Tarea 3: Clasificación de Raza

EL7008 – Procesamiento Avanzado de Imágenes

Nombre:	Sebastián Parra
Profesor:	Javier Ruiz del Solar
Auxiliar:	Patricio Loncomilla
Ayudante:	Francisco Leiva
	Nicolás Cruz
Fecha:	5 de octubre de 2018

Introducción

El presente informe trata sobre las actividades realizadas en el marco de la tercera tarea del curso EL7008 – Procesamiento Avanzado de Imágenes, cuyo objetivo es lograr una familiarización con la implementación de histogramas LBP uniformes y algoritmos clasificadores para aplicaciones de clasificación de rostros humanos, estudiando los grados de libertad que debe manejar un diseñador de este tipo de sistemas, y cómo cada parámetro influye en los resultados obtenidos, todo esto utilizando librerías estándares de procesamiento de imágenes (OpenCV en C++).

Para lograr estos objetivos, la tarea consistió en diseñar un sistema que permita clasificar rostros humanos según su raza, existiendo las categorías “Asian”, “Black”, “Indian”, “Others”, y “White”. Para esto, se entregó una base de datos de 1000 imágenes, 200 muestras de cada clase, de 200 x 200 pixeles, a las cuales se les calculó sus histogramas LBP uniformes considerando 4 regiones por imagen, para luego ser introducidos en clasificadores tipo SVM y Random Forest para obtener predicciones, utilizando un 70% de las imágenes para entrenamiento y el resto para el conjunto de prueba. Finalmente, se solicitó utilizar este procedimiento para resolver los problemas de clasificación binaria (usando sólo las clases “Asian” y “Black”) y el problema completo con las 5 clases, comparando cómo van variando los resultados dependiendo del tipo de clasificador utilizado y sus parámetros escogidos.

A lo largo de este documento se presenta un marco teórico donde se explican las principales herramientas utilizadas para resolver el problema propuesto junto a los conocimientos necesarios para comprender la solución implementada, seguido de una sección de desarrollo y resultados donde se explicará cómo se realizó la programación de cada paso del procedimiento pedido y se mostrarán los resultados de aplicar el sistema diseñado para resolver el problema de clasificación binaria y multiclase, para finalmente obtener conclusiones sobre los aprendizajes obtenidos con esta tarea, mencionando las dificultades encontradas y si los resultados obtenidos concuerdan con lo esperado teóricamente.

Marco Teórico

Para lograr comprender la programación de la solución de esta tarea, es necesario describir algunos conceptos, los cuales serán explicados en esta sección.

Transformada LBP (Local Binary Patterns)

La transformada LBP es un descriptor utilizado comúnmente para resolver problemas de clasificación en visión computacional debido a su capacidad de ser un poderoso descriptor de texturas, a la vez utilizando mínimos recursos computacionales. El algoritmo para la obtención del vector de características se describe en los siguientes pasos:

1. Para cada pixel en la imagen a transformar, se compara su valor con el de sus vecinos, los cuales estarán dispuestos según una circunferencia con centro en el pixel a examinar y un radio definido por el usuario. Luego, los vecinos corresponderán a los n puntos resultantes de dividir la circunferencia en n valores equidistantes, realizando interpolaciones en caso de que algún punto no quedase justo en el centro de un pixel. Se debe notar que, para el caso de esta tarea, se eligió n = 8 y un radio de 1, por lo que los vecinos examinados correspondieron a los pixeles que rodean el pixel examinado. En la figura 1 se pueden apreciar distintas elecciones de vecinos para distintos radios y números de puntos.

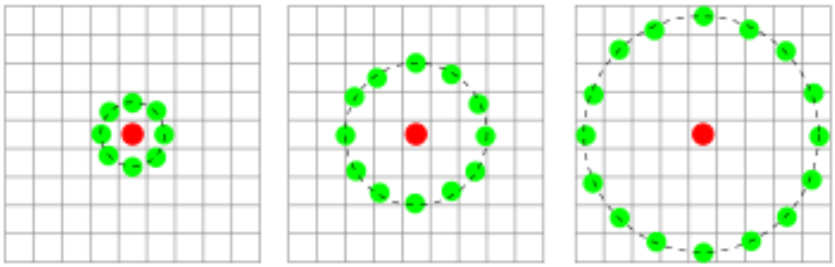

Figura 1: Distintas configuraciones para realizar la selección de vecinos en LBP: (izquierda) R=1, n=8; (centro) R=2.5, n=12; (derecha) R=4, n=16.

2. Recorriendo los vecinos en sentido horario, si el vecino es mayor al valor central, se escribe un "1", en caso contrario se escribe un "0", como se muestra en la figura 2.

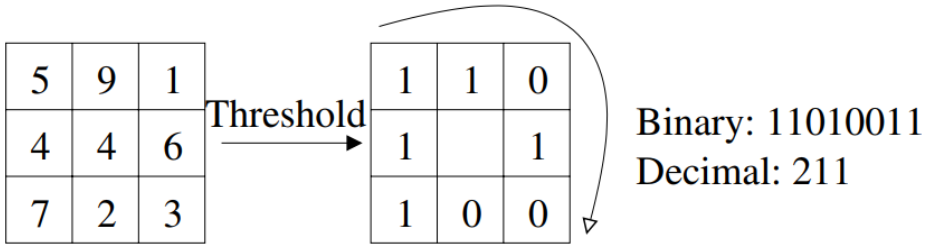

Figura 2: Visualización de la construcción del string binario de una transformada LBP. (Fuente: T. Ahonen, A. Hadid, M. Pietikainen, "Face recognition with local binary patterns.)

3. Esta combinación de ceros y unos se concatena para dar como resultado un número binario de 8 dígitos, el cual es transformado a decimal (un valor entre 0 y 255) y se guarda como el valor de la transformada en la posición correspondiente al pixel que se estaba analizando.

Una vez calculada la transformada LBP para todos los pixeles en la figura original (salvo los pixeles del borde, que suelen ser descartados al no tener todos sus vecinos definidos), se puede imprimir en pantalla la imagen resultante, obteniendo dibujos similares a los que se muestran en la figura 3.

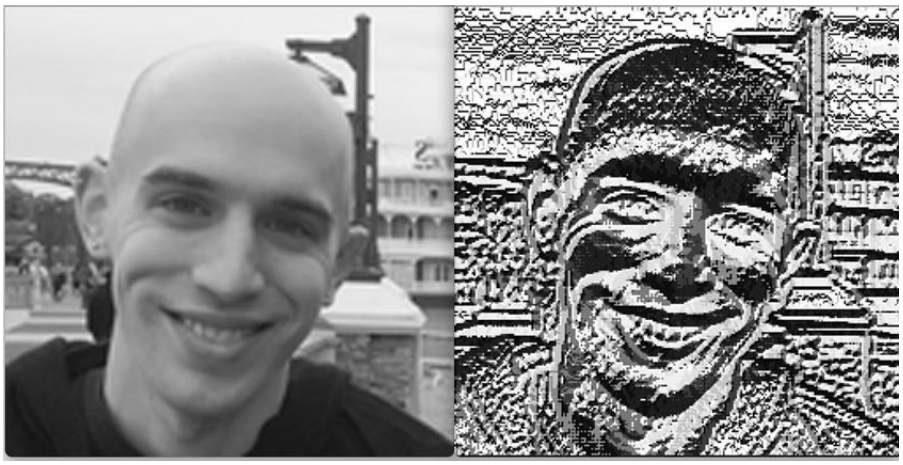

Figura 3: Visualización de un rostro humano (izquierda) y su transformada LBP (derecha).

Descriptor LBP uniforme

Una vez que se creó el descriptor LBP y se demostró su buen desempeño en problemas de clasificación, surgieron una serie de trabajos que intentaron mejorar el rendimiento del algoritmo realizándole pequeñas modificaciones, y una de las más populares es la variante uniforme del descriptor. Ésta trata de compactar la información útil obtenida a través de un LBP estándar al introducir el concepto de patrones uniformes: En la sección anterior, se mencionó que el algoritmo LBP busca obtener un código de 8 bits, el cual es convertido a entero y reemplaza a su pixel correspondiente. En este contexto, se define como uniforme a todos los códigos (o patrones) que contengan a lo más dos transiciones de bits, o sea, a lo más dos cambios de 0 a 1 o de 1 a 0. Así, el descriptor LBP uniforme asigna un valor entero a cada cadena de números que resulte uniforme, mientras que los patrones que no lo sean serán mapeados a un único entero. De esta manera, se reduce el número de valores posibles que puede tomar este descriptor de 256 a 59 (existen 58 códigos uniformes de 8 bits, más un valor para los no uniformes).

Una explicación de por qué esta modificación resulta útil, además de por la compresión de la información, es debido a que la gran mayoría de las texturas importantes en imágenes, ya sea regiones planas, bordes, y esquinas, presentan características uniformes, como se puede apreciar en la figura 4.

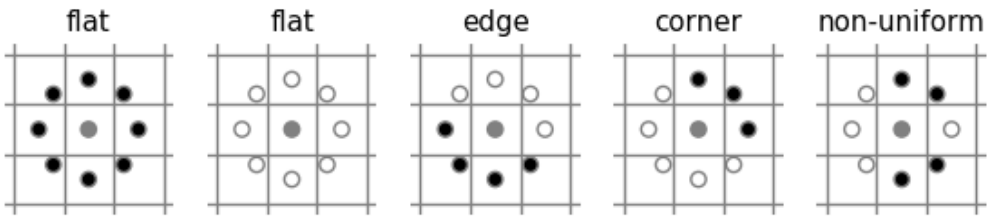

Figura 4: Diagrama ilustrativo que explica la intuición detrás de considerar sólo los patrones uniformes, notando que las texturas más importantes en imágenes presentan esta característica.

Histogramas LBP

La generación de histogramas corresponde al último paso del proceso de obtención de descriptores LBP de una imagen. La idea de utilizar histogramas es que permiten compactar aún más la información contenida en la imagen LBP, aunque esto viene con el costo de perder información regional de las imágenes. Debido a esto, se opta por utilizar una solución intermedia, que consiste en dividir cada imagen en 4 regiones (obteniendo 4 submatrices para las esquinas superior izquierda, superior derecha, inferior izquierda, e inferior derecha), calcular los histogramas LBP uniformes de cada región, y finalmente obtener un descriptor para la figura que consiste en la concatenación de los 4 histogramas, como se puede apreciar en la figura 5. Como cada histograma corresponde a un vector que indica cuántas veces aparece cada intensidad de pixel en cada una de las regiones de la imagen LBP uniforme, se tendrá que cada histograma almacenará un total de 236 datos.

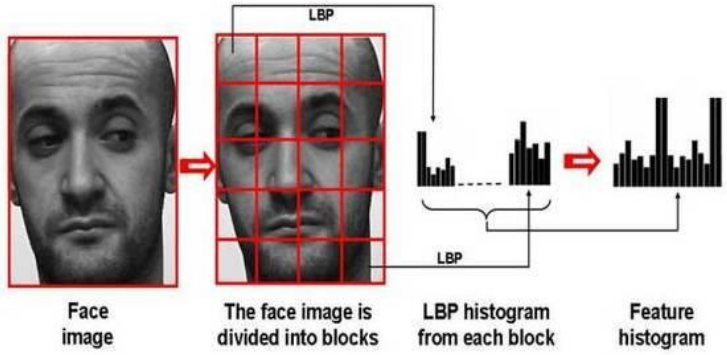

Figura 5: Diagrama ilustrativo del procedimiento de división de la imagen en regiones, el cálculo de histogramas LBP por región, y su concatenación.

Cabe destacar que, si bien en este trabajo se utilizó una división de 4 regiones, este valor se puede modificar, según el usuario estime conveniente.

Support Vector Machine (SVM)

Se llama SVM a una serie de modelos de aprendizaje supervisado utilizados para resolver problemas de clasificación y regresión. Estos algoritmos se basan en encontrar un hiperplano que separe los datos de distintas clases en el espacio de características. Para esto, se plantea un problema de optimización a resolver, el cual intenta encontrar el hiperplano que maximice el margen de separación entre las clases por medio de la elección de "vectores de soporte" que corresponden a un subconjunto de las muestras que actúan como puntos que determinarán la inclinación del hiperplano, como se puede apreciar en la figura 6.

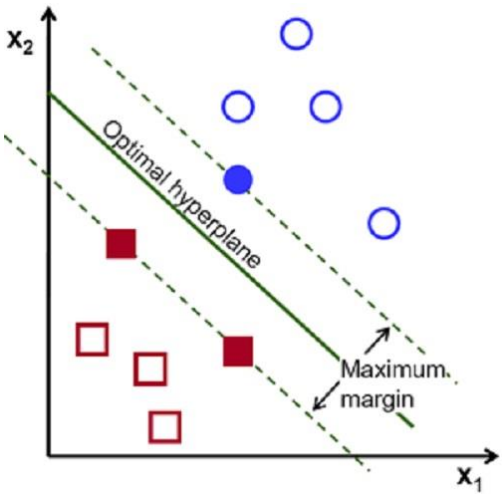

Figura 6: Imagen ilustrativa de cómo funciona una máquina de soporte vectorial

Adicionalmente, existen ciertos hiperparámetros que pueden modificar el comportamiento del clasificador, destacándose la variable C, que permite controlar cuánta importancia se le quiere dar al tamaño del margen y cuánta se le quiere dar a que todos los datos se encuentren correctamente clasificados, los cuales suelen ser metas mutuamente excluyentes (o sea, si se desea mejorar una, la otra debe empeorar). La variable C agrega un nuevo coeficiente a la función de costo que penaliza las malas clasificaciones de datos. De esta manera, valores muy altos de C tenderán a entregar resultados donde la

gran mayoría de los datos de entrenamiento se encuentren bien clasificados, aunque se corre el riesgo de que aumente el error de generalización por *overfitting*. En la figura 7 se puede apreciar un ejemplo simple de cómo el parámetro C puede influir en el resultado de una SVM.

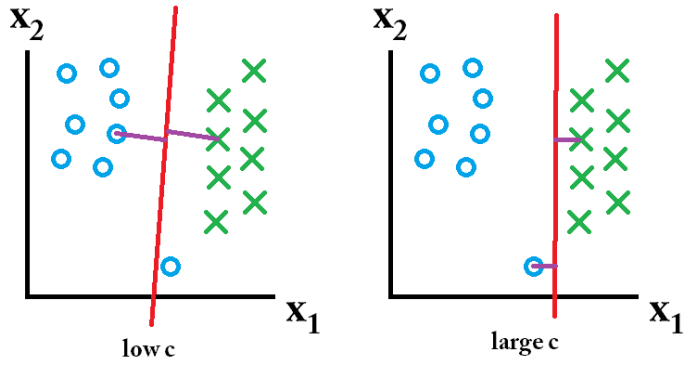

Figura 7: En la izquierda, el resultado de aplicar una SVM con C bajo para clasificar un conjunto determinado de muestras. En la derecha, el resultado de aplicar una SVM con C alto sobre el mismo conjunto de datos.

Si bien las figuras y definiciones anteriores dan una intuición de que las SVM sólo pueden encontrar un hiperplano separador lineal entre clases, esta limitación se soluciona utilizando una técnica conocida como el "truco del Kernel", la cual consiste en utilizar funciones Kernel para aumentar la dimensionalidad del espacio de características de los datos, encontrar un hiperplano separador en el nuevo espacio aumentado, y finalmente volver al espacio original para obtener el hiperplano separador no-lineal. Una visualización simplificada de este procedimiento se puede apreciar en la figura 8.

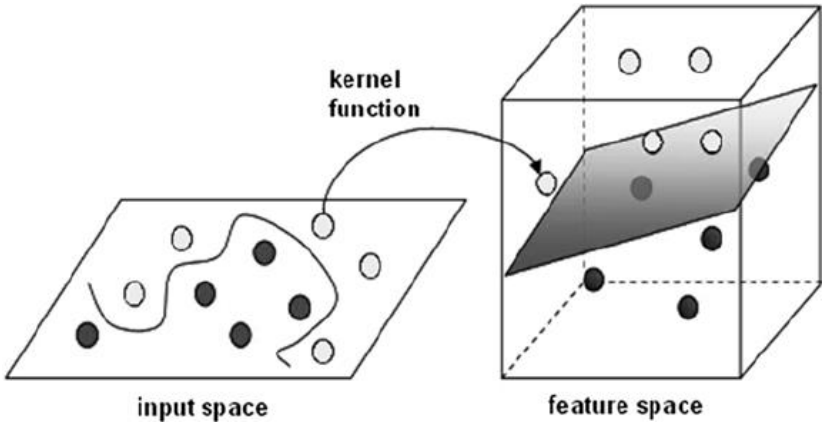

Figura 8: Diagrama simplificado de cómo funciona el truco del Kernel en una SVM

Sin embargo, para que el truco del Kernel funcione correctamente, es necesario utilizar una función de Kernel adecuada para el problema. Debido a esto, existen una serie de familias de kernels que son lo suficientemente flexibles para, luego de un ajuste de hiperparámetros, poder resolver prácticamente cualquier problema de clasificación con clases separables.

Por otro lado, otra limitación de las SVM que se podría inferir a través de lo explicado hasta ahora es que sólo funciona para problemas de clasificación binaria. Sin embargo, existen dos formas principales de solucionar esta limitación. La primera es conocida como SVM *one-vs-all* o *one-vs-rest*, y se basa en entrenar un clasificador para cada clase del problema, considerando como casos negativos a todo el resto

de las clases, lo cual resulta en tener que entrenar N máquinas de soporte distintas (donde N es el número de clases), pero viene con el costo de que, como se consideran todas las otras clases como ejemplos negativos, las clasificaciones se harán bajo condiciones de desbalance de clases, lo cual puede perjudicar mucho el desempeño del clasificador. Por otro lado, la segunda técnica se conoce como SVM *all-vs-all*, y consiste en entrenar un clasificador para cada par de clases, lo cual elimina el problema del desbalance de clases, aunque viene con el costo de tener que entrenar $\frac{N(N-1)}{2}$clasificadores, aumentando considerablemente los costos computacionales. Finalmente, en la figura 9 se pueden apreciar los distintos resultados que se pueden obtener al aplicar ambos tipos de SVM sobre un mismo conjunto de datos.

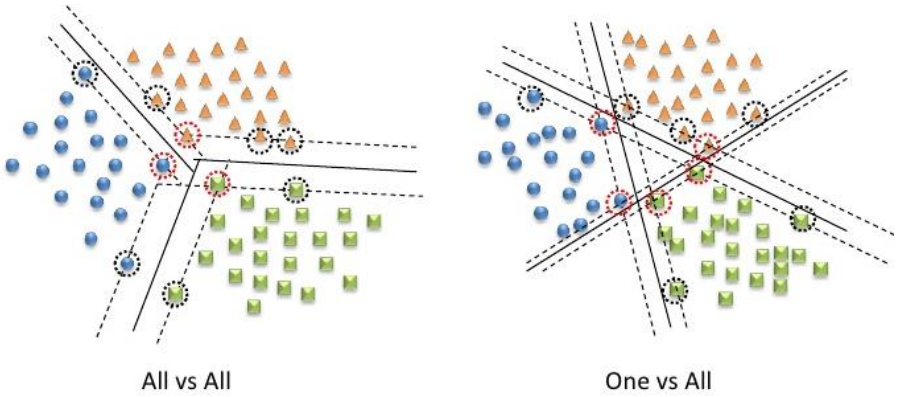

Figura 9: Comparación entre una SVM *all-vs-all* y una *one-vs-all* para una problema de clasificación multiclase

Random Forest

Los Random Forest son un método de aprendizaje por ensamble utilizado para resolver problemas de clasificación y regresión, que se basan en la aplicación de técnicas de bagging y la utilización de árboles de decisión para realizar la predicción de clases. En resumen, este método utiliza un algoritmo que genera B conjuntos de muestras aleatorias con reemplazo de los datos de entrenamiento, luego entrena B árboles de decisión con estos datos (a cada árbol le corresponde un único conjunto), donde en el proceso de entrenamiento de estos árboles se selecciona un subconjunto aleatorio de características cada vez que se crea una nueva división. Finalmente, para realizar predicciones se introducen las muestras a predecir en cada uno de los B árboles entrenados de manera paralela, donde la predicción final se realizará mediante un sistema de votación simple. Un diagrama de este procedimiento de predicción se muestra en la figura 10.

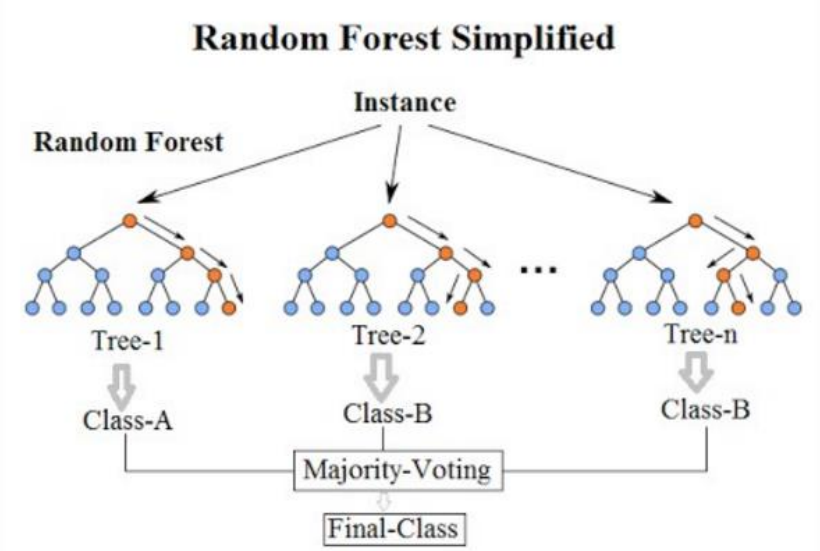

Figura 9: Diagrama simplificado que ilustra el procedimiento de predicción para un Random Forest.

Desarrollo y Resultados

Parte 1: Implementación de la transformada LBP uniforme

Para programar este requisito, primero se creó una función llamada *LBPTransform()*, la cual toma como argumento una matriz, y retorna su transformada LBP estándar (con valores entre 0 y 255), sin considerar sus esquinas, por lo que la imagen de salida tendrá dos filas y dos columnas menos que la original. A continuación, se presenta una descripción en palabras de cómo se implementó esta función.

1. Se convierte la imagen de entrada a escala de grises
2. Para cada pixel que no se encuentra en el borde, realizar las siguientes acciones (ver figura 10):
    a. Se inicializa el string LBP de 8 bits en cero.
    b. Se recorren los 8 vecinos del pixel en orden horario. Para cada uno de los vecinos, realizar las siguientes acciones:
        i. Se hace un desplazamiento a la izquierda del string LBP (equivalente a multiplicarlo por 2)
        ii. Si el pixel vecino es mayor al pixel central (el obtenido en 2.), se suma 1 al string de 8 bits
    c. Una vez obtenido el string de 8 bits final al haber recorrido todos los vecinos, se guarda este valor en la matriz de salida, en la posición equivalente a la ubicación del pixel central.

```cpp
// Get center pixel and reset binary pattern
float center = src_gray.at<float>(r,c);
unsigned char binPatt = 0;

// For each pixel, check its 8 neighbors in a clockwise manner
int neighR = r-1;
int neighC = c-1;
for(int direction=0; direction<4; direction++)
{
    for (int step=1; step<3; step++)
    {
        switch (direction) // This is to implement clockwise movement
        {
            case 0:
                neighC++;
                break;
            case 1:
                neighR++;
                break;
            case 2:
                neighC--;
                break;
            case 3:
                neighR--;
                break;
        }
        // Multiply binary pattern by 2 (equivalent to left bit shift)
        binPatt *= 2;
        // For each neighbor, check if I(neigh) > I(center). If true,
        // append a 1 to the binary pattern
        float neighbor = src_gray.at<float>(neighR, neighC);
        if (neighbor > center)
        {
            binPatt++;
        }
    }
}

// Insert LBP transform of center pixel in LBP imagen
LBP.at<unsigned char>(r-1, c-1) = binPatt;
```

Figura 10: Extracto de la función *LBPTransform()* que muestra cómo se fue obteniendo el patrón LBP para cada pixel original.

Posteriormente, se decidió utilizar una *lookup table* para poder transformar una imagen LBP a su variante uniforme. Para esto, se creó la función *getUniformLBPTable()*, la cual se dedica a crear una matriz de 1 fila y 256 columnas que asigna a cada número dentro de este rango, un valor entre 0 y 58, donde se asignó el cero como valor para los patrones no uniformes. En resumen, la función realiza las siguientes operaciones:

1. Se preasigna memoria para guardar la matriz de 1 fila y 256 columnas que corresponderá a la salida
2. Se inicializa el valor al cual el próximo número uniforme será mapeado en 1.
3. Para todos los strings de 8 bits entre 0 y 255, realizar las siguientes acciones (ver figura 11):
    a. Obtener una copia del número binario y aplicarle un desplazamiento de bit a la izquierda (equivalente a multiplicar por 2).
    b. Aplicar el operador binario XOR entre el número original y su copia desplazada.
    c. Contar el número de unos en el número binario resultante, lo cual será igual a el número de transiciones de bit del número original. Notar que si el número binario original termina en 1 (lo que es equivalente a que sea impar), se debe restar 1 a este valor, debido a que la operación de desplazamiento de bit fuerza la aparición de un 0 en el extremo derecho del string binario.

d. Si el número de transiciones es mayor a 2, el número se mapea a un 0 en la *lookup table*. Si esto no ocurre, el número se mapea al valor guardado en la variable que indica el valor al cual será mapeado el próximo número uniforme, y posteriormente se aumenta esta variable en 1

```
// Get a copy of i left-shifted and XOR it with original i value.
auto original = (unsigned char) i;
unsigned char shifted = original << 1;
unsigned char transitions = original ^ shifted;
size_t counts = std::bitset<8>(transitions).count();
// If least significant bit is 1, substract a value to number of transitions
if (i % 2 == 1)
{
    counts--;
}
// If number is not uniform, map to 0. Else, map to current value set for next uniform bitstring
if (counts > 2)
{
    lookup.at<unsigned char>(0, i) = 0;
}
else
{
    lookup.at<unsigned char>(0, i) = uniformMap;
    uniformMap++;
}
```

Figura 11: Extracto de la función getUniform*LBPTable()* que muestra cómo se fue obteniendo el valor de la *lookup table* para cada entero entre 0 y 255.

Una vez programadas estas dos funciones, la obtención de imágenes LBP uniformes consistió en, primero y antes que todo, ejecutar una única vez la función *getUniformLBPTable()* para obtener la *lookup table* que será utilizada para todas las imágenes que se desee procesar, luego, utilizar la función *LBPTransform()* para calcular la transformada LBP estándar de cada imagen a transformar, y finalmente, determinar la transformada uniforme mediante la utilización de la función *LUT()* , la cual aplica la *lookup table* obtenida anteriormente sobre la imagen LBP para obtener su variante uniforme.

Parte 2: Implementación de histogramas LBP

Para esta sección se creó una función llamada *uniformLBPHistogram()*, la cual, a partir de una imagen LBP y una lookup table, calcula su histograma LBP uniforme considerando una división de 4 regiones idénticas. Su funcionamiento se describe a continuación.

1. Mediante los pasos explicados en la sección anterior, se obtiene la transformada LBP uniforme de la imagen LBP estándar a partir de una *lookup table*.
2. Mediante objetos tipo *Rect()* de OpenCV, se obtienen referencias a las 4 submatrices que representan las regiones de la transformada LBP uniforme, correspondientes a esquinas superior-izquierda, superior-derecha, inferior-izquierda, e inferior-derecha, como se puede apreciar en la figura 12.

```
// Get uniform LBP values with lookup table and split matrix in 4 regions
// (top-left, top-right, bottom-left, bottom-right)
cv::Mat TL, TR, BL, BR;
cv::LUT(imLBP, lookupTable, uniformLBP);

TL = uniformLBP(cv::Rect(0, 0, uniformLBP.cols/2, uniformLBP.rows/2));
TR = uniformLBP(cv::Rect(0, uniformLBP.cols/2, uniformLBP.cols/2, uniformLBP.rows/2));
BL = uniformLBP(cv::Rect(uniformLBP.rows/2, 0, uniformLBP.cols/2, uniformLBP.rows/2));
BR = uniformLBP(cv::Rect(uniformLBP.rows/2, uniformLBP.cols/2, uniformLBP.cols/2, uniformLBP.rows/2));
```

Figura 12: Extracto de la función uniform*LBPhistogram()* que muestra cómo se obtienen las 4 regiones de la imagen LBP uniforme.

3. Mediante la función *calcHist()* de OpenCV, se calculan los histogramas de cada una de las submatrices, considerando 59 bins, uno para cada valor entre 0 y 58, como se puede apreciar en la figura 13.

```
// Create top-left, top-right, bottom-left and bottom-right histograms
cv::Mat histTL, histTR, histBL, histBR;
int histSize = 59;
float range[] = {0, 59};
const float* histRange = {range};

cv::calcHist(&TL, 1, 0, cv::Mat(), histTL, 1, &histSize, &histRange, true, false);
cv::calcHist(&TR, 1, 0, cv::Mat(), histTR, 1, &histSize, &histRange, true, false);
cv::calcHist(&BL, 1, 0, cv::Mat(), histBL, 1, &histSize, &histRange, true, false);
cv::calcHist(&BR, 1, 0, cv::Mat(), histBR, 1, &histSize, &histRange, true, false);
```

Figura 13: Extracto de la función uniform*LBPhistogram()* que muestra cómo se obtienen los histogramas para cada una de las 4 regiones de la imagen LBP uniforme.

4. Se concatenan los histogramas, y se retorna este resultado como un vector fila.

Parte 3: Lectura de datos y separación de conjuntos de entrenamiento y prueba

Una vez implementada la funcionalidad del cálculo de histogramas LBP uniformes, se pudo proceder a leer la base de datos de imágenes y realizar la separación de los conjuntos de entrenamiento y prueba, lo cual también considera un previo reordenamiento aleatorio de los datos.

Para implementar la lectura de datos, se aprovechó de la función *glob()* de openCV, la cual, a partir de un directorio que especifique una terminación de archivo, como por ejemplo el string ".../separated/*.jpg", realiza una búsqueda recurrente de todos los archivos con esa terminación que se encuentren dentro del directorio especificado, guardando la ubicación de estos archivos en un vector de strings. Luego, basta recorrer este vector con un ciclo *for()* para poder realizar operaciones sobre las imágenes de la base de datos, como se puede apreciar en la figura 14.

```
// Get images
path = "../separated/*.jpg";
std::vector<cv::String> fn;
cv::glob(path, fn, true);
for (size_t k = 0; k < fn.size(); ++k) {
```


```
// Get images
path = "../separated/*.jpg";
std::vector<cv::String> fn;
cv::glob(path, fn, true);
for (size_t k = 0; k < fn.size(); ++k) {
```


Figura 13: Extracto de la función *main()* que muestra cómo se leyeron las imágenes de la base de datos.

Luego, dentro del ciclo *for()*, se procedió a procesar cada una de las imágenes, leyéndolas con *imread()* para posteriormente obtener su transformada LBP uniforme por medio de las funciones descritas en los pasos anteriores, lo cual entrega el vector de características de la figura, mientras que para las etiquetas se le asignó a cada imagen una variable que comenzara en 0, y fuese sumando 1 a su valor cada 200 imágenes, efectivamente entregándoles estiquetas distintas a cada clase. Una vez obtenido el vector de características y la etiqueta de una fotografía, estos valores fueron concatenados a una matriz de características y una de etiquetas de la base de datos, respectivamente. La implementación de este procedimiento se puede apreciar en la figura 14.

```
int label = -1;
for (size_t k = 0; k < fn.size(); ++k) {
    cv::Mat src = cv::imread(fn[k]);
    // Process only successful attempts
    if (src.empty()) {
        continue;
    }
    // Get LBP image
    cv::Mat LBP = LBPTransform(src);
    // Get uniform LBP histogram of LBP image
    cv::Mat hist = uniformLBPHistogram(LBP, lookupLBP);
    // Get feature value (0 for Asian, 1 for Black, 2 for Indian, 3 for Others, and 4 for White)
    if (k % 200 == 0) {
        label++;
    }
    cv::Mat currentLabel(1, 1, CV_32SC1, cv::Scalar(label));

    // Append to feature and label matrices
    if (k == 0) {
        features = hist;
        labels = currentLabel;
    } else {
        cv::vconcat(features, hist, features);
        cv::vconcat(labels, currentLabel, labels);
    }
}
```

Figura 14: Implementación del proceso de lectura de la base de datos y obtención de la matriz de características y el vector de etiquetas.

Una vez procesada la base de datos, sólo resta realizar la separación de las características en conjuntos de entrenamiento y prueba. Para esto, se aprovechó de la clase *TrainData()* de OpenCV, la cual se puede construir a partir de una matriz de características y un vector de etiquetas, y permite realizar la separación de los conjuntos, considerando un previo reordenamiento aleatorio de las muestras, mediante el método *setTrainTestSplitRatio().* Posteriormente, es posible obtener las matrices de características y vectores de etiquetas para cada conjunto mediante las funciones *getSamples()* y *getResponses()*, como se puede apreciar en la figura 15.

```
// Create TrainData object. Observations are rows and features are columns
cv::Ptr<cv::ml::TrainData> faceData = cv::ml::TrainData::create(features, cv::ml::ROW_SAMPLE, labels);
// Set 0.3 test ratio and shuffle data
faceData->setTrainTestSplitRatio(0.7, true);

// Convert labels to int. For some reason, they get treated as float if this is not done, which conflicts
// with SVM, since SVM needs labels as int vector
cv::Mat trainFeatures = faceData->getTrainSamples();
cv::Mat trainLabels = faceData->getTrainResponses();
trainLabels.convertTo(trainLabels, CV_32S);

cv::Mat testFeatures = faceData->getTestSamples();
cv::Mat testLabels = faceData->getTestResponses();
testLabels.convertTo(testLabels, CV_32S);
```

Figura 15: Implementación de la separación de la matriz de características y el vector de etiquetas en los conjuntos de entrenamiento y prueba.

Parte 4: Entrenamiento de la SVM

Esta parte del código fue programada utilizando la clase *ml::SVM()* de OpenCV, la cual presenta métodos, tales como *setType()*, *setTermCriteria()*, *setKernel()*, y *setC()*, que permiten cambiar sus parámetros según se estime conveniente. Una vez establecidos los parámetros del clasificador, los procesos de entrenamiento y evaluación resultan sencillos, ya que sólo se necesita llamar a los métodos *train()* y *calcError()*. En la figura 16 se puede apreciar la implementación de los procesos de configuración, entrenamiento y evaluación de la SVM, donde se decidió utilizar un kernel lineal por simplicidad.

```
// Create SVM
auto svm = cv::ml::SVM::create();
// SVM parameters
svm->setType(svm->C_SVC); // C-Support Vector Classifier
svm->setTermCriteria(cv::TermCriteria(cv::TermCriteria::MAX_ITER, 100000, 1e-6)); // max iterations, tolerance
svm->setKernel(svm->LINEAR);

float binaryCValues[] = {1e-6, 1e-5, 1e-4, 0.001, 0.01};
float multiclassCValues[] = {1e-8, 1e-7, 1e-6, 1e-5, 1e-4};
float CValues[5];

if (problemType == 1)
    std::copy(binaryCValues, binaryCValues+5, CValues);
else if (problemType == 2)
    std::copy(multiclassCValues, multiclassCValues+5, CValues);

// Train on different parameter configurations
for (const auto& C : CValues) {
    // Set C value
    svm->setC(C);

    svm->train(trainFeatures, 0, trainLabels);

    // Get train and test errors
    float trainErr = svm->calcError(faceData, false, cv::noArray());
    float testErr = svm->calcError(faceData, true, cv::noArray());
```

Figura 16: Implementación del clasificador SVM. Primero se crea el clasificador, luego se configuran sus parámetros, después se entrena, y finalmente se calcula el porcentaje de error de clasificación con *calcError()*.

Parte 5: Entrenamiento del Random Forest

La implementación de este clasificador se realizó se forma similar a la SVM, mediante la clase *ml::RTrees* de OpenCV. Como sólo se solicitó probar este clasificador con sus parámetros por defecto, en este parte sólo fue necesario utilizar los métodos *train()* y *calcError()*, como se muestra en la figura 17.

```
// Create RF classifier
auto rf = cv::ml::RTrees::create();

rf->train(trainFeatures, 0, trainLabels);

float trainErr = rf->calcError(faceData, false, cv::noArray());
float testErr = rf->calcError(faceData, true, cv::noArray());

std::cout << "Accuracy (%) for RF classifier with default parameters" << std::endl;
std::cout << "Train: " << 100-trainErr << ", Test: " << 100-testErr << std::endl;
```


```
// Create RF classifier
auto rf = cv::ml::RTrees::create();

rf->train(trainFeatures, 0, trainLabels);

float trainErr = rf->calcError(faceData, false, cv::noArray());
float testErr = rf->calcError(faceData, true, cv::noArray());

std::cout << "Accuracy (%) for RF classifier with default parameters" << std::endl;
std::cout << "Train: " << 100-trainErr << ", Test: " << 100-testErr << std::endl;
```


Figura 17: Implementación del clasificador Random Forest. Primero se crea el clasificador, luego se entrena con sus parámetros por defecto, y finalmente se calcula el porcentaje de error de clasificación con *calcError()*.

Resultados

Cálculo de la transformada LBP

En la figura 18 se puede apreciar el resultado de calcular la transformada LBP en sus versiones estándar y uniforme sobre la imagen *0.jpg* de la clase “Indian”.

Figura 18: Transformada LBP estándar (izquierda) y uniforme (derecha) de la imagen *0.jpg* de la clase “Indian”.

Problema de clasificación binario

A continuación, en la figura 19 se muestran los resultados obtenidos para el problema de clasificación que sólo considera las clases "Asian" y "Black". Se utilizó una SVM con kernel lineal, probando distintos valores del parámetro C, mientras que el Random Forest se entrenó con los parámetros por defecto

```
Accuracy (%) for linear SVM classifier with parameters
C = 1e-06
Train: 82.1429, Test: 73.3333

Accuracy (%) for linear SVM classifier with parameters
C = 1e-05
Train: 87.5, Test: 77.5

Accuracy (%) for linear SVM classifier with parameters
C = 0.0001
Train: 93.5714, Test: 73.3333

Accuracy (%) for linear SVM classifier with parameters
C = 0.001
Train: 99.2857, Test: 74.1667

Accuracy (%) for linear SVM classifier with parameters
C = 0.01
Train: 100, Test: 72.5

Accuracy (%) for RF classifier with default parameters
Train: 97.1429, Test: 71.6667
```

Figura 19: Tasas de acierto (en porcentaje) para el problema de clasificación binario, utilizando casificadores SVM y Random Forest.

Problema de clasificación multiclase

En la figura 20 se muestran los resultados para el problema de clasificación considerando las 5 clases (Asian, Black, Indian, Others, y White), utilizando una SVM de kernel lineal con distintos parámetros de C y un Random Forest con sus parámetros por defecto.

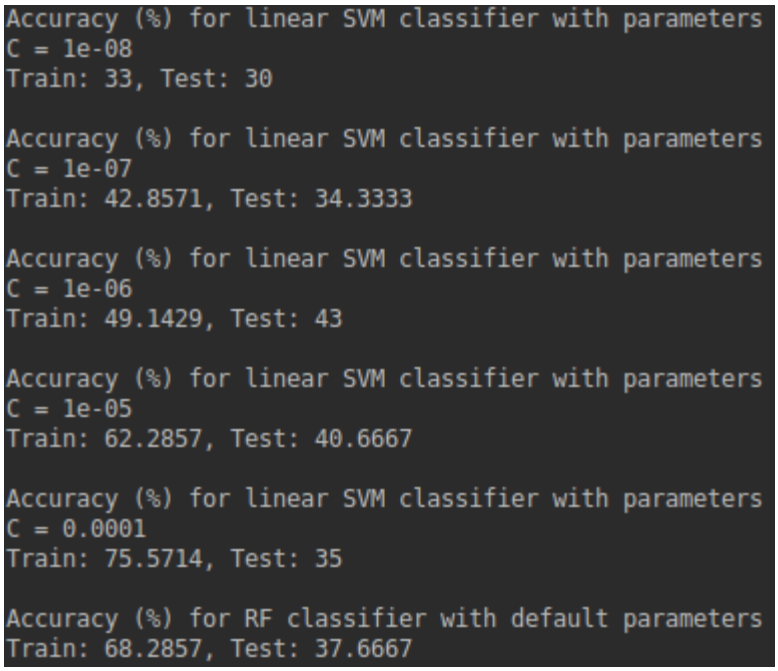

Figura 20: Porcentaje de acierto para el problema de clasificación multiclase, utilizando clasificadores SVM y Random Forest.

Conclusiones

En el presente trabajo se implementó correctamente la obtención de características LBP para una imagen, el cual es un descriptor muy utilizado en aplicaciones de visión computacional, e incluso se pudo programar una versión mejorada de este algoritmo, la cual, al tomar en cuenta sólo los patrones uniformes, permite realizar ahorros en cuanto a costos computacionales. Además, en esta tarea se logró utilizar algoritmos de clasificación comunes en el área de *Machine Learning,* como lo son las SVM y los Random Forest, pudiendo trabajar con implementaciones específicas proporcionadas por OpenCV en C++. Finalmente, los resultados obtenidos permiten demostrar que, si bien la elección de parámetros de los clasificadores resulta importante para obtener buenos resultados, la realización de un buen proceso de selección de características es de carácter esencial.

A partir de las actividades realizadas en esta tarea, se pudo aprender a utilizar módulos más avanzados de OpenCV, pudiendo aprender a resolver no sólo los problemas que se solicitó solucionar directamente mediante el enunciado, sino que también otras necesidades secundarias que surgieron debido a estos requisitos explícitos, como fue el proceso de lectura de la base de datos y extracción de características, cuya eficiencia en su implementación tendrá gran influencia en el costo computacional de ejecutar el programa, especialmente a medida que los tamaños de las bases de datos escalan.

En cuanto a las dificultades encontradas, se destaca la implementación del algoritmo LBP uniforme, especialmente al intentar realizar el recorrido en sentido horario de los vecinos de un pixel central, y en el proceso de búsqueda de una solución para transformar los 256 patrones LBP distintos en 59 patrones uniformes. Otra dificultad que cabe mencionar fue la búsqueda de parámetros que entregaran un buen comportamiento para la SVM, puesto que inicialmente se intentó utilizar una con un kernel RBF, topándose con la sorpresa que el algoritmo no lograba converger, y en cambio asignaba a todas las muestras una misma clase.

Finalmente, en cuanto a los resultados, en primer lugar, se puede mencionar que, si se observa la figura 18, se puede notar que la transformada LBP uniforme permite compactar la información del algoritmo LBP estándar, notándose que ambas imágenes muestran prácticamente las mismas texturas, lo cual concuerda con lo esperado teóricamente, puesto que, como se explicó en el marco teórico, la gran mayoría de las texturas importantes (bordes, esquinas, etc.) deberían presentar patrones uniformes. En segundo lugar, si se observa la figura 19, se puede apreciar que los clasificadores logran altas tasas de acierto, notando que la SVM parece obtener sus mejores resultandos con C=1e-05, mientras que para valores mayores se puede observar una disminución del desempeño debido al overfit. Teniendo esto en mente, es posible concluir que el sistema diseñado puede resolver exitosamente el problema de clasificación binario. Sin embargo, si se observa la figura 20, se puede notar que el desempeño del sistema es considerablemente peor, obteniéndose una tasa de acierto del 43% para la mejor configuración de parámetros, lo cual si bien es mejor que un clasificador nulo (i.e. clasificar “lanzando una moneda”), el cual debería tener una tasa de acierto teórica del 20%, los valores obtenidos no se encuentran dentro del rango de lo aceptable, por lo que se concluye que el sistema de clasificación por raza utilizando descriptores LBP no es adecuado para solucionar este problema, pudiendo inferirse que el descriptor LBP a pesar de ser muy bueno para detectar texturas, esta información no es tan útil para realizar una predicción sobre la raza de la persona, por lo que se debe buscar una mejor alternativa. En este contexto, se propone como idea para mejorar la clasificación utilizar algún descriptor que le dé importancia a la geometría y color de los rostros, los cuales se intuye podrían ser características de alta relevancia al momento de clasificar por raza.