در پیاده سازی مقاله تابع KeyClustering یک آرایه ورودی از نوع ndarray و تعداد خوشه ها را میگیرد.
در این پیاده سازی تعداد همسایه ها برابر 1 (k = 1) و g برابر 2 در نظر گرفته شده است.

```
self.C_target = C_target
(self.N = len(data
self.k = 1
self.g = 2
```

در ابتدا D_orginal یک آرایه دو بعدی به ابعاد N*N که فاصله هر عنصر در ورودی را با عناصر دیگر محاسبه میکند.

```
((self.D_orginal = np.zeros((self.N, self.N
```

```
(self.init_D_orginal(data
```

```
:(def init_D_orginal(self, data
:(for i in range(self.N
:(for j in range(self.N
([self.D_orginal[i, j] = self.dist(data[i], data[j
```

سپس آرایه R مقدار دهی میشود. این آرایه شامل هر عنصر و k همسایه آن است. پیدا کردن همسایه های نزدیک از روی ماتریس D_orginal است به طوری که خود عنصر و k عضوی که کمترین فاصله با عنصر را دارد به عنوان همسایه آن درنظر گرفته میشوند.

```
(self.R = np.zeros((self.N, self.k+1), dtype=np.uint
```

```
()self.init_R
```

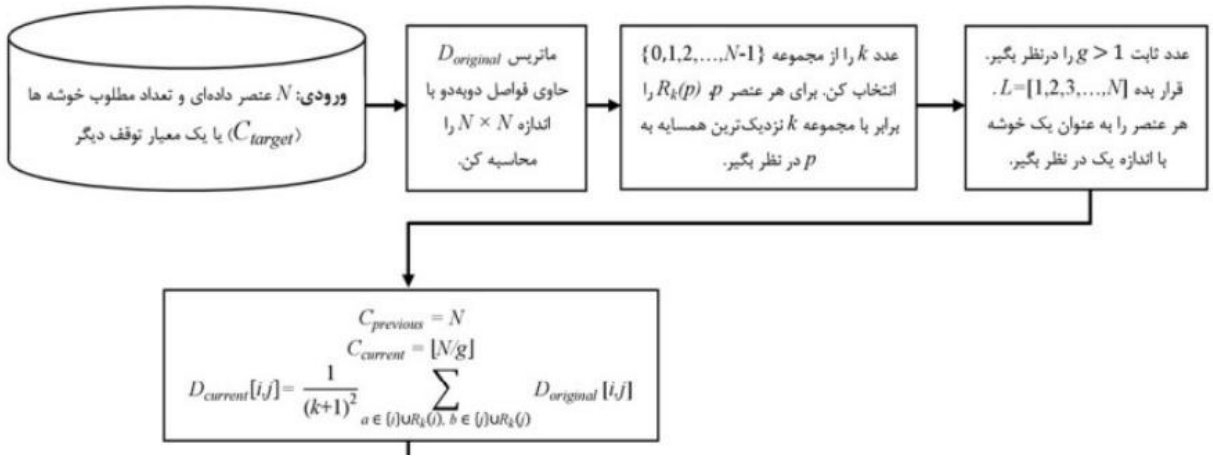
```
:(def init_R(self
()D_copy = self.D_orginal.copy
max_val = self.D_orginal.max() + 1
:(for i in range(self.N
self.R[i, 0] = i
D_copy[i, i] = max_val
:(for j in range(1, self.k+1
([self.R[i, j] = np.argmin(D_copy[i
D_copy[i, self.R[i, j]] = max_val
```

در D_current با سایز اولیه N*N که N سایز دیتای ورودی است، بجای اینکه فاصله هر نقطه تک به تک با بقیه نقاط محاسبه شود، فاصله هر نقطه و همسایه هایش با سایر نقاط و همسایه هایشان محاسبه میشود. که باز هم محاسبه آن از روی D_orginal به سادگی قابل انجام است.
تا به اینجا که مربوط به تابع init بود، شامل قسمت زیر از مقاله میشود.

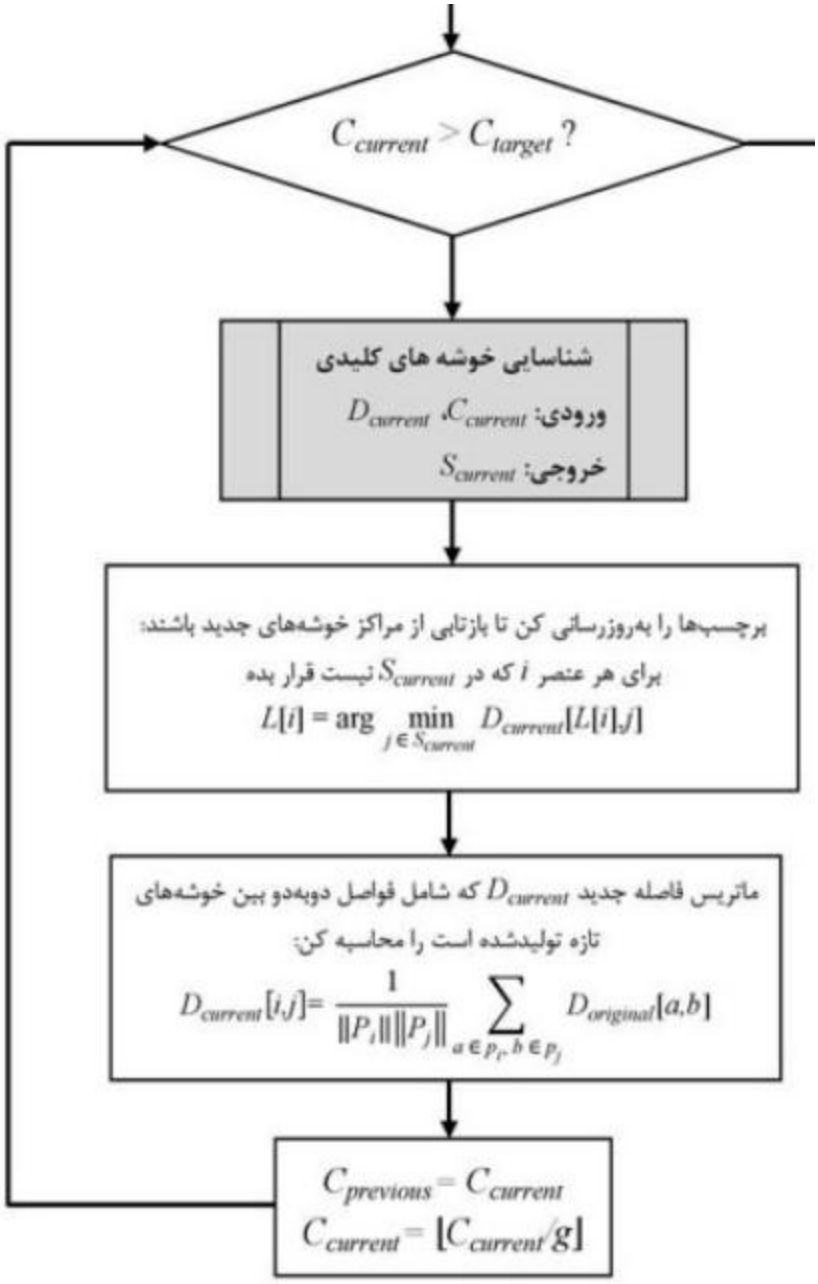
```
((self.D_current = np.zeros((self.N, self.N
```

```
()self.init_D_current
```

```
:(def init_D_current(self
const = 1 / (self.k+1)**2
:(for i in range(self.N
:(for j in range(self.N
sum = 0
:[for a in self.R[i
:[for b in self.R[j
:if i != j
[sum += self.D_orginal[a, b
self.D_current[i, j] = const * sum
```



با صدا زدن تابع fit مراحل اصلی خوشه بندی آغاز میشود که شامل قسمت زیر در مقاله میشود.



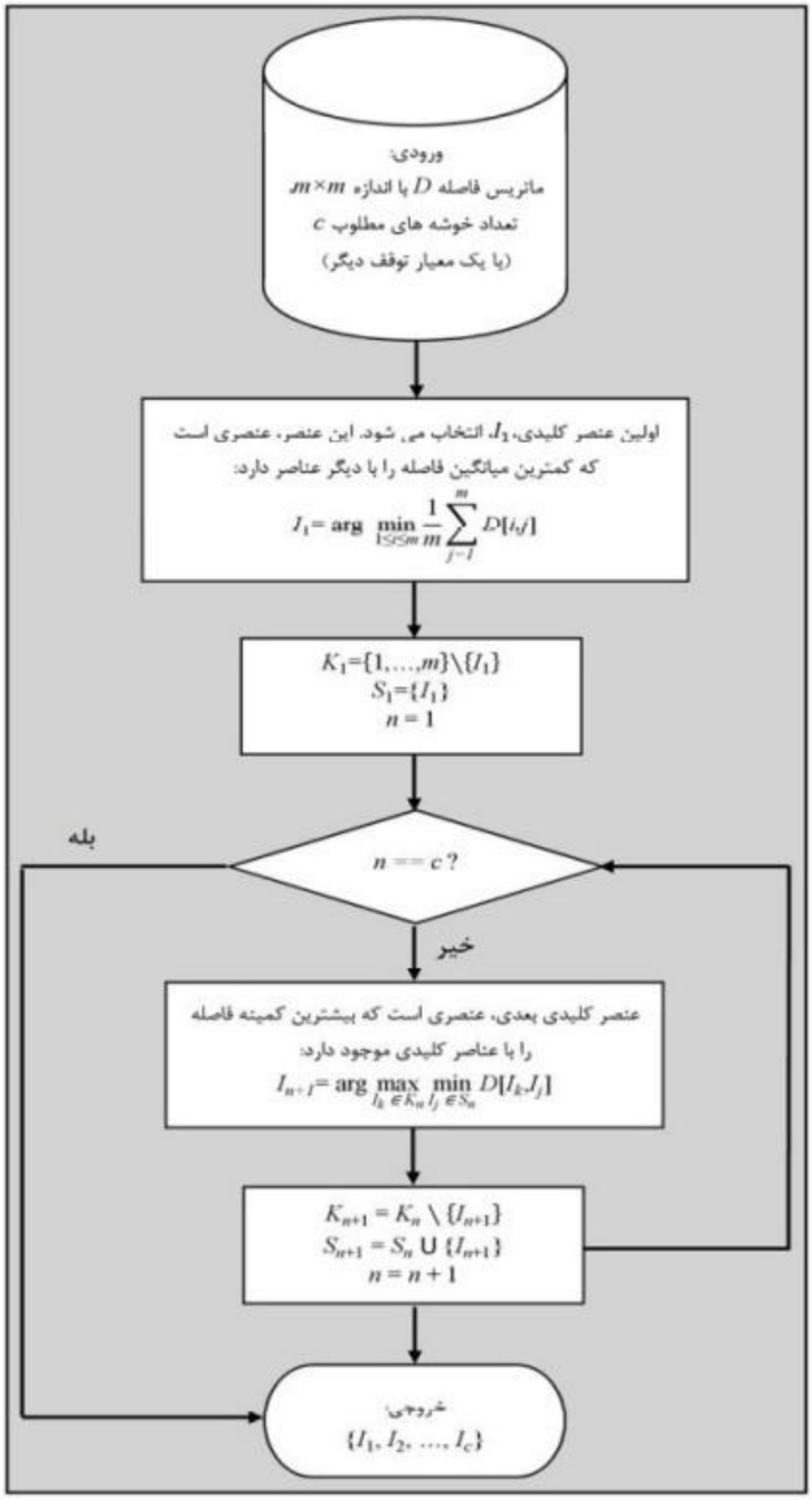

```
:while self.C_current > self.C_target
self.S_current, self.K_current =
(self.find_key_clusters(self.C_current

()self.update_labels
```

```
()self.D_current = self.update_new_distances


()self.update_C
```

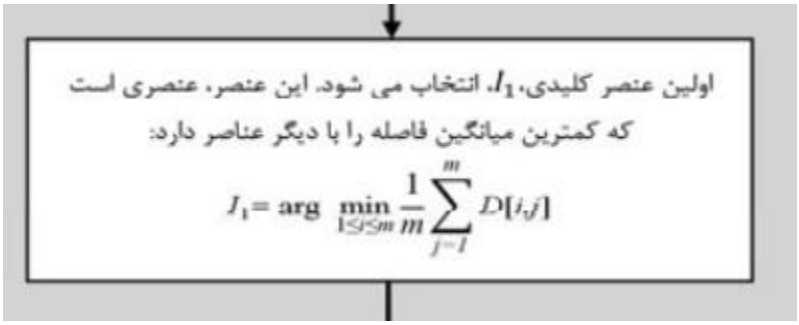
در ابتدا باید خوشه های کلیدی محاسبه شوند، خود این شناسایی عناصر کلیدی خودش دارای چندین پیمایش است، که شامل قسمت زیر از مقاله است:


ورودی:
ماتریس فاصله D با اندازه $m \times m$
تعداد خوشه های مطلوب c
(یا یک معیار توقف دیگر)
اولین عنصر کلیدی، I_1، انتخاب می شود. این عنصر، عنصری است
که کمترین میانگین فاصله را با دیگر عناصر دارد:
$I_1 = \arg \min_{1 \le i \le m} \frac{1}{m} \sum_{j=1}^{m} D[i,j]$
$K_1 = \{1,\ldots,m\} \setminus \{I_1\}$
$S_1 = \{I_1\}$
$n = 1$
بله
$n == c$?
خیر
عنصر کلیدی بعدی، عنصری است که بیشترین کمینه فاصله
را با عناصر کلیدی موجود دارد:
$I_{n+1} = \arg \max_{I_k \in K_n} \min_{I_j \in S_n} D[I_k, I_j]$
$K_{n+1} = K_n \setminus \{I_{n+1}\}$
$S_{n+1} = S_n \cup \{I_{n+1}\}$
$n = n + 1$
خروجی:
$\{I_1, I_2, \ldots, I_c\}$

برای شناسایی خوشه های کلیدی یک شرط توقف لازم است، که آن شرط تعداد خوشه های موردنظر در آن پیمایش است که برابر C_current میباشد.
ماتریس distance را میسازیم تا مجموع فاصله هر نقطه تا نقاط دیگر را داشته باشیم:

```
distance = np.array([np.sum(self.D_current[i]) for i in
([(range(self.C_current
```

برای پیدا کردن اولین خوشه کافی است اندیس عنصری با کمترین مقدار در distance را محاسبه کنیم




```
(I1 = np.argmin(distance

[] = K_curr
:((for i in range(len(self.D_current
:if i != I1
(K_curr.append(i
(K_curr = np.array(K_curr

[S_curr = [I1
```

به S_curr عنصری که انتخاب شد (I1) و به K_curr بقیه عناصر را اضافه میکنیم. سپس تا زمانی که تعداد خوشه ها برابر C_current نشد بیشترین فاصله بین عناصر S_curr و K_curr را محاسبه و به S_curr اضافه میکنیم.
در این پیاده سازی علاوه بر S_curr در خروجی K_curr هم برگردانده شده است.

```
n = 1
:while n != C_stop

[] = next_key_distances
:for item_k in K_curr
[] = tmp
```

```
:for item_S in S_curr
([tmp.append(self.D_current[item_k, item_S

((next_key_distances.append(np.min(tmp

(I_n = np.argmax(next_key_distances

([S_curr.append(K_curr[I_n
(K_curr = np.delete(K_curr, I_n


n += 1
```

K_current = K_curr
S_current = S_curr

سپس نوبت به آپدیت برچسب ها است. سپس برای هر آیتمی که در K_current است نزدیک ترین عنصر را از S_current انتخاب کرده و برچسب آنرا روی آیتم میگذاریم.

```
:for item_K in self.K_current
[] = tmp
:for item_S in self.S_current
([tmp.append(self.D_current[item_K, item_S

(arg_min = np.argmin(tmp

[self.L[np.where(self.L==item_K)] = self.S_current[arg_min
```

چون ممکن است label ها به ترتیب از 0 به بالا نباشند، به هرکدام یک عدد از 0 تا C_current نسبت میدهیم و label ها را به رنج 0 و C_current تغییر میدهیم

```
{} = new_label
:(for i in range(self.C_current
new_label[self.S_current[i]] = i

:((for i in range(len(self.L
[[self.L[i] = new_label[self.L[i
```

سپس فاصله های جدید که فاصله هر خوشه با خوشه های دیگر است محاسبه میشود. فاصله خوشه ها از هم بر اساس عنصر های هر خوشه و همسایه عناصر هر خوشه محاسبه میشود.

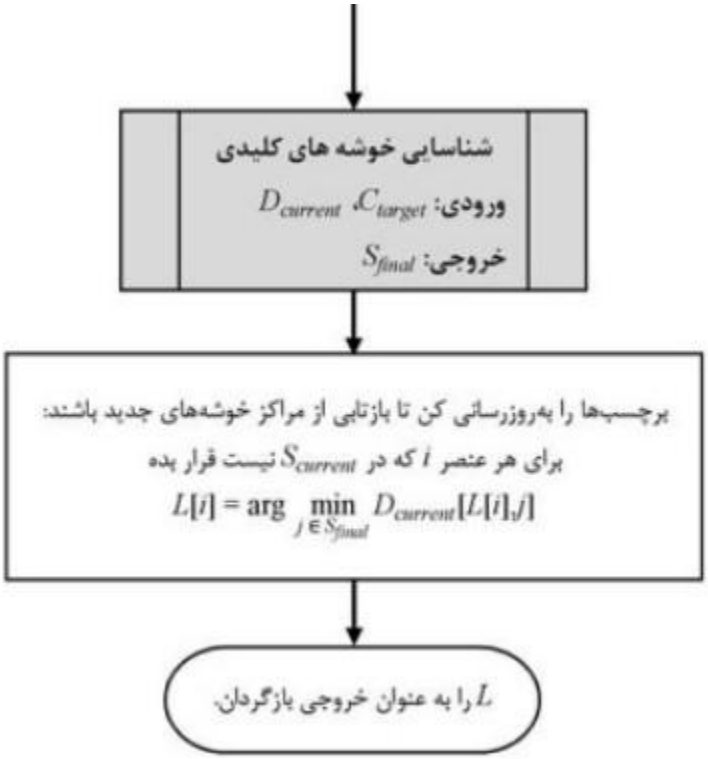
```
:(for i in range(self.C_current
([c_1_len = len(clusters[i
:(for j in range(self.C_current
:if i == j
continue
([c_2_len = len(clusters[j

sum = 0
:[for c1 in clusters[i
:[for c2 in clusters[j
[sum += self.D_orginal[c1, c2
D_curr[i, j] = 1 / (c_1_len * c_2_len) * sum
```

بعد از آن `C_current` بر g تقسیم میشود.

```
:(def update_C(self
self.C_previous = self.C_current
self.C_current = self.C_current // self.g
```

زمانی که C_current برابر تعداد خوشه های مشخص شده شد، یکبار دیگر شناسایی خوشه های کلیدی با C_target خوشه و آپدیت برچسب ها را انجام میدهیم.



```
self.C_current = self.C_target
self.S_current, self.K_current =
(self.find_key_clusters(self.C_current

()self.update_labels
```