Tarea 4: LBP features on Age Classification with SVM & Random Forest

Estudiante: Luis Escares
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla

I. Introducción

En este informe se diseña y construye un sistema de clasificación de edad utilizando características tipo histogramas LBP y un clasificador estadístico *SVM* o un *Random Forest*. Para esto en la tarea, sé programar una función que retorna la transformada LBP de la imagen, luego genera un "descriptor local" llamado histograma LBP, se concatenan cuatro descriptores locales de cada imagen, lo que sirve como un vector de características. Se ingresa dicho vector al clasificador y resulta la clase del intervalo de edad al que pertenece la persona.

También, en esta tarea se programará el entrenamiento de los clasificadores SVM y Random Forest usando funciones predefinidas de OpenCV. Se programará la división de la base de datos en 70 % de imágenes para Train y 30 % para Test. Los clasificadores se entrenarán con los vectores de características provenientes de las características LBP extraídas usando un ciclo for de la base de datos Train. Finalmente, se programará la evaluación de los clasificadores usando las características LBP de la base de datos Test y se programará el calculo del accuracy como métrica de evaluación.

Las secciones del informe corresponden a un marco teórico donde se expone la transformada LBP, es decir, como calcular las características LBP de una imagen y se describe el uso de patrones uniformes, es decir, la forma en que cada valor LBP de píxel "vota" para armar el histograma LBP. Luego, se describe los niveles de localidad en torno a la generación del histograma usando una grilla n x n y se aterriza este concepto en una grilla de 2 x 2. Luego se expone sobre el SVM biclase de kernel lineal, el uso del kernel trick y los SVM en modalidad one-vs-one y one-vs-all. Finalmente, se describe el clasificador Random Forest.

La sección de desarrollo se ve paso a paso explicando las etapas del programa. En general, se insertan y explican las partes relevantes del código. Como la función que retorna la transformada LBP, explicando porque resultan así las imágenes, y la función que retorna los histogramas LBP. Se explica como se implementa el sistema de ejecución del código sin ninguna intervención del usuario más que para seleccionar el tipo de clasificador; es decir, se explica como las imágenes son leídas desde diferentes rutas usando un ciclo flor. Se explica como se construyen las funciones que entrenan y ejecutan los clasificadores. Finalmente, se explica como se presentan los resultados, explicándolos y haciendo una comparación entre lo teóricamente esperado y lo obtenido.

En la conclusión se hace una mirada más general de los resultados obtenidos y de sus análisis. Se discute hasta qué punto se observan los comportamientos teóricos esperados para los accuracy, identificando las causas de este y se hacen observaciones de como mejorarlo. Se comenta sobre las dificultades superadas para realizar esta experiencia y se finaliza con los aprendizajes y habilidades desarrolladas.

Este informe destaca porque en el se exponen técnicas novedosas de abordar el problema de clasificación de clases en que el objeto es protagonista en la imagen; en este caso, el objeto son caras y las clases son intervalos de edad de personas. Es por esto que te invito lector a descubrir en las siguientes paginas las ingeniosas técnicas de histogramas LBP y clasificadores SVM y Random Forest.

II. Marco Teórico

II-A. Transformada LBP

La transformada LBP, que significa local binary pattern, es un tipo de descriptor visual que se usa para los sistemas de clasificación de caras. Consiste en un algoritmo que se le aplica a una imagen de la que se obtiene otra imagen con las características LBP. Con la imagen LBP se genera un histograma que sera explicado en la siguiente sección. AL aplicar la transformada a una imagen de a x b píxeles se obtiene una imagen más pequeña de tamaño (a-2)x(b-2) píxeles.

El algoritmo de la transformada LBP consiste en recorrer secuencialmente píxel a píxel de la imagen de entrada, es decir, ir recorriendo uno a uno los píxeles de la imagen, no importando si es en orden por filas o por columnas. Entonces, se entiende una imagen de a x b como una matriz de a filas y b columnas en que el primer píxel está en las coordenadas (0, 0). Luego, el algoritmo parte analizando el píxel de coordenadas (1,1) en que considera la vecindad entorno al píxel como los vecinos verticales, horizontales y en las diagonales inmediatas. Para evitar problemas con

los bordes estos no se procesan. En el ejemplo de la figura 1 la vecindad del píxel 4 de en medio son los píxeles de valor 5, 9, 1, 6, 3, 2, 7 y 0.

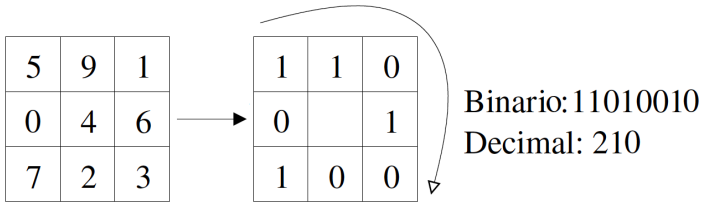


Figura 1. Ejemplo vecindad de un píxel

Para cada píxel analizado se crea un vector de 8 ceros. Entonces, el algoritmo parte analizando el píxel superior izquierdo. Si el 5 es mayor o igual que el píxel del centro de la matriz anota un 1 en el primer índice del vector, si no, se conserva el cero. Luego, avanzando en el sentido de las agujas del reloj, analiza el píxel superior (el 9), si es mayor que el central, anota un 1 en el segundo índice del vector y así sucesivamente. Finalmente, se pasa el vector a decimal, en que la primera cifra corresponde a la cifra más significativa de 128.

Lo anteriormente descrito corresponde a la versión más estándar del LBP. El algoritmo se puede partir desde cualquiera de los píxeles de la vecindad, lo importante es ser consistente en la asignación de unos en la cifra del numero binario correspondiente al orden descrito anteriormente. Además, se puede ejecutar el algoritmo en el sentido contrario de las agujas del reloj, conservando la asignación expuesta. Finalmente, la ultima generalización del algoritmo es que se puede considerar los píxeles alejados en dos píxeles del centro, con lo que el valor a considerar es una extrapolación entre los píxeles que tocan los puntos negros de la figura 2, cuando el punto negro no cae en el centro de un píxel de vecindad. Y cuando cae en el centro de un píxel comparar con el valor de dicho píxel.

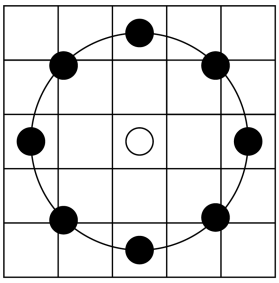


Figura 2. Extrapolación de píxeles en vecindades más grandes

II-B. Uso de patrones uniformes

Cada numero binario obtenido de cada píxel es su patrón binario. Entonces, el uso de patrones uniformes significa que los números binarios se consideran como patrones binarios uniformes. Un patrón binario es **uniforme** cuando contiene **a lo más 2 transiciones entre 0 y 1 o entre 1 y 0** cuando el binario es considerado un conjunto de caracteres circular (string o "palabra" circular). Lo anterior significa que, si en el numero binario, la ultima cifra es un 1 y la primera es un 0 se considera como una transición (y viceversa). Por ejemplo, se clasifica los siguientes binarios en uniforme y no uniforme.

$$\begin{array}{ll} 00000001 \rightarrow uniforme & 11111011 \rightarrow uniforme \\ 00001000 \rightarrow uniforme & 11000101 \rightarrow no-uniforme \\ 00001110 \rightarrow uniforme & 01111010 \rightarrow no-uniforme \\ 10000011 \rightarrow uniforme & 00101010 \rightarrow no-uniforme \end{array}$$

El patrón de un píxel es no-uniforme vota como un 0 en el histograma LBP. En el caso de los píxeles uniformes se indexan correlativamente en el histograma; los histogramas son array que parten en 0, es decir, el primer elemento del array se le llama casilla 0. El binario que representa el 0 (que es el primer numero uniforme obtenible) se le asigna la casilla de índice 1 del histograma. El que representa el 1 (00000001) se le asigna la casilla de índice 2. El binario 2 (00000010) se le asigna la casilla de índice 3; y así, hasta el píxel que tenga valor el binario 5 (00000101) que no es uniforme porque tienen cuatro transiciones, entonces vota en la casilla 0. Así sucesivamente se van llenando las 59 diferentes casillas posibles. A continuación se presenta una tabla con los primeros patrones binarios e índices de las casillas de array que se les asignan.

Natural	Patron Binario	Uniformidad	Casilla Histograma
0	00000000	uniforme	1
1	00000001	uniforme	2
2	00000010	uniforme	3
3	00000011	uniforme	4
4	00000100	uniforme	5
5	00000101	NO-uniforme	0
6	00000110	uniforme	6
7	00000111	uniforme	7
8	00001000	uniforme	8

II-C. Generación de histograma usando grilla nxn

La generación de histograma usando grilla nxn significa que la imagen resultante de la transformada LBP se divide en n·n regiones rectangulares, dividiendo el largo por n y el ancho por n. Luego, en cada región creada se va recorriendo y se va armando el histograma LBP local de la región.

Una vez recorridas las n·n regiones, se obtienen n·n histogramas que son arreglos de 59 valores enteros, los cuales se concatenan en un solo arreglo de 59·n·n elementos. No existe una regla en cuanto al orden en que se deben concatenar los histogramas. Lo importante es ser consistente y concatenar los histogramas de todas las imágenes de la misma forma. En esta tarea lo que se

implementa es que primero se concatenen los histogramas en una fila y luego ese resultado se concatena columna por columna.

Por ejemplo en el caso de la generación de histograma usando grilla 2x2 significa que la imagen se divide en 4 regiones rectangulares, dividiendo el largo y el ancho por la mitad, formando estas 4 regiones rectangulares de un cuarto del área original. Luego, se calcula la transformada LBP de cada región obteniendo 4 histogramas con los que se obtendrá un arreglo de 236 elementos.

Entonces, la concatenación consiste en tomar el histograma de la región 0,0 y concatenarlo con la 0,1. Tomar el histograma de la región 1,0 y concatenarlo con la 1,1. Finalmente, concatenar ambos resultados dejando como primer histograma el de la regio 0,0.

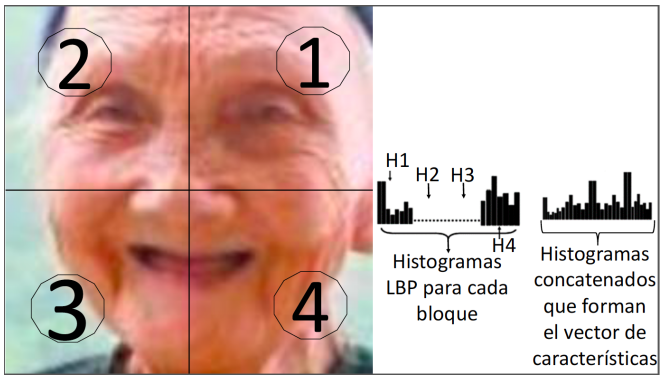


Figura 3. Algoritmo LBP

II-D. SVM de dos clase con kernel lineal

El SVM, Máquinas de vectores de soporte por su sigla en ingles, es un clasificador cuyo objetivo es definir un hiperplano que separe el espacio de características en dos regiones. El aprendizaje equivale a encontrar el hiperplano que maximice la distancia entre el y los puntos llamados soportes, que son los puntos más cercanos al hiperplano. Dicha distancia es llamada margen. Entonces, este hiperplano generado es un lugar geométrico que sirve como regla de decisión para separar dos clases. Que en la figura 4 están representadas por equis verdes y círculos rojos. Si una muestra está a un lado del hiperplano, entonces es de la clase más frecuente en esa región.

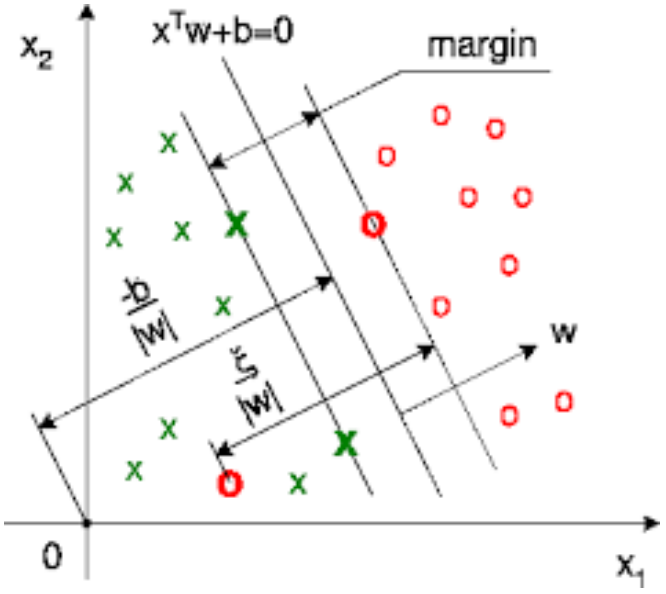


Figura 4. Esquema general SVM

Naturalmente, puede haber puntos que están de un lado del hiperplano que no corresponde a su clase; como en la figura 4 el circulo rojo de la izquierda. Estos son los errores del clasificador y realmente, entrenar un SVM implica dejar algunos puntos mal clasificados. Porque si el hiperplano se dobla mucho para abarcar todos esos puntos, se dice que está en *overfitting*.

El caso particular del SVM con *kernel lineal* es un caso particular del *kernel trick* y corresponde a un aumento de dimensionalidad, en que las distancias en la nueva dimensión se miden con la función de kernel lineal. Lo que permite estimar los parámetros del nuevo hiperplano. En la siguiente ecuación se presenta la función kernel lineal, en que x y x' representan vectores de características. En la siguiente párrafo, se introduce la problemática que soluciona el kernel lineal y se explica su funcionamiento.

$$K(x, x') =< x, x' > \tag{1}$$

II-E. SVM con kernel trick

En el caso ideal de la formulación teórica del SVM siempre existe un hiperplano que separa las clases (o por lo menos comete pocos errores). En la realidad, los datos censados y proyectados en un espacio n-dimensional casi nunca son separables, es decir, casi nunca existe dicho hiperplano. Es por esto que se recurre al uso de funciones kernel como el kernel lineal, lo que es conocido como kernel trick. Esto significa que los puntos del primer espacio n-dimensional se proyectan en otro espacio de dimensión mayor, la dimensión que se agrega al vector de características es la función ϕ evaluada en el vector de características original. También, se le llama función de

mapeo. Entonces, un kernel es una función[1] que cumple con que:

$$K(x, x') =< \phi(x), \phi(x') > \quad (2)$$

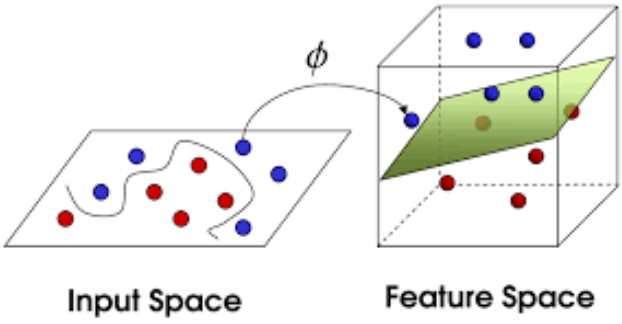


Figura 5. Proyección de datos en kernel trick

Entonces, se aumenta la dimensionalidad con el objetivo de procesar nuevas características que permitan estimar un hiperplano, como se esquematiza en la figura 5. Esta estimación es en el fondo maximizar distancias llamadas margenes, para lo que se necesita medir distancias. Luego, realmente no importa cual es el mapeo sino que como calcular las distancias entre puntos en el nuevo mapa. Es entonces que se define el kernel como la métrica, es decir, la función con que se calculan la distancias y, luego con esta como hiperparametro, se estiman los W y b. Finalmente, en el caso del kernel lineal la función kernel con que se mide distancia es:

$$K(x, x') =< x, x' > \quad (3)$$

II-F. SVM one-vs-all y one-vs-one

El SVM one-vs-all se usa cuando se trabaja con más de dos clases, consiste en buscar un hiperplano que separe una clase con el resto. En el caso general de N clase, se entrenan N clasificadores. En el caso particular del presente informe se clasifican 5 clases. Entonces, un SVM one-vs-all es encontrar tomar una de las 5 clases y encontrar un hiperplano que separe esa clase de las otras 4, luego tomar otra clase distinta y encontrar otro clasificador.Así hasta encontrar los 5 clasificadores.

El SVM one-vs-one corresponde a encontrar un clasificador por cada par de clases. Cuando se tienen N clases, se entrenan N(N-1)/2 clasificadores. Entonces, cuando se tienen más de dos clases, como en el presente informe; no es trivial ni mucho menos directo, afirmar que la separación entre una clase y las otras cuatro es igual o una especie de "promedio" entre los hiperplanos que se encuentran al buscar la separación entre pares de clases. Es por esto, que es interesante abordar cada caso y es lo que se expone en este informe.

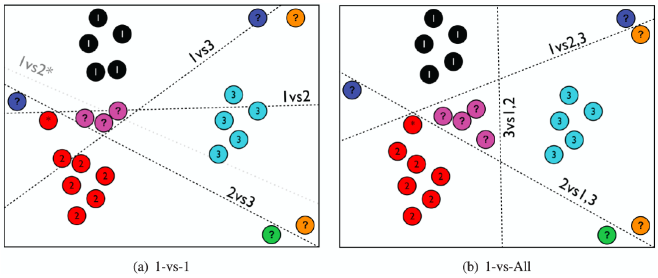


Figura 6. SVM one-vs-one y one-vs-all

II-G. Random Forest

Random forest es un algoritmo de aprendizaje supervisado que consiste en "cultivar" o desarrollar múltiples arboles de decisión en lugar de solo uno. Cada árbol da una clasificación llamada voto y finalmente random forest clasifica la muestra en la clase que obtiene más votos. En principio cada vez que se ejecuta el algoritmo se crean aleatoriamente arboles de decisión con lo que la clase resultante es pseudo aleatoria, entonces, cuantos más arboles haya en el bosque más robusto sera el método, es decir, cada vez que se ejecute tendera a entregar la misma clase y por ende lo sera la precisión del algoritmo. Random forest tiene sus bases en el algoritmo de arboles de decisión el cual se describe en el siguiente párrafo.

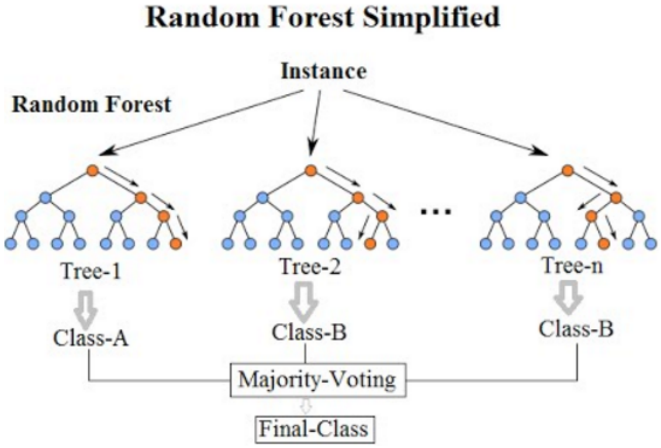


Figura 7. Ejemplo de random forest

Un árbol de decisión es un algoritmo de clasificación de aprendizaje supervisado que consiste en dividir los ejemplos en dos o más subconjuntos basados en las características más significativas de estos. A diferencia de los métodos lineales como SVM logra clasificar bastante bien las agrupaciones que no son linealmente separables

por una única frontera (hiperplano).

La construcción del árbol consiste en que la variable más significativa se usa en la raíz del árbol creando una condición. A partir de esta, se generan dos opciones, una en que dicha condición es verdadera y en la que es falsa. Cada opción llega a una nueva condición basada en la segunda variable más significativa y así sucesivamente hasta separar las regiones de puntos considerando un margen de error.

No existe una regla especifica de como definir las condiciones que separan el espacio de características, ni mucho menos la cantidad de condiciones. Es por esto que en suma cobra sentido realizar el algoritmo Random Forest. En el que, las múltiples repeticiones de arboles de decisión, en que al construirlos las decisiones de cual de las posibles fronteras con sentido elegir se hace de manera aleatoria, se logra concretar la idea de que en promedio la muestra es clasificada satisfactoriamente. Además, logra entregar una noción de pertenencia de la muestra a la clase, es decir, que tan seguro está el clasificador de que la muestra pertenece a la clase predicha.

III. Desarrollo

En la presente sección se agregan y explican las partes relevantes del código, además, se presentan y analizan las imágenes obtenidas en cada paso del programa.

III-A. Lectura de datos y separación train 70 % / test 30 %

Para no repetir código se implementaron dos funciones: *makeTrainTest_db1* y *makeTrainTest_db2*. Lo que hacen es recibir el path de donde esta la carpeta build, el valor de n y retornar (más bien modificar) las matrices train_features, test_features, train_labels y test_labels. Separando la base de datos y dejándola balanceada. Entonces, *makeTrainTest_db1* lee las imágenes de la db1, extrae histogramas LBP y separa dichos conjuntos. Lo mismo para *makeTrainTest_db2* sólo que para la db2.

Las funciones más importantes dentro de ellas son *glob()* que rellena un arreglo de string con las direcciones o *path* de las imágenes. Lo importante es entender su primer argumento: pre_path_db1 = "db_tarea_4/db1/*.jpg". Eso significa que se buscan todos los archivos terminados en .jpg dentro de la carpeta db1. El tercer argumento se llama nested si es True, se busca en las carpetas dentro de la carpeta db1, que es lo que se hace en nuestro caso.

```
vector<String> path_imagenes_db1;
string pre_path_db1 = "db_tarea_4/db1/*.jpg";
string path_db1 = path_build + pre_path_db1;
glob(path_db1,path_imagenes_db1,true);
```

Luego se recorre el arreglo de direcciones path con un for y se leen las imágenes con imread(). Se calcula el histograma LBP usando el siguiente código, lo importante es que aquí se implementa llamar a la funciÓn que calcula los histogramas con un numero n que se puede variar.

```
Mat imagenLBP=transformadaLBP(src);
Mat histograma_actual=nhistogramaLBP(imagenLBP, n);
```

Se observa que los primeros 200 path son de una clase, los otros 200 de otra y así dependiendo de la db. Entonces, se aprovecha esto para asignar el label de la muestra según el Índice con que es leída mediante bloques if:

```
if(i < 200){
  label=0;
}
if(i >= 200 && i < 400){
  label=2;
}
if(i >= 400 && i < 600){
  label=1;
}
cv::Mat label_actual(1, 1, textbf{CV\_32SC} cv::Scalar(label));
```

Lo más importante de esto es que el label se tiene que definir como una mat de formato textbfCV_32SC, para que la librería de SVM y RF de OpenCV reconozca que el problema es multiclase.Luego, se concatenan el mat de features con la matriz grande de features y el mat de label con la matriz grande de labels.

Aquí se llega con una gran matriz de features y otra gran matriz con los labels. Por ejemplo , en el caso de la db1 y n=4. Llegamos con una matriz de 400x945. Los ejemplos son 400 de ahí las 400 filas. Las columnas son 944==59*16, porque las features son la concatenación de 16 histogramas de 59 bins, más una gran matriz de una columna que representa la clases. Las primeras 200 columnas son ejemplos de la clase 1 y las otras 200 son de la clases 5. Lo importante es que hay dos caminos.

El primero es usar una funcion shuffle que mezcla, es decir, intercambia aleatoriamente las posiciones de las filas, por así decirlo, "baraja" las columnas. Luego hacer split que separa las primeras columnas en el 70 % para train y 30 %para test. Esto es importante porque si no se hiciera shuffle, el conjunto de train quedaría desbalanceado. Porque el 70 % de 400 es 280, entonces, split tomaría para el conjunto de train las 200 imágenes de clase 1 más 80 imágenes de clase 5; y el conjunto de test quedaría con 120 imágenes todas de clase 5. Lo que es un conjunto de entrenamiento totalmente desbalanceado. Al menos en todas las documentaciones leídas de openCV 2 Y 3 en como implementen la separación del data set en ninguna se menciona que la funcion de split (por ejemplo setTrainTestSplitRatio o set_train_test_split) incluya el shuffle por eso me pareció importante mencionarlo.

La consecuencia de tener los conjuntos desvanecidos es que se obtendrán resultados demasiado buenos. Sin embargo, si se revisan una a una las predicciones nos daremos cuenta de que realmente lo que se pierde es la

capacidad de generalización del clasificador.

Entonces, aunque se aplique shuffle, igual queda el problema de que en pocos casos los conjuntos de train y test queden 50/50 balanceados, porque shuffle baraja las muestras aleatoriamente. Entonces, como se conocen entre que índices están las muestras de las clases y realmente son pocas muestras, lo que no añade tiempos de ejecución altos. Lo que se hace es dividir manualmente el dataset. Quedando el set de train con 140 muestras de clase 1 y 140 muestras de clase 5. El set de test queda con 60 muestras de clase 1 y 60 muestras de clase 5.

El código que implementa lo anterior se muestra a continuación. Consiste recorrer la matrices features y labels con un cuatro ciclo for de 160, 160, 60 y 60 iteraciones. Cada ciclo for arma una de las matrices:

```
for(int i=0;i<140;i++){//para train clase 0
  for (int f=0;f<features.cols;f++){
        train_features_clase0.at<float>(i,f)=
        features.at<float>(i,f);
  }
  train_labels_clase0.at<int>(i,0) = labels.at<int>(i,0);
}

for(int i=140;i<200;i++){//para test clase 0
  for (int f=0;f<features.cols;f++){
        test_features_clase0.at<float>(i-140,f)=
        features.at<float>(i,f);
  }
  test_labels_clase0.at<int>(i-140,0) = labels.at<int>(i,0);
}
//análogo para clase 1
```

Se comenta que es ineficiente hacer un ciclo for de 400 iteraciones con 4 bloques if adentro porque para cada muestra solo un if siempre va a dar positivo y se evalúan 3 bloques if en vano.

Por ultimo se concatenan las matrices de features y labels generadas para cada clase usando vconcat. El primer parámetro es la primera matriz a concatenar, el segundo la segunda, y el tercero la matriz más grande ya concatenada. Por ejemplo:

```
vconcat(train_features_clase0, train_features_clase1,
        train_features);
vconcat(train_labels_clase0, train_labels_clase1,
        train_labels);

vconcat(test_features_clase0, test_features_clase1,
        test_features);
vconcat(test_labels_clase0, test_labels_clase1,
        test_labels);
```

En fin se obtiene una base de datos calibrada de las matrices features y labels: del 0 al 139 son muestras para train clase 0, del 139 al 199 son para test clase 0; del 200 al 339 son para train clase 1 y del 340 al 399 son para test clase 1.

Es vital que al definir las funciones *makeTrainTest_db1* o *makeTrainTest_db2* los argumentos hay que ponerlos con −>&<−, por ejemplo:

```
void makeTrainTest_db2(int n, Mat &train_features,
Mat &test_features, Mat &train_labels, Mat &test_labels,
string path_build)
```

Porque si no, las matrices train_features, test_labels, etc. Se definen en la localidad de la funcion, luego no se exportan a la funcion main, cuando se les llama. Eso puede originar el problema de que se evalúen los clasificadores con muestras de una única clase. Entonces, poniendo el **&** eso se soluciona.

1) Implementación transformada LBP con patrones uniformes: La implementación de la transformada LBP considerando los patrones uniformes previamente descritos en el marco teórico se realiza con la función: *trasformadaLBP*. Dicha función recibe una imagen en un objeto Mat y retorna otro objeto Mat que representa la transformada LBP. Es decir, cada píxel de dicha imagen es un numero entre 0 y 58.

A la función ingresa la imagen y esta se pasa a escala de grises usando la función predefinida *cvtColor*. Luego se crea una Mat llamada LBP que representa la transformada LBP y es con la que principalmente se trabaja. Lo importante es que se crea con 2 fila y con 2 columnas menos que la imagen de entrada, porque en esta implementación el *padding* consiste en no considerar los bordes porque la imagen es suficientemente grande y los bordes de la foto no aportan mucha información a la edad, no así la cara.

Entonces, considerando que se conocen todos los enteros en patrón binario uniforme del 0 al 256. Se crea un array con todos los números que se le debe asignar a cada patrón binario para rellenar la imagen resultante de la transformada LBP y luego para armar el histograma. El array se llama LUT, por el patrón de diseño Look Up Table.

Las imágenes se va recorriendo con un doble ciclo for sin considerar los bordes. Para cada píxel las comparaciones para determinar el patrón binario de la vecindad se implementan con 8 bloques if. La transformación binario a decimal con la función pow() y se guarda en la variable auxiliar píxel. Luego, se consulta: LUT[pixel].

Entender el arreglo LUT es la parte más importante de esta explicación. Es un arreglo de 255 elementos (porque existen 255 combinaciones posibles de números binarios de 8 cifras). Entonces, en el índice i tiene el numero que se le debe escribir al píxel resultante de la LBP, que representa la casilla del histograma LBP donde votara el píxel. Esta escritura se implementa con:

```
int LUT[]={1,2,3,4,5,0,6,7,8,0,0,0,9,0,10,11,12,0,0,0,0,0,0,
0,13,0,0,0,14,0,15,16,17,0,0,0,0,0,0,0,0,0,0,0,0,0,0,18,0,
0,0,0,0,0,0,19,0,0,0,20,0,21,22,23,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,24,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,25,0,0,0,0,0,0,0,26,0,0,0,27,0,28,29,30,31,0,32,0,0,
0,33,0,0,0,0,0,0,34,0,0,0,0,0,0,0,0,0,0,0,0,0,0,35,0,0,0,
```

```
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,36,37,
38,0,39,0,0,0,40,0,0,0,0,0,0,0,41,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,42,43,44,0,45,0,0,0,46,0,0,0,0,0,0,0,47,48,49,0,50,0,0,0,51,
52,53,0,54,55,56,57,58};
//Se consulta como:
LBP.at<float>(f-1,c-1) = LUT[pixel];
```

Por ejemplo: el patrón binario de la vecindad del píxel es 0, es decir, 00000000, que es un patrón uniforme. El script consulta el índice 0 del array LUT. Entonces LUT[0] es igual 1, entonces, en el píxel de la imagen final (llamada LBP) se anota un 1. Y al calcular el histograma ese píxel va a votar en la casilla 1.

Para un píxel con un patrón NO-uniforme como el 5, este píxel tiene que tener valor 0 en la imagen resultante. Entonces, el script consulta LUT[5] que es igual 0. Entonces, en el píxel final se escribe un 0. Al calcular el histograma ese píxel va a votar en la casilla 0.

El ultimo caso de borde es el 255, cuya vecindad tienen patrón binario 11111111, es un patrón uniforme, pues tiene 0 transiciones. Luego, al consultar LUT[255] resulta 58. Entonces, en el píxel final se escribe un 58. Al calcular el histograma ese píxel va a votar en la casilla 58 que es la ultima del histograma.

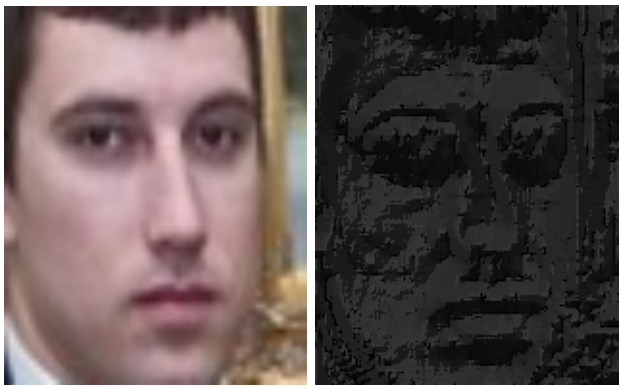
Por último el mat LBP se trasforma a tipo de dato CV_8UC1 usando la función: convertTo. Esto permite que se pueda imprimir en pantalla las imágenes LBP. A continuación se muestran algunos ejemplos:

(a) Clase 1-4 años (b) Su Transf. LBP

Figura 8. Ejemplo imagen Clase 1-4 años y su Transf. LBP

(a) Clase 5-27 años (b) Su Transf. LBP

Figura 9. Ejemplo imagen Clase 5-27 años y su Transf. LBP



(a) Clase 28 o + años (b) Su Transf. LBP

Figura 10. Ejemplo imagen Clase 28 o + años y su Transf. LBP

III-B. Implementación y descripción histograma usando grilla nxn

La implementación del histograma se realiza con la función *nhistogramaLBP()*. Esta función recibe una Mat con la imagen LBP proveniente de la función *transformadaLBP()*. Además, recibe un int n que sirve para indicar la cantidad de ventanas en que se dividirá la imagen LBP, es decir, se dividirá en nxn ventanas. La *nhistogramaLBP()* retorna una Mat de 1 fila y 59*n*n columnas en formato CV_8UC1.

Entonces, las partes importantes es que primero la función calcula la cantidad de columnas y filas que tendrá cada ventana con el siguiente código(todos las ventanas quedan iguales):

```
int filas_ventana = imagenLBP.rows/n;
int columnas_ventana = imagenLBP.cols/n;
```

La función es en lo general un cuádruple ciclo for anidado. Los primeros dos ciclos for anidados son para

posicionarnos en las ventanas. Por lo que las variables f y c van desde 0 hasta n-1. Una vez posicionado en la ventana, por ejemplo: para la primera f=0 y c=0. Con los otros dos ciclos for anidados se hace un slice de esa región de la imagen.

```
for(int fv = 0; fv < filas\_ventana; fv++){
  for(int cv = 0; cv < columnas\_ventana; cv++){
        ventana.at<uchar>(fv,cv) = imagenLBP.at<uchar>
        ((f*filas\_ventana) + fv,(c*columnas\_ventana) + cv);
  }
}
```

En el código anterior lo importante es observar el caso borde con n's complicados , como cuando las divisiones imagenLBP.rows/n no den con resto 0. En esos se da el problema de que para las regiones (0,n) ; (n,0) y (n,n). Se hubiesen generado problemas si definíamos de antemano los largos de la región porque esas hubiesen quedado más cortas. En esta implementación simplemente se da el punto de partida y cuanto avanzar. Entonces, las ultimas regiones quedan 2 o 3 píxeles más cortas pero no importa, porque la funcion que calcula el histograma recibe cualquier tamaño de región.

La siguiente parte importante que es calcular el histograma. Esto se implementa con la función predefinida *calcHist()*. Esta función recibe un Mat junto con varios parámetros y retorna un mat que es el histograma basado en los valores de dicho mat, lo útil en nuestro caso es que no le importan las dimensiones del mat. Entonces, la los parámetros de la funcion son:

```
calcHist( &mat\_input, 1, 0, Mat(), Mat\_histograma, 1,
&histSize, &histRange, uniform, accumulate );
```

Primero entra el mat de la imagen al que se le quiere sacar el histograma, el 2º parámetro es el numero de arreglos que sirven de fuente, en nuestro caso es 1 porque solo usamos una mat. El 3º es la dimensión a ser medida, como es una sola mat en nuestro caso es 0 pues es su primera dimensión (y solo tiene una). El 4º es una mascara que sirve para indicar que píxeles ignorar al hacer el histograma. El 5º es la mat que terminara con el histogram de salida. el 6º es la dimensionalidad del histograma 1 en nuestro caso.

El 7º es el histSize, es decir, el numero de bines de nuestro histograma, es es 59, porque son 58 posibles patrones uniformes que votan en casillas distintas más la casila de todos los patrones no-uniformes. histRange es el rango de valores que tiene cada bin, en este caso se pone de 0 a 59, luego a cada bien le toca solo 1 valor. Los últimos dos parámetros con la uniform, es decir, 1 en nuestro caso porque todos los histogramas a calcular serán iguales; y accumulate, en nuestro caso 0 porque el histograma que creamos se pone en 0 cuando iniciamos la función.

```
cv::Mat histograma\_ventana;
int histSize = 59;
float range[] = {0, 59};
const float* histRange = {range};
cv::calcHist(&ventana, 1, 0, cv::Mat(),
histogram\a_ventana, 1, &histSize, &histRange, true, false);
```

Entonces, de *calcHist()* sale una mat de (59*n*n) x 1, llamada histograma_ventana. Se traspone esta mat con *.t()* y se concatena a histograma_concatenated con hconcat. Es destacable que la funcion hconcat tiene problemas con el histograma de la primera ventana porque no puede concatenar vacio con algo. Entonces, esto se soluciona con un bloque if que iguala el primer histograma calculado con el histograma final a retornar. Luego, no hay problemas con los siguientes histograma que se concatenan. Finalmente se retorna el histograma resultante de todas las concatenaciones.

```
if(f==0 && c==0){
  histograma\_concatenated = histograma_ventana.t();
}       else        {
  cv::hconcat(histograma\_concatenated,
  histograma\_ventana.t(),histograma\_concatenated);
}
```

III-C. Clasificadores

Como no tiene sentido poner 8 secciones una para cada clasificador, porque basicamente seria muy largo y tedioso de leer, lo que haremos sera lo siguente: En las siguintes dos secciones se explca como se programa el SVM y el Random Forest. EN la seccion Funcionamiento general del script se explica la arquitectura del scrip para que el ayudante no tenga que modificar el codigo y en Overview de resultados se presenta una tabla con todos los resultados de todos los clasificadores y se hacen conclusiones.

III-D. Entrenamiento y evaluación SVM

Entonces, a esta parte llegamos luego de haber llamado a la funcion *makeTrainTest_db1* o *makeTrainTest_db2*. Por lo que constamos en la base de datos separada y balanceada. Contamos con los conjuntos: train_features, test_features, train_labels y test_labels.

Primero se define el objeto SVM con SVM::create(). Luego con las lineas set se definen los hiperparametros en este caso que sea de kernel lineal, de tipo c_SVC, los errores para los soportes, etc. Luego se entrena con el método train, lo importante es que el parámetro ROW_SAMPLE significa que cada fila es un ejemplo.

```
Ptr<SVM> svm = SVM::create();
svm->setType(SVM::C\_SVC);
svm->setKernel(SVM::LINEAR);
svm->setTermCriteria(TermCriteria(
        TermCriteria::MAX\_ITER, 100, 1e-6));

svm->train(train\_features, ROW\_SAMPLE, train\_labels);
```

Entonces, una vez entrenado, se evalué usando svm->predict. Primero se crea una matriz en formato

CV_32SC1 para guardar las predicciones y la variable sum_vp_y_vn para ir calculando la suma de la diagonal. Con un for sobre la cantidad de muestras de test se va prediciendo una a una los labels del conjunto de test con:int prediction = svm->predict(test_features.row(i)). Lo cual retorna un int con la clase predicha. Si la clase predicha coincide con la clase real se suma 1 a la variable sum_vp_y_vn.
Finalmente se calcula el accuracy con: (sum_vp_y_vn/120.0)*100 y se presentan los resultados con un print.

```
cv::Mat predicted_labels(120, 1, CV_32SC1, cv::Scalar(1));
int sum_vp_y_vn=0;
for(int i=0; i<test_features.rows; i++){
  int prediction = svm->predict(test_features.row(i));
  predicted_labels.at<int>(i,0)=prediction;
  if(test_labels.at<int>(i,0)==predicted_labels.
    at<int>(i,0)){
        sum_vp_y_vn = sum_vp_y_vn+1;
  }
}
cout<<"SUMA DE LA DIAGONAL = "
    << sum_vp_y_vn<<endl;
cout<<"accuracy SVM 2 clases (n=4)  : "
    << (sum_vp_y_vn/120.0)*100<<endl;
```

III-E. *Entrenamiento y evaluación Random Forest*

Debido a que en OpenCV los clasificadores son intancia de una clase que implementa clasificadores estadisticos, la sintaxis es muy parecida. Primero de instancia el objeto rf que representara al clasificador. Luego, se puede setear sus hiperparametros usando: rf->set(hiperparametro a setear). El entrenamiento se hace usando f->train(train_features,ROW_SAMPLE, train_labels). Al igual que con SVM, el argumente ROW_SAMPLE significa que las filas de la matriz train_features son muestras.

```
//create RandomForest classifier
auto rf = cv::ml::RTrees::create();
//train RandomForest
rf->train(train_features,ROW_SAMPLE, train_labels);
```

La clasificación es exactamente igual usando la notación de flecha –>. Se crea la amtriz que guardara las predicciones. Luego, Es importante hacer las predicciones una a una usando un ciclo for. Si la clase predicha es igual a la clase real se suma 1 a la variables sum_vp_y_vn con la que se calcula el accuracy. EL cual finalemente se presenta con el ultimo print.

```
//Predict with RandomForest
cv::Mat predicted_labels(180, 1, CV_32SC1, cv::Scalar(1));

int sum_vp_y_vn=0;
for(int i=0; i<test_features.rows; i++)
{
        int prediction = rf->predict(test_features.row(i));
        predicted_labels.at<int>(i,0)=prediction;
        //Evaluation of predition
        if(test_labels.at<int>(i,0)==
            predicted_labels.at<int>(i,0))
        {
                sum_vp_y_vn = sum_vp_y_vn+1;
        }
}
//Show accuracy
cout<<"SUMA DE LA DIAGONAL = " <<
    sum_vp_y_vn<<endl;
cout<<"accuracy RandomForest 3 clases (n=2)  : " <<
(sum_vp_y_vn/180.0)*100<<endl;
```

III-F. *Funcionamiento general del script*

Para hacer funcionar el scrip debes:
[0]Crear una carpeta llamada build, copiar la carpeta db_tarea_4 dentro de la carpeta build. [1]Abrir una terminal dentro de la carpeta build. [2]Escribir:"cmake .." [3]Escribir:"make", para compilar el codigo [4]Escribir: "./tarea4", para ejecutar el codigo. [5]Buscar la direccion de la carpeta build. Por ejemplo: /home/luis/Escritorio/tarea4-2019/build/ [6] Ingresar ese path a la consola [7] Si quiere que se muestren todos los resultados, entrenando todos los clasificadores presione 0 [8] Si quiere el resultado de un clasificador en particular presione el numero correspondiente.
[9] Los resultados comenzaran a mostrarse en la terminal.
El script lo que hace es primero preguntarte el path de la carpeta build, como se explico anteriormente. Luego te muestra las opciones de clasificadores. Luego, si apretas 0 entra en un ciclo for que llama 8 veces a la funcion caller. la funcion caller ejecuta la separación de la db, los histogramas con el n deseado y preintea el resultado. Si aprientas otro numero se llama especificamente a la funcion caller para ese clasificador. lo hice asi para ahorrar espacio

III-G. *Overview de resultados*

La siguiente es una tabla con todos los accuracy de las posibles combinaciones de clasificadores:

Clasificador	n=8	n=4	n=2
SVM 2 clases	90	83	66
RF 2 clases	90.8	75	72
SVM 3 clases	67.2	60	44
RF 3 clases	62.2	55	55

La primera observación se obtuvieron mejores resultados con n = 4 que con n=2. Se experimenta con n=8 y se obtiene mucho mejores resultados para todos los clasificadores que con n=4. Así que se observa que aumentando n se mejoran los resultados. Esto se explica porque en el fondo se extrae más información. El inconveniente de que aumenta el tiempo de cómputo.
.
También se observa que los clasificadores triclase tienen peor desempeño que los biclase. Esto se debe a que en general el problema biclase es más simple que el multiclase y que para este problema en particular de clasificación de edad se cuenta con muy pocas imágenes y porque sé los clasificadores SVM y RF no fueron pensado desde el principio en su arquitectura para ser usados como imágenes.
.
Algo destacable es que en teoría LBP sirve para extraer información con respecto a texturas de las imágenes y

realmente se obtienen resultados bastante buenos considerando lo expuesto en el párrafo anterior. Además los tiempos de entrenamiento son bajisimos en comparación a cualquier CNN. Así que se confirma la teoría en el sentido de que efectivamente LBP sirve para extraer información útil de texturas de imágenes.
.

Teóricamente, ni SVM ni RF son uno mejor que el otro para problemas relacionados con imágenes. Asi que no se puede afirmar que se confirme la teórica en base a compararlos. Se observa que para n=8 y n=4 SVM se comporta mejor en biclase y triclase. Por el contrario con n=2 RF tiene accuracy en ambas categorías. En suma, no se puede afirmar que alguno de los 2 sea mejor para esta tarea.
.

IV. Conclusión

En resumen, en esta tarea se realiza la extracción características LBP de imágenes considerando patrones uniformes. Luego, se implementa la localidad a nivel de región dividiendo la imagen en nxn regiones. Se implementa un código que para todo n logra dividir la imagen en nxn regiones y calcula el histograma LBP de cada región. Para luego implementar la localidad a nivel global concatenando dichos histogramas.

Posteriormente, se resuelve el problema de clasificar intervalos de edad. Para lo que se programa la extracción de características de imágenes de las bases de datos db1(400 imágenes biclase) y db2(600 imágenes triclase). Cada clase es un intervalo de edad. Se construyen los conjuntos de entrenamiento y test en proporciones 70/30 y procurando que queden igualmente balanceados para cada clase. Luego, se entrenan y evalúan clasificadores SVM y Random Forest con considerando regiones de n igual a 4 y a 2. En suma se implementa todo el proceso en una 'pseudo-interfaz de usuario' en la consola, que permite elegir entre entrenar y evaluar todos los clasificadores o entrenar y evaluar uno en específico.

Los resultados obtenidos para la LBP son los esperados en función de lo que indica la teoría, al igual que los histogramas, los cuales son obtenidos en un tiempo razonable. Puede haber variaciones con otras implementaciones dependiendo de con cual píxel de la vecindad se considera el dígito de la cifra más significativa del número binario resultante. Lo importante es que se fue consistente en aplicar el mismo orden a todas las imágenes.

Para los resultado obtenidos en clasificadores, lo esperado es accuracy bajos por sobre el 50 % y no sobre el 90 %. Debido a la baja cantidad de imágenes, a que LBP no es de los mejores métodos para abordar un problema tan complejo como este y que la arquitectura de SVM y RandomForest tampoco esta pensada desde su inicio para clasificar imágenes. El peor clasificador fue el SVM triclase con n=2 con 44 % de accuracy y el mejor fue el SVM biclase con n=4 con 83 %, lo que deja a todos los clasificadores entrenados y evaluados dentro de lo esperado teóricamente. Lo importante es que se procura minuciosamente de balancear los conjuntos de train/test y revisar predicción por predicción, que no se estuviera usando como test muestras de una sola clase. Así se logra identificar cuando un clasificador resulta sospechosamente bien y se logra corregir.

Los aprendizajes obtenidos al realizar esta tarea son profundizar la programación en C++ y el trabajo con la distribución de Linux Ubuntu. Lo más importante es que se aprendió a trabajar con objetos que implementan los clasificadores SVM y RandomForest en OpenCV y se aprendió como es el patrón de uso para los objetos clasificadores. Es decir, se ve a cada tipo de clasificador como una instancia de la clase que implementa varios tipos de clasificadores. Lo que hace que compartan métodos como train y predict y se llamen usando la notación de flecha ->. También, se aumentan los conocimientos en temas operativos muy útiles como concatenar strings, paths y pensar en la arquitectura del script al definir funciones. Finalmente, se aprende a escribir scripts de rápida ejecución en la máquina por ser de C++ y que procesan todas las imágenes de una carpeta fuente sin necesidad de compilar/correr el script varias veces.

Las dificultades que se encontraron durante el desarrollo de la experiencia fue que se debía aprender C++ que es un lenguaje más alejado del usuario, es decir, los códigos son menos intuitivos. La principal dificultad fue aprender a usar los objetos de la librería de machine learning de OpenCV, aprender a usar la notación de flecha -> y lograr encontrar la implementación que se ajustara a la versión de OpenCV (v2.3) con que se estaba escribiendo el código. Otra dificultad fue el debug porque se compila y debugea desde la consola con lo que fue difícil identificar los errores. Finalmente, las dificultades se solucionaron preguntando a los compañeros de clase, investigando en la ficha de cada clase en la documentación de OpenCV y leyendo dudas parecidas en Stackoverflow.

El modo en que podrían mejorarse los resultados es haciendo *data augmentation* sobre la base de datos o consiguiendo más muestras para las clases. También, se pueden mejorar haciendo *Fine-Tuning* sobre los hiperparametros de los clasificadores. Por ejemplo, elegir otro tipo de kernel que se adapte mejor que el lineal. En general, se obtuvieron mejores resultados con n = 4 que con n=2. Se experimenta con n=8 y se obtiene mucho mejores resultados para todos los clasificadores que con n=4. Así que se concluye que aumentando n se mejoran los resultados (porque en el fondo se extrae más información) con el inconveniente de que aumenta el tiempo de cómputo.
.