4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

При разработке программного модуля оценки качества зерна были реализованы некоторые нестандартные алгоритмы, такие как построение ROC-кривой, классификация, изменение цветовой системы.

4.1 Построение ROC-кривой

Для представления пользователю качества классификации, производится построение ROC-кривой. Кривая строится на основе предоставленных пользователем идеальных данных классификатора и результатов полученных при работе классификатора. Производится перезапись данных из статического массива с идеальными данными в динамический.

```
for(int i = 0; i < seedVector.length(); i++)
{
    idealClusterData[i] = idealClusterDataStatic[i];
}
```

В дальнейшем, расстояние от разделяющей классы прямой преобразуется в вероятность принадлежности объекта к одному из классов. Эти данные переписываются из вектора содержащего все объекты в динамический массив целочисленного типа int.

```
for(int j = 0; j < n; j++)
{
   scores[j] = seedVector[j].probability;
   scores2[j] = 1 - seedVector[j].probability;
}
```

Далее вычисляются параметры true positive и false positive для каждого объекта в массиве `seedVector` и сохраняются в массивы `TPmas` и `FPmas`. Данные точки высчитываются путем деления переменных `TP` и `FP` переменные `P` и `N` соответственно. `P` – количество правильных результатов классификации для класса, `N` – неправильных результатов.

```
if(fi != fprev)
{
    fprev = fi;
    FPprev = FP;
    TPprev = TP;
    this->TPmas.push_back(TP/P);
    this->FPmas.push_back(FP/N);
}
if(label == posclass) {
```

```
        TP = TP + 1;
    }
    else
    {
        FP = FP + 1;
    }

    this->TPmas.push_back(TP/P);
    this->FPmas.push_back(FP/N);
```

После построения по полученным точкам графика ROC-кривой в большей части случаев необходимо производить процесс сглаживания. Данный процесс необходим для более удачного визуального представления кривой.

```
for(int i=0;i<a.size()-1;i++)
{
   if(i+NO_OF_NEIGHBOURS+1<a.size())
   {
       for(int j=1;j<NO_OF_NEIGHBOURS;j++)
       {
          a[i].first += a[i+j].first;
          a[i].second += a[i+j].second;
       }
       a[i].first /= NO_OF_NEIGHBOURS;
       a[i].second /= NO_OF_NEIGHBOURS;
} else
{
   for(int j=1;j<NO_OF_NEIGHBOURS;j++)
   {
       a[i].first += tmp[i-j].first;
       a[i].second += tmp[i-j].second;
   }
   a[i].first /= NO_OF_NEIGHBOURS;
   a[i].second /= NO_OF_NEIGHBOURS;
}
```

4.2 Классификация

Разбиение всех найденных объектов на отдельные классы называется классификацией. За классификацию отвечает модуль принятия решений. Построен он на основе метода опорных векторов. Для начала процесса классификации методу требуются предоставленные пользователем данные. Данные содержат объекты с посчитанными характеристиками, которые пользователь предпочел выбрать как образец. Для каждого класса необходимо предоставить по три объекта относящихся к нему. Число три было выбрано для того чтобы увеличить шанс правильного отнесения объекта к классу.

Алгоритм классификации реализуется путем выполнения следующей

последовательности шагов:

Шаг 1. Выбор обучающих данных:

1) Инициализация переменных. В методе используются следующие переменные:

– `seedVect = QVector<Seed>(seedVector)` – инициализирует массив с объектами семян из которых будет выбираться обучающая выборка;

– `countOfClusters = countOfClusters` – инициализируется количество кластеров для которых требуется заполнить обучающую выборку;

2) При нажатии мышью получаются координаты точки положения курсора;

3) Определяется к какому объекту на изображении относится координаты клика;

4) Закрашивается контур объекта цветом относящимся к выбираемому классу;

```
     for(int i = 0; i < seedVect.length(); i++)
     {
        if(allocObjMat.at<Vec3b>(imgY,         imgX)[0]         ==
seedVect[i].GetColor().val[0] &&
           allocObjMat.at<Vec3b>(imgY,         imgX)[1]         ==
seedVect[i].GetColor().val[1] &&
           allocObjMat.at<Vec3b>(imgY,         imgX)[2]         ==
seedVect[i].GetColor().val[2])
             {

               if(isEnoughObjForCluster(currentCluster))
               {
                   currentCluster++;
                   countOfObjsInCluster = 0;
               }
               countOfObjsInCluster++;

               trainDataObjsVectrs[currentCluster].push_back(i);
               fillLabels(i);

               contourDetection(seedVect[i].GetColor());
               showOnSrcLabel(this->srcImg);

               if(countOfObjsInCluster % countOfObjsInGrBox == 0)
               {
                   setCheckForCheckBox(currentCluster,true);
                   currentCluster++;
                   countOfObjsInCluster = 0;
               }

               break;
             }
```

5) Заносится выбранный объект в trainDataObjsVectrs и отобразить на интерфейсе, то, что объект добавлен.

6) Производится проверка на количество объектов для класса, если объектов достаточно, то ClusterGroupBox переводится в состояние enable и происходит переход к следующему классу.

7) После заполнения всех требуемых данных процесс завершается и диалог возвращает значение trainDataObjsVectrs;

Шаг 2. Обучение классификатора:

1) Инициализация переменных. В методе используются следующие переменные:

– seedVect = QVector<Seed>(seedVector) – инициализирует массив с объектами семян из которых будет выбираться обучающая выборка;

– featVect = QVector<int>(featVector) – инициализируется массив с признаками объекта, которые были выбраны ранее для того, чтобы быть условием классификации;

– trainigDataObjs = QVector<QVector<int> > (trainDataObj) – массив объектов выбранных для того, чтобы быть обучающей выборкой классификатора;

2) Производится подсчет количества переданных объектов в обучающей выборке trainigDataObjs;

```
for(int i = 0; i < trainigDataObjs.length(); i++)
{
    for(int j = 0; j < trainigDataObjs[i].length(); j++)
    {
        countOfOb++;
    }
}
```

3) Каждый объект представляется в виде динамического массива целочисленного типа int;

```
for(int i = 0; i < trainigDataObjs.length(); i++)
{
   for(int j = 0; j < trainigDataObjs[i].length(); j++)
   {
      labels[indexOfObject++] = i;
         fillObject(trainingData[k++],
trainigDataObjs[i][j]);
   }
}
```

4) Все преобразованные данные аккумулируются в trainingDataMat типа Mat, размерность матрицы задает количество объектов;

```
Mat trainingDataMat(countOfOb, featVect.length(), CV_32FC1,
trainingData);
```

5) Производится обучение классификатора путем вызова метода `train` у объекта SVM. В качестве параметра в метод передаются данные выбранные заранее пользователем – `trainingDataMat`;

```
CvSVMParams params;
params.svm_type    = CvSVM::C_SVC;
params.kernel_type = CvSVM::LINEAR;
params.term_crit    = cvTermCriteria(CV_TERMCRIT_ITER, 100,
1e-6);

CvSVM SVM;
SVM.train(trainingDataMat, labelsMat, Mat(), Mat(), params);
```

6) Все объекты преобразуются в динамические массивы целочисленного типа `int`;

7) Основным шагом в классификации является вызов метода `predict` у объекта SVM. Метод производит классификацию и возвращает для каждого объекта значение расстояния от разделяющей классы прямой, по нему определяется принадлежность объекта к классу;

8) Заключительный этап классификации – это преобразование расстояния до разделяющей прямой от объекта в вероятность принадлежности объекта к классу;

```
float confidence;
for(int i = 0; i < seedVect.length(); i++)
{
    fillObject(object, i);
    Mat objectMat(1, featureLength, CV_32FC1, object);
    float cl2 = SVM.predict(objectMat, true);
    confidence = 1.0 / (1.0 + exp(-cl2));
    float cl = SVM.predict(objectMat);
    seedVect[i].SetCluster(cl);
    seedVect[i].probability = confidence;
}
```

4.3 Изменение цветовой системы

Часто, для решения разного рода задач требуется различное представление данных. Получить цветовая система RGB не всегда предоставляет требуемые данные, в таких случаях возникает потребность в расширении или изменении цветовых каналов изображения. В дипломной работе предоставлена возможность использования четырех дополнительных видов цветовых систем: HLS, HSV, CMYK и Lab.

Рассмотрим алгоритм функции преобразующей цветовую систему RGB

в цветовую систему CMYK содержит следующие пункты:

1) Инициализация данных в которые будут сохранены преобразованные каналы;

```
Mat channel[4];

for (int i = 0; i < 4; i++)
{
        cmyk.push_back(cv::Mat(img.size(), CV_8UC1));
        channel[i] = Mat::eye(img.size(), CV_8UC1);
}
```

2) Разделение изображения в цветовой системе RGB на отдельные, составляющие ее, каналы: Red, Green, Blue.

```
std::vector<cv::Mat> rgb;
cv::split(img, rgb);
```

3) Производится перекодирование каналов RGB в каналы CMYK. Требуется произвести перебор всех пикселей изображения и, используя весовые коэффициенты, вычислить новые значения пикселей.

```
for (int i = 0; i < img.rows; i++)
{
        for (int j = 0; j < img.cols; j++)
     {
                float r = (int)rgb[2].at<uchar>(i, j) / 255.;
                float g = (int)rgb[1].at<uchar>(i, j) / 255.;
                float b = (int)rgb[0].at<uchar>(i, j) / 255.;
                float k = std::min(std::min(1- r, 1- g), 1-
b);

                uchar ch = channel[0].at<uchar>(i, j);
                channel[0].at<uchar>(i, j) = (1 - r - k) / (1
- k) * 255.;
                channel[1].at<uchar>(i, j) = (1 - g - k) / (1
- k) * 255.;
                channel[2].at<uchar>(i, j) = (1 - b - k) / (1
- k) * 255.;
                channel[3].at<uchar>(i, j) = k * 255.;
          }
 }
```

4) Записать полученные каналы в выходной буфер, который передается в метод по указателю.

```
 cmyk[0] = channel[0].clone();
 cmyk[1] = channel[1].clone();
```

```
cmyk[2] = channel[2].clone();
cmyk[3] = channel[3].clone();
```

5) Для более удобного восприятия каналов пользователем, требуется придать каждому каналу свой цвет.

```
std::string cmyk_labels[3] = {"[C]yan", "[M]agenta",
"[Y]ellow"};
double std_values[3][3] = {{255, 255, 0}, {255, 0, 255}, {0,
255, 255}};
cmykColored = showChannels(img, channel, cmyk_labels,
std_values, COLOR_HSV2BGR, true);
```

Преобразование в цветовую систему CMYK позволяет расширить каналы RGB.

Функция преобразующая RGB в цветовую систему HSV содержит следующие пункты:

1) Инициализация данных в которые будут сохранены преобразованные каналы;

```
Mat channel[3];
for (int i = 0; i < 3; i++)
{
   hsvVector.push_back(cv::Mat(img.size(), CV_8UC1));
}
```

2) Вызов функции преобразования из цветовой системы RGB в систему HSV.

```
cvtColor(img,hsv,CV_BGR2HSV);
```

3) Разделение преобразованного изображения в цветовой системе HSV на отдельные, составляющие ее, каналы: Hue, Saturation, Value.

```
split(hsv, channel);
```

4) Запись полученных разделенных каналов в выходной буфер, переданный в метод по указателю.

```
hsvVector[0] = channel[0].clone();
hsvVector[1] = channel[1].clone();
hsvVector[2] = channel[2].clone();
```

5) Преобразование каналов из полутона в цветное изображение для лучшего визуального восприятия путем добавления каждому каналу своего цвета.

```
    std::string hsv_labels[3] = {"[H]ue", "[S]aturation",
"[V]alue"};
        double std_values[3][3] = {{1, 255, 255}, {179, 1, 255},
{179, 0, 1}};
        hsvVectorColored = showChannels(img, channel,
hsv_labels, std_values, COLOR_HSV2BGR, false);
```

Рассмотрим алгоритм функции преобразующей цветовую RGB в цветовую систему HLS содержит следующие пункты:

1) Инициализация данных в которые будут сохранены преобразованные каналы:

– `vector<Mat> hlsVectorColored = NULL` – vector в который будут помещаться каналы уже преобразованный из полутона в цвет, инициализируется `NULL`;

– `Mat hls = Mat()` – изображение в цветовой системе HLS, инициализируется пустой матрицей;

– `Mat channel[3]` – статический массив для сохранения каналов;

```
for (int i = 0; i < 3; i++)
{
     hlsVector.push_back(cv::Mat(img.size(), CV_8UC1));
}
```

2) Вызов функции преобразования из цветовой системы RGB в систему HLS.

```
cvtColor(img,hls,CV_BGR2HLS);
```

3) Разделение преобразованного изображения в цветовой системе HLS на отдельные, составляющие ее, каналы: Hue, Lightness, Saturation.

```
split(hls, channel);
```

4) Запись полученных разделенных каналов в выходной буфер, переданный в метод по указателю.

```
hlsVector[0] = channel[0].clone();
hlsVector[1] = channel[1].clone();
hlsVector[2] = channel[2].clone();
```

5) Преобразование каналов из полутона в цветное изображение для лучшего визуального восприятия путем добавления каждому каналу своего цвета.

```
    std::string hls_labels[3] = {"[H]ue", "[L]ightness",
"[S]aturation"};
```

```
    double std_values[3][3] = {{1, 100, 50}, {179, 1, 255}, {179, 69, 1}};
    hlsVectorColored = showChannels(img, channel, hls_labels, std_values, COLOR_HLS2BGR, false);
```

Рассмотрим алгоритм функции преобразующей цветовую RGB в цветовую систему HLS содержит следующие пункты:

1) Инициализация данных в которые будут сохранены преобразованные каналы:

– `vector<Mat> labVectorColored = NULL` – vector в который будут помещаться каналы уже преобразованный из полутона в цвет, инициализируется NULL;

– `Mat lab = Mat()` – изображение в цветовой системе Lab, инициализируется пустой матрицей;

– `Mat channel[3]` – статический массив для сохранения каналов;

```
for (int i = 0; i < 3; i++)
{
    labVector.push_back(Mat(img.size(), CV_8UC1));
}
```

2) Вызов функции преобразования из цветовой системы RGB в систему HLS.

```
cvtColor(img,lab,CV_BGR2Lab);
```

3) Разделение преобразованного изображения в цветовой системе Lab на отдельные, составляющие ее, каналы: Luminance, Adimension, Bdimention.

```
split(lab, channel);
```

4) Запись полученных разделенных каналов в выходной буфер, переданный в метод по указателю.

```
labVector[0] = channel[0].clone();
labVector[1] = channel[1].clone();
labVector[2] = channel[2].clone();
```

5) Преобразование каналов из полутона в цветное изображение для лучшего визуального восприятия путем добавления каждому каналу своего цвета.