ECE661 - Assignment 8

Caluadewa Dharshaka Tharindu Mathew (mathewc@purdue.edu)

November 5, 2016

1 LBP Feature Extraction

The LBP algorithm was implemented using the following steps:

1. For each pixel the equally spaced difference of the points for P points on a circle of radius R and the pixel is given as:

$$\Delta u = R \cdot cos(\frac{2\pi p}{P}), \Delta v = R \cdot sin(\frac{2\pi p}{P})$$

   where u is the downwards direction, v is the left direction of the image and $p = 0, 1..., P-1$. We choose $R = 1, P = 8$ for the purpose of this implementation.

2. The gray value at each location x on the circle needs to be interpolated, and this is done by using the formula:

$$\text{value(x)} = (1-\Delta u)(1-\Delta v)A + (1-\Delta u)\Delta v B + \Delta u(1-\Delta v)C + \Delta u \Delta v D$$

   where A, B, C, and D are the top left, top right, bottom left, bottom right of the rectangle around the point x. Here the inter-pixel sampling interval is assumed to be a unit distance along the vertical and horizontal axes.

3. Now, each interpolated gray value of the P points will be thresholded based on the gray value of the center pixel under consideration. If the interpolated pixel's gray value is less than the gray value of the center pixel, the bit value will be 0. It will be 1 otherwise. This will create a 8 bit pattern for P = 8.

4. Then we rotate this bit pattern until we find the minimum value of the bits. Let this be *minbits*. This is how LBP becomes rotationally invariant.

5. Then we encode *minbits* based on the runs of 1s (this was determined to be more important by the LBP creators) in the bit pattern as follows:

   - If there are two runs or less, then the number of the longest run of 1s is r, then the encoding is r. This means a run of zeros is 0, a full run of 1s is P
   - If the number of runs is larger than 2 is P+1

6. Using the above encoding scheme, we create a LBP histogram of P+2 bins for each image.

2 NN Classification Method

2.1 Training

The training part of the NN classification was done simply by calculating the LBP vector of each image, and storing it in a data struct along with the class that it belongs to.

2.2 Classification

The classification of the testing images is carried out with the following algorithm:

1. Calculate the LBP vector of the testing image, I_{test}
2. For each training image I^i_{train}, calculate the Euclidean distance of LBP vector between I_{test} and I^i_{train}, and store it in a data struct along with I^i_{train} class.
3. Find the entries with k minimum distances (k=5 for this implementation)
4. Within the k minimum entries, the class that appears the most will be the classified class

3 Observations

1. The algorithm performs well overall, giving an accuracy of 70%, which is well above the baseline of randomly picking a class, which is 25% for this case.
2. It performs well for trees and mountains, where the naturally occurring patterns of textures are unique and dominate most of the scene.
3. For the case of cars, the algorithm misclassifies some of the images due to the cluttered backgrounds with buildings. We can see that the misclassifications pick the buildings due to this case.
4. For buildings, the misclassifications are picking the mountain class, most probably due to the presence of the sky in the building images. The single misclassification of the mountain also picks the building class. This means whenever there's a shared texture among two classes the classifier can easily get confused and a more expressive measurement and/or classification method is needed.
5. The above observation of the sky being present in the misclassification can also be thought of as the reason for the single misclassification of the tree class which picks the building instead.

4 Results

4.1 LBP Histogram Feature Vectors

imagesDatabaseHW8/training/**building**/01.jpg -

$[0:2653 \quad 1:5192 \quad 2:1753 \quad 3:4010 \quad 4:6496 \quad 5:6953 \quad 6:2199 \quad 7:5300 \quad 8:7130 \quad 9:7832]$

imagesDatabaseHW8/training/**car**/01.jpg

$[0:1781 \quad 1:3356 \quad 2:1383 \quad 3:3095 \quad 4:6215 \quad 5:5980 \quad 6:2332 \quad 7:3854 \quad 8:17089 \quad 9:4328]$

imagesDatabaseHW8/training/**mountain**/01.jpg

$[0:2924 \quad 1:3314 \quad 2:2176 \quad 3:3111 \quad 4:3679 \quad 5:5697 \quad 6:2797 \quad 7:3744 \quad 8:16170 \quad 9:5732]$

imagesDatabaseHW8/training/**tree**/01.jpg

$[0:4857 \quad 1:4902 \quad 2:3414 \quad 3:4330 \quad 4:5051 \quad 5:3845 \quad 6:3143 \quad 7:4883 \quad 8:5846 \quad 9:9142]$

4.2 Confusion Matrix

	tree	building	car	mountain
tree	4	1	0	0
building	0	3	0	2
car	0	2	3	0
mountain	0	1	0	4

4.3 Overall Accuracy

The overall accuracy of the algorithm is : 70%

5 Source Code

```
#pragma once
#include "opencv2/core/core.hpp"
#include <functional>
#include <unordered_map>

typedef std::map<int, double> LBPHist;

class Textures
{
public:
        void normalize_histogram(LBPHist& lbp_hist);
        LBPHist create_lbp_histogram(int rows, int cols, std::function<unsigned char
              (int, int)> get_value);
        double calc_euclidean(const LBPHist& one, const LBPHist& two);
        Textures();
        ~Textures();
};
```

```
#include "Textures.h"
#include <bitset>
#include <unordered_map>
#include <iostream>
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <iomanip>


#define PI CV_PI

template <std::size_t N>
inline
void
rotate(std::bitset<N>& b, unsigned m)
{
        b = b << m | b >> (N - m);
}

template <std::size_t N>
inline
std::vector<std::pair<unsigned char, unsigned>> calculate_runs(const std::bitset<N>&
      b) {

        std::vector<std::pair<unsigned char, unsigned>> runs;

        unsigned char last_v = b.at(N - 1);
        int count = 1;

        if (N - 2 > 0) {
                for (int i = N - 2; i >= 0; --i) {
                        unsigned char curr_v = b.at(i);
                        if (curr_v == last_v) {
                                count++;
                        }
                        else {
                                runs.push_back(std::make_pair(last_v, count));
                                last_v = curr_v;
                                count = 1;
                        }
                }
        }
        runs.push_back(std::make_pair(last_v, count));

        return runs;
}

void Textures::normalize_histogram(LBPHist& lbp_hist) {
        double sum = 0.0;
        for (auto& pair : lbp_hist) {
                sum += pair.second;
        }
        for (auto& pair : lbp_hist) {
                pair.second /= sum;
        }
}

LBPHist Textures::create_lbp_histogram(int rows, int cols, std::function<unsigned
     char(int, int)> get_value) {

        const int P = 8;
        LBPHist lbp_histogram;

        auto increment_histogram = [&](int p)
        {
                auto itr = lbp_histogram.find(p);
                if (itr != lbp_histogram.end()) {
                        itr->second += 1;
                } else {
                        lbp_histogram[p] = 1.0;
                }
        };

        for (int row = 1; row < rows - 1; ++row) {
                for (int col = 1; col < cols - 1; ++col) {
```

```
                        unsigned char current_pxl_color = get_value(row, col);
                        // for each pixel compute P = 8, R = 1
                        const float R = 1;
                        std::bitset<P> circle_bits;

                        for (int p = 0; p < P; ++p) {
                                float del_u = R * std::cos(2 * PI * p / (float)P);
                                float del_v = R * std::sin(2 * PI * p / (float)P);

                                // retaining integer value, i.e. floor values to
                                    calulate A of A B C D corner points
                                float v = col + del_v;
                                float u = row + del_u;

                                cv::Point2i A(v, u);

                                float delta_u = u - A.y;
                                float delta_v = v - A.x;

                                float circle_point_interpolated_color;

                                if (delta_u < 1e-3 && delta_v < 1e-3) {
                                        circle_point_interpolated_color = get_value(
                                            A.y, A.x);
                                } else if (delta_v < 1e-3) {
                                        circle_point_interpolated_color = (1 -
                                            delta_u) * get_value(A.y, A.x) +
                                            delta_u * get_value(A.y + 1, A.x);
                                } else if (delta_u < 1e-3) {
                                        circle_point_interpolated_color = (1 -
                                            delta_v) * get_value(A.y, A.x) +
                                            delta_v * get_value(A.y, A.x + 1);
                                } else {
                                        circle_point_interpolated_color = (1 -
                                            delta_u) * (1 - delta_v) * get_value(A.
                                            y, A.x) + (1 - delta_u) * delta_v *
                                            get_value(A.y, A.x + 1)
                                               + delta_u * (1 - delta_v) *
                                                    get_value(A.y + 1, A.x) +
                                                    delta_u * delta_v * get_value(A
                                                    .y + 1, A.x + 1);
                                }


                                circle_bits.set(p, !(circle_point_interpolated_color
                                    < current_pxl_color));
                        }

                        //std::cout << "pattern : " << circle_bits << "\n";

                        // rotate until min is found
                        std::bitset<P> min = circle_bits;
                        for (int i = 0; i < 8; ++i) {
                                if (circle_bits.to_ulong() < min.to_ulong()) {
                                        min = circle_bits;
                                }
                                rotate(circle_bits, 1);
                                //std::cout << "rotation: " << circle_bits << "\n";
                        }

                        //std::cout << "minimum bits : " << min << "\n";

                        // create lbp histogram runs
                        auto runs = calculate_runs(min);

                        int encoding = -1;
                        if (runs.size() > 2) {
                                encoding = P + 1;
                        } else if (runs.size() == 1 && runs[0].first == 0) {
                                encoding = 0;
                        } else if (runs.size() == 1 && runs[0].first == 1) {
                                encoding = P;
                        } else {
                                encoding = runs[1].second;
                        }

                        increment_histogram(encoding);

                        //std::cout << "encoding : " << encoding << "\n";

                }
        }


        return lbp_histogram;

}

double Textures::calc_euclidean(const LBPHist& one, const LBPHist& two) {
        double norm = 0.0;
        for (auto& bin_one : one) {
                auto result = two.find(bin_one.first);
                if (result != two.end()) {
                        norm += std::pow(bin_one.second - result->second, 2);
                };
        }
        return std::sqrt(norm);
}
```

```
Textures::Textures()
{
}


Textures::~Textures()
{
}

int main(int argc, char** argv) {

        unsigned char img_arr[] = { 5, 4, 2, 4, 2, 2, 4, 0,
                4, 2, 1, 2, 1, 0, 0, 2,
                2, 4, 4, 0, 4, 0, 2, 4 };

        //cv::Mat img(3, 8, CV_8U, img_arr);


        Textures textures;
        //textures.create_lbp_histogram(img.rows, img.cols, get_pxl);

        std::string classes[] = { "building", "car", "mountain", "tree" };

         std::vector<std::pair<std::string, LBPHist>> trained_data;

        std::string training_path("imagesDatabaseHW8/training/");

        // calculate lbp of training data
        for (auto& classifying_class : classes) {
                for (int i = 1; i <= 20; ++i) {
                        std::stringstream ss;
                        ss << training_path << classifying_class << "/" << std::
                             setfill('0') << std::setw(2) << i << ".jpg";
                        std::string filename = ss.str();
                        cv::Mat training_img = cv::imread(filename,
                             CV_LOAD_IMAGE_GRAYSCALE);

                        auto get_pxl = [&](int row, int col)
                        {
                                return training_img.at<unsigned char>(row, col);
                        };

                        auto lbp_hist = textures.create_lbp_histogram(training_img.
                             rows, training_img.cols, get_pxl);

                        trained_data.push_back(std::make_pair(classifying_class,
                             lbp_hist));

                        if (i == 1) {
                                std::cout << filename << " ";
                                // print vector
                                std::cout << "\\begin{bmatrix} ";

                                for (auto& entry : lbp_hist) {
                                        std::cout << entry.first << " : " << entry.
                                             second << " & ";
                                }

                                std::cout << "\\end{bmatrix}\n ";
                        }
                }
        }

        std::string testing_path("imagesDatabaseHW8/testing/");

        std::unordered_map <std::string, std::unordered_map<std::string, int>>
             confusion_map;
        for (auto & actual_class : classes) {
                for (auto & classified_class : classes) {
                        confusion_map[actual_class][classified_class] = 0;
                }
        }

        int total_test_cases = 0;
        int correct_test_cases = 0;

        for (auto & actual_class : classes) {
                for (int i = 1; i <= 5; ++i) {
                        std::stringstream ss;
                        ss << testing_path << actual_class << "_" << i << ".jpg";
                        std::string filename = ss.str();
                        cv::Mat testing_img = cv::imread(filename,
                             CV_LOAD_IMAGE_GRAYSCALE);

                        auto get_pxl = [&](int row, int col)
                        {
                                return testing_img.at<unsigned char>(row, col);
                        };

                        // calculate test img lbp
                        auto test_img_lbp_hist = textures.create_lbp_histogram(
                             testing_img.rows, testing_img.cols, get_pxl);

                        struct ClassifiedClass {
                                std::string classified_class;
                                double distance;
                        };

                        std::vector<ClassifiedClass> nearest_neighbor_finder;
```

```
                        for (auto& trained_vec : trained_data) {
                                auto trained_lbp_hist = trained_vec.second;

                                double distance = textures.calc_euclidean(
                                        trained_lbp_hist, test_img_lbp_hist);

                                ClassifiedClass temp_classifier;
                                temp_classifier.classified_class = trained_vec.first
                                        ;
                                temp_classifier.distance = distance;

                                nearest_neighbor_finder.push_back(temp_classifier);
                        }

                        // find k nearest neibors
                        std::sort(nearest_neighbor_finder.begin(),
                             nearest_neighbor_finder.end(), [&](const
                             ClassifiedClass& left, const ClassifiedClass& right)
                        {
                                return left.distance < right.distance;
                        });

                        std::unordered_map<std::string, int> class_counter;
                        for (int k = 0; k < 5; ++k) {
                                auto near_class = nearest_neighbor_finder[k].
                                        classified_class;
                                //std::cout << "near neighbor : " << near_class << "
                                       distance : " << nearest_neighbor_finder[k].
                                       distance << " \n";
                                auto result = class_counter.find(near_class);
                                if (result != class_counter.end()) {
                                        result->second++;
                                } else {
                                        class_counter[near_class] = 1;
                                }
                        }

                        int max_count = -1;
                        std::string classified_class;

                        for (auto& counter_item : class_counter) {
                                if (counter_item.second > max_count) {
                                        max_count = counter_item.second;
                                        classified_class = counter_item.first;
                                }
                        }

                        confusion_map[actual_class][classified_class]++;

                        //std::cout << filename << " actual : " << actual_class << "
                              classified : " << classified_class << "\n";
                        if (actual_class == classified_class) {
                                correct_test_cases++;
                        }
                        total_test_cases++;

                }
        }

        std::cout << "Accuracy : " << correct_test_cases / (double)(total_test_cases
             ) << "\n";

        // print confusion matrix
        std::cout << "\n & ";
        for (auto & actual_classes : confusion_map) {
                std::cout << actual_classes.first << " & ";
        }
        std::cout << "\n";
        for (auto & actual_classes : confusion_map) {
                std::cout << actual_classes.first << " & ";
                for (auto & classified_classes : actual_classes.second) {
                        std::cout << classified_classes.second << " & ";
                }
                std::cout << "\n";
        }


}
```