

Primena fazi logike u obradi slika

Jelena Mrdak, mi15021

Tijana Jevtić

5. januar 2019

Sadržaj

1 FCM	1
1.1 Binarizacija slike	3
1.2 FCM i k-means	6
2 Fuzzy Edge Detection	8

1 FCM

Fuzzy C-means (FCM) je jedan od najpopularnijih algoritama za fazi klasterovanje. U ovom poglavlju ćemo ga najpre detaljno opisati, a zatim ćemo ga iskoristiti za binarizaciju slike.

Cilj ovog algoritma je da skup $X = \{x_1, x_2, \dots, x_n\}$ particioniše na k delova (klastera) po nekom kriterijumu. Preciznije, kriterijum je minimizacija sledeće funkcije:

$$F(\bar{w}, \bar{c}) = \sum_{i=1}^n \sum_{j=1}^k w_{ij}^m \|x_i - c_j\|^2,$$

gde $w_{ij} \in [0, 1]$ predstavlja pripadnost tačke x_i j -tom klasteru i $\sum_{j=1}^k w_{ij} = 1$, dok je c_j centroid j -tog klastera. Realni parametar $m > 1$ predstavlja faktor fazifikacije i on se zadaje unapred. U nastavku ćemo preciznije odrediti ove koeficijente. Sada ćemo samo ukratko opisati korake algoritma.

FCM je veoma sličan algoritmu k-means i sastoji se iz sledećih koraka:

- Izabrati broj klastera k .
- Svakoj tački x_i dodeliti koeficijente $w_{ij} \in [0, 1], j = 1, 2, \dots, k$.
- Ponavljati sve dok ne dođe do konvergencije:

- Izračunati centroide za svaki klaster.
- Ažurirati koeficijente.
- Tačku x_i dodeliti klasteru kom najviše pripada, tj. r -tom klasteru, gde je $w_{ir} = \max_j w_{ij}$.

Teorema 1.1. *Potrebni uslovi za minimizator (\bar{w}^*, \bar{c}^*) funkcije $F(\bar{w}, \bar{c})$ su:*

$$c_j = \frac{\sum_{i=1}^n w_{ij}^m \cdot x_i}{\sum_{i=1}^n w_{ij}^m} \quad (1)$$

i

$$w_{ij} = \frac{1}{\sum_{u=1}^k \left(\frac{\|x_i - c_j\|}{\|x_i - c_u\|} \right)^{\frac{2}{m-1}}} \quad (2)$$

Dokaz. Pronaći ćemo potencijalne tačke lokalnih uslovnih ekstremuma. Koristićemo Lagranževe množioce. Posmatraćemo pomoćnu funkciju:

$$J(\bar{w}, \bar{c}, \bar{\lambda}) = \sum_{i=1}^n \sum_{j=1}^k w_{ij}^m \|x_i - c_j\|^2 - \sum_{i=1}^n \lambda_i \left(\sum_{j=1}^k w_{ij} - 1 \right).$$

Tačke koje tražimo moraju da zadovoljavaju uslov $\nabla J = \mathbf{0}$. Dakle,

$$\frac{\partial J}{\partial c_j} = 0, 1 \leq j \leq k \quad (3)$$

$$\frac{\partial J}{\partial w_{ij}} = 0, 1 \leq i \leq n, 1 \leq j \leq k \quad (4)$$

$$\frac{\partial J}{\partial \lambda_i} = 0, 1 \leq i \leq n \quad (5)$$

Rešavanjem (3) dobijamo (1). Iz (4) imamo

$$m w_{ij}^{m-1} \|x_i - c_j\|^2 - \lambda_i = 0,$$

odnosno

$$w_{ij} = \left(\frac{\lambda_i}{m \|x_i - c_j\|^2} \right)^{\frac{1}{m-1}}. \quad (6)$$

Iz (5) dobijamo:

$$\begin{aligned}
1 &= \sum_{u=1}^k w_{iu} \\
&= \sum_{u=1}^k \left(\frac{\lambda_i}{m \|x_i - c_u\|^2} \right)^{\frac{1}{m-1}} \\
&= \sum_{u=1}^k \left(\frac{m \|x_i - c_u\|^2}{\lambda_i} \right)^{\frac{1}{1-m}} \\
&= \sum_{u=1}^k \frac{(m \|x_i - c_u\|^2)^{\frac{1}{1-m}}}{\lambda_i^{\frac{1}{1-m}}} \\
&= \frac{1}{\lambda_i^{\frac{1}{1-m}}} \sum_{u=1}^k (m \|x_i - c_u\|^2)^{\frac{1}{1-m}},
\end{aligned}$$

pa zaključujemo da je

$$\lambda_i = \left(\sum_{u=1}^k (m \|x_i - c_u\|^2)^{\frac{1}{1-m}} \right)^{1-m}.$$

Konačno, zamenjujući poslednju jednakost u (6), dobijamo (2). □

FCM algoritam za određivanje minimizatora funkcije F je iteracija kroz potrebne uslove.

1.1 Binarizacija slike

Ispod je prikazan kod za binarizaciju slike koji koristi FCM algoritam. Napominjemo da se zbog čitljivosti koda u ovom delu nismo odlučili za efikasnu implementaciju. O tome će biti više reči u narednoj sekciji.

```

#include <iostream>
#include <opencv2/highgui/highgui.hpp>

int main(int argc, const char *argv[])
{
    if (argc != 2) {
        std::cerr << "Usage: ./binarization path_to_img" << std::endl;
        return 1;
    }

    // read image
    cv::Mat img = cv::imread(argv[1], cv::IMREAD_GRAYSCALE);
    cv::Mat img_binary = cv::Mat(img.rows, img.cols, CV_8UC1,
        ↪ cv::Scalar(255));

```

```

// weights
std::vector<std::vector<float>> w1(img.rows,
    ↪ std::vector<float>(img.cols, 0));
std::vector<std::vector<float>> w2(img.rows,
    ↪ std::vector<float>(img.cols, 0));
// fuzzification factor
double m = 2;
// centroids
std::pair<float, float> c;

// init weights
for (int i = 0; i < img.rows; i++) {
    for (int j = 0; j < img.cols; j++) {
        w1[i][j] = img.at<unsigned char>(i,j)/255.0;
        w2[i][j] = 1 - w1[i][j];
    }
}

// stopping criteria
float eps = 0;

do {
    // calculate centroids
    std::pair<float, float> c1_fraction{0,0};
    std::pair<float, float> c2_fraction{0,0};

    for (int i = 0; i < img.rows; i++) {
        for (int j = 0; j < img.cols; j++) {
            c1_fraction.first += std::pow(w1[i][j], m)*img.at<unsigned
                ↪ char>(i,j);
            c1_fraction.second += std::pow(w1[i][j], m);
            c2_fraction.first += std::pow(w2[i][j], m)*img.at<unsigned
                ↪ char>(i,j);
            c2_fraction.second += std::pow(w2[i][j], m);
        }
    }

    auto old_c = c;
    c = {c1_fraction.first/c1_fraction.second,
        ↪ c2_fraction.first/c2_fraction.second};
    eps = (old_c.first-c.first)*(old_c.first-c.first) +
        ↪ (old_c.second-c.second)*(old_c.second-c.second);

    // update weights
    for (int i = 0; i < img.rows; i++) {

```

```

    for (int j = 0; j < img.cols; j++) {
        float d1 = std::abs(img.at<unsigned char>(i,j)-c.first);
        float d2 = std::abs(img.at<unsigned char>(i,j)-c.second);
        w1[i][j] = 1/(std::pow(d1/d1, 2/(m-1)) + std::pow(d1/d2,
            ↪ 2/(m-1)));
        w2[i][j] = 1/(std::pow(d2/d1, 2/(m-1)) + std::pow(d2/d2,
            ↪ 2/(m-1)));
    }
}

} while(eps > 1);

// cluster pixels based on weights
for (int i = 0; i < img_binary.rows; i++) {
    for (int j = 0; j < img_binary.cols; j++) {
        img_binary.at<unsigned char>(i,j) = (w1[i][j] > w2[i][j]) ? 255 :
            ↪ 0;
    }
}

// show and save binary image
namedWindow("Display window", cv::WINDOW_AUTOSIZE);
imshow("Display window", img_binary);
cv::waitKey(0);
imwrite("fcm.png", img_binary);

return 0;
}

```

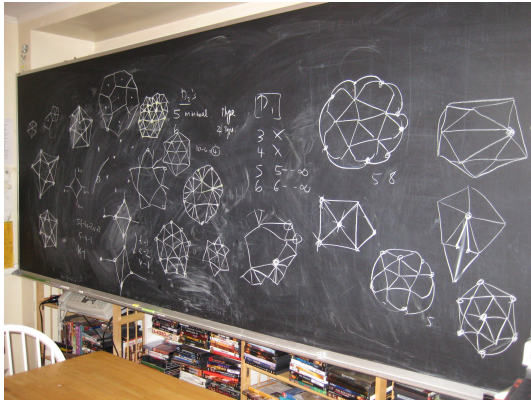
Rezultat izvršavanja algoritma je prikazan ispod.



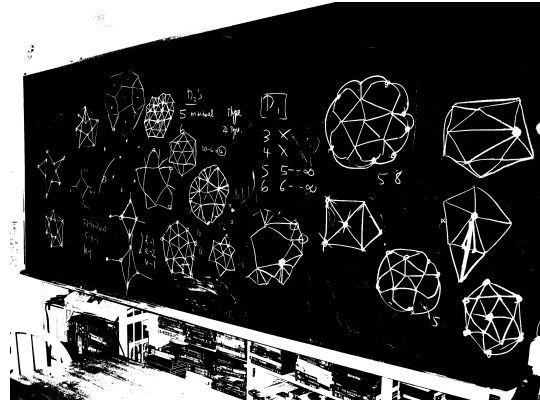
Slika 1: input



Slika 2: output



Slika 3: input



Slika 4: output

1.2 FCM i k-means

U ovom odeljku ćemo uporediti rezultate algoritama FCM i k-means, kao i vremena njihovih izvršavanja.

Napomena 1.1. Koristićemo efikasniju implementaciju FCM algoritma od one date u sekciji 1.1.

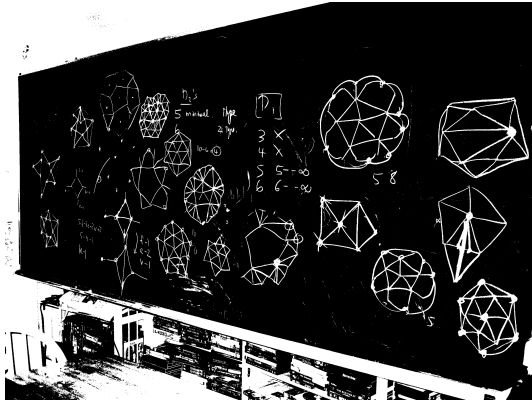
Na sledećim slikama su prikazani rezultati.



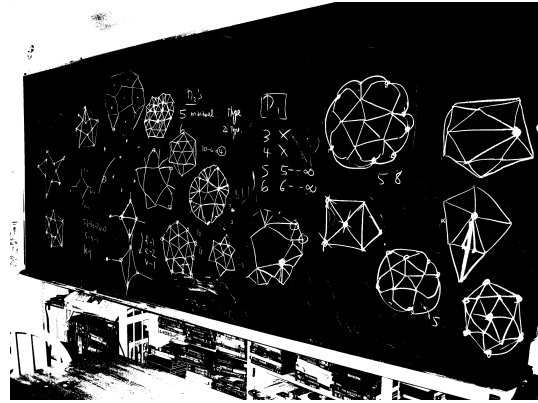
Slika 5: FCM output
Broj iteracija: 7



Slika 6: k-means output
Broj iteracija: 6



Slika 7: FCM output
Broj iteracija: 7



Slika 8: k-means output
Broj iteracija: 7

Možemo primetiti da su slike 5 i 6 identične, dok se slike 7 i 8 neznatno razlikuju. Međutim, vremena izvršavanja se primetno razlikuju. U Tabeli 1 su prikazana vremena (u sekundama) potrebna da algoritmi obrade Sliku 1 500, 1000 i 1500 puta. Slično, u Tabeli 2 su prikazana vremena potrebna da se obradi Slika 3 50, 100 i 150 puta.

broj izvršavanja	FCM	k-means
500	4	21
1000	8	42
1500	12	63

Tabela 1: Input Slika 1

broj izvršavanja	FCM	k-means
50	8	44
100	17	89
150	25	133

Tabela 2: Input Slika 3

Prikazaćemo još dva testa urađena na dve nove slike.

broj izvršavanja	FCM	k-means
100	15	119
150	22	179
200	31	238

Tabela 3:

broj izvršavanja	FCM	k-means
1000	3	36
1500	5	53
2000	6	71

Tabela 4:

Na osnovu podataka iz tabela, zaključujemo:

test	k-means/FCM
1	5
2	5
3	8
4	12

Tabela 5: Koliko puta je FCM brži od k-means

Treba napomenuti da su ovi rezultati okvirni, jer u velikoj meri zavise od implementacije samih algoritama. Naime, za centoride u k-means algoritmu je korišćen celobrojni tip

(int), dok je centroide u FCM algoritmu korišćen realni tip (float). U oba slučaja su se algoritmi zaustavljali kad je promena u centroidima bila manja od jedan. Dakle, već tu može doći do razlike u broju iteracija. Međutim, i dalje očekujemo da će FCM biti brži od k-means.

Takođe, ističemo da FCM koristi više memorije nego k-means.

2 Fuzzy Edge Detection

```
#include <opencv2/highgui/highgui.hpp>
#include <iostream>

int main(int argc, const char *argv[])
{
    if (argc != 3) {
        std::cerr << "Usage: ./binarization path_to_img threshold" <<
            ↪ std::endl;
        return 1;
    }

    // read image
    cv::Mat img = cv::imread(argv[1], cv::IMREAD_GRAYSCALE);

    float threshold = std::atof(argv[2]);
    const int L = 256;
    std::vector<std::vector<float>> mi(img.rows,
        ↪ std::vector<float>(img.cols, 0));
    auto output = gaussian_blur(img);

    float maxm = 0;
    for (int i = 1; i < output.rows-1; i++) {
        for (int j = 1; j < output.cols-1; j++) {
            unsigned s = 0;
            for (int x = -1; x <= 1; x++) {
                for (int y = -1; y <= 1; y++) {
                    s += std::abs(output.at<unsigned char>(i,j)-output.at<unsigned
                        ↪ char>(i+x,j+y));
                }
            }
            mi[i][j] = (1.0*s)/(s+(L-1));
            maxm = std::max(maxm, mi[i][j]);
        }
    }

    for (int i = 1; i < output.rows-1; ++i) {
```



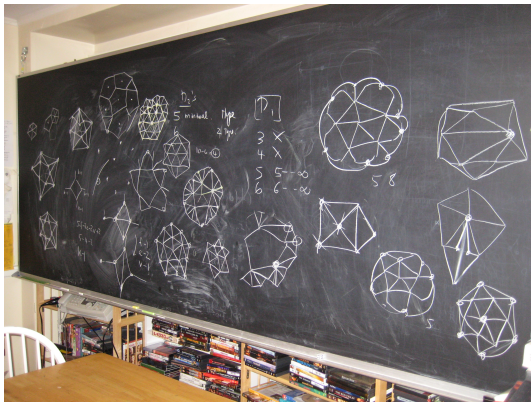
```

for (int j = 1; j < output.cols-1; j++) {
    output.at<unsigned char>(i,j) = (mi[i][j]/maxm < threshold) ? 0 :
        ↪ 255;
}
}

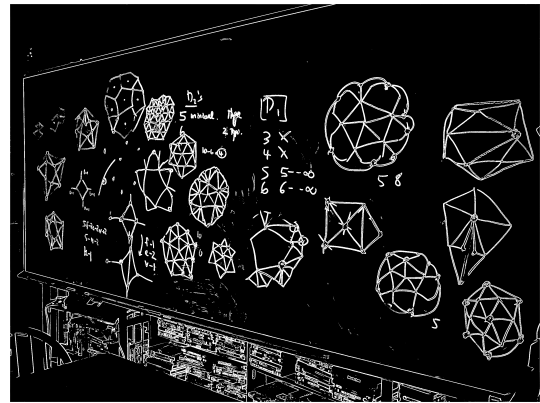
namedWindow("Display window", cv::WINDOW_AUTOSIZE);
imshow("Display window", output);
cv::waitKey(0);
imwrite("edge_detection.png", output);

return 0;
}

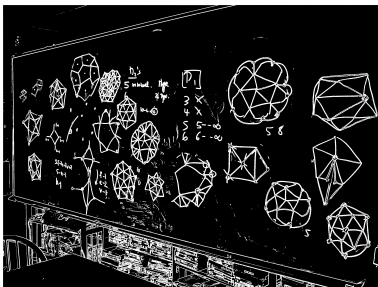
```



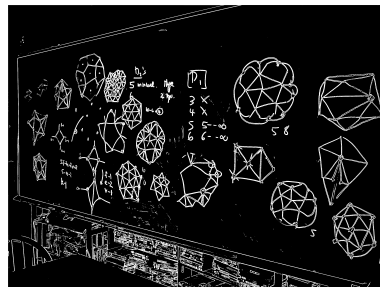
Slika 9: input



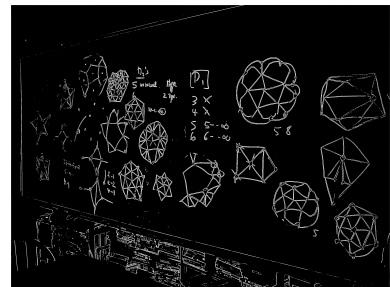
Slika 10: output



Slika 11: threshold = 0.25



Slika 12: threshold = 0.35



Slika 13: threshold = 0.5