Cascade Classification K-Means

Dayan Bravo Fraga

March 2023

1 Instrucciones del Proyecto

El objetivo de este proyecto es el de hace una clasificación en cascada de una imagen a color en formato CIE-Lab, utilizando únicamente los componentes ab de cada pixel.

La clasificación en cascada opera como sigue: en un primer nivel, cada pixel de la imagen se clasificará como perteneciente a dos clases. Esto se realizará utilizando el algoritmo de k-medias, con un valor de K=2. A los pilos clasificados en cada una de las dos clases, se clasificarán a su vez en otras dos posibles clases, utilizando también el algoritmo de k-medias. Se procede de esta manera con cada subclase encontrada hasta que no pueda ser subdividida. El algoritmo deberá producir un árbol binario en donde en cada iteración, a partir de las hojas del árbol, se pueda generar una imagen que sería una versión simplificada, en cuanto al número de colores, de la imagen original.

El algoritmo de K-medios se debe ejecutar primero utilizando la distancia Euclidiana como métrica, hasta que converja o hasta que el número de píxeles que cambia de clase es lo suficientemente pequeño. Una vez que ocurra eso, se deberá continuar ejecutando, pero utilizando la distancia de Mahalanobis como métrica.

Se debe entregar el programa funcionando, y un reporte en donde se reporte lo hecho y los resultados obtenidos.

2 Solución

En esta sección se describe el algoritmo utilizado para resolver el problema planteado. Al mismo tiempo, se explican los pasos que se deben seguir para poder ejecutar el programa. Así como un ejemplo de ejecución del programa utilizando una imagen de prueba.

2.1 Preparación del ambiente

Para poder ejecutar el programa, se debe tener instalado el compilador de C++ `g++` y la herramienta `make`. Además, se debe tener instalado el paquete `libopencv-dev` para poder utilizar la librería `OpenCV`.

Para instalar `libopencv-dev`, se debe ejecutar el Comando 1.

Listing 1: Install OpenCV

```
sudo apt-get install libopencv-dev
```

2.1.1 Compilación

Para compilar el programa, se debe ejecutar el Comando 2.

Listing 2: Compilation

```
make
```

2.1.2 Ejecución

El programa tiene un parámetro opcional, que es el nombre de la imagen a utilizar.
Si no se especifica el nombre de la imagen, se utilizará una imagen por defecto.
Que es una imagen de prueba de 3x3.
Para ejecutar el programa, se debe ejecutar el siguiente comando:

Listing 3: Ejecution

```
./main [image_name]
```

2.2 Algoritmo

2.2.1 Importar imagen

Importar imagen a color en formato CIE-Lab.
Para esto se utiliza la Función 4.

Listing 4: Load Image

```
void ImageRepo::byName(
    Mat &image,
    Mat &imageOriginal,
    const string &name
    ) {
    // Read image.
    imageOriginal = imread(name);
    // Convert to float values.
    Mat imageFloat;
    imageOriginal.convertTo(imageFloat, CV_32FC3);
    // Normalize.
    imageFloat /= 255.0;
    // Convert to BGR2Lab.
    cvtColor(imageFloat, image, COLOR_BGR2Lab);
}
```

En este ejemplo, se utilizará la imagen de la Figura 1.

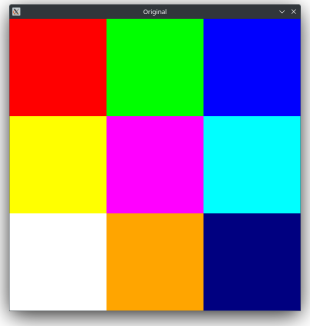

Figure 1: Imagen de prueba de 3x3.

2.2.2 Generar máscara

Se genera una máscara de la misma dimensión que la imagen, que contiene las clases a las que pertenece cada pixel.
(Inicialmente, todos los píxeles pertenecen a la misma clase "0").

El Código 5 muestra como se inicializa la máscara.

Listing 5: Compilation

```
// init mask with zeros.
mask = Mat::zeros(image.size(), CV_8UC1);
```

Al inicializar la máscara, todos los píxeles pertenecen a la misma clase, por lo que la máscara se inicializa con todos los píxeles con el valor 0.

La máscara es la siguiente matriz:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

2.2.3 Calcular estadísticas

Ahora se procede a calcular las estadísticas (vector promedio y matriz de covarianza) de cada clase.

(Como solo existe la clase 0, se calculan solo las estadísticas de esta clase).

Para esto se utiliza la Función 6.

Listing 6: Mean And Covariance

```
int MatTools::meanCovariance(Mat &image, Mat &mask, map<int,
    ImageStats> &stats, vector<int> &labels, bool covariance) {
    auto result = MatTools::mean(image, mask, stats, labels);
    if (0 != result) {
        return result;
    }
    if (covariance) {
        // calculate covariance
        auto itMask = mask.begin<uchar>();
        auto itMaskEnd = mask.end<uchar>();
        auto itImage = image.begin<Vec3f>();

        while (itMask != itMaskEnd) {
            auto label = *itMask;
            auto &imageStats = stats[label];
            auto a = (*itImage)[1];
            auto b = (*itImage)[2];
            auto ad = a - imageStats.mean.a;
            auto bd = b - imageStats.mean.b;
            imageStats.covariance.s1 += ad * ad;
            imageStats.covariance.s2 += bd * bd;
            imageStats.covariance.s3 += ad * bd;
            itMask++;
            itImage++;
        }

        for (auto &item: stats) {
            auto &imageStats = item.second;
            auto &count = imageStats.count;
            if (count < 2) {
                continue;
            }
            auto iCont = (float) (1. / (count - 1));
            imageStats.covariance.s1 *= iCont;
            imageStats.covariance.s2 *= iCont;
            imageStats.covariance.s3 *= iCont;
        }

    }
    return 0;
}
```

Como se puede observar, las estadísticas se almacenan en un diccionario, donde la clave es el número de la clase y el valor es una estructura de tipo `ImageStats`.

La estructura `ImageStats` contiene el vector promedio y la matriz de covarianza de la clase.

Lo podemos ver en Código 7.

Listing 7: Image Stats

```cpp
struct ImageStats {
    // vector of Lab averages
    struct {
        float l = 0;
        float a = 0;
        float b = 0;
    } mean;
    // covariance matrix
    struct {
        float s1 = 1;
        float s2 = 1;
        float s3 = 0;
    } covariance;
    // number of pixels
    int count = 0;
    // class label
    int label = 0;
};
```

Siguiendo con el ejemplo, ahora tenemos disponible la media y la matriz de covarianza de la clase 0.

En la Figura 2, depuramos el código y podemos ver estas estadísticas:

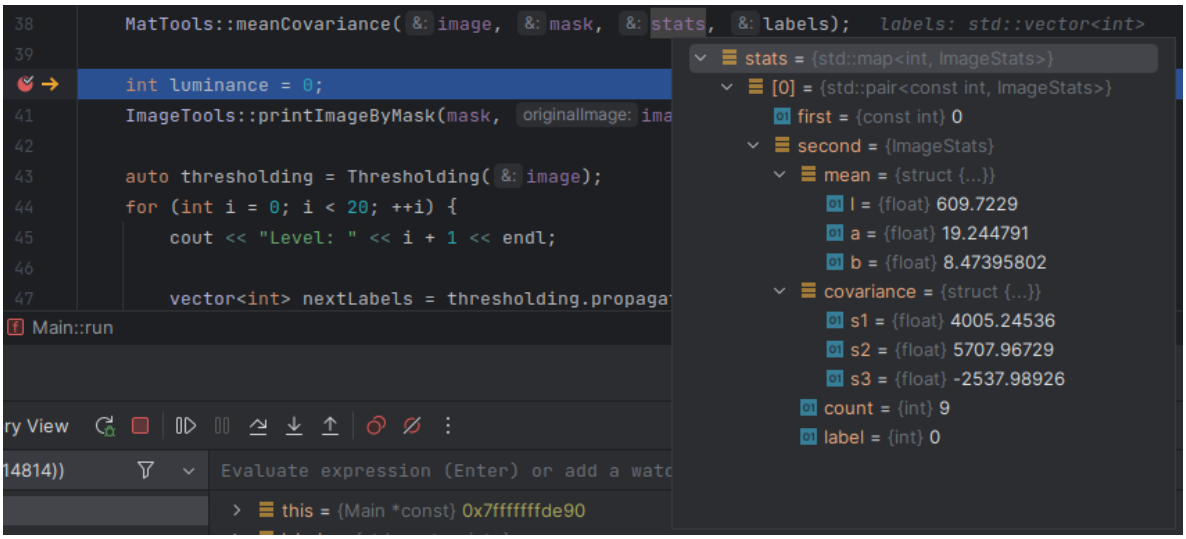

Figure 2: Estadísticas de la clase 0.

2.2.4 Construir imagen

Ahora, con información de la máscara y las estadísticas podríamos construir una imagen en formato CIE-Lab, donde cada píxel sea el vector promedio de la clase a la que pertenece.

La Figura 3 nos muestra la imagen construida con la clase 0.

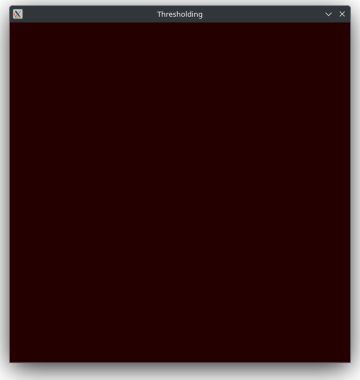

Figure 3: Imagen construida con la clase 0 (Luminosidad: 0).

Tener en cuenta que la luminosidad de la imagen es 0.
El objetivo de esto es que se vean iguales los pixeles que pertenecen a la misma clase.

2.2.5 Dividir máscara

Ahora, se procede a dividir la máscara (en este caso la 0) en dos nuevas máscaras (1 y 2).
Esto se realiza con la Función 8.

Listing 8: Divide Class

```
vector<int> Thresholding::divideClass(Mat &mask, int label) {
    int newLabelA = Thresholding::lastLabel++;
    int newLabelB = Thresholding::lastLabel++;
    int changesCount = 0;
    auto it = mask.begin<uchar>();
    auto itEnd = mask.end<uchar>();
    for (; it != itEnd; ++it) {
        if (*it == label) {
            *it = (0 == (changesCount % 2)) ? newLabelA : newLabelB;
            changesCount++;
        }
    }
    return vector<int>({newLabelA, newLabelB});
}
```

El cual vemos que pide por parámetro la máscara y la clase a dividir.
Se decidió para este caso ir alternando la asignación de las nuevas clases, de esta forma se garantiza que no se generen clases vacías.
La función devuelve un vector con los dos nuevos labels en un vector de enteros.

2.2.6 Distancia de Mahalanobis

El resultado de la acción anterior nos devuelve una nueva máscara con dos clases.
Pero debemos hacer un proceso de umbralización para determinar agrupar los píxeles en la clase de la que más se acerquen.
Para esto vamos a necesitar una función que nos permita calcular la distancia de Mahalanobis entre un píxel y una clase.
La Función 9 nos permite calcular la distancia de Mahalanobis entre un píxel y una clase.

Listing 9: Mahalanobis Distance

```
double MatTools::distanceMahalanobisNormalized(double a, double b,
    ImageStats imageStats) {
     auto &mean = imageStats.mean;
     auto &covariance = imageStats.covariance;
     auto ad = a - mean.a;
     auto bd = b - mean.b;
     auto s1 = covariance.s1;
     auto s2 = covariance.s2;
     auto s3 = covariance.s3;
     auto d = (ad * ad) / s1 + (bd * bd) / s2 - 2 * (ad * bd) / s3;

     return ad * ad + bd * bd;
}
```

La función recibe como parámetros el píxel a calcular la distancia y la estructura de estadísticas de la clase.

2.2.7 Reordenar píxeles

Esta función que calcula la Distancia de Mahalanobis será utilizada en la Función 10 `reorderPixels` para reordenar los píxeles de la máscara.

Listing 10: Reorder Pixels

```
int Thresholding::reorderPixels(
        Mat &mask,
        vector<int> &labels,
        map<int, ImageStats> &stats,
        bool &hasClassA,
        bool &hasClassB
) {
    int classA = labels[0];
    int classB = labels[1];
    ImageStats imageStatsClassA = stats[classA];
    ImageStats imageStatsClassB = stats[classB];

    auto itImage = this->image.begin<Vec3f>();
    auto itMask = mask.begin<uchar>();
    auto itEnd = mask.end<uchar>();
    int changesCount = 0;
    while (itMask != itEnd) {
        auto &label = *itMask;
        if (classA != label && classB != label) {
            itImage++;
            itMask++;
            continue;
        }
        Vec3f pixel = *itImage;
        float a = pixel[1];
        float b = pixel[2];
        // Calculate distance.
        double distanceA = MatTools::distanceMahalanobisNormalized(a,
           b, imageStatsClassA);
        double distanceB = MatTools::distanceMahalanobisNormalized(a,
           b, imageStatsClassB);
        if (distanceA < distanceB) {
            hasClassA = true;
            if (classA != label) {
                label = classA;
                changesCount++;
            }
        } else {
            hasClassB = true;
            if (classB != label) {
                label = classB;
                changesCount++;
            }
        }
        itImage++;
        itMask++;
    }

    return changesCount;
}
```

Esta función recibe por parámetro la máscara, los labels de las clases a reordenar, las estadísticas de las clases y dos variables booleanas que indican si la clase A y la clase B tienen píxeles asignados.

2.2.8 Umbralizar

Ahora con estas funciones disponibles, podemos proceder a umbralizar la imagen.

La Función 11 `toThreshold` es la encargada de realizar el proceso de umbralización.

Listing 11: Mahalanobis Distance

```
int Thresholding::toThreshold(
        Mat &mask,
        vector<int> &labels,
        map<int, ImageStats> &stats,
        bool &hasClassA,
        bool &hasClassB,
        bool covariance
) {
    int limit = covariance ? iterationsLimitWithConvergence :
       iterationsLimitWithoutConvergence;
    for (int i = 0; i < limit; i++) {
        cout << "Iteration: " << i << endl;

        hasClassA = false;
        hasClassB = false;

        int classA = labels[0];
        int classB = labels[1];

        cOutImageStats(stats[classA]);
        cOutImageStats(stats[classB]);

        auto changesCount = this->reorderPixels(
                mask,
                labels,
                stats,
                hasClassA,
                hasClassB
        );

        if (0 == changesCount) {
            cout << "El algoritmo convergi  en " << i << "
               iteraciones." << endl;
            return 0;
        } else {
            cout << "Hubo: " << changesCount << " cambios." << endl;
            MatTools::meanCovariance(this->image, mask, stats, labels,
                covariance);
        }
    }

    return 1;
}
```

La función recibe por parámetro la máscara, los labels de las clases a umbralizar, las estadísticas de las clases, dos variables booleanas que indican si la clase A y la clase B tienen píxeles asignados y un booleano que indica si se debe utilizar la covarianza o no.

Este último parámetro es utilizado en las primeras iteraciones para que se utilice la distancia Euclidiana (Mahalanobis sin covarianza) y en las siguientes iteraciones se utilice la distancia de Mahalanobis.

2.2.9 Primera iteración

Luego de la umbralización, se obtiene una máscara con dos clases (1 y 2) en la cual, cada clase contiene píxeles más cercanos.

Para el ejemplo que estamos utilizando, la máscara resultante es la siguiente:

$$\begin{bmatrix} 2 & 2 & 1 \\ 2 & 1 & 2 \\ 2 & 2 & 1 \end{bmatrix}$$

Si construimos la imagen a partir de la máscara, obtenemos la siguiente imagen:

La Figura 4 nos muestra la imagen construida con la máscara actual (conformada por las clases 1 y 2).

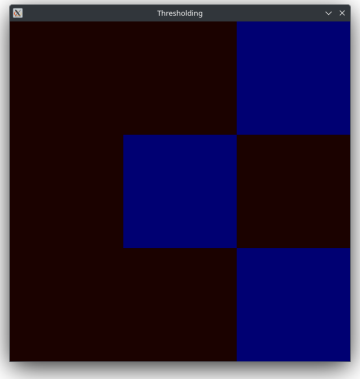

Figure 4: Imagen construida con las clases 1 y 2 (Luminosidad: 0).

Como podemos observar, todos los píxeles que pertenecen a la clase 1 se observan del mismo color (rojo oscuro) y los píxeles que pertenecen a la clase 2 se observan del mismo color (azul oscuro).

2.2.10 Propagación

Lo siguiente que debemos hacer es continuar iterando para generar más clases y así obtener una imagen con más colores.

En la iteracion siguiente, se espera que se generen para la clase 1, dos clases más (3 y 4) y para la clase 2, dos clases más (5 y 6).

Este proceso se repite hasta que las clases no se puedan dividir más.

La Función 12 `propagate` se encarga de la propagación de las clases.

Listing 12: Función propagate.

```
vector<int> Thresholding::propagate(Mat &mask, vector<int> &labels,
    map<int, ImageStats> &stats) {
    // labels que de deben propagar en la siguiente iteracion
    vector<int> nextLabels = {};

    for (int label: labels) {
        auto count = this->countClass(mask, label);
        if (count < 2) {
            continue;
        }
        vector<int> newLabels = this->divideClass(mask, label);

        MatTools::meanCovariance(this->image, mask, stats, newLabels,
            false);

        bool hasClassA = false, hasClassB = false;
        this->toThreshold(
                mask,
                newLabels,
                stats,
                hasClassA,
                hasClassB,
                false
        );

        if (hasClassA && hasClassB) {
            this->toThreshold(
                    mask,
                    newLabels,
                    stats,
                    hasClassA,
                    hasClassB,
                    true
            );
        }
        if (hasClassA) {
            int classA = newLabels[0];
            if (hasClassB) {
                nextLabels.push_back(classA);
            }
        }
        if (hasClassB) {
            int classB = newLabels[1];
            if (hasClassA) {
                nextLabels.push_back(classB);
            }
        }
    }
    return nextLabels;
}
```

La función recibe por parámetro la máscara, los labels de las clases a propagar, las estadísticas de las clases y devuelve los labels de las clases que se deben propagar en la siguiente iteración.