# PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

## FACULTAD DE CIENCIAS E INGENIERÍA


# Diseño de un modelo algorítmico basado en visión computacional para la detección y clasificación de retinopatía diabética en imágenes retinográficas digitales.

Tesis para optar el Título de **Ingeniero Informático**, que presenta el bachiller:

Daekef Rosendo Abarca Cusimayta

ASESOR: Dr. César A. Beltrán Castañón

Lima, mayo de 2018

# Contenido

## 1. Código Desarrollado

**NOTA: Sólo se incluyen las funciones maestras. No están incluidas las funciones satélite.

### 1.1. Función Principal - Clasificador

```
int main(int argc, char *argv[]) {

    //SE INICIA LA APLICACIÓN QT
    cout<<"se inicia aplicacion"<<endl;
    QApplication app(argc, argv);
    app.setQuitOnLastWindowClosed(true);
    cout<<"se declara form"<<endl;
    MainForm mainForm;
    cout<<"se muestra ventana"<<endl;
    mainForm.show();
    cout<<"se mostro ventana"<<endl;

    //SE CONFIGURA LAS VARIABLES PARA LEER DE LOS ARCHIVOS
    string urlBase = "";
    string image_name = "";
    Mat bgMask, image, tmp;
    int labels[numberOfImages * nCategories];
    float trainingData[numberOfImages * nCategories][nChars];

    int flag = 1;
    int nImage = 0;
    string svmFileName = "svm_training.xml";
    string trainingFileName = "training_data.csv";
    ifstream svmFile(svmFileName);
    ifstream trainingFile(trainingFileName);
    ofstream trainingFileOutput;
    Ptr<TrainData> td;
    CharsImage ci;
    int ncols = 2240, nrows = 1488;
    int debugFlag = 0;
    if (!svmFile.is_open() || debugFlag) {
        if (!trainingFile.is_open() || debugFlag) {
            if (!debugFlag) trainingFileOutput.open(trainingFileName);
            for (int i = 0; i < nCategories; i++) {
                switch (i) {
                    case 0:
                        urlBase = "image/0-dataset/0 - normal/";
                        image_name = "n_image";
                        break;
                    case 1:
                        urlBase = "image/0-dataset/1 - leve/";
```

```
                image_name = "l_image";
                break;
            case 2:
                urlBase = "image/0-dataset/2 - moderado/";
                image_name = "m_image";
                break;
            case 3:
                urlBase = "image/0-dataset/3 - severo/";
                image_name = "s_image";
                break;

        }
        for (int n = 1; n <= numberOfImages; n++) {
            ci.nImage = n;

            image = imread(urlBase + image_name + SSTR(n) + ".tif", 1);
            if (image.cols != ncols || image.rows != nrows) {
                resize(image, image, Size(ncols, nrows));
            }
            //----INICIO DEL PRE-PROCESAMIENTO
            if (flag) {
                //SE QUITA EL FONDO DE LA IMAGEN
                backgroundSegmentation(image, bgMask);
                readInGreenChannel("image/bgMask.tif",bgMask);
                flag = 0;
            }
            bgMask.copyTo(tmp);
            //SE QUITA EL DISCO OPTICO DE LA IMAGEN
            opticDiscSegmentation(tmp, image, ci);
            //----FIN DEL PRE-PROCESAMIENTO

            //----INICIO EXTRACCION DE CARACTERISTICAS

            //SE EXTRAEN LOS MICROANEURISMAS Y LAS HEMORRAGIAS
            darkLessionSegmentation(tmp, image, ci);
            //SE EXTRAEN LOS EXUDADOS
            brightLessionSegmentation(tmp, image, ci);

            //----FIN EXTRACCION DE CARACTERISTICAS

            //----INICIO CLASIFICADOR
            //SE ARMAN LOS VECTORES DE CARACTERISTICAS PARA CADA
IMAGEN
            cout << "Imagen " << nImage << ": " << "(Tipo - " << i << ") " <<
ci.areaDarkZone << " " << ci.numberDarkZone << " " << ci.areaBrightZone << " " <<
ci.numberBrightZones << endl;
```

```
                trainingFileOutput << ci.areaDarkZone << "," << ci.numberDarkZone <<
"," << ci.areaBrightZone << "," << ci.numberBrightZones << "," << i << endl;

                labels[nImage] = i;

                trainingData[nImage][0] = ci.areaDarkZone;
                trainingData[nImage][1] = ci.numberDarkZone;
                trainingData[nImage][2] = ci.areaBrightZone;
                trainingData[nImage][3] = ci.numberBrightZones;
                nImage++;
                if (debugFlag) {
                    return 0;
                }

            }

        }
        //SE ENTRENA EL CLASIFICADOR CON LOS DATOS OBTENIDOS DE
LOS VECTORES DE CARACTERISTICAS
        trainingFileOutput.close();
        Mat labelsMat(numberOfImages*nCategories, 1, CV_32SC1, labels);
        Mat trainingDataMat(numberOfImages*nCategories, nChars, CV_32FC1,
trainingData);
        td = TrainData::create(trainingDataMat, ROW_SAMPLE, labelsMat);
    } else {
        td = TrainData::loadFromCSV(trainingFileName, -1);
        svmFile.close();
    }
    // SE FIJAN LAS VARIABLES DEL CLASIFICADOR SVM
    Ptr<SVM> svm = SVM::create();
    svm->setType(ml::SVM::C_SVC);
    svm->setKernel(ml::SVM::RBF);
    svm->setGamma(9.3750000000000002e-002);
    svm->setTermCriteria(TermCriteria(TermCriteria::MAX_ITER, 1000,
1.1920928955078125e-007));
    svm->setC(2.5000000000000000e+000);
    ParamGrid noParams;
    noParams.logStep = 0;

    svm->trainAuto(td, 10, noParams,
            noParams,
            noParams,
            noParams,
            noParams,
            noParams,
            true);
    ;
    svm->save(svmFileName);
```

```
    } else {
        cout << "File Exists" << endl;

    }


    cout << "App exec:"<<app.exec()<<endl;
    return 0;
    //----FIN CLASIFICADOR
}
```

### 1.2. Función ReadInGreenChannel

```
//FUNCION QUE LEE EN EL CANAL VERDE
void readInGreenChannel(const String& path, Mat& image) {
    Mat im;
    im = imread(path, 1);
    vector<Mat> channels;
    split(im, channels);
    image = channels[1];
}

void readInGreenChannel(Mat& src, Mat& image) {

    vector<Mat> channels;
    split(src, channels);
    image = channels[1];
}
```

### 1.3. Función backGroundSegmentation

```
void backgroundSegmentation(Mat& src, Mat& bgMask) {
    //Se itera un bloque de tamaño nRows y nCols y se calcula la máscara resultante
    CV_Assert(src.depth() != sizeof (uchar));
    bgMask = Mat::zeros(src.rows, src.cols, src.type());
    int nRows = src.rows;
    int nCols = src.cols;
    nRows /= w;
    nCols /= w;
    Mat block;

    Mat d;
    for (int i = 0; i < nRows; i++) {

        for (int j = 0; j < nCols; j++) {
            block = Mat(src, Rect(j*w, i*w, w, w));
            d = Mat(bgMask, Rect(j*w, i*w, w, w));
```

```
            retrieveBackgroundMask(block, d);

        }
    }

    Mat backgroundImageResult;
    medianBlur(bgMask, bgMask, 11);
    Mat element = getStructuringElement(MORPH_ELLIPSE, Size(5, 5));
    morphologyEx(bgMask, bgMask, MORPH_OPEN, element, Point(-1, -1), 2);

    vector<Mat> channels;
    split(bgMask, channels);
    bgMask = channels[1];


    vector<vector<Point> > contours;
    vector<Vec4i> hierarchy;
    findContours(bgMask, contours, hierarchy, CV_RETR_TREE,
CV_CHAIN_APPROX_SIMPLE, Point(0, 0));
    vector<vector<Point> > contours_poly(contours.size());
    vector<Point2f>center(contours.size());
    vector<float>radius(contours.size());
    for (int i = 0; i < 1; i++) {
        approxPolyDP(Mat(contours[i]), contours_poly[i], 3, true);
        minEnclosingCircle((Mat) contours_poly[i], center[i], radius[i]);
    }
    Mat fineMask = Mat::zeros(bgMask.rows, bgMask.cols, CV_8UC1);
    for( int i = 0; i<1; i++ )
     {
        cout<<radius[i]<<endl;
        cout<<center[i]<<endl;
       circle(fineMask, Point(nCols/2, nRows/2), radius[i]-67, Scalar(255), -1, 8, 0);
     }


    bgMask = fineMask;
    src.copyTo(src, bgMask);

}
```

### 1.4. Función OpticDiscSegmentation

```
void opticDiscSegmentation(Mat& bgMask, Mat& image, CharsImage& ci) {
    Mat roi;
    int fixedCircleSize = 80;
    int windowSize = 400;
    vector<Mat> channels;
    split(image, channels);


    //Se busca el disco óptico en el tercio central de la imagen
    Mat greenChannel;
    cvtColor(image, greenChannel, CV_BGR2GRAY);
    int y = greenChannel.rows / 3;
    int x = 0;
    int w = greenChannel.cols;
    int h = y;
    roi = Mat(greenChannel, Rect(x, y, w, h));


    Mat element = getStructuringElement(MORPH_ELLIPSE, Size(11, 11));
    Mat aux;
    morphologyEx(roi, roi, MORPH_DILATE, element, Point(-1, -1), 2);

    GaussianBlur(aux, aux, Size(31, 31), 1.0, 1.0);


    vector<Vec3f> circles;
    //se arma una función hough para encontrar el cículo más predominante
    HoughCircles(roi, circles, CV_HOUGH_GRADIENT, 1, image.cols / 2, 20, 10, 50,
250);
    //se marca el círculo encontrado y se crea una máscara para poder quitarla de la
imagen original
    Mat opticDMask = Mat(image.rows, image.cols, CV_8UC1, Scalar(255));
    if (circles.size() > 0) {
        Point center(cvRound(circles[0][0]), cvRound(circles[0][1]));
        int radius = cvRound(circles[0][2]);

        center.x = center.x;
        center.y = center.y + image.rows / 3;

        circle(opticDMask, center, radius + fixedCircleSize, Scalar(0), -1, 8, 0);

    } else {
        Point maxLoc;
        minMaxLoc(roi, NULL, NULL, NULL, &maxLoc);
        maxLoc.x = maxLoc.x;
        maxLoc.y = maxLoc.y + image.rows / 3;
        circle(opticDMask, maxLoc, 100, Scalar(0), -1, 8, 0);
```

```
    }

    Mat finalImage, finalMask;
    bitwise_and(bgMask, opticDMask, bgMask);
    image.copyTo(finalImage, bgMask);
    image = finalImage;
}
```

### 1.5. Función darkLessionSegmentation

```
void darkLessionSegmentation(Mat& bgMask, Mat& image, CharsImage& ci) {



    Mat invG, tmpGC;
    readInGreenChannel(image, invG);



    Mat element = getStructuringElement(MORPH_RECT, Size(3, 3));

    //Se trabaja en el canal verde invertido
    bitwise_not(invG, invG, bgMask);
    invG.copyTo(tmpGC);

    Scalar mImg= mean(tmpGC, bgMask);

    Mat medianFilter, topHat;

    Ptr<CLAHE> ptr = createCLAHE();
    ptr->setClipLimit(5);
    ptr->apply(invG, invG);

    medianBlur(invG, medianFilter, 105);
    invG = invG - medianFilter;
    morphologyEx(invG, topHat, CV_MOP_TOPHAT, element);
    invG = invG - topHat;
    GaussianBlur(invG, invG, Size(7, 7), 0, 0);
    //se aplica un umbral para solo quedarnos con las zonas osucras
    double max, min;
    minMaxLoc(invG, &min, &max, NULL, NULL, bgMask);
    threshold(invG, invG, 10, 255, CV_THRESH_BINARY);

    vector<vector<Point> > contours;
    vector<Vec4i> hierarchy;

    //se busca las zonas oscuras y se las marca
```

```
    findContours(invG, contours, hierarchy, CV_RETR_CCOMP,
CHAIN_APPROX_SIMPLE);

    //se cuenta las zonas oscuras y se las filtra segun su area, tamaño y circularidad
    Mat maImage = Mat::zeros(invG.rows, invG.cols, invG.type());
    double area;
    Rect r;
    vector<vector<Point> > filteredContours;
    double k;
    for (int i = 0; i < contours.size(); i++) {
        drawContours(maImage, contours, i, Scalar(255), CV_FILLED, 8, hierarchy, 0,
Point());
        area = contourArea(contours.at(i));
        r = boundingRect(contours.at(i));
        Scalar m = mean(tmpGC(r), maImage(r));
        if (r.height > r.width) {
            max = r.height;
            min = r.width;
        } else {
            max = r.width;
            min = r.height;
        }

        k = min / max;

        if ((k >= 0.8) && area <= 800 && area >= 0.50 * r.height * r.width && area >= 20
&& m.val[0] >= mImg.val[0]*1.1) filteredContours.push_back(contours.at(i));


    }
    //se crea una máscara con las zonas marcadas
    maImage = Mat::zeros(invG.rows, invG.cols, invG.type());
    for (int i = 0; i < filteredContours.size(); i++) {
        drawContours(maImage, filteredContours, i, Scalar(255), CV_FILLED, 8,
hierarchy, 0, Point());
    }

    //se marca el vector de características con los resultados obtenidos
    ci.areaDarkZone = countNonZero(maImage);
    ci.numberDarkZone = filteredContours.size();


}
```

### 1.6. Función brightLessionSegmentation

```
void brightLessionSegmentation(Mat& mask, Mat& image, CharsImage& ci) {


    //Se trabaja en el espacio de color LAB
    vector<Mat> channels;
    split(image, channels);
    Mat gc = channels[1];
    Mat greenChannel;
    gc.copyTo(greenChannel);

    Scalar mImg=mean(greenChannel, mask);
    Mat lab;
    Mat l_chann;

    cvtColor(image, lab, CV_BGR2Lab);

    split(lab, channels);
    l_chann = channels[0];

    Mat element = getStructuringElement(MORPH_ELLIPSE, Size(15, 15));
    Mat topHat, bottomHat;
    l_chann.copyTo(topHat);
    l_chann.copyTo(bottomHat);
    //Se ejecutan operadores morfologicos
    Mat opened;
    erode(l_chann, opened, element);
    dilate(opened, opened, element);

    topHat = topHat - opened;

    Mat closed;
    dilate(l_chann, closed, element);
    erode(closed, closed, element);

    bottomHat = closed - bottomHat;

    l_chann = l_chann + topHat - bottomHat;

    equalizeHist(l_chann, l_chann);
    double min, max;
    minMaxLoc(l_chann, &min, &max);

    double t = max - 0.01 * max;
    threshold(l_chann, l_chann, t, 255, CV_THRESH_BINARY);

    Mat i1, i2;
    element = getStructuringElement(MORPH_ELLIPSE, Size(15, 15));
```

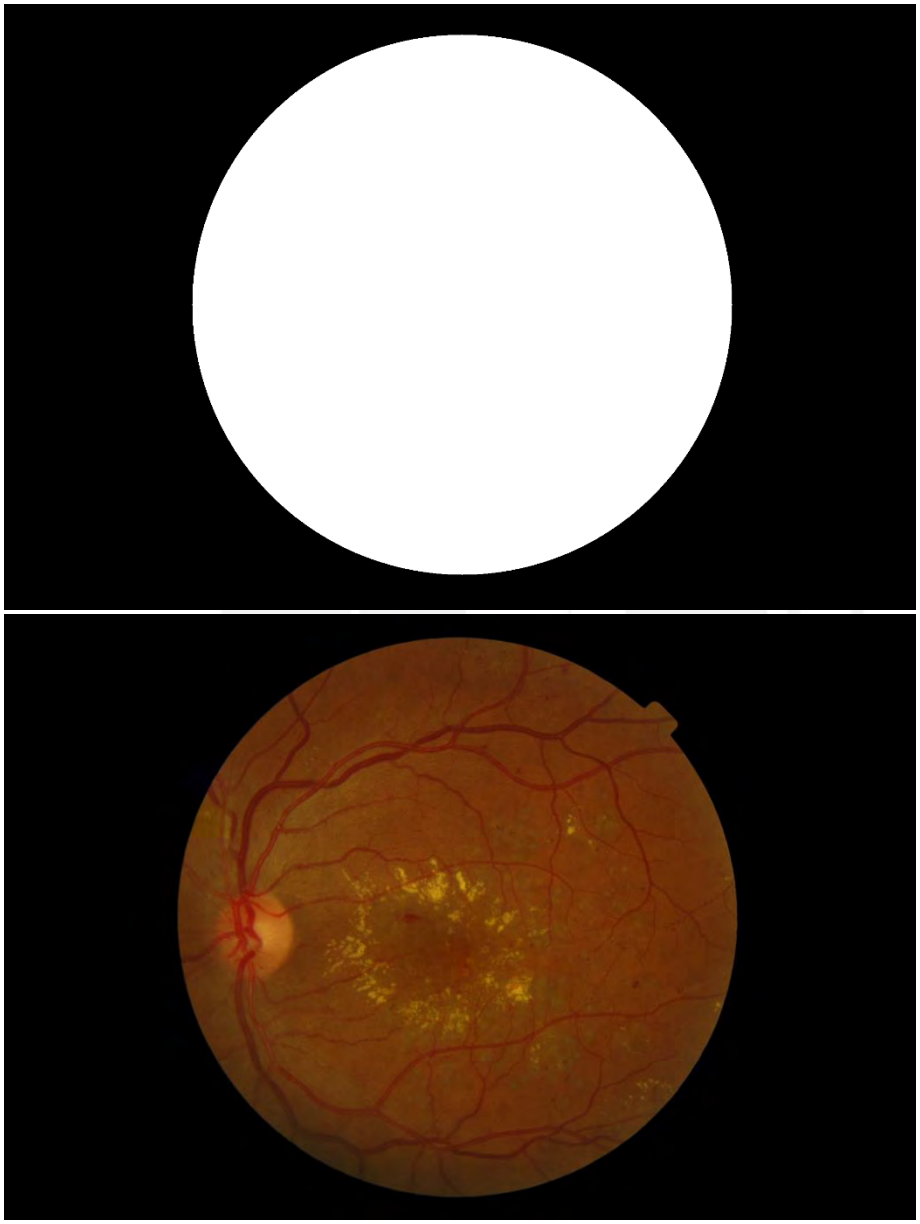
```
    dilate(greenChannel, i1, element);
    element = getStructuringElement(MORPH_ELLIPSE, Size(7, 7));
    dilate(greenChannel, i2, element);

    greenChannel = i1 - i2;
    threshold(greenChannel, greenChannel, 0.04 * 255, 255,
CV_THRESH_BINARY);

    Mat c1, c2;
    element = getStructuringElement(MORPH_ELLIPSE, Size(5, 5));
    dilate(greenChannel, c1, element);
    erode(c1, c1, element);

    element = getStructuringElement(MORPH_ELLIPSE, Size(23, 23));
    dilate(greenChannel, c2, element);
    erode(c2, c2, element);

    bitwise_and(c1, l_chann, c1, mask);
    bitwise_and(c2, l_chann, c2, mask);


    Mat i5, i6;
    bitwise_xor(c1, c2, i5, mask);
    bitwise_and(c1, c2, i6, mask);

    Mat finalOutput;

    finalOutput = i5 + i6;

    element = getStructuringElement(MORPH_ELLIPSE, Size(3, 3));
    erode(finalOutput, finalOutput, element);
    dilate(finalOutput, finalOutput, element);


    vector<vector<Point> > contours;
    vector<Vec4i> hierarchy;

    findContours(finalOutput, contours, hierarchy, CV_RETR_CCOMP,
CHAIN_APPROX_SIMPLE);

    vector<vector<Point> > filteredContours;
    double k, area;
    //Se marca las zonas brillosas encontradas
    Rect r;
    Mat maImage = Mat::zeros(finalOutput.rows, finalOutput.cols, finalOutput.type());
    for (int i = 0; i < contours.size(); i++) {
        drawContours(maImage, contours, i, Scalar(255), CV_FILLED, 8, hierarchy, 0,
Point());
```

```
        area = contourArea(contours.at(i));
        r = boundingRect(contours.at(i));
        if (r.height > r.width) {
            max = r.height;
            min = r.width;
        } else {
            max = r.width;
            min = r.height;
        }
        Scalar m = mean(gc(r), maImage(r));
        Scalar mImgOrig=mean(image(r),maImage(r));
        k = min / max;

        if ((k >= 0.4) && area >= 10 && area >= 0.3 * r.height * r.width &&
m.val[0]>=mImg.val[0] && mImgOrig.val[0]<=50 && abs(mImgOrig.val[1]-
mImgOrig[2])<=100) filteredContours.push_back(contours.at(i));


    }

    maImage = Mat::zeros(finalOutput.rows, finalOutput.cols, finalOutput.type());
    for (int i = 0; i < filteredContours.size(); i++) {
        drawContours(maImage, filteredContours, i, Scalar(255), CV_FILLED, 8,
hierarchy, 0, Point());
    }
    //Se marca el vector de características con el resultado obtenido
    ci.areaBrightZone = countNonZero(maImage);
    ci.numberBrightZones = filteredContours.size();
}
```

## 2. Imágenes por fase

### 2.1. Extracción del background

Imágenes resultantes luego de extraer el fondo de las imágenes retinográficas.



### 2.2. Extracción del disco óptico con máscara

Máscara resultante luego de aplicar la transformada hough a la imagen.

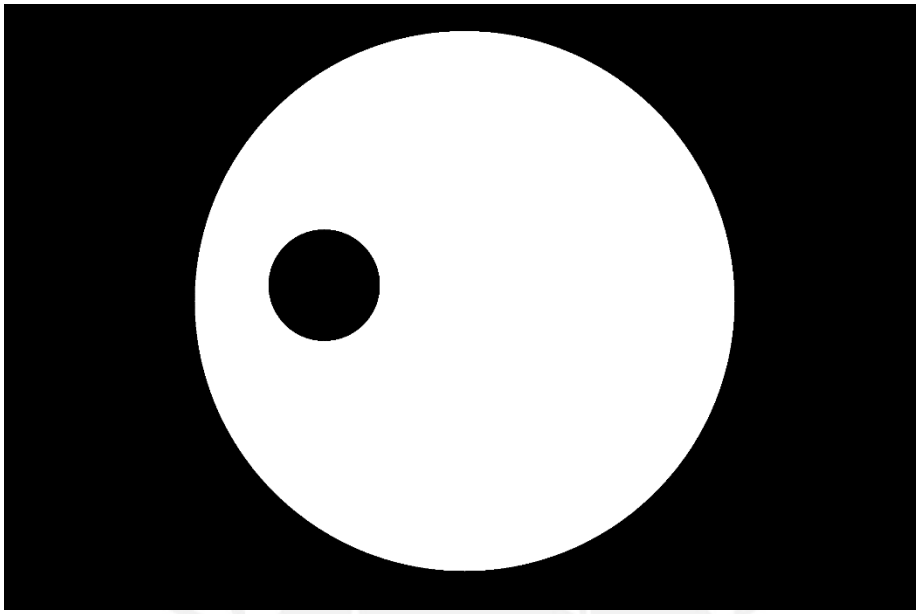
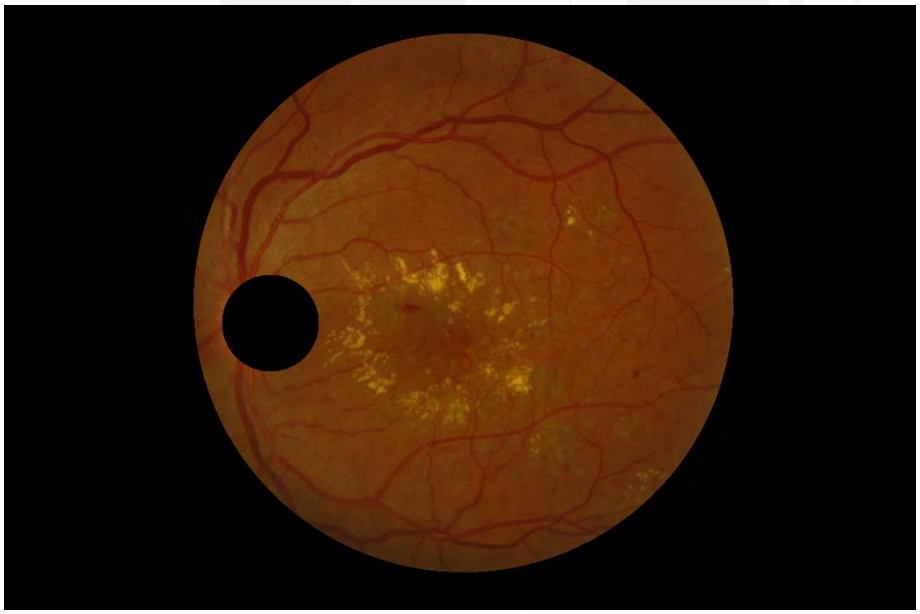


Imagen resultante luego de aplicar la máscara anterior.



### 2.3. Extracción de las lesiones oscuras (microaneurismas y hemorragias)

Hemorragias y microaneurismas detectadas en la imagen.

### 2.4. Extracción de lesiones brillosas (exudados)

Exudados detectados en la imagen.

## 3. Archivo de configuración del clasificador SVM

Este archivo contiene todos los datos de los hiperplanos generados a partir del vector de características.


```
<?xml version="1.0"?>
<opencv_storage>
<opencv_ml_svm>
  <format>3</format>
  <svmType>C_SVC</svmType>
  <kernel>
    <type>RBF</type>
    <gamma>9.3750000000000000e-002</gamma></kernel>
  <C>2.5000000000000000e+000</C>
  <term_criteria><iterations>1000</iterations></term_criteria>
  <var_count>4</var_count>
  <class_count>4</class_count>
  <class_labels type_id="opencv-matrix">
    <rows>4</rows>
    <cols>1</cols>
    <dt>i</dt>
    <data>
      0 1 2 3</data></class_labels>
  <sv_total>581</sv_total>
```

4. Archivo de configuración del clasificador SVM

Vector de características:

- Área de lesiones oscuras
- Número de elementos de lesiones oscuras
- Área de lesiones brillosas
- Número de elementos de lesiones brillosas
- Severidad de la retinopatía
    - 0: Normal (Sin retinopatía)
    - 1: Leve
    - 2: Moderado
    - 3: Severo

| Área lesiones oscuras | Num Lesiones Oscuras | Área lesiones brillosa | Num Lesiones Brillosas | Severidad de Retinopatía |
|---|---|---|---|---|
| 0 | 0 | 402 | 11 | 0 |
| 0 | 0 | 917 | 27 | 0 |
| 0 | 0 | 887 | 14 | 0 |
| 42 | 1 | 437 | 10 | 0 |
| 41 | 1 | 86 | 3 | 0 |
| 30 | 1 | 74 | 2 | 0 |
| 77 | 2 | 127 | 3 | 0 |
| 0 | 0 | 401 | 11 | 0 |
| 0 | 0 | 600 | 12 | 1 |
| 226 | 3 | 195 | 6 | 1 |
| 0 | 0 | 110 | 4 | 1 |
| 0 | 0 | 361 | 10 | 1 |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 432 | 6 | 1 |
| 374 | 7 | 0 | 0 | 2 |
| 72 | 1 | 2954 | 47 | 2 |
| 40 | 1 | 1772 | 37 | 2 |
| 0 | 0 | 161 | 4 | 2 |
| 42 | 1 | 1019 | 11 | 2 |
| 0 | 0 | 817 | 9 | 2 |
| 464 | 5 | 224 | 9 | 3 |
| 385 | 5 | 0 | 0 | 3 |
| 0 | 0 | 212 | 5 | 3 |
| 555 | 9 | 138 | 2 | 3 |
| 148 | 1 | 735 | 12 | 3 |
| 598 | 9 | 896 | 19 | 3 |
| 717 | 10 | 377 | 5 | 3 |