*coef1 = model->sv\_coef[j - 1];

double \*coef2 = model->sv\_coef[i];

for (k = 0; k<ci; k++)

sum += coef1[si + k] \* kvalue[si + k];

for (k = 0; k<cj; k++)

sum += coef2[sj + k] \* kvalue[sj + k];

sum -= model->rho[p];

dec\_values[p] = sum;

if (dec\_values[p] > 0)

++vote[i];

else

++vote[j];

p++;

}

int vote\_max\_idx = 0;

for (i = 1; i<nr\_class; i++)

if (vote[i] > vote[vote\_max\_idx])

vote\_max\_idx = i;

free(kvalue);

free(start);

free(vote);

return model->label[vote\_max\_idx];

}

}

double svm\_predict(const svm\_model \*model, const svm\_node \*x){

int nr\_class = model->nr\_class;

double \*dec\_values;

if (model->param.svm\_type == ONE\_CLASS ||

model->param.svm\_type == EPSILON\_SVR ||

model->param.svm\_type == NU\_SVR)

dec\_values = Malloc(double, 1);

else

dec\_values = Malloc(double, nr\_class\*(nr\_class - 1) / 2);

double pred\_result = svm\_predict\_values(model, x, dec\_values);

free(dec\_values);

return pred\_result;

}

double svm\_predict\_probability(

const svm\_model \*model, const svm\_node \*x, double \*prob\_estimates){

if ((model->param.svm\_type == C\_SVC || model->param.svm\_type == NU\_SVC)

&& model->probA != NULL && model->probB != NULL) {

int i;

int nr\_class = model->nr\_class;

double \*dec\_values = Malloc(double, nr\_class\*(nr\_class - 1) / 2);

svm\_predict\_values(model, x, dec\_values);

double min\_prob = 1e-7;

double \*\*pairwise\_prob = Malloc(double \*, nr\_class);

for (i = 0; i<nr\_class; i++)

pairwise\_prob[i] = Malloc(double, nr\_class);

int k = 0;

for (i = 0; i<nr\_class; i++)

for (int j = i + 1; j<nr\_class; j++) {

pairwise\_prob[i][j] = min(max(sigmoid\_predict(dec\_values[k], model-

>probA[k], model->probB[k]), min\_prob), 1 - min\_prob);

pairwise\_prob[j][i] = 1 - pairwise\_prob[i][j];

k++;

}

multiclass\_probability(nr\_class, pairwise\_prob, prob\_estimates);

int prob\_max\_idx = 0;

for (i = 1; i<nr\_class; i++)

if (prob\_estimates[i] > prob\_estimates[prob\_max\_idx])

prob\_max\_idx = i;

for (i = 0; i<nr\_class; i++)

free(pairwise\_prob[i]);

free(dec\_values);

free(pairwise\_prob);

return model->label[prob\_max\_idx];

}

else

return svm\_predict(model, x);

}

static const char \*svm\_type\_table[] = {

"c\_svc","nu\_svc","one\_class","epsilon\_svr","nu\_svr",NULL

};

static const char \*kernel\_type\_table[] = {

"linear","polynomial","rbf","sigmoid","precomputed",NULL

};

int svm\_save\_model(const char \*model\_file\_name, const svm\_model \*model) {

FILE \*fp = fopen(model\_file\_name, "w");

if (fp == NULL) return -1;

char \*old\_locale = setlocale(LC\_ALL, NULL);

if (old\_locale) {

old\_locale = \_strdup(old\_locale);

}

setlocale(LC\_ALL, "C");

const svm\_parameter& param = model->param;

fprintf(fp, "svm\_type %s\n", svm\_type\_table[param.svm\_type]);

fprintf(fp, "kernel\_type %s\n", kernel\_type\_table[param.kernel\_type]);

if (param.kernel\_type == POLY)

fprintf(fp, "degree %d\n", param.degree);

if (param.kernel\_type == POLY || param.kernel\_type == RBF || param.kernel\_type

== SIGMOID)

fprintf(fp, "gamma %g\n", param.gamma);

if (param.kernel\_type == POLY || param.kernel\_type == SIGMOID)

fprintf(fp, "coef0 %g\n", param.coef0);

int nr\_class = model->nr\_class;

int l = model->l;

fprintf(fp, "nr\_class %d\n", nr\_class);

fprintf(fp, "total\_sv %d\n", l);

{

fprintf(fp, "rho");

for (int i = 0; i<nr\_class\*(nr\_class - 1) / 2; i++)

fprintf(fp, " %g", model->rho[i]);

fprintf(fp, "\n");

}

if (model->label){

fprintf(fp, "label");

for (int i = 0; i<nr\_class; i++)

fprintf(fp, " %d", model->label[i]);

fprintf(fp, "\n");

}

if (model->probA) // regression has probA only{

fprintf(fp, "probA");

for (int i = 0; i<nr\_class\*(nr\_class - 1) / 2; i++)

fprintf(fp, " %g", model->probA[i]);

fprintf(fp, "\n");

}

if (model->probB){

fprintf(fp, "probB");

for (int i = 0; i<nr\_class\*(nr\_class - 1) / 2; i++)

fprintf(fp, " %g", model->probB[i]);

fprintf(fp, "\n");

}

if (model->nSV){

fprintf(fp, "nr\_sv");

for (int i = 0; i<nr\_class; i++)

fprintf(fp, " %d", model->nSV[i]);

fprintf(fp, "\n");

}

fprintf(fp, "SV\n");

const double \* const \*sv\_coef = model->sv\_coef;

const svm\_node \* const \*SV = model->SV;

for (int i = 0; i<l; i++){

for (int j = 0; j<nr\_class - 1; j++)

fprintf(fp, "%.16g ", sv\_coef[j][i]);

const svm\_node \*p = SV[i];

if (param.kernel\_type == PRECOMPUTED)

fprintf(fp, "0:%d ", (int)(p->value));

else

while (p->index != -1)

{

fprintf(fp, "%d:%.8g ", p->index, p->value);

p++;

}

fprintf(fp, "\n");

}

setlocale(LC\_ALL, old\_locale);

free(old\_locale);

if (ferror(fp) != 0 || fclose(fp) != 0) return -1;

else return 0;

}

static char \*line = NULL;

static int max\_line\_len;

static char\* readline(FILE \*input){

int len;

if (fgets(line, max\_line\_len, input) == NULL)

return NULL;

while (strrchr(line, '\n') == NULL){

max\_line\_len \*= 2;

line = (char \*)realloc(line, max\_line\_len);

len = (int)strlen(line);

if (fgets(line + len, max\_line\_len - len, input) == NULL)

break;

}

return line;

}

// FSCANF helps to handle fscanf failures.

// Its do-while block avoids the ambiguity when

// if (...)

// FSCANF();

// is used

#define FSCANF(\_stream, \_format, \_var) do{ if (fscanf(\_stream, \_format, \_var) != 1) return false; }while(0)

bool read\_model\_header(FILE \*fp, svm\_model\* model){

svm\_parameter& param = model->param;

char cmd[81];

while (1){

FSCANF(fp, "%80s", cmd);

if (strcmp(cmd, "svm\_type") == 0){

FSCANF(fp, "%80s", cmd);

int i;

for (i = 0; svm\_type\_table[i]; i++){

if (strcmp(svm\_type\_table[i], cmd) == 0){

param.svm\_type = i;

break;

}

}

if (svm\_type\_table[i] == NULL){

fprintf(stderr, "unknown svm type.\n");

return false;

}

}

else if (strcmp(cmd, "kernel\_type") == 0){

FSCANF(fp, "%80s", cmd);

int i;

for (i = 0; kernel\_type\_table[i]; i++){

if (strcmp(kernel\_type\_table[i], cmd) == 0){

param.kernel\_type = i;

break;

}

}

if (kernel\_type\_table[i] == NULL){

fprintf(stderr, "unknown kernel function.\n");

return false;

}

}

else if (strcmp(cmd, "degree") == 0)

FSCANF(fp, "%d", &param.degree);

else if (strcmp(cmd, "gamma") == 0)

FSCANF(fp, "%lf", &param.gamma);

else if (strcmp(cmd, "coef0") == 0)

FSCANF(fp, "%lf", &param.coef0);

else if (strcmp(cmd, "nr\_class") == 0)

FSCANF(fp, "%d", &model->nr\_class);

else if (strcmp(cmd, "total\_sv") == 0)

FSCANF(fp, "%d", &model->l);

else if (strcmp(cmd, "rho") == 0){

int n = model->nr\_class \* (model->nr\_class - 1) / 2;

model->rho = Malloc(double, n);

for (int i = 0; i<n; i++)

FSCANF(fp, "%lf", &model->rho[i]);

}

else if (strcmp(cmd, "label") == 0) {

int n = model->nr\_class;

model->label = Malloc(int, n);

for (int i = 0; i<n; i++)

FSCANF(fp, "%d", &model->label[i]);

}

else if (strcmp(cmd, "probA") == 0) {

int n = model->nr\_class \* (model->nr\_class - 1) / 2;

model->probA = Malloc(double, n);

for (int i = 0; i<n; i++)

FSCANF(fp, "%lf", &model->probA[i]);

}

else if (strcmp(cmd, "probB") == 0){

int n = model->nr\_class \* (model->nr\_class - 1) / 2;

model->probB = Malloc(double, n);

for (int i = 0; i<n; i++)

FSCANF(fp, "%lf", &model->probB[i]);

}

else if (strcmp(cmd, "nr\_sv") == 0){

int n = model->nr\_class;

model->nSV = Malloc(int, n);

for (int i = 0; i<n; i++)

FSCANF(fp, "%d", &model->nSV[i]);

}

else if (strcmp(cmd, "SV") == 0){

while (1){

int c = getc(fp);

if (c == EOF || c == '\n') break;

}

break;

}

else{

fprintf(stderr, "unknown text in model file: [%s]\n", cmd);

return false;

}

}

return true;

}

svm\_model \*svm\_load\_model(const char \*model\_file\_name){

FILE \*fp = fopen(model\_file\_name, "rb");

if (fp == NULL) return NULL;

char \*old\_locale = setlocale(LC\_ALL, NULL);

if (old\_locale) {

old\_locale = \_strdup(old\_locale);

}

setlocale(LC\_ALL, "C");

// read parameters

svm\_model \*model = Malloc(svm\_model, 1);

model->rho = NULL;

model->probA = NULL;

model->probB = NULL;

model->sv\_indices = NULL;

model->label = NULL;

model->nSV = NULL;

// read header

if (!read\_model\_header(fp, model)){

fprintf(stderr, "ERROR: fscanf failed to read model\n");

setlocale(LC\_ALL, old\_locale);

free(old\_locale);

free(model->rho);

free(model->label);

free(model->nSV);

free(model);

return NULL;

}

// read sv\_coef and SV

int elements = 0;

long pos = ftell(fp);

max\_line\_len = 1024;

line = Malloc(char, max\_line\_len);

char \*p, \*endptr, \*idx, \*val;

while (readline(fp) != NULL){

p = strtok(line, ":");

while (1) {

p = strtok(NULL, ":");

if (p == NULL)

break;

++elements;

}

}

elements += model->l;

fseek(fp, pos, SEEK\_SET);

int m = model->nr\_class - 1;

int l = model->l;

model->sv\_coef = Malloc(double \*, m);

int i;

for (i = 0; i<m; i++)

model->sv\_coef[i] = Malloc(double, l);

model->SV = Malloc(svm\_node\*, l);

svm\_node \*x\_space = NULL;

if (l>0) x\_space = Malloc(svm\_node, elements);

int j = 0;

for (i = 0; i<l; i++){

readline(fp);

model->SV[i] = &x\_space[j];

p = strtok(line, " \t");

model->sv\_coef[0][i] = strtod(p, &endptr);

for (int k = 1; k<m; k++){

p = strtok(NULL, " \t");

model->sv\_coef[k][i] = strtod(p, &endptr);

}

while (1){

idx = strtok(NULL, ":");

val = strtok(NULL, " \t");

if (val == NULL)

break;

x\_space[j].index = (int)strtol(idx, &endptr, 10);

x\_space[j].value = strtod(val, &endptr);

++j;

}

x\_space[j++].index = -1;

}

free(line);

setlocale(LC\_ALL, old\_locale);

free(old\_locale);

if (ferror(fp) != 0 || fclose(fp) != 0)

return NULL;

model->free\_sv = 1; // XXX

return model;

}

void svm\_free\_model\_content(svm\_model\* model\_ptr){

if (model\_ptr->free\_sv && model\_ptr->l > 0 && model\_ptr->SV != NULL)

free((void \*)(model\_ptr->SV[0]));

if (model\_ptr->sv\_coef){

for (int i = 0; i<model\_ptr->nr\_class - 1; i++)

free(model\_ptr->sv\_coef[i]);

}

free(model\_ptr->SV);

model\_ptr->SV = NULL;

free(model\_ptr->sv\_coef);

model\_ptr->sv\_coef = NULL;

free(model\_ptr->rho);

model\_ptr->rho = NULL;

free(model\_ptr->label);

model\_ptr->label = NULL;

free(model\_ptr->probA);

model\_ptr->probA = NULL;

free(model\_ptr->probB);

model\_ptr->probB = NULL;

free(model\_ptr->sv\_indices);

model\_ptr->sv\_indices = NULL;

free(model\_ptr->nSV);

model\_ptr->nSV = NULL;

}

void svm\_free\_and\_destroy\_model(svm\_model\*\* model\_ptr\_ptr){

if (model\_ptr\_ptr != NULL && \*model\_ptr\_ptr != NULL){

svm\_free\_model\_content(\*model\_ptr\_ptr);

free(\*model\_ptr\_ptr);

\*model\_ptr\_ptr = NULL;

}

}

void svm\_destroy\_param(svm\_parameter\* param){

free(param->weight\_label);

free(param->weight);

}

const char \*svm\_check\_parameter(const svm\_problem \*prob, const svm\_parameter \*param) {

// svm\_type

int svm\_type = param->svm\_type;

if (svm\_type != C\_SVC &&

svm\_type != NU\_SVC &&

svm\_type != ONE\_CLASS &&

svm\_type != EPSILON\_SVR &&

svm\_type != NU\_SVR)

return "unknown svm type";

// kernel\_type, degree

int kernel\_type = param->kernel\_type;

if (kernel\_type != LINEAR &&

kernel\_type != POLY &&

kernel\_type != RBF &&

kernel\_type != SIGMOID &&

kernel\_type != PRECOMPUTED)

return "unknown kernel type";

if (param->gamma < 0)

return "gamma < 0";

if (param->degree < 0)

return "degree of polynomial kernel < 0";

// cache\_size,eps,C,nu,p,shrinking

if (param->cache\_size <= 0)

return "cache\_size <= 0";

if (param->eps <= 0)

return "eps <= 0";

if (svm\_type == C\_SVC ||

svm\_type == EPSILON\_SVR ||

svm\_type == NU\_SVR)

if (param->C <= 0)

return "C <= 0";

if (svm\_type == NU\_SVC ||

svm\_type == ONE\_CLASS ||

svm\_type == NU\_SVR)

if (param->nu <= 0 || param->nu > 1)

return "nu <= 0 or nu > 1";

if (svm\_type == EPSILON\_SVR)

if (param->p < 0)

return "p < 0";

if (param->shrinking != 0 &&

param->shrinking != 1)

return "shrinking != 0 and shrinking != 1";

if (param->probability != 0 &&

param->probability != 1)

return "probability != 0 and probability != 1";

if (param->probability == 1 &&

svm\_type == ONE\_CLASS)

return "one-class SVM probability output not supported yet";

// check whether nu-svc is feasible

if (svm\_type == NU\_SVC){

int l = prob->l;

int max\_nr\_class = 16;

int nr\_class = 0;

int \*label = Malloc(int, max\_nr\_class);

int \*count = Malloc(int, max\_nr\_class);

int i;

for (i = 0; i<l; i++) {

int this\_label = (int)prob->y[i];

int j;

for (j = 0; j<nr\_class; j++)

if (this\_label == label[j]){

++count[j];

break;

}

if (j == nr\_class){

if (nr\_class == max\_nr\_class){

max\_nr\_class \*= 2;

label = (int \*)realloc(label, max\_nr\_class\*sizeof(int));

count = (int \*)realloc(count, max\_nr\_class\*sizeof(int));

}

label[nr\_class] = this\_label;

count[nr\_class] = 1;

++nr\_class;

}

}

for (i = 0; i<nr\_class; i++){

int n1 = count[i];

for (int j = i + 1; j<nr\_class; j++){

int n2 = count[j];

if (param->nu\*(n1 + n2) / 2 > min(n1, n2)){

free(label);

free(count);

return "specified nu is infeasible";

}

}

}

free(label);

free(count);

}

return NULL;

}

int svm\_check\_probability\_model(const svm\_model \*model){

return ((model->param.svm\_type == C\_SVC || model->param.svm\_type ==

NU\_SVC) && model->probA != NULL && model->probB != NULL) ||

((model->param.svm\_type == EPSILON\_SVR || model->param.svm\_type ==

NU\_SVR) && model->probA != NULL);

}

void svm\_set\_print\_string\_function(void(\*print\_func)(const char \*)){

if (print\_func == NULL)

svm\_print\_string = &print\_string\_stdout;

else

svm\_print\_string = print\_func;

}

## toCSV.cpp

#include<opencv2/core/core.hpp>

#include<opencv2/highgui/highgui.hpp>

#include<opencv2/imgproc/imgproc.hpp>

#include<opencv2/ml/ml.hpp>

#include"Segmentation.h"

#include<fstream>

#include<istream>

#include<iostream>

#include<string>

#include<vector>

using namespace std;

using namespace cv;

// Image File Converted To Csv

int getCSV(Mat img){

int val;

ofstream myfile;

myfile.open("example.csv");

myfile << "0";

myfile << " ,";

for (int y = 0; y < img.rows; y++) {

for (int x = 0; x < img.cols; x++) {

val = img.at<uchar>(Point(x, y));

myfile << val << ",";

}

}

myfile << "\n";

myfile.close();

int label=toSVMformat();

return label;

}

## toSVMformat.cpp

#include<opencv2/core/core.hpp>

#include<opencv2/highgui/highgui.hpp>

#include<opencv2/imgproc/imgproc.hpp>

#include<opencv2/ml/ml.hpp>

#include"Segmentation.h"

#include<fstream>

#include<istream>

#include<iostream>

#include<string>

#include<vector>

using namespace std;

using namespace cv;

int toSVMformat() {

ifstream file;

ofstream data;

string value;

file.open("example.csv");

data.open("exmpleData.data");

for (int x = 0; x < 785; x++) {

getline(file, value, ','); // read a string until next comma

int t = atoi(value.c\_str());

if ((t > 0) && (value != " ")) {

float val = (float)t / 255;

string s= to\_string(val);

data << x << ":" << s << " ";

}

}

file.close();

data.close();

int label = svmPredict();

return label;

}