Univerzitet u Sarajevu
Elektrotehnički fakultet
Odsjek za automatiku i elektroniku
Predmet: Optimizacija resursa
Akademska: 2017/2018.g.

Klasterizacija slika primjenom PSO algoritma

Student:
Besim Arnautović 1404/16920

Januar 2018.

Sadržaj

Popis slika

1 Uvod

Klasificiranje slika je proces indentifikovanja skupova sličnih komponenti slike. Ove komponente slike mogu biti pikseli, regije, linije, itd. zavisno od problema kojeg rješavamo. Mnogo osnovnih tenika za procesiranje slika kao što su kvantizacija, segmentacija mogu se posmatrati kao različite instance problema klasterizacije[1].

Postoje dva glavna pristupa klasificiranju slika: sa i bez nadzora. U metodi sa nadzorom, broj i numeričke karakteristike (kao što su srednja vrijednost i varijansa) klasa u slici su unaprijed poznate i koriste se u koraku treniranja, što prethodi koraku klasificiranja. Postoji nekoliko popularnih algoritama sa nadzorom kao što su minimum-distance-to-mean, paralelopiped i Gausovi klasifikatori na bazi maksimuma vjerovatnoće. Za pristupe bez nadzora, klase su nepoznate i pristup polazi od segmentiranja slike u grupe (ili klastere), prema zadanoj mjeri sličnosti. Stoga, klasifikatori bez nadzora su također poznati kao problemi klasterizacije. Generalno, pristup bez nadzora ima par prednosti u odnosu na pristup sa nadzorom, od toga su najbitniji:

- Za pristupe bez nadzora nema potrebe da se u skupu podataka slika sve klase specificiraju unaprijed. Algoritam klasterizacije će automatski pronaći disjunktne klase, što će mnogo smanjiti posao traženja *a priori* informacija.
- Karakteristike objekata koje se klasificiraju mogu da se mijenjaju kroz vrijeme; pristup bez nadzora je savršen način da se ove promjene mjere i posmatraju
- Neke karakteristike objekta mogu bit nepoznate unaprijed. Pristupi bez nadzora će automatski obilježiti ove karakteristike.

Fokus u ovom radu se stavlja na pristup bez nadzora (tzv. klasterizacija slika). Postoji par algoritama koji pripadaju ovoj klasi. Ovi algoritmi se mogu kategorizirati u dvije grupe: hijerarhijski i particionalni. Za hijerarhijske algoritme klasterizacije, izlaz je stablo koje pokazuje sekvencu klasterizacije gdje je svaki klaster dio skupa podataka. Ovi algoritmi imaju iduće prednosti:

- Broj klastera ne mora biti zadan *a priori*.
- Nezavisni su od početnih uslova.

Ali, ovi algoritmi također imaju i par mana:

- Statički su, npr. pikseli dodijeljeni klasteru ne mogu se premjestiti u drugi klaster.
- Nekada mogu ne uspjeti da separiraju klastere koji se preklapaju zbog nedostatka informacija o globalnoj veličini klastera.

U drugu ruku, particionalni algoritmi klasterizacije dijele skup podataka u određen broj klastera. Ovi algoritmi imaju za cilj minimizirati određeni kriterij (npr. kvadratnu funkciju greške). Stoga, mogu biti tretirani kao problem optimizacije. Prednosti hijerarhijskih algoritama su mane particionalnih i vice versa.

Algoritam koji se najviše primjenjuje je iterativni K-means algoritam. Za K-means klasterizaciju, algoritam starta sa K klaster centara ili *centroida* (početne vrijednosti

centroida su randomly izabrane ili izvedene iz *a priori* informacija). Onda, svaki piksel u slici je dodijeljen najbližem klasteru (tj. najbližem centroidu). Konačno, centoridi se računaju ponovo poštoujući dodijeljene piksele. Ovaj proces se ponavlja do konvergencije algoritma. Postoji par nedostataka ovog algoritma:

- ovaj algoritam ovisi o podacima
- također je pohlepni algoritam koji ovisi o početnim uslovima, koji mogu izazvati da algoritam zapadne u lokalno umjesto u globalno rješenje i
- korisnik mora da specificira broj klasa unaprijed.

Postoje još mnogi algoritmi koji su našli primjenu gdje je i K-means algotiram, a najznačajniji su:

- ISODATA i SYNERACT - ovi algoritmi su unaprijeđenje K-means algoritma i daju mnogo bolje rezultate
- Fuzzy C-means (FCM) algoritam - u ovom algoritmi pikseli mogu pripadati više klastera, što je određeno fuzzy koeficijentom koji je pridružen svakom pikselu.
- Expectation–Maximization (EM) algoritam - unaprijeđena verzija ovog algoritma se koristi za segmentaciju slika magnetnih rezonanci mozga
- K-harmonic means (KHM) algoritam
- Artificial Neural Networks (ANN) algoritmi
- Genetički algoritmi (GA)
- PSO algoritam i mnogi drugi.

2 Particle Swarm Optimization

Particle Swarm Optimizers (PSO) predstavljaju optimizaciju baziranu na populaciji modeliranoj na bazi simulacije socijalnog ponašanja ptica u jatu[2] [3]. PSO održava jato mogućih rješenja problema optimizacije pod razmatranjem. Svako moguće rješenje se naziva čestica. Ako problem optimizacije ima n varijabli, onda svaka čestica predstavlja n-dimenzionalnu tačku u problemskom prostoru. Kvaliteta, ili fitnes, čestice mjeri se koristeći fitnes funkciju. Funkcija fitnesa kvantificira koliko je blizu čestica optimalnom rješenju.

Svaka čestica se kreće kroz problemski prostor, korigirajući svoju poziciju na osnovu distance između svoje trenutne i svoje najbolje pozicije i distance između svoje trenutne i globalno najbolje pozicije u jatu. Performansa čestice, tj. koliko je čestica blizu globalnom optimumu, mjerena je koristeći funkciju fitnesa oja zavisi od problema optimizacije.

Svaka čestica je opisana sa idućim informacijama:

- $\mathbf{x}_i$, *trenutna pozicija* čestice

- $\mathbf{v}_i$, *trenutna brzina* kretanja čestice i
- $\mathbf{y}_i$, *personalna najbolja pozicija* čestice.

Personalno najbolja pozicija koja se veže za česticu i je najbolja pozicija koju je čestica imala do tada, tj. pozicija u kojoj je čestica imala najveću vrijednost fitnesa. Personalna najbolja vrijednost pozicije stoga služi kao neka vrsta memorije. Ako sa f označimo funkciju koju minimalizujemo, onda najbolja vrijednost pozicije čestice u trenutku t se računa po formuli 2.1:

$$\mathbf{y}_i(t+1) = \begin{cases} \mathbf{y}_i(t) & ako\ f(\mathbf{x}_i(t+1)) \geq f(\mathbf{y}_i(t)) \\ \mathbf{x}_i(t+1) & ako\ f(\mathbf{x}_i(t+1) < f(\mathbf{y}_i(t)) \end{cases} \tag{2.1}$$

Jedan od jedinstvenih principa PSO algoritma je razmjena informacija između pripadnika jata. Ova razmjena informacija se koristi da bi se utvrdile najbolja čestica i najbolja pozicija u jatu tako da ostale čestice mogu da prilagode svoju poziciju prema najboljoj. Prve socijalne topologije koje su razvijene su bile topologija zvijezde i topologija prstena. Topologija zvijezde dopušta da svaka čestica komunicira sa svim ostalim česticama. Rezultat topologije zvijezdee je da je najbolja čestica u jatu određena i sve ostale čestice se kreću ka globalno najboljoj čestici. Rezultirajući algoritam se često naziva *gbest* PSO. Topologija prstena, u drugu ruku, definiše preklapajuća susjedstva čestica. Čestice u susjedstvu komuniciraju da bi indentifikovale najbolju česticu u tom susjedstvu. Potom, sve čestice u tom susjedstvu se kreću prema najboljoj lokalnoj čestici (tj. najboljoj čestici u tom susjedstvu). Rezultirajući algoritam se naziva *lbest* PSO.

Za *gbest* PSO algoritam, gdje je najbolja čestica određena iz cijelog jata, najbolja čestica je 2.2:

$$\hat{\mathbf{y}}_i(t) \in \{\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_s\} = \min\{f(\mathbf{y}_0(t)), f(\mathbf{y}_1(t)), \ldots, f(\mathbf{y}_s(t))\} \tag{2.2}$$

gdje je s ukupni broj čestica u jatu. Za *lbest* PSO model, susjedstvo se definiše jednačinom 2.3:

$$N_j = \{\mathbf{y}_{i-l}(t), \mathbf{y}_{i-l+1}(t), \ldots, \mathbf{y}_{i-1}(t), \mathbf{y}_i(t), \mathbf{y}_{i+l}(t), \ldots, \mathbf{y}_{i+l-1}(t), \mathbf{y}_{i+l}(t)\} \tag{2.3}$$

i najbolja čestica u susjedstvu N_j je:

$$\hat{\mathbf{y}}_j(t+1) \in N_j | f(\hat{\mathbf{y}}_j(t+1)) = \min\{f(y_i)\},\ \forall y_i \in N_j$$

Susjedstva su obično definisana koristeći indekse čestica, iako se topološka susjedstva također koriste. Model *gbest* PSO je samo specijalan slučaj *lbest* sa $l = s$; tj. susjedstvo je čitavo jato. Dok *lbest* PSO ima veću raznolikost nego *gbest* PSO, značajno je sporiji od *gbest* PSO-a. Ostatak rada je koncentrisan na brži *gbest* PSO.

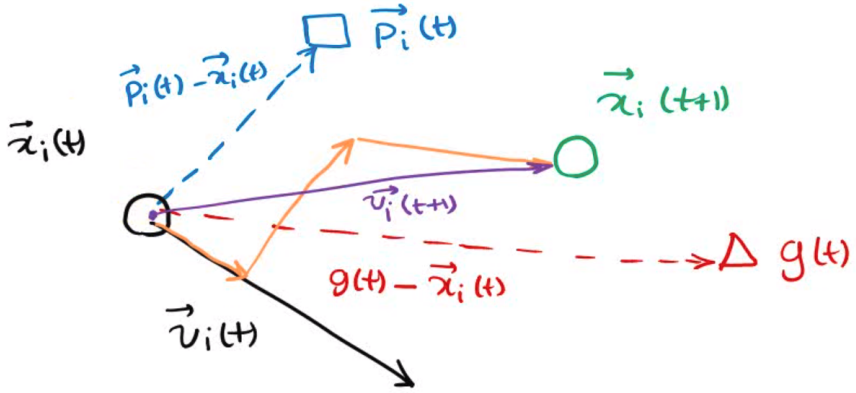
Za svaku iteraciju *gbest* PSO algoritma, $\mathbf{v}_i$ i $\mathbf{x}_i$ se mjenjaju po jednačinama 2.4 i 2:

$$\mathbf{v}_i(t+1) = w\mathbf{v}_i(t) + c_1\mathbf{r}_1(t)(\mathbf{y}_i(t) - \mathbf{x}_i(t)) + c_2\mathbf{r}_2(t)(\hat{\mathbf{y}}(t) - \mathbf{x}_i(t)) \tag{2.4}$$

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1) \tag{2.5}$$

gdje je w težinski koeficijent inercije, c_1 i c_2 su konstante ubrzanja i $\mathbf{r}_1(t)$ i $\mathbf{r}_2(t)$ su vektori sa random elementima raspoređenih normalnom distribucijom, $U(0,1)$. Jednačina 2.4 se sastoji od tri komponente:

- *Inercioni dio*, koji služi kao memorija prijašnjih brzina. Težinski koeficijent inercije kontroliše uticaj prijašnjih brzina: veliki koeficijent inercije favorizuje istraživanje (više random kretanje čestica) dok manji koeficijent inercije favorizuje eksploataciju.
- *Kognitivni dio*, $\mathbf{y}_i(t) - \mathbf{x}_i(t)$, koji predstavlja iskustvo same čestice, tj. informaciju dosadašnje najbolje pozicije koju je čestica imala. Kognitivna komponenta služi kao memorija za prijašnju najbolju poziciju svake čestice.
- *Socijalna komponenta*, $\hat{\mathbf{y}}(t) - \mathbf{x}_i(t)$, koja predstavlja informaciju gdje se nalazi najbolja globalna pozicija neke čestice (slika 2.1).



Slika 2.1: Grafički prikaz navedenih komponenti, gdje je p_i personalna najbolja pozicija čestice, a g_i globalna najbolja pozicija.

Performansa PSO algoritma je osjetljiva na parametre w, c_1 i c_2. Postoji par savjeta (baziranim na empirijskim istraživanjima) za dobro određivanje parametara, ali teoretska istraživanja su bila dovoljna da daju granice za ove vrijednosti:

$$w > \frac{1}{2}(c_1 + c_2), \quad w < 1$$

Ako izaberemo konstante na ovaj način, PSO pokazuje konvergentno ponašanje. Ako gornji uslov nije ispunjen, PSO pokazuje ciklično ili divergentno ponašanje [4] [5].
Da bi osigurali to da brzine čestica nisu prevelike (što može izazvati da čestice napuste problemski prostor), računanje brzine se može ograničiti. Ograničavanje računanja brzine je ipak problemski ovisno.

3 Klasterizacija slike

Za potrebe ovog paragrafa, definisati ćemo iduće pojmove:

- N_b predstavlja broj spektralnih traka u slici
- N_p predstavlja broj piksela u slici
- N_c predstavlja broj spektralnih klasa (ovaj podatak unosi korisnik)
- $\mathbf{z}_p$ predstavlja N_p komponenti piksela p
- m_j predstavlja srednju vrijednost klastera j.

3.1 PSO bazirana klasterizacija slike

U smislu klasterizacije slike, jedna čestica predstavlja N_c srednjih vrijednosti klastera. To znači, da je svaka čestica $\mathbf{x}_i$ sastavljena od $\mathbf{x}_i = \{\mathbf{m}_{i1}, \ldots, \mathbf{m}_{ij}, \ldots, \mathbf{m}_{iN_c})$ gdje $\mathbf{m}_{ij}$ predstavlja j-ti centroid vektor klastera i-te čestice. Stoga, jato predstavlja broj kandidata klasterizacije slike.

Da bi se izračunala kvaliteta svake čestice u jatu, tj. funkcija fitnesa, koristit ćemo jednačinu 3.1:

$$f(\mathbf{x}_i, \mathbf{Z}_i) = w_1 \bar{d}_{max}(\mathbf{Z}_i, \mathbf{x}_i) + w_2(z_{max} - d_{min}(\mathbf{x}_i)) \tag{3.1}$$

gdje z_{max} predstavlja maksimalnu vrijednost piksela u slici (npr. $z_{max} = 2^b - 1$ za b-bitnu sliku); $\mathbf{Z}_i$ je matrica pripadnosti piksela klasteru čestice i. Svaki element z_{ijp} indicira da li piksel $\mathbf{z}_p$ pripada klasteru C_{ij} čestice i. Konstante w_1 i w_2 su korisnički definisane konstante. Također:

$$\bar{d}_{max}(\mathbf{Z}_i, \mathbf{x}_i) = \max_{j=1,\ldots,N_c} \left\{ \sum_{\forall \mathbf{z}_p \in C_{ij}} \frac{d(\mathbf{z}_p, \mathbf{m}_{ij}}{|C_{ij}|} \right\} \tag{3.2}$$

je maksimalna srednja euklidska udaljenost čestice od svoje dodijeljene klase, i

$$d_{min}(\mathbf{x}_i) = \min_{\forall j_1, j_2, j_1 \neq j_2} \Big\{ d(\mathbf{m}_{ij_1}, \mathbf{m}_{ij_2}) \Big\} \tag{3.3}$$

je minimalna euklidska udaljenost između bilo kojeg para klustera. U gore navedenim jednačinama, $|C_{ij}|$ je kardinalni broj skupa $|C_{ij}|$.

Ova funkcija fitnesa ima zadatak da istovremeno:

- minimizira intra-udaljenost između piksela i pridruženih srednjih vrijednosti klastera, što predstavlja član $\bar{d}_{max}(\mathbf{Z}_i, \mathbf{x}_i)$ i
- maksimizuje inter-udaljenost između bilo koja dva para klastera, što predstavlja $d_{min}(\mathbf{x}_i)$.

Funkcija fitnesa je zbog ovoga ima više funkcionalnosti. Pristupi rješavanju problema sa više funkcionalnosti su razvijeni većinom samo za evolucione računarske pristupe. Nedavno je razvijen pristup za više-funkcionalnu optimizaciju koristeći PSO algoritam. Pošto se mi ovdje nećemo koncentrisati na rješavanje više-funkcionalnih problema, koristit ćemo drugačije prioritete za pod-funkcionalnosti kroz odgovarajuće inicijalizacije vrijednosti w_1 i w_2.
Pseudokod PSO algoritma je:

1. Svaka čestica se inicijalizira da sadrži N_c randmoly izabranih srednjih vrijednosti klastera
2. For $t = 1$ to t_{max}
   (a) Za svaku česticu i
      i. Za svaki piksel $\mathbf{z}_p$
         - izračunati $d(\mathbf{z}_p, \mathbf{m}_{ij})$ za sve klastere C_{ij}

- dodijeliti $\mathbf{z}_p$ klasteru C_{ij} gdje je $d(\mathbf{z}_p, \mathbf{m}_{ij}) = \min_{\forall c=1,\ldots,N_c}\{d(\mathbf{z}_p, \mathbf{m}_{ic})\}$

ii. Izračunati fintes $f(\mathbf{x}_i(t), \mathbf{Z}_i)$

(b) Pronaći globalno najbolje rješenje $\hat{\mathbf{y}}(t)$

(c) Promjeniti centroide klastera koristeći jednačine 2.4 i 2.

Prednosti korištenja PSO algoritma je to da izvršava paralelno pretraživanje za optimalne klastere. Ovakvo pretraživanje koje je bazirano na populaciji smanjuje efekat početnih vrijednosti u poređenju sa K-means algoritmom (posebno za relativno velike dimenzije jata).

4 Implementacija

U ovom poglavlju će biti detaljno objašnjen kod za implementaciju[1]PSO algoritma za klusterizaciju slika.

4.1 Klasa *Particle*

U ovoj klasi su implementirane metode kreiranja i ažuriranja svih centroida (srednjih vrijednosti klastera), fitnesa čestice i pozicije čestice. U idućim poglavljima ćemo objasniti rad svake metode ove klase, krenuvši od konstruktora.

4.1.1 Konstruktor

```
class Particle(object):
    def __init__(self, number_clusters, im_shape):
        self.fitness = float
        self.position = np.random.randint(0, 256, (
            number_clusters, im_shape[2]))
        self.best_fitness = None
        self.best_position = self.position
        self.velocity = 0.0
        self.clusters = []
        self.num_clusters = number_clusters
        self.im_shape = im_shape
```

Listing 1: Kod konstrukotra klase Particle

Kao što je rečeno u prijašnjim poglavljima, veličinu klastera slike i same dimenzije slike u PSO algoritmu zadaje korisnik, što su i sam konstruktor klase *Particle* prima kao ulazne podatke. Osnovni atributi klase *Particle*, tj. jedne čestice, su, kao što je rečeno, trenutna pozicija, *self.position*, koja se pri inicijalizaciji čestice popunjava random brojevima koji imaju vrijednosti između minimalne i maximalne vrijednosti piksela u slici i imaju istu dimenziju kao dimenzije piksela u slici. Atributi *self.fitness, self.velocity, self.best_fitness* će se definirati nakon pozivanja metoda koje ćemo objasniti u idućim poglavljima. Atribut *self.best_position* se postavlja na trenutnu poziciju jer je to jedina pozicija koju je čestica posjetila, tako da je to automatski čini i najboljom. Također se definišu i klasteri koje ta čestica sadrži u sebi kao prazan niz.

[1]Projekat je implementiran koristeći python3

4.1.2 Metoda *update_cluster*

```
    def update_clusters(self, image):
        self.clusters = [np.zeros((self.im_shape[0], self.
            im_shape[1]), dtype=np.bool) for _ in range(self.
            num_clusters)]
        dim_x = image.shape[0]
        dim_y = image.shape[1]
        for i in range(dim_x):
            for j in range(dim_y):
                minimum = sqrt(((image[i, j, :] - self.position
                    [0, :])**2).sum())
                num_of_cluster = 0
                for k in range(1, self.num_clusters):
                    distance = sqrt(((image[i, j, :] - self.
                        position[k, :])**2).sum())
                    if distance < minimum:
                        minimum = distance
                        num_of_cluster = k
                self.clusters[num_of_cluster][i, j] = True
```

Listing 2: Kod metode update_cluster koja mjenja centoride jedne čestice

Ova metoda kao ulazne podatke (pored instance nad kojom se poziva) prima i sliku nad kojom vršimo obradu. U prvoj liniji popunjavamo klastere nulama i postavljamo ih na odgovarajuću dimenziju. U ovoj metodi kroz dvije *for* petlje tražimo centroide svih klastera čestice, ažuriramo ih i nakon toga dodjeljujemo piksel najbližem klasteru. *For* petlje koje definišu brojači i i j ažuriraju centroide sadržane u čestici, dok *for petlja* po brojaču k traži minimalnu udaljenost piksela od svih centroida i pridružuje piksel najbližem klasteru na osnovu izračunatog minimuma.

4.1.3 Metoda *calculate_fitness*

```
    def d_max(self, image):
        maximum = 0
        for i in range(self.num_clusters):
            if self.clusters[i].sum() == 0:
                continue
            pixels_in_cluster = image[np.where(self.clusters[i])]
            d_prvo = (pixels_in_cluster-self.position[i, :])**2
            d_prvo = d_prvo.reshape((d_prvo.shape[0], d_prvo.
                shape[1], 1))
            d_prvo = np.sqrt(d_prvo.sum(axis=2)).sum().sum()
            d = d_prvo / self.clusters[i].sum()
            if d > maximum:
                maximum = d
        return maximum

    def d_min(self):
        minimum = float
        for i in range(self.num_clusters):
            for j in range(i + 1, self.num_clusters):
                p1 = self.position[i, :]
                p2 = self.position[j, :]
                d = np.sqrt(((p1 - p2) ** 2).sum())
                if i == 0 and j == 1:
```

```
                    minimum = d
                else:
                    if d < minimum:
                        minimum = d
        return minimum

    def calculate_fitness(self, image):
        global w1, w2, z_max
        self.fitness = w1*self.d_max(image) + w2*(z_max - self.
            d_min())
        if self.best_fitness is None:
            self.best_fitness = self.fitness
        elif self.fitness < self.best_fitness:
            self.best_fitness = self.fitness
            self.best_position = self.position
```

Listing 3: Listing kodova metoda koje se koriste za računanje fintesa čestice

Ove tri metode navedene na listingu iznad implementiraju jednačine 3.1, 3.2 i 3.3. U metodi *calculate_fitness* se pored ažuriranja fitnesa čestice ažurira i najbolja pozicija i fitnes koje je čestica imala, tj. ažuriraju se atributi *self.best_position* i *self.best_fitness*.

4.1.4 Metode *delete_clusters* i *update_position*

```
    def delete_clusters(self):
        self.clusters = []

    def update_position(self):
        global w, c1, c2, global_best_position, w_damp
        r1 = c1*np.random.random((self.num_clusters, 1))
        r2 = c2*np.random.random((self.num_clusters, 1))
        self.velocity = w*self.velocity + \
                        r1*(self.best_position - self.position) +
                            \
                        r2*(global_best_position - self.position)
        self.position = self.position + self.velocity
        w *= w_damp
```

Listing 4: Listing za metode koje služe za brisanje kreiranih klastera i za ažuriranje trenutne brzine i pozicije čestice u jatu

U prvoj metodi na Listingu 4 brišu kreirani klasteri u metodi *update_clusters* opisanoj u poglavlju 4.1.2. Pošto obrada slika zahtjeva puno memorije, pamćenje svih kreiranih klastera bi mnogo trošilo resurse mašine na kojoj se algoritam pokreće. Pošto je klaster totalno definisan sa česticom koja ima najbolju poziciju i najbolji fitnes (tj. tu su sadržani podaci centroida svih klastera), veoma lagano možemo rekonstruisati klastere nakon izvršetka algoritma i kada se pronađe najbolja čestica.

Metoda *update_position* implementira jednačine 2.4 i 2. Nakon što je trenutna vrijednost pozicije čestice promjenjena, smanjujemo uticaj inercione komponente u jednačini 2.4 da bi smanjili "lutanje" čestice po slici.

4.1.5 *main()*

```
def main():
    slika = mpimg.imread("/home/besim/Documents/Fax/OR/projekat/
        boja.jpg")
    if slika.ndim < 3:
        slika = slika.reshape((slika.shape[0], slika.shape[1], 1)
            )
    print(slika.shape)
    global swarm_size, global_best_position, global_best_fitness,
        iterations, num_clusters
    particles = [Particle(num_clusters, im_shape=slika.shape) for
        _ in range(swarm_size)]
    particles[0].update_clusters(slika)
    particles[0].calculate_fitness(slika)
    global_best_fitness = particles[0].fitness
    global_best_position = particles[0].position

    for i in tqdm(range(iterations)):
        for particle in particles:
            particle.update_clusters(slika)
            particle.calculate_fitness(slika)
            particle.delete_clusters()
            if particle.fitness < global_best_fitness:
                global_best_fitness = particle.fitness
                global_best_position = particle.position

        for particle in particles:
            particle.update_position()
    particles[0].position=global_best_position
    particles[0].update_clusters(slika)
    for i in range(num_clusters):
        cluster = particles[0].clusters[i]
        np.save("/home/besim/Documents/Fax/OR/projekat/
            cluster_boja"+str(i), cluster)
        plt.figure()
        cluster = cluster.reshape((cluster.shape[0], cluster.
            shape[1], 1))
        cluster = cluster*slika
        plt.imshow(cluster)

    plt.show()
```

Listing 5: Listing koda u kojem se kreira jato i izvršava samo PSO algoritam

U *main()* funkciji se prvo učitava slika nad kojom ćemo vršiti obradu. Poslije toga, definišu se osnovne konstante koje su potrebne za pokretanje PSO algoritma kao što su veličina jata (*swarm_size*), globalna najbolja pozicija (*global_best_position*) i globalno najbolji fitnes (*global_best_fitnes*), maksimalni broj iteracija (jer je to uslov zaustavljanja našeg algoritma, *iterations*) i broj klastera na koji ćemo razdvojiti sliku, *num_clusters*. Nakon toga postavljamo početne vrijednosto globalno najboljeg fitnesa i pozicije i započinjemo algoritam.

Kreiramo jato sa zadanim brojem čestica i pokrećemo PSO algoritam. Unutar *for petlje* pozivamo prethodno objašnjenje funkcije *update_clusters(image)*, *calculate_fitness(image)*, *delete_clusters()* i ažuriramo globalno najbolji fitnes i poziciju. Nakon toga ažuriramo

poziciju čestice pozivom metoda *update_position()*.

Nakon što se algoritam završi, iz globalno najbolje pozicije rekonstruišemo klastere, spašavamo ih i prikazujemo rezultate.

5 Eksperimentalni rezltati

PSO bazirana klasterizacija slike je primjenjena na dvije slike, jednu crno bijelu sliku formata 128x128 bita i jednu sliku u boji istog formata. Slike su preuzete potpuno nasumično sa interneta. Osnovne konstante su inicijalizirane kao što je prikazano na Listingu

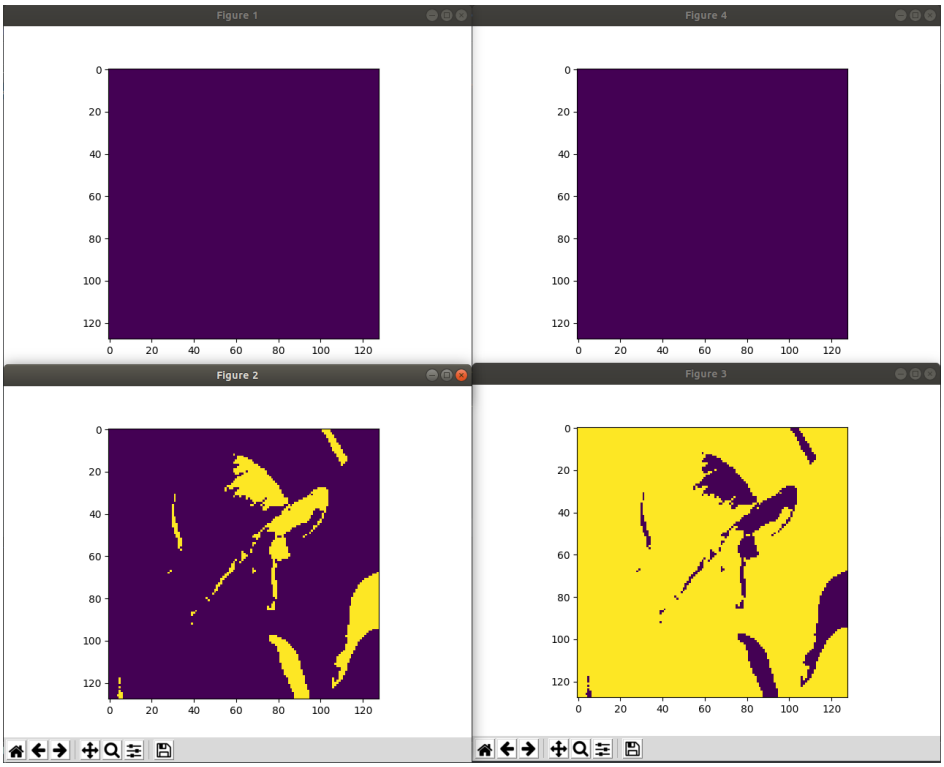
```
iterations = 100
swarm_size = 15
num_clusters = 4

w = 0.729844  # Inertia weight to prevent velocities becoming too
    large
c1 = 1.496180  # Scaling co-efficient on the social component
c2 = 1.496180  # Scaling co-efficient on the cognitive component
w_damp = 0.99   # Damping factor for velocity of particle

global_best_position = None
global_best_fitness = None

w1 = 0.5    # Fitness coefficient
w2 = 0.5    # Fitness coefficient
z_max = 255
```

Listing 6: Podaci korišteni za inicijalizaciju PSO algoritma

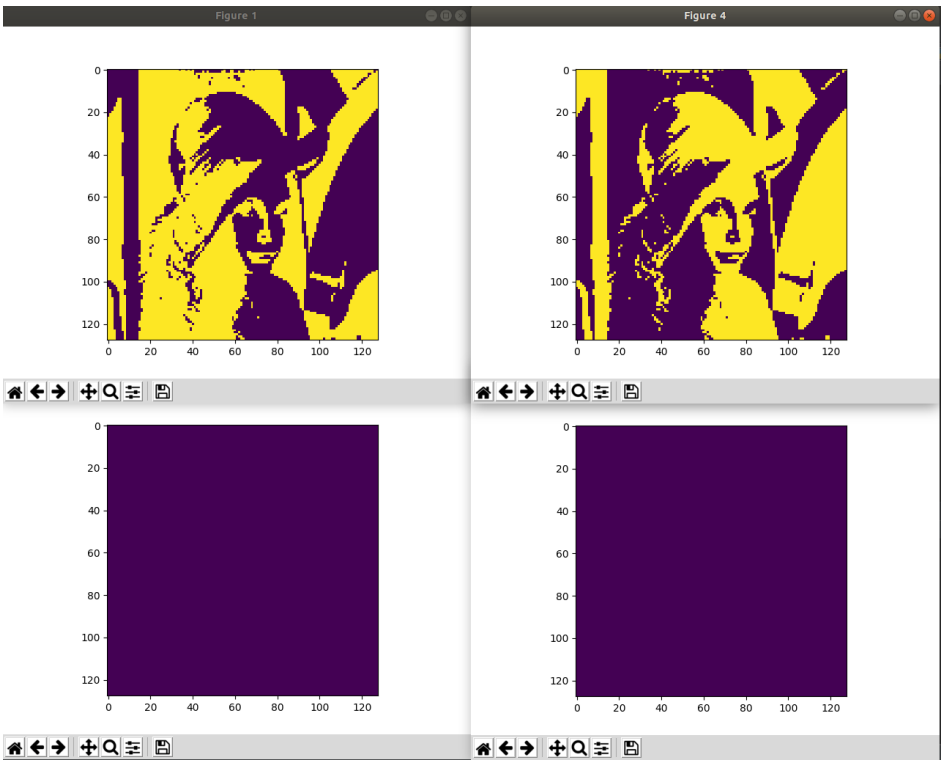
Na crno bijelu sliku, prikazanu na slici 5.1, smo primjenili algoritam dva puta, sa različitim brojem iteracija.

Slika 5.1: Crno-bijela testna slika formata 128x128.

Prvi put smo imali 25 iteracija dok smo pri drugom primjenjivanju imali 100 iteracija. Rezulati algoritma sa 25 iteracija su prikazani na slici , dok su rezultati pokretanja algoritma sa 100 iteracija prikazani na slici :



Slika 5.2: Rezultat primjenjivanja PSO algoritma na sliku 5.1 sa 25 iteracija.

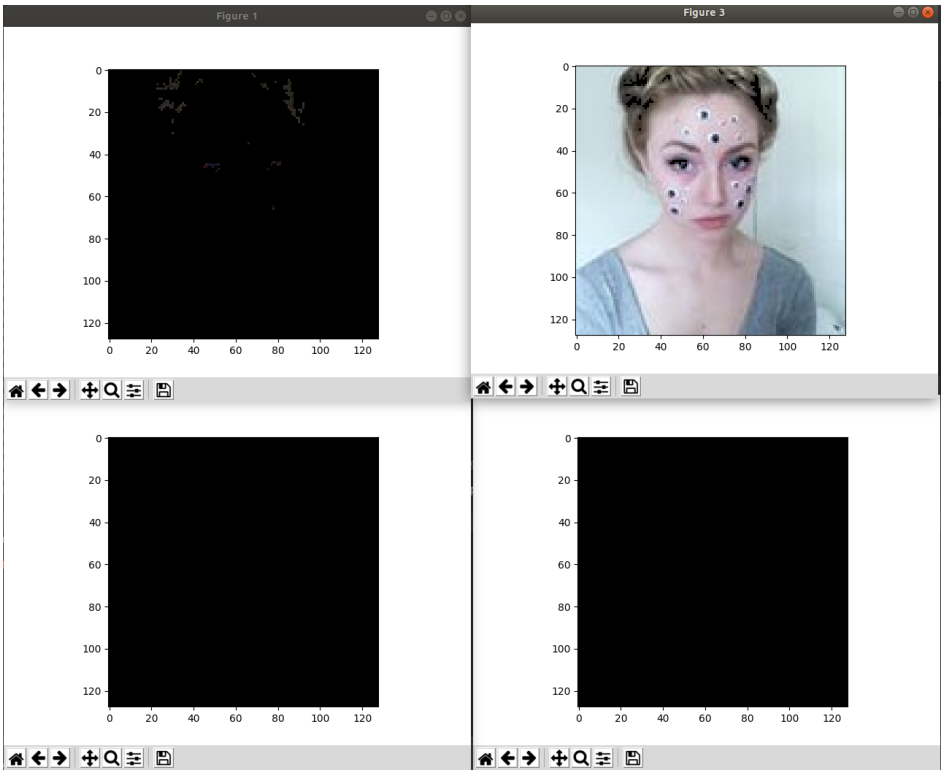


Slika 5.3: Rezultat primjenjivanja PSO algoritma na sliku 5.1 sa 100 iteracija.

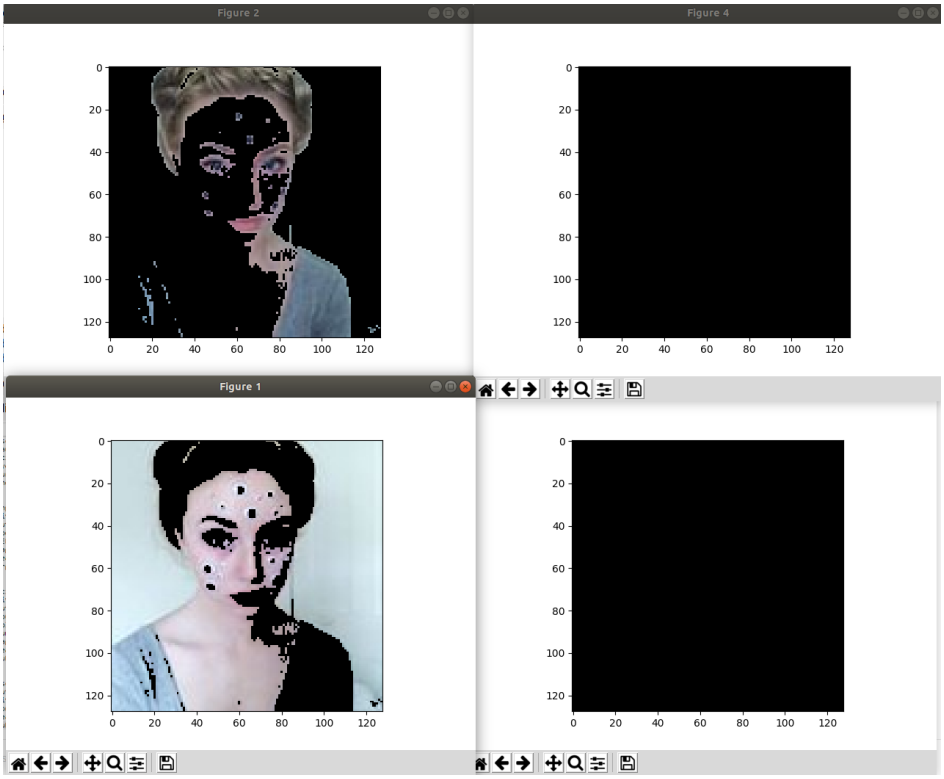
Vidimo da je algoritam pri prvom primjenjivanju sa manje iteracija uspio da nađe dva klastera na ovoj crnobijeloj slici, dok je pri drugom primjenjivanju sa 100 iteracija našao također samo dva klastera. Zbog veoma malog broja čestica u jatu (samo 15) i malog broja iteracija, algoritam pri prvom pokretanju je uspio da dostigne konvergenciju, ali nije uspio svim pikselima da dodijeli odgovarajuće klastere i zato ivice na slici 5.2 nisu veoma dobro izražene, jer je algoritam prekinuo uslov zaustavljanja (dostignut je maksimalan broj iteracija). Ali, samo sa 100 iteracija (što je ipak četiri puta više iteracija, ali u poređenju sa nekim drugim algoritmima koji zahtjevaju mnogo više iteracija da bi dostigli konvergenciju) naš algoritam je uspio da ovu sliku podijeli samo u dva klastera, što je bilo očekivano radi prirode slike. Na osnovu ovih rezultata zaključujemo da PSO algoritam zaista brzo konvergira i da uspješno pronalazi klastere i za 25 i za 100 iteracija, iako postoji vidno poboljšanje u performansi algoritma za povećan broj iteracija izvršavanja.
Isti eksperiment ćemo ponoviti za sliku u boji, prikazanoj na slici 5.4:

Slika 5.4: Slika u boji formata 128x128x3.

Pošto je rad sa slikama u boji mnogo zahtjevniji, jer imaju dodatnu dimenziju koja predstavlja podatke o RGB komponentama piksela, bilo je očekivano da za isti broj iteracija kao i za crnobijelu sliku, PSO algoritam neće dati ni približno dobre rezultate, što se može vidjeti na slikama 5.5 i 5.6:



Slika 5.5: Rezultat primjenjivanja PSO algoritma na sliku 5.4 sa 25 iteracija.



Slika 5.6: Rezultat primjenjivanja PSO algoritma na sliku 5.4 sa 100 iteracija.

Kao što se može vidjeti na slikama iznad, PSO algoritam nema toliko brzu konvergenciju kada se primjeni na ne crnobijele slike. Ovaj algoritam mnogo usporava razna množenja matrica koje vrši u računanju fitnesa, kao i stalno pristupanje pikselima slike kod računanja istog. Ovaj algoritam bi se mogao unaprijediti ako bi povećali broj

iteracija izvršavanja (što mi nismo uradili u ovom radu, radi poređenja performansi PSO algoritma na crnobijelim i obojenim slikama pri istom broju iteracija). Također, moguće je promjeniti implementaciju algoritma tako da se optimizuje i smanje broj množenja i pristupa slici koji se vrše pri računanju funkcije fitnesa.

6 Zaključak

Particle Swarm Optimization je jedan veoma jak algoritam optimizacije. U ovom radu je predstavljena njegova aplikacija na problem klasterizacije slika. Ipak, ovaj algoritam se može primjeniti na razne vrste problema optimizacije podataka jer zbog toga što je particionalni algoritam klasterizacije, može se primjenjivati kao optimizator.
Postoje razna poboljšanja ovog algoritma, kao što su GCPSO i DCPSO. U radu[6] je prezentovano koliko je PSO algoritam pokazuje bolje performanse upoređen sa trenutno aktuelnim algoritmima klasterizacije.

Iako je u ovom radu korištena funkcija u ovom PSO pristupu imala više funkcionalnosti, kao što je opisano u poglavlju 3.1, nije korišten niti jedan posebni optimizator sa više funkcionalnosti. Ovaj algoritam bi se najvjerovatnije znatno unaprijedio ako bi se razmatrao pristup sa više funkcionalnosti. Također, ovaj algoritam nije pokazao osobinu brze konvergencije kada je primjenjen na slike sa bojama, tako da je preporučljivo razviti strategiju u kojoj se dinamično određuje optimalan broj klastera u slici.

Literatura

[1] Samim Konjicija. Predavanja iz predmeta Optimizacija resursa, Decembar 2017.

[2] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm intelligence*, 1(1):33–57, 2007.

[3] James Kennedy, Russell C Eberhart, and Y Shi. Swarm intelligence, 2001. *Kaufmann, San Francisco*, 1:700–720, 2001.

[4] Frans van den Bergh and Andries P Engelbrecht. A new locally convergent particle swarm optimiser. In *Systems, Man and Cybernetics, 2002 IEEE International Conference on*, volume 3, pages 6–pp. IEEE, 2002.

[5] Frans Van Den Bergh. *An analysis of particle swarm optimizers*. PhD thesis, University of Pretoria, 2007.

[6] M Omran, Andries Petrus Engelbrecht, and A Salman. Particle swarm optimization method for image clustering. *International Journal of Pattern Recognition and Artificial Intelligence*, 19(03):297–321, 2005.