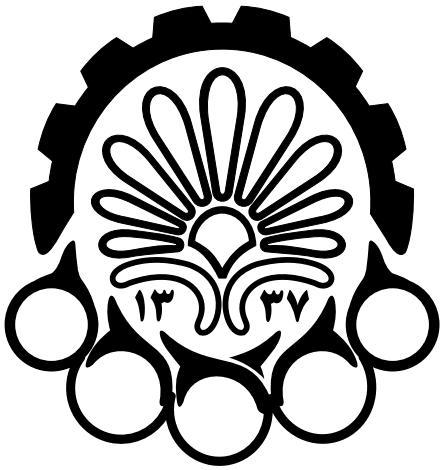


دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر

تمر ین چهارم درس یادگیری ماشین

دکتر ناظرفرد

غلامرضا دار ۴۰۰۱۳۱۰۱۸

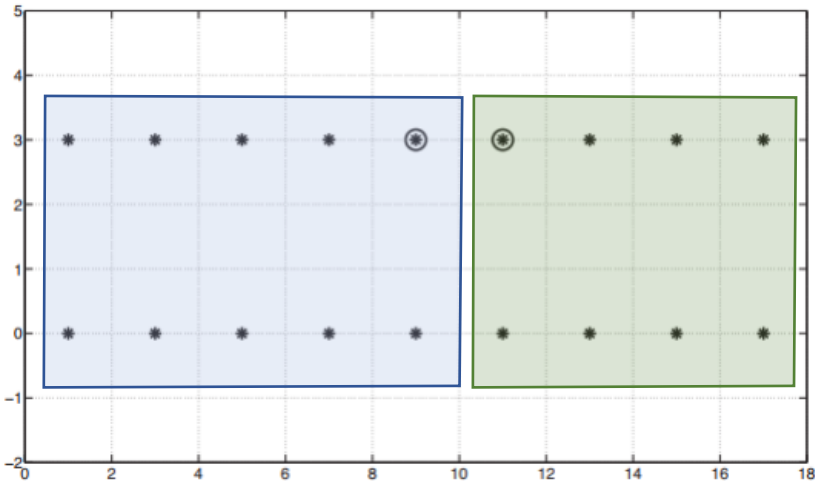
زمستان ۱۴۰۰

فهرست مطالب

بخش اول: پرسشهای تشریحی

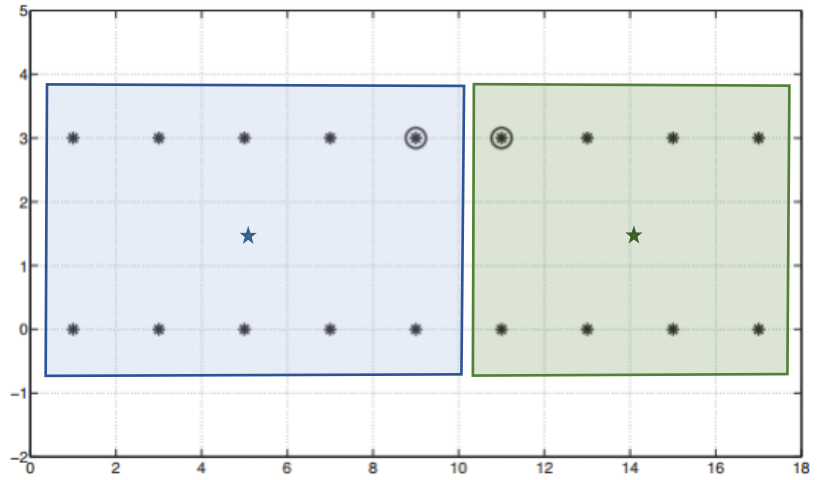
سوال ۱)

در مرحله اول نقاطی که به هر کلاستر تعلق دارند را مشخص میکنیم، سپس برای محاسبه مرکز جدید هر کلاستر بین نقاط هر کلاستر میانگین میگیریم. خواهیم دید پس از ۱ مرحله دیگر مرکز ها تغییر نمیکنند و الگوریتم متوقف میشود.



$$\mu_1 = \frac{\left(\binom{1}{0} + \binom{3}{0} + \binom{5}{0} + \binom{7}{0} + \binom{9}{0} + \binom{1}{3} + \binom{3}{3} + \binom{5}{3} + \binom{7}{3} + \binom{9}{3}\right)}{10} = \binom{5}{1.5}$$

$$\mu_2 = \frac{\left(\binom{11}{0} + \binom{13}{0} + \binom{15}{0} + \binom{17}{0} + \binom{11}{3} + \binom{13}{3} + \binom{15}{3} + \binom{17}{3}\right)}{8} = \binom{14}{1.5}$$



سوال ۲)

Customer Segmentation: با انجام خوشه بندی بر اساس ویژگی های مختلفی مانند تاریخچه خرید مشتری، سن، جنسیت و ... میتوان مشتری ها را در دسته های شبیه به هم خوشه بندی کرد. با این کار تعدادی خوشه داریم که افراد با ویژگی های مشابه در یک خوشه قرار دارند. به عنوان مثال یک فروشگاه زنجیره ای را در نظر بگیرید. این فروشگاه اگر بتواند خانم های باردار را شناسایی کند و برای آنها پیام تبلیغاتی در مورد وسایل نوزاد بفرستد شانس زیادی برای به تعامل گرفتن مشتری دارد. همچنین اگر در آینده یک مشتری جدید وارد سیستم شود و تعدادی خرید انجام دهد، با استفاده از خوشه های موجود و میزان شباهت مشتری جدید با آن خوشه ها میتوان مشتری جدید را نیز به آسانی شناسایی کرد و خدمات مخصوص به آن نوع مشتری را به او داد.

Dimensionality Reduction: یکی از تکنیک هایی که برای کاهش بعد با استفاده از خوشه بندی وجود دارد به این شکل است که ابتدا تعداد خوشه ها را برابر با تعداد بعد مقصد در نظر میگیریم، سپس عمل خوشه بندی را انجام میدهیم. در نهایت فاصله نقطه ها از مراکز مختلف خوشه ها فیچر های جدید هستند که تعداد این فیچر ها قاعدتا به تعداد مراکز خوشه ها و در نتیجه به تعداد خوشه هاست.

Anomaly Detection: یکی از کاربرد های مهم خوشه بندی شناسایی داده پرت و غیرعادی است. برای شناسایی داده نویز یا غیر عادی کافیست فاصله هر داده را تا مرکز خوشه ای که به آن تعلق دارد حساب کنیم. قاعدتا داده های نویز و غیر عادی نسبت به داده های عادی فاصله بیشتری تا مرکز خوشه ها دارند چون داده های غیرعادی به طور کلی از خود داده های عادی فاصله زیادی دارند و مراکز خوشه ها از میانگین فاصله همه نقاط بدست می آیند.

Image Segmentation: یکی دیگر از کاربردهای خوشه بندی Image segmentation است. فرض کنید میخواهید در یک تصویر هوایی از منطقه کشاورزی، زمین های زراعی و زمین های خشک یا رودخانه و ... را از هم جدا کنید. یک راه این است که پیکسل هایی با شباهت رنگی زیاد را در یک دسته قرار دهید و با این کار احتمالا پیکسل های متناظر با منطقه زراعی(سبز) و رودخانه(آبی) به خوبی از هم جدا میشوند و در دسته های مختلف قرار میگیرند. این دسته بندی به راحتی با روش های خوشه بندی مانند Kmeans انجام میپذیرد. بهتر است در همچین مسائلی مقدار k را به اندازه تعداد سوژه های مختلف (رودخانه، کشتزار و ...) یا بیشتر از آن قرار دهید.

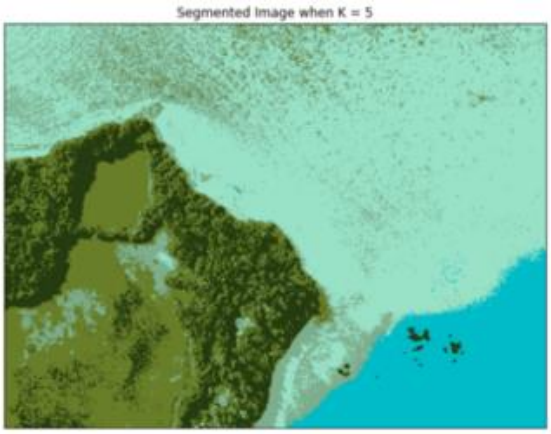


Image from https://www.kdnuggets.com/2019/08/introduction-image-segmentation-k-means-clustering.html

سوال ۳)

پارامتر های DBSCAN (min_pts, ϵ) در واقع ابرپارامتر هستند و نیاز به یافتن مقادیر مناسبی برای آنها وجود دارد.

به طور کلی بهتر است هر چه تعداد داده ها زیاد تر میشود یا تعداد داده های تکراری افزایش می‌یابد، مقدار min_pts را افزایش دهیم. به عنوان یک نقطه شروع مقدار **2*dim** پیشنهاد میشود.

یکی از تکنیک هایی که برای پیدا کردن مقدار مناسب ϵ پیشنهاد میشود این است که فاصله میانگین هر نقطه تا k نزدیکترین همسایه آن را محاسبه کنیم. اگر نمودار میانگین فاصله را رسم کنیم(محور افقی شماره نقطه، محور عمودی میانگین فاصله آن نقطه تا k نزدیک ترین همسایه اش) یک نمودار صعودی بدست می‌آید. میتوانیم مشابه روش elbow شکستگی نمودار را پیدا کنیم و فاصله میانگین متناظر با آن نقطه را به عنوان مقدار بهینه ϵ استفاده کنیم. بهتر است این کار را به ازای مقادیر مختلف k آزمایش کنیم.

اما در هر صورت روش قطعی ای برای پیدا کردن مقدار بهینه برای این پارامتر ها وجود ندارد و برای هر مسئله میتواند متفاوت عمل کند.

سوال ٤)

الف) معیار Single Linkage به این صورت عمل میکند که در هر مرحله برای ترکیب دو کلاستر به دو عنصر در دو کلاستر نگاه میکند که فاصله **کمینه** ای داشته باشند. هر دو کلاستری که در این معیار **کوچکتر** باشند با هم ترکیب میشوند. Complete Linkage به این صورت عمل میکند که در هر مرحله برای ترکیب دو کلاستر به دو عنصر در دو کلاستر نگاه میکند که فاصله **بیشینه** ای داشته باشند. هر دو کلاستری که در این معیار **کوچکتر** باشند با هم ترکیب میشوند. Average Linkage نیز همانطور که از اسمش پیداست به جای در نظر گرفتن فاصله کمینه یا بیشینه، فاصله میانگین را مقایسه میکند. آن دو کلاستری که فاصله **میانگین** عناصر آنها **کوچکتر** باشد با هم ترکیب میشوند.

پیچیدگی زمانی معیار Single Linkage = $\boldsymbol{O(n^2)}$

پیچیدگی زمانی معیار Complete Linkage = $\boldsymbol{O(n^2 logn)}$

پیچیدگی زمانی معیار Average Linkage = $\boldsymbol{O(n^2 logn)}$

از لحاظ مقاومت در برابر نویز، معیار های Single Linkage و Complete Linkage چون در نهایت به یک داده وابسته هستند و تمام تصمیمشان را بر اساس یک داده(نزدیکترین یا دورترین) میگیرند در مقابل نویز اصلا مقاوم نیستند و حتی یک داده نویزی میتواند نتیجه آنها را به کلی عوض کند. اما معیار Average Linkage چون میانگین میگیرد در مقابل نویز مقاوم تر است و اثر نویز را با میانگین گیری از بین میبرد.

ب)

Single Linkage: **C1= {1,2} C2= {3,4}**

Complete Linkage: **C1= {1,3} C2= {2,4}**

Average Linkage: **C1= {1,2} C2= {3,4}**

ج)

شکل b) **Single Linkage**

شکل c) **هیچکدام**

سوال ۵)

الگوریتم Policy Iteration پیچیده تر(دارای دو مرحله پی در پی) ولی سریعتر از الگوریتم Value Iteration است و زودتر همگرا میشود. یکی از مشکلاتی که الگوریتم Value Iteration دارد این است که در طی اجرای این الگوریتم از یک جایی به بعد Policy حاصل ثابت میشود اما Value ها همچنان در حال همگرایی هستند به این دلیل که تغییر های بسیار کوچک در Value ها باعث تغییر Policy بهینه نمیشود.

در الگوریتم Policy Iteration ما با یک Policy رندوم شروع میکنیم و در هر مرحله دو عمل Policy Evaluation و Policy Improvement را تکرار میکنیم.

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s')$$

اما الگوریتم Value Iteration که در هر مرحله فقط یک کار انجام میدهد به این شکل عمل میکند که در ابتدا Value ها را برابر صفر در نظر میگیرد و هر بار بر اساس رابطه Bellman این مقادیر آپدیت میشوند.

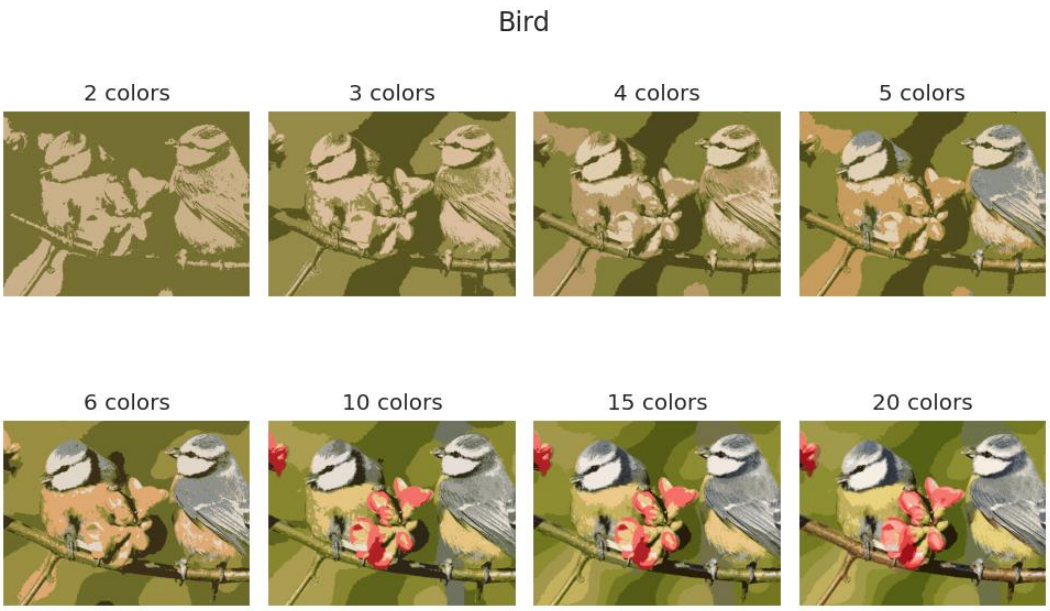
$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U_i(s')$$

بخش دوم: پیاده سازی

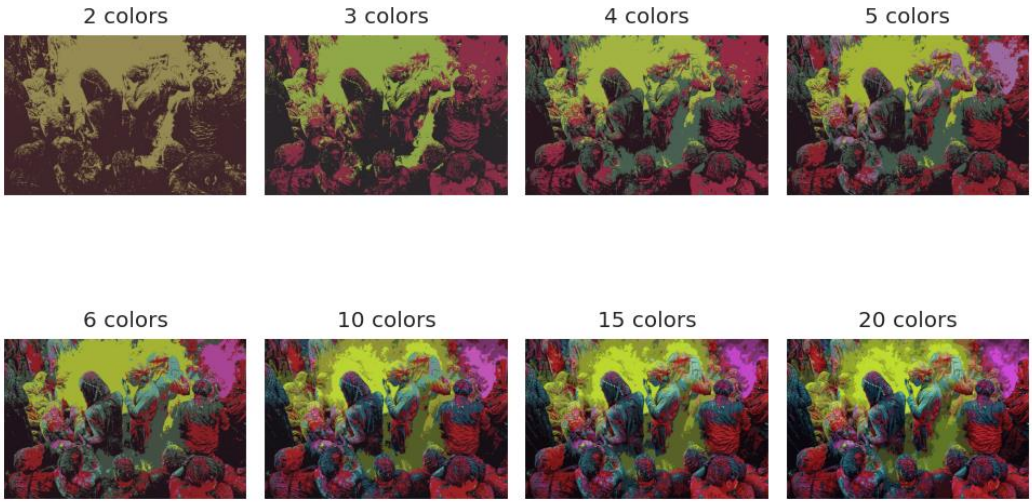
سوال ۱)

نوت بوک مربوطه https://colab.research.google.com/drive/1CCtc6MqozuSDKs71baKaLs_M_BeH-IDF?usp=sharing

قسمت الف و ب سوال را همزمان انجام میدهیم.

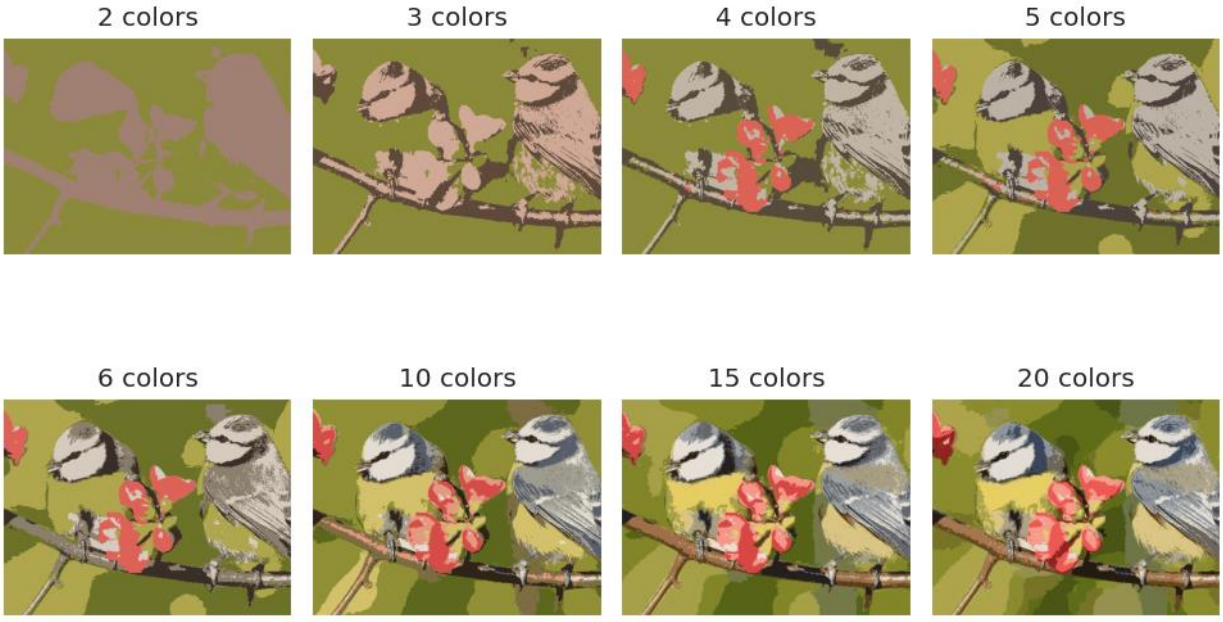


Holi



همچنین چون در این کاربرد فاصله رنگ ها مطرح است بهتر است از Color Space های دیگری مانند lab استفاده کنیم.(به سینه پرنده نگاه کنید زرد ≠سفید)

Bird



Car

2 colors

3 colors

4 colors

5 colors

6 colors

10 colors

15 colors

20 colors

Holi

2 colors

3 colors

4 colors

5 colors

6 colors

10 colors

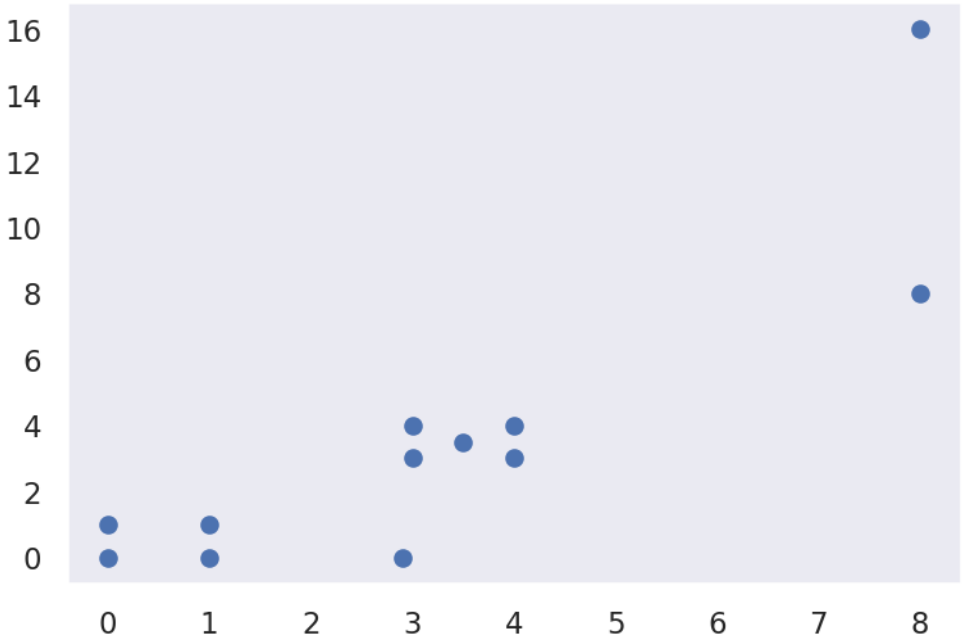
15 colors

20 colors

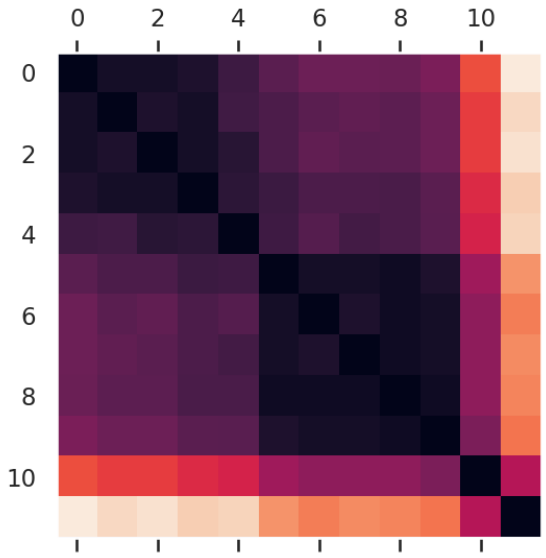
سوال ۲)

نوت بوک مربوطه https://colab.research.google.com/drive/1k71Vfr-UZ4mBWcBNXuLoUOZ6x2fxPCWv?usp=sharing

برای اطمینان از صحت پیاده سازی DBSCAN خود آن را بر روی یک Toy Example آزمایش میکنیم.



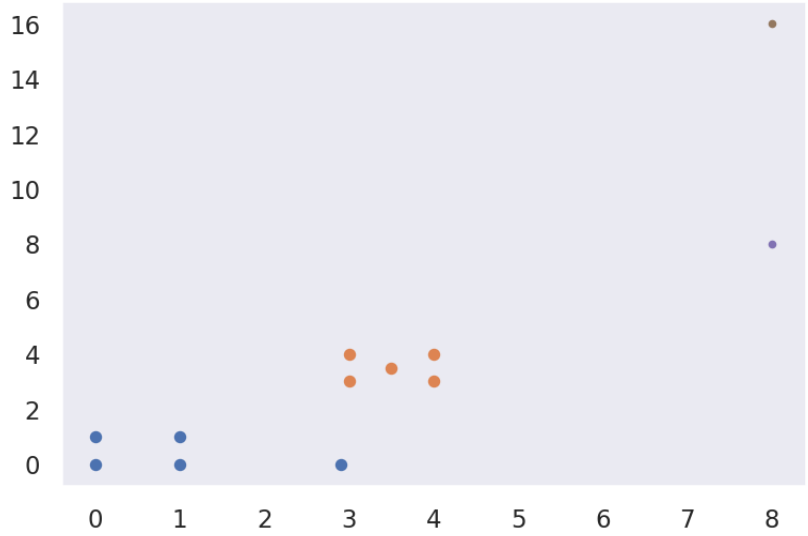
برای آسان شدن محاسبات آینده Distance Matrix بین نقاط را تولید میکنیم. (پیکسل های روشن تر نشان دهنده فاصله بیشتر بین نقاط هستند)



الگوریتم را بر روی داده ها اجرا میکنیم (هر عدد نشان دهنده آیدی کلاستر آن نقطه است).

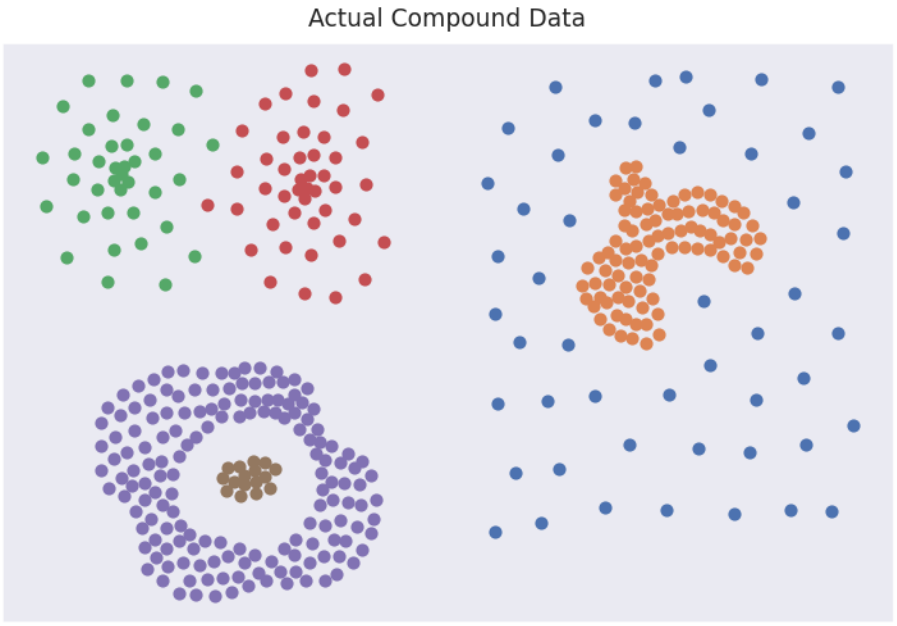
```
------------------------------------------------
0)
[ 0.  0.  0.  0.  0. -1. -1. -1. -1. -1. -1. -1.]

------------------------------------------------
1)
1 is already in a cluster
[ 0.  0.  0.  0.  0. -1. -1. -1. -1. -1. -1. -1.]

------------------------------------------------
2)
2 is already in a cluster
[ 0.  0.  0.  0.  0. -1. -1. -1. -1. -1. -1. -1.]

------------------------------------------------
3)
3 is already in a cluster
[ 0.  0.  0.  0.  0. -1. -1. -1. -1. -1. -1. -1.]

------------------------------------------------
4)
4 is already in a cluster
[ 0.  0.  0.  0.  0. -1. -1. -1. -1. -1. -1. -1.]

------------------------------------------------
5)
[ 0.  0.  0.  0.  0.  1.  1.  1.  1.  1. -1. -1.]

------------------------------------------------
6)
6 is already in a cluster
[ 0.  0.  0.  0.  0.  1.  1.  1.  1.  1. -1. -1.]

------------------------------------------------
7)
7 is already in a cluster
[ 0.  0.  0.  0.  0.  1.  1.  1.  1.  1. -1. -1.]

------------------------------------------------
```

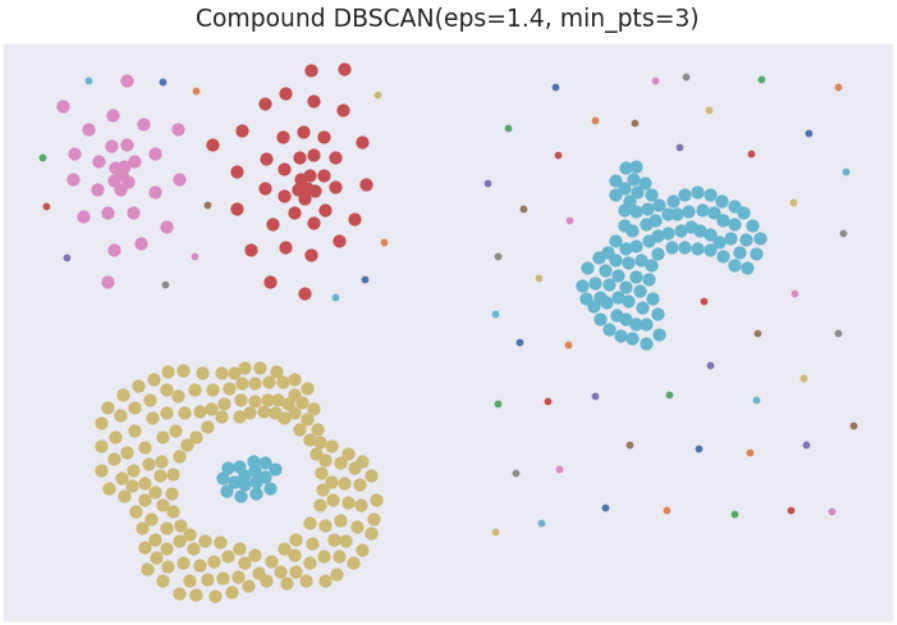
و نتیجه کلاسترینگ :



در این بخش الگوریتم DBSCAN مان را بر روی دیتاست های مورد نظر سوال اجرا میکنیم. در این مرحله برای هر دیتاست با آزمون و خطا مقدار مناسبی برای پارامترهای ϵ و min_pts می‌یابیم.

Compound:



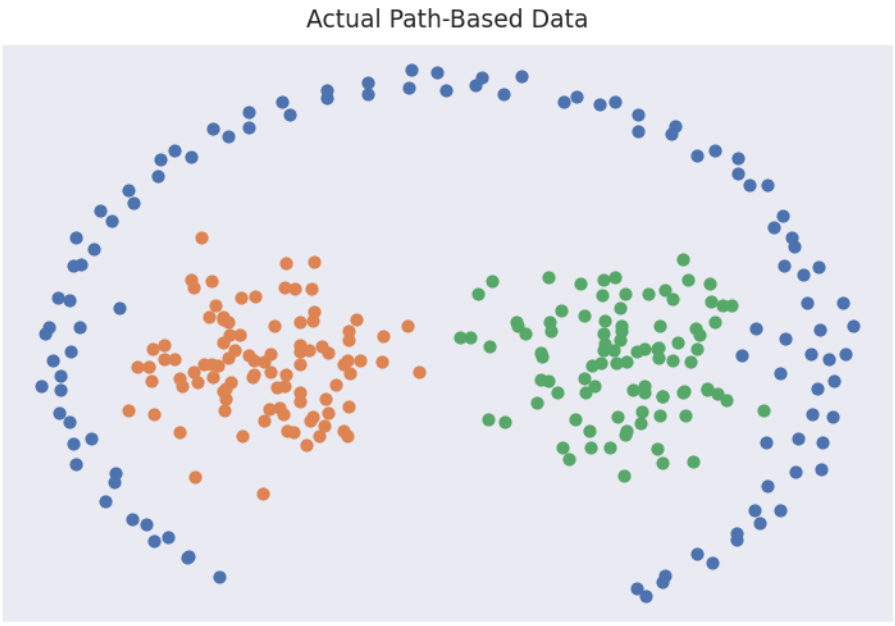


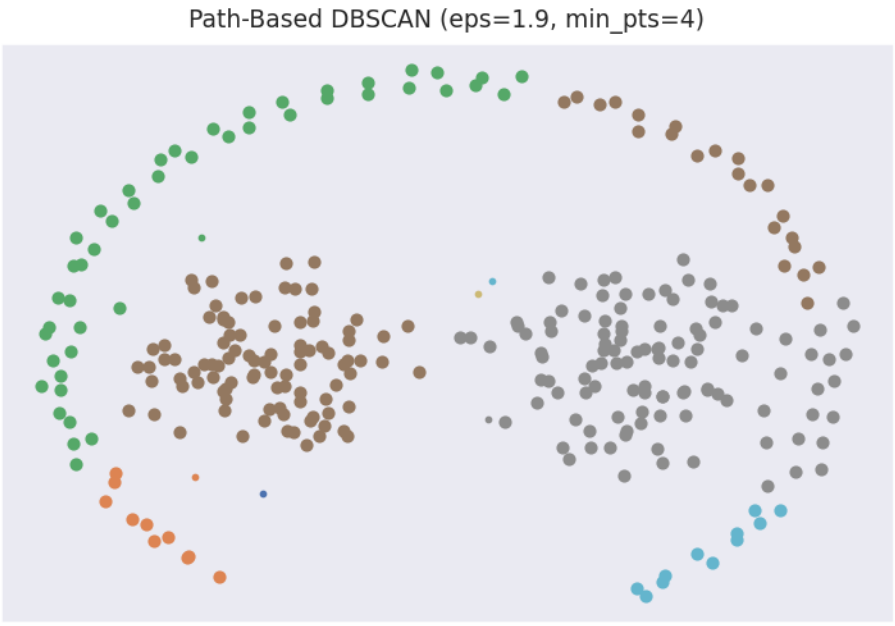
Of Clusters: 5

Of Noise: 62

Purity: 0.965

Path-Based:



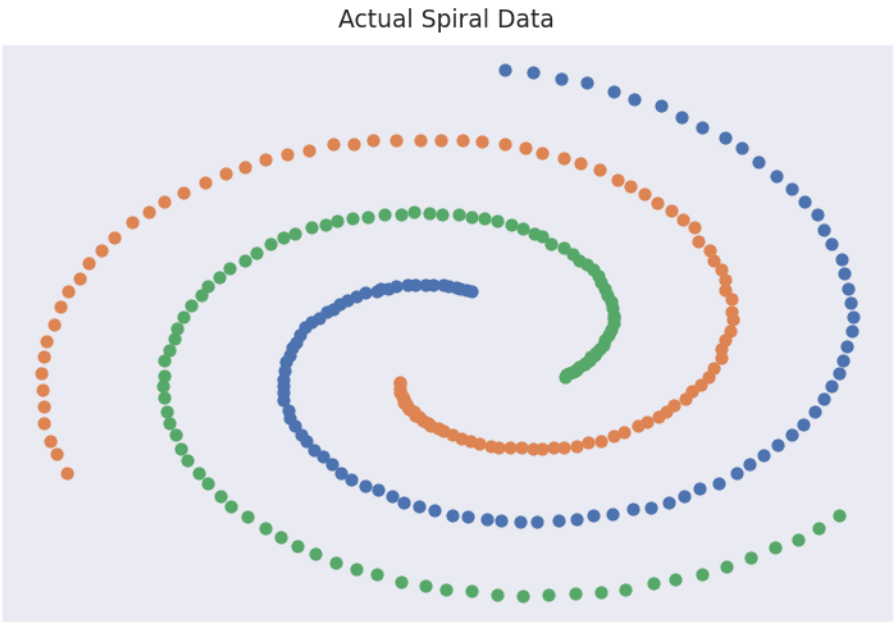


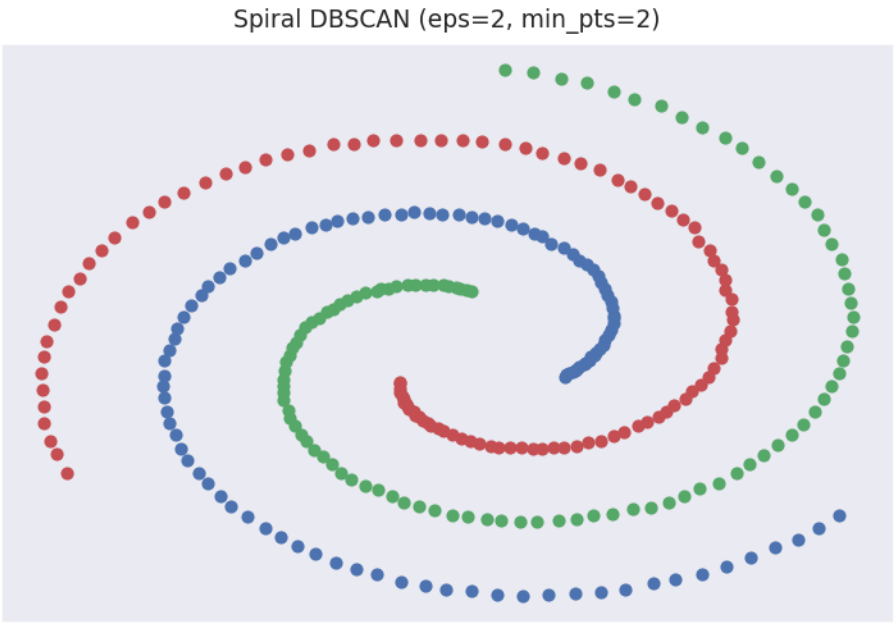
Of Clusters: 6

Of Noise: 6

Purity: 0.769

Spiral:



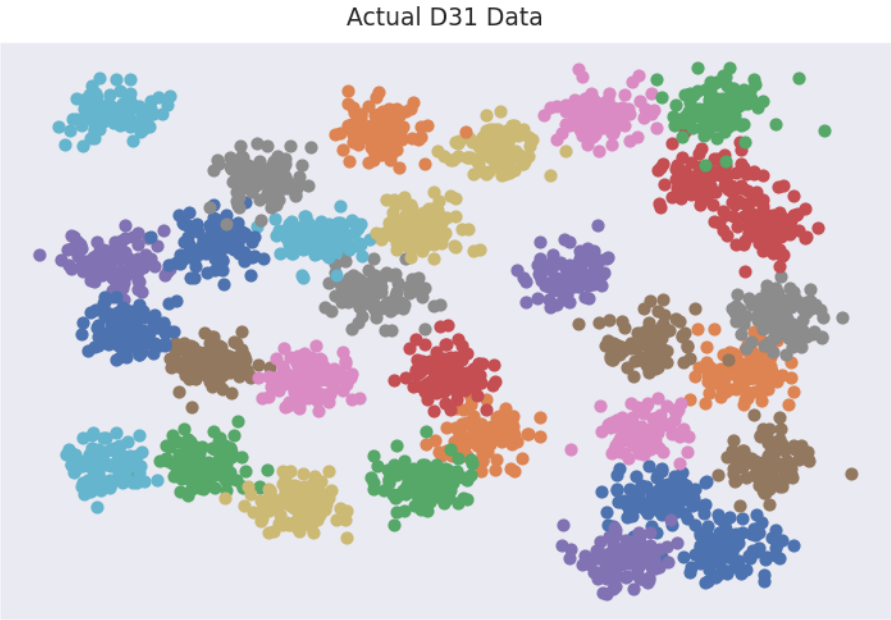


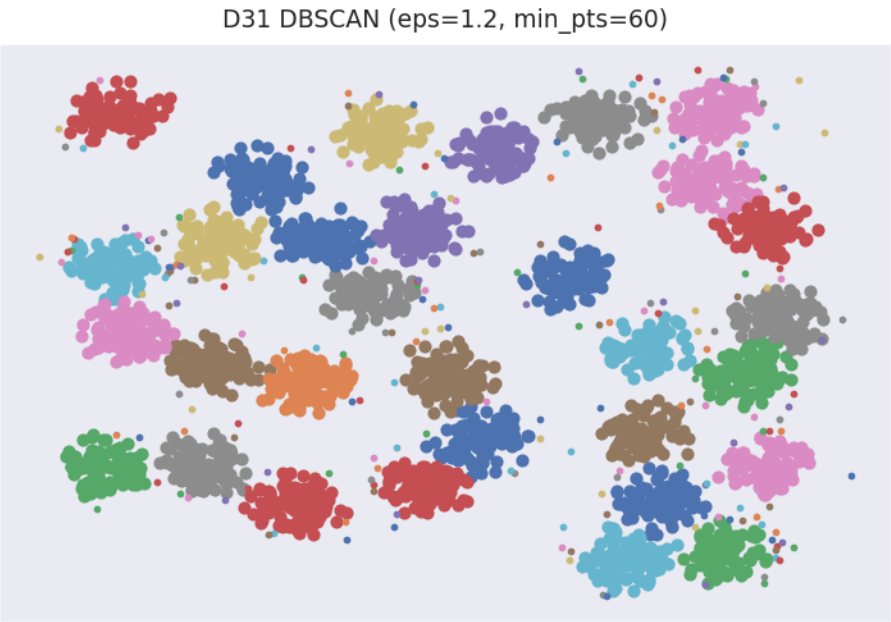
Of Clusters: 3

Of Noise: 0

Purity: 1.000

D31:



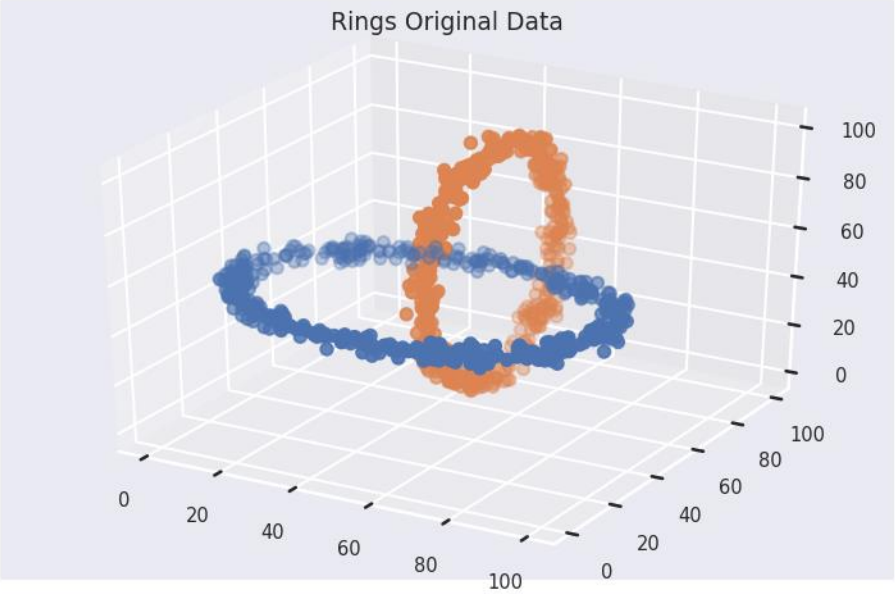


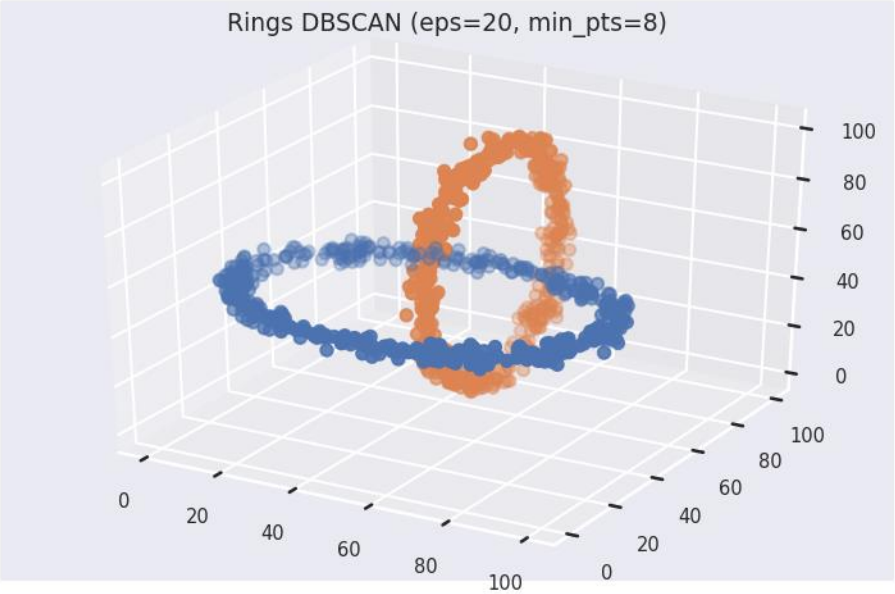
Of Clusters: 31

Of Noise: 207

Purity: 0.914

Rings:





Of Clusters: 2

Of Noise: 0

Purity: 1.000

ب) گزارش مقدار Purity هر دیتاست پس از اجرای DBSCAN

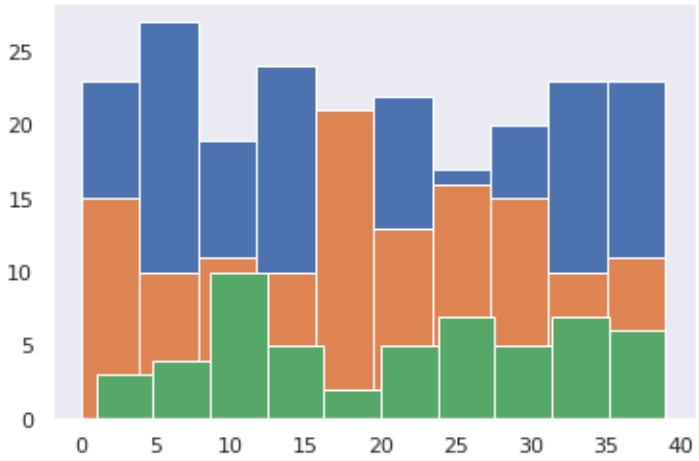
```
Purity Score for 'Compound' is   0.965
Purity Score for 'Path-based' is 0.769
Purity Score for 'Spiral' is     1.000
Purity Score for 'D31' is        0.914
Purity Score for 'Rings' is      1.000
```

سوال ۳)

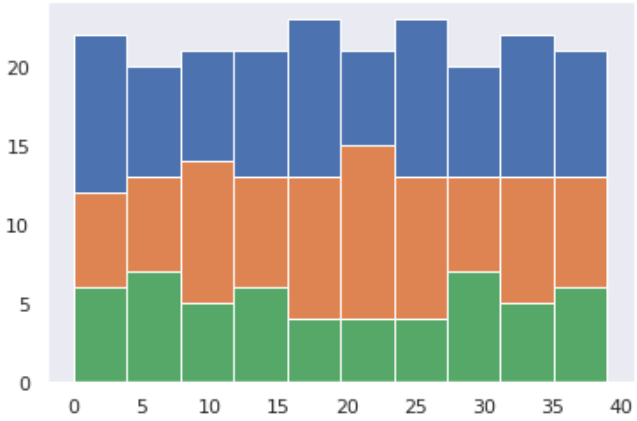
نوت بوک مربوطه https://colab.research.google.com/drive/1e3mT5WPcXRm5zhwEp0HM2cI0EtGJph00?usp=sharing

الف) یعنی موقعی که میخواهیم داده ها را به دسته های Train, Valid, Test تبدیل کنیم دقت کنیم که تعداد داده از هر کلاس در این دسته ها تقریبا برابر باشد. به عنوان مثال اینطوری نباشد که در دسته Train تعداد بسیار زیادی داده از کلاس ۱ داشته باشیم ولی در دسته Test از این کلاس داده ای موجود نباشد.

نمودار توزیع مربوط به دسته بندی غیر Stratified را در تصویر زیر مشاهده میکنید. در این تصویر محور افقی کلاس ها هستند و محور عمودی میزان داده موجود در هر کدام از سه دسته (Train, Test, Valid) که با رنگ های آبی، سبز، نارنجی مشخص شده اند را نشان میدهد.

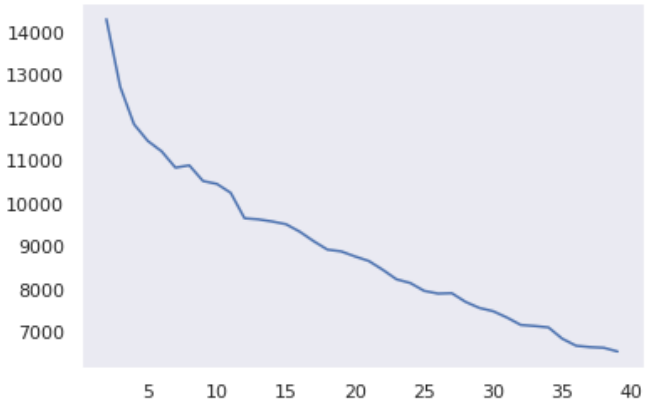


حالا دسته بندی را با رعایت برابری بین کلاس ها و به صورت Stratified انجام میدهیم.

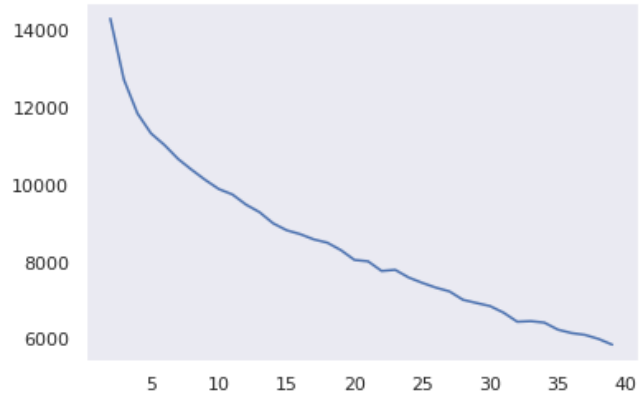


همانطورکه مشخص است در این نوع دسته بندی تعداد داده موجود در دسته ها از هر کلاس تقریبا برابر و برابر با $\frac{1}{3}$ کل داده های موجود است.

ب) در این مرحله Kmeans را به ازای k های مختلف از ۲ تا ۴۰ اجرا میکنیم و در هر مورد SSE را حساب میکنیم.



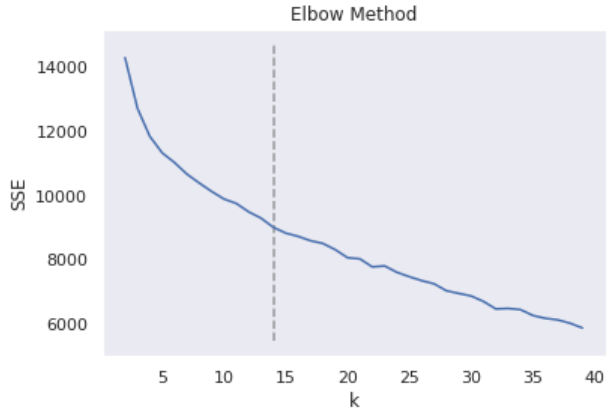
همچنین برای اطمینان از صحت انجام مرحله بالا و صحت پیاده سازی *Kmeans* مان، آن را با پیاده سازی *Kmeans* از لایبرری *SKlearn* نیز مقایسه کردیم.



همانطور که به نظر میرسد پیاده سازی ما تقریبا درست کار میکند.

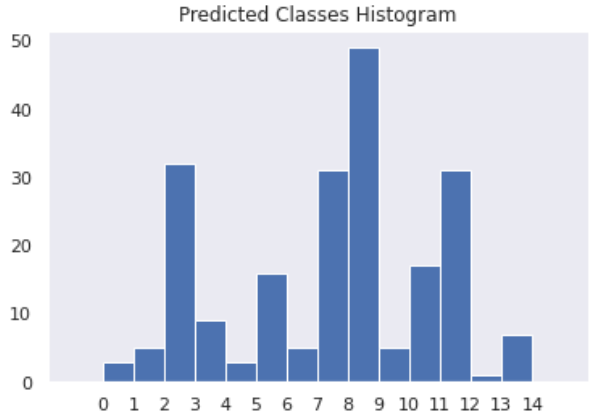
نمودارهای k-*SSE* را در تصاویر بالا مشاهده میکنید. حالا با کمک روش *Elbow* میتوانیم بهترین مقدار k را از روی این نمودار ها بدست آوریم.

در این نوت بوک از لایبرری *Kneed* برای یافتن نقطه *Elbow* استفاده میکنیم.



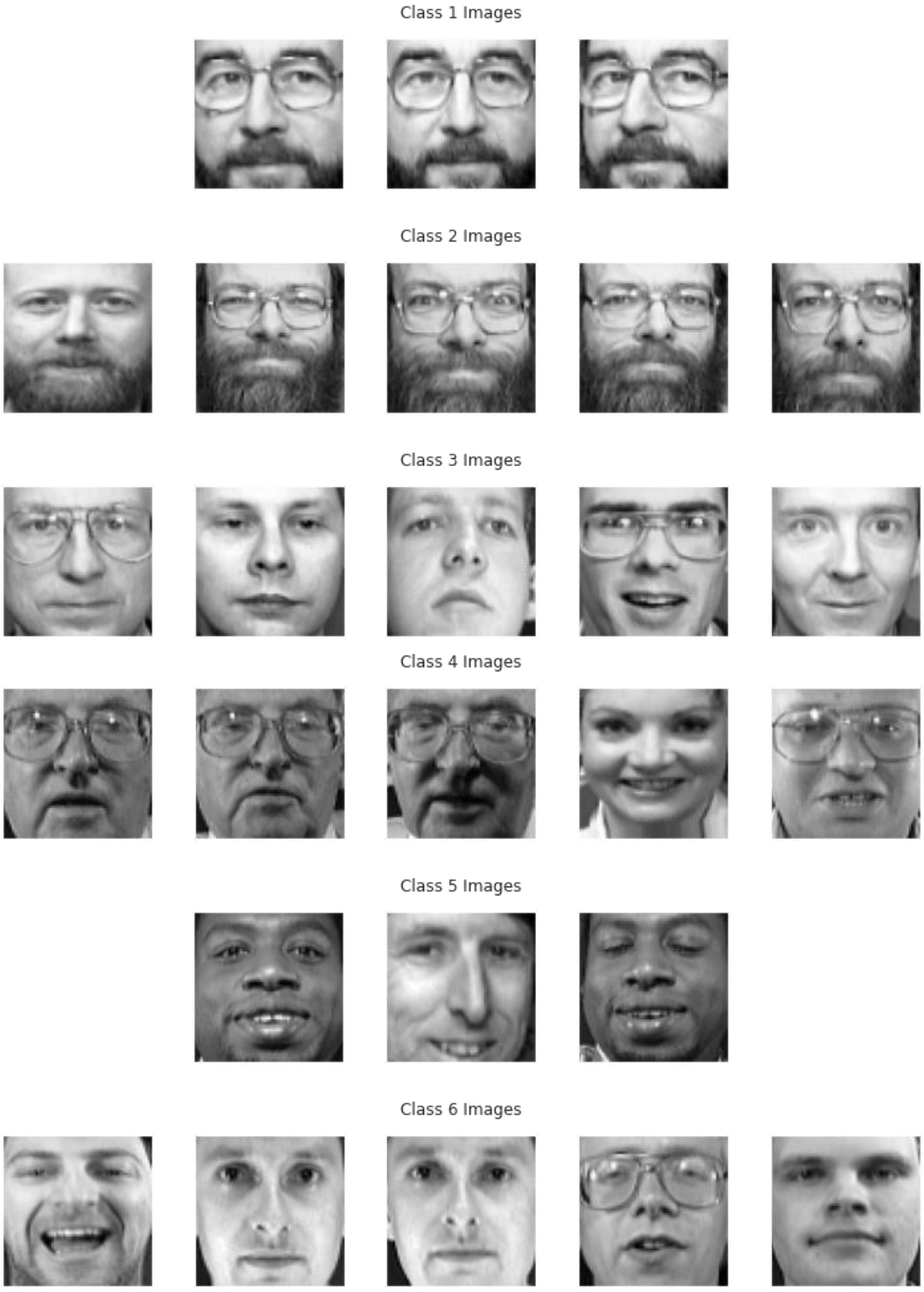
طبق نتیجه بالا بهترین مقدار K برای این مسئله عدد ۱۴ است.

این یعنی قرار است ۴۰۰ تصویر ورودی را به ۱۴ خوشه مختلف تقسیم بندی کنیم. امیدواریم پس از خوشه بندی تصاویری که در هر خوشه قرار میگیرند یا از یک شخص باشند یا بسیار به هم شباهت داشته باشند.



پس از اجرای خوشه بندی با $K=14$، میتوانیم توزیع تصاویر در خوشه های مختلف را ببینیم. به عنوان مثال خوشه شماره ۸ دارای حدود ۵۰ تصویر است.

ج) در ادامه تعدادی از تصاویر هر خوشه را نمایش میدهیم تا ببینیم خوشه بندی توانسته تصاویر شبیه به هم را جدا کند یا خیر.(تعداد تصاویر نمایش داده شده از هر دسته برابر با $\min(k_c, 5)$ در نظر گرفته شده است. که k_c تعداد تصاویر در کلاستر k است.)



Class 7 Images

Class 8 Images

Class 9 Images

Class 10 Images

Class 11 Images

Class 12 Images

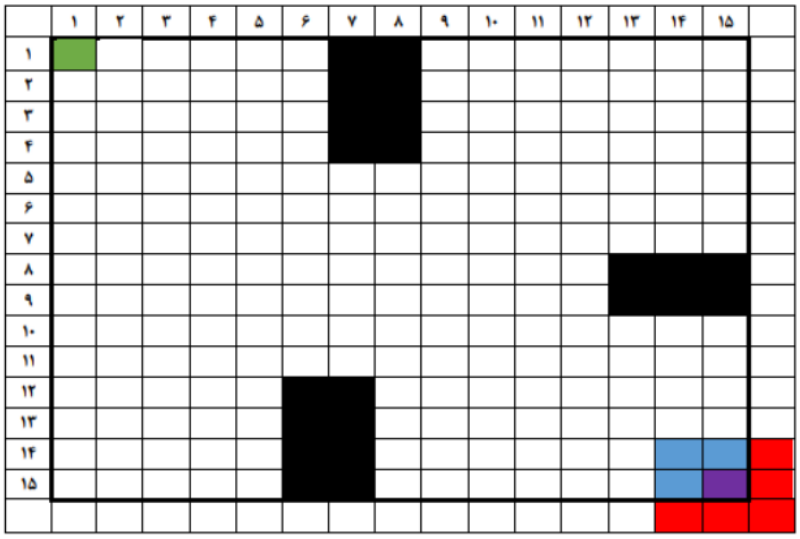
Class 13 Images

Class 14 Images

همانطور که مشاهده میشود نتیجه خوشه بندی بسیار **قابل قبول** است. یکی از دلایلی که این روش توانست اینقدر خوب عمل کند از خاصیت های این دیتاست است که تصاویر چهره افراد برش داده شده اند و چشم و دهان و بینی و ... در تمامی تصاویر تقریبا در یک نقطه قرار گرفته اند. اگر این شرایط وجود نداشت ممکن بود به روش های پیچیده تری برای بررسی میزان شباهت بین چهره ها نیاز پیدا کنیم(در این مثال صرفا به تصاویر به چشم یک بردار ۶۴ در ۶۴ نگاه کردیم و برای بررسی میزان شباهت از نرم اختلاف دو تصویر استفاده شد).

سوال ٤)

برای این سوال باید یک نوت بوک ناقص که از لایبرری amalearn و gym استفاده میکرد را کامل میکردیم. هدف این تمرین پیاده سازی الگوریتم های Value Iteration , Policy Iteration بر روی یک مسئله مسیریابی ربات بود.



شکل ۱ – ربات در ابتدا در خانه بنفش – هدف رسیدن به خانه سبز – نقاط مشکلی مانع

گزارش این سوال به سه بخش تقسیم شده است:

بخش اول) پاسخ دادن به سوال هایی که در طرح این تمرین مطرح شده بود.

بخش دوم) دیدن مثالی مشابه اما با محیطی متفاوت(اضافه کردن موانع در وسط صفحه).

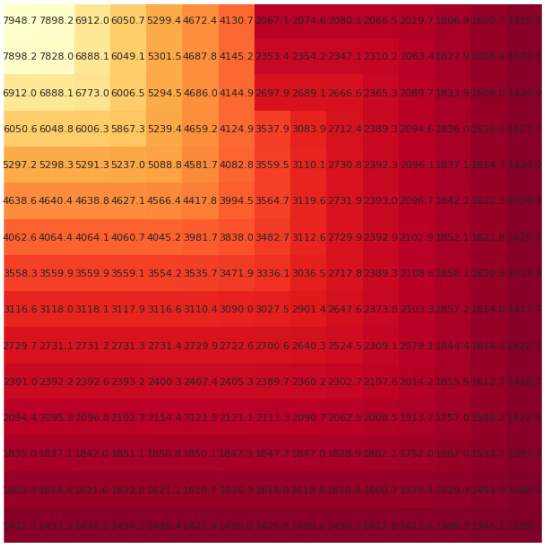
بخش سوم) توضیحاتی در مورد gym و نحوه عملکرد کلی کلاس های Agent و Environment و تابع اصلی take_action()

نوت بوک مربوط به این سوال در فایل فشرده تمرین وجود دارد. همچنین با توجه به اینکه اجرا شدن این نوت بوک کمی زمان بر است، یک **فایل ویدیویی** که مربوط به اجرای **انیمیشن** قسمت ***Main*** این نوت بوک است نیز در فایل فشرده موجود است.

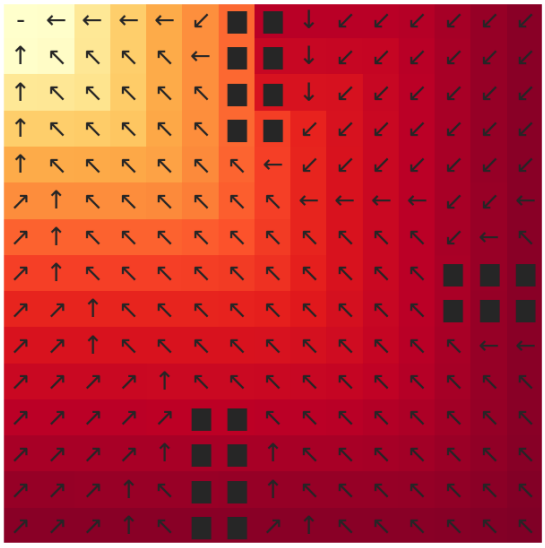
بخش اول)

۱) در این قسمت سه تصویر مشاهده خواهید کرد. تصویر اول مربوط به نتیجه Value Function نهایی (V^*) پس از اجرای **Policy Iteration**، تصویر دوم مربوط به نتیجه Policy نهایی (π^*) پس از اجرای **Policy Iteration** و تصویر سوم مسیر رسم شده با دنبال کردن Policy نهایی از مبدا تا مقصد میباشد.

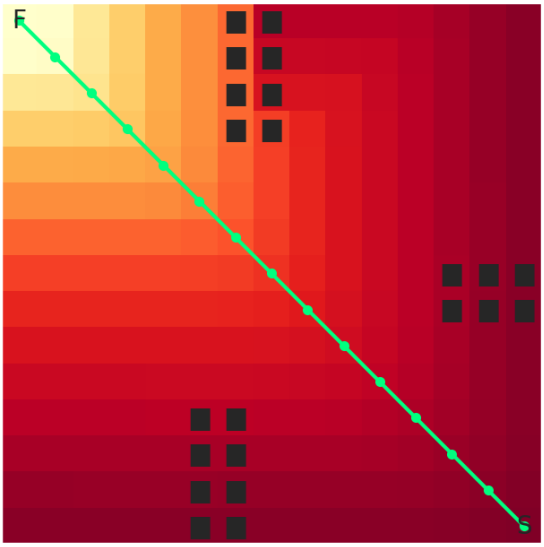
Value - Base Conditions



Policy - Base Conditions

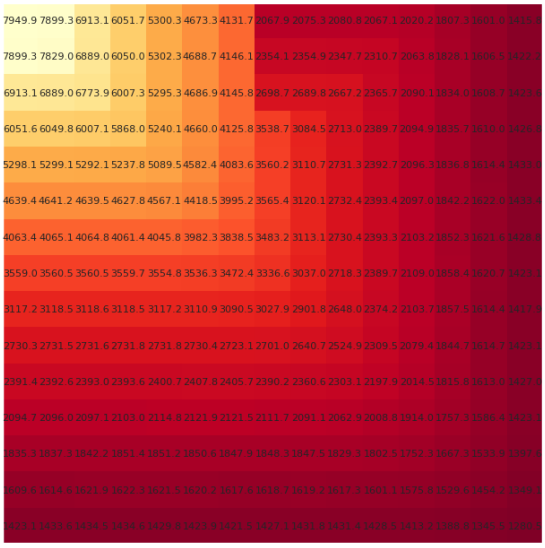


Path - Base Conditions

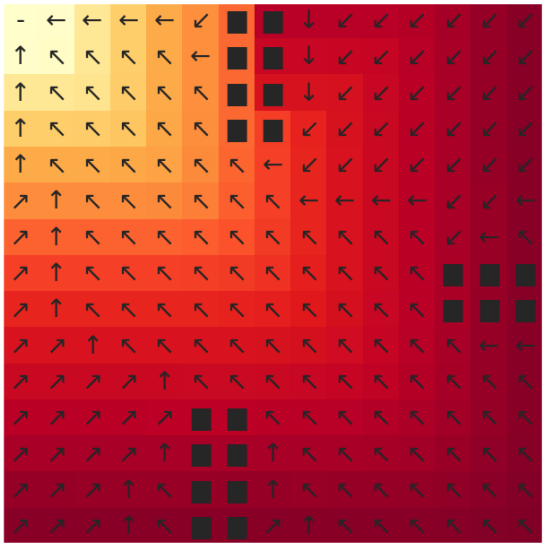


۲) در این قسمت سه تصویر مشاهده خواهید کرد. تصویر اول مربوط به نتیجه Value Function نهایی (V^*) پس از اجرای **Policy Iteration**، تصویر دوم مربوط به نتیجه Policy نهایی (π^*) پس از اجرای **Policy Iteration** و تصویر سوم مسیر رسم شده با دنبال کردن Policy نهایی از مبدا تا مقصد میباشد.

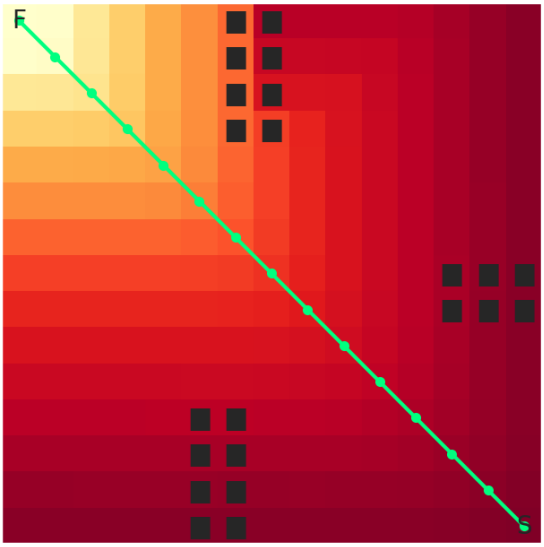
Value - No Friction



Policy - No Friction

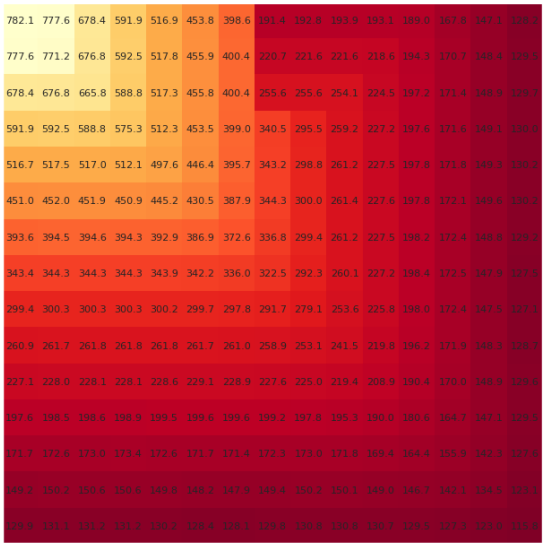


Path - No Friction

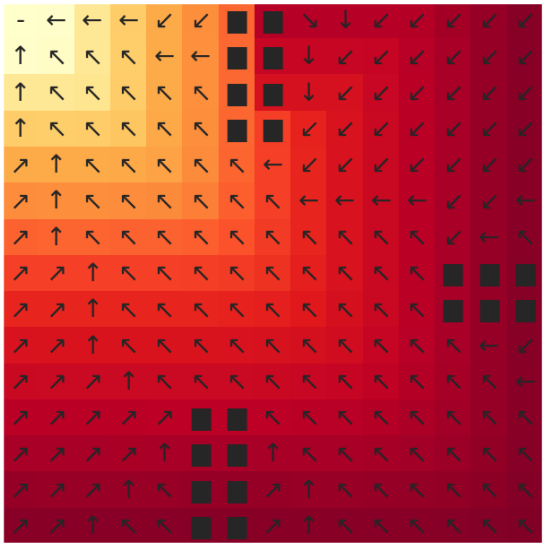


۳) در این قسمت سه تصویر مشاهده خواهید کرد. تصویر اول مربوط به نتیجه Value Function نهایی (V^*) پس از اجرای **Policy Iteration**، تصویر دوم مربوط به نتیجه Policy نهایی (π^*) پس از اجرای **Policy Iteration** و تصویر سوم مسیر رسم شده با دنبال کردن Policy نهایی از مبدا تا مقصد میباشد.

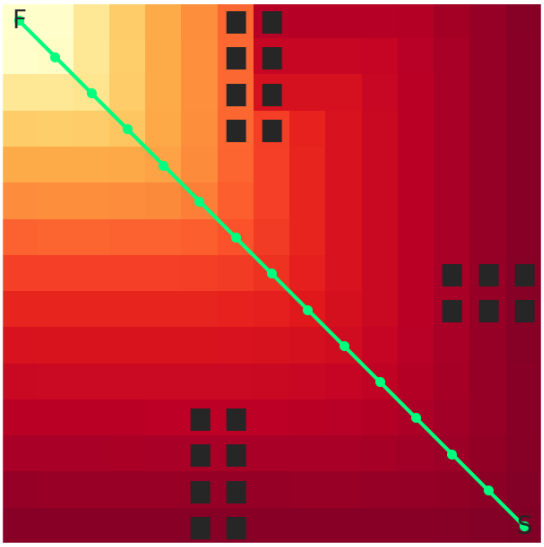
Value - High Friction



Policy - High Friction

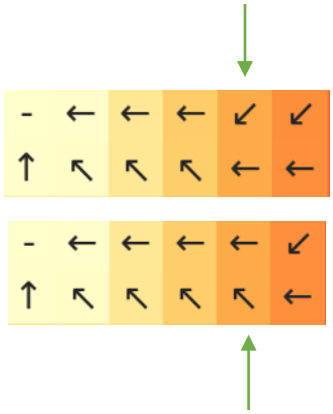


Path - High Friction

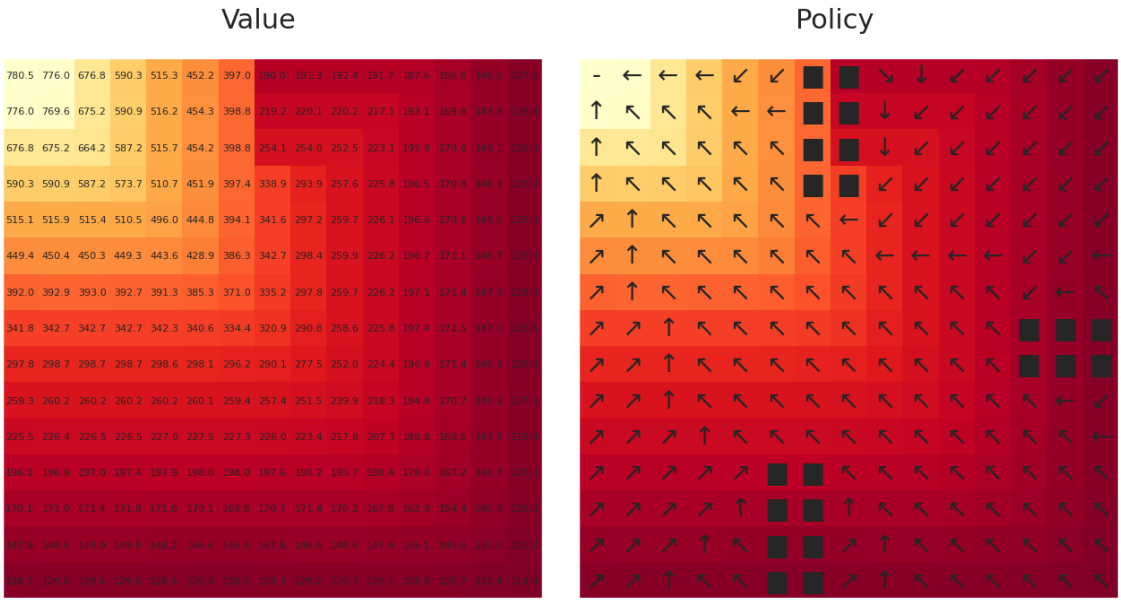


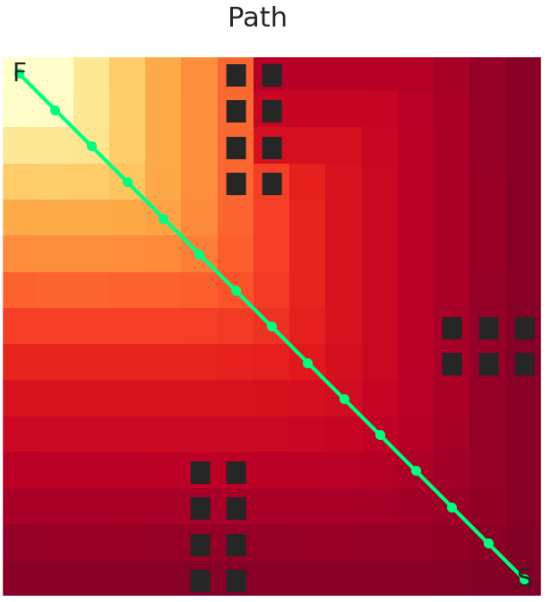
۴) همانطور که دیدید به ازای هر سه حالتِ محیط، **Path بدست آمده برابر شد**(به صورت یک خط صاف از S به F) اما مقادیر V^* در این سه حالت همانطور که انتظار میرفت متفاوت است. این تفاوت اما به اندازه ای نیست که بتواند در π^* تغییر اساسی ایجاد کند[1]. در بخش بعدی گزارش این سوال به بررسی صحت پیاده سازی ها با استفاده از یک محیط دیگر میپردازیم. **همچنین مقادیر مختلفی برای Discount را نیز در آن محیط آزمایش میکنیم**. بنابراین در این محیط، هر سه حالت مسیر با طول یکسان و برابری تولید کردند. در نتیجه در مرحله بعد (Value Iteration) صرفا از حالت پایه محیط استفاده میکنیم.

۱- لازم به ذکر است با اینکه مسیر نهایی تغییر نکرد اما تعدادی از خانه های Policy (که در مسیر تاثیری نداشتند) تغییر کردند به عنوان مثال سلول های سطر ۱ و ستون ۵ و همینطور سلول های سطر ۲ و ستون ۵ در دو حالت آخر با هم تفاوت دارند. نسخه بزرگنمایی شده این دو Policy را میتوان در شکل زیر مشاهده کرد.



۵) در این قسمت سه تصویر مشاهده خواهید کرد. تصویر اول مربوط به نتیجه Value Function نهایی (V^*) پس از اجرای **Value Iteration**، تصویر دوم مربوط به نتیجه Policy نهایی (π^*) پس از اجرای **Value Iteration** و تصویر سوم مسیر رسم شده با دنبال کردن Policy نهایی از مبدا تا مقصد میباشد. همانطورکه انتظار میرفت زمان همگرایی برای این الگوریتم بیشتر از الگوریتم Policy Iteration بود.

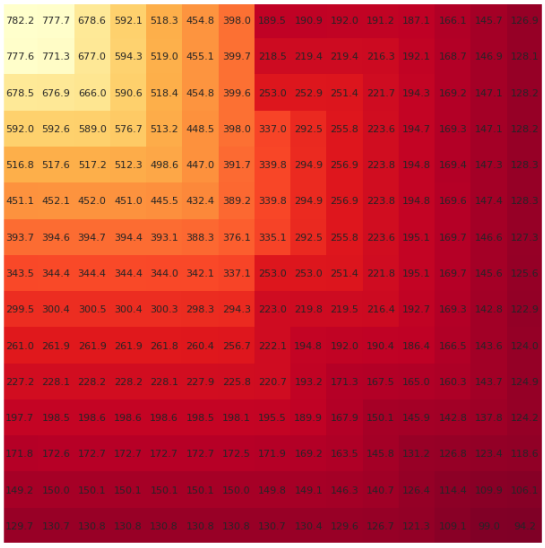




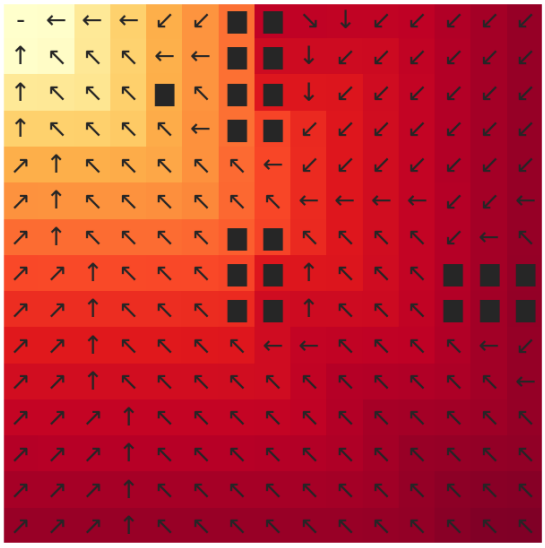
بخش دوم) به دلیل ساده بودن محیط داده شده در صورت سوال امکان وجود اشتباه در پیاده سازی وجود داشت. به همین دلیل الگوریتم ها را بر روی محیط پیچیده تری هم آزمایش کردیم. همانطور که مشاهده میشود هم Policy نهایی هم مسیر بدست آمده به خوبی محاسبه شده اند. در ادامه این بخش، با ثابت نگه داشتن شرایط محیط و تغییر Discount factor الگوریتم را اجرا کردیم.

Discount Factor = 0.9

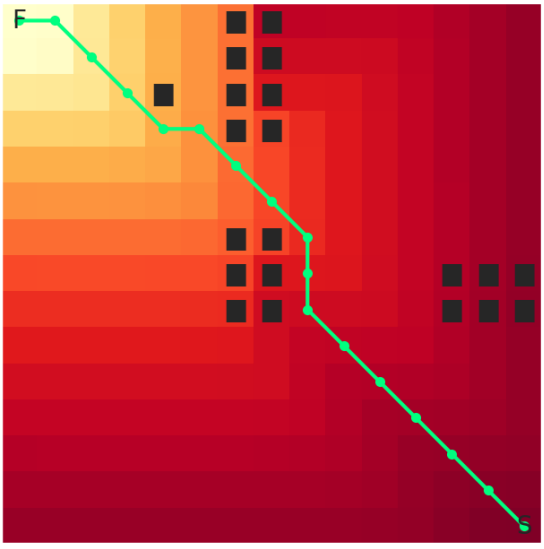
Value - Base Conditions



Policy - Base Conditions



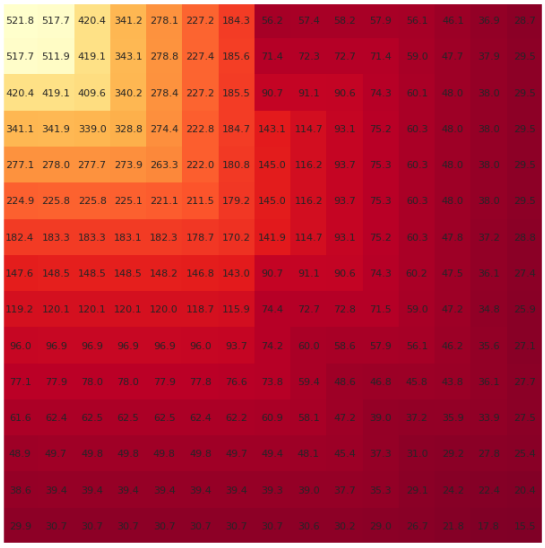
Path - Base Conditions



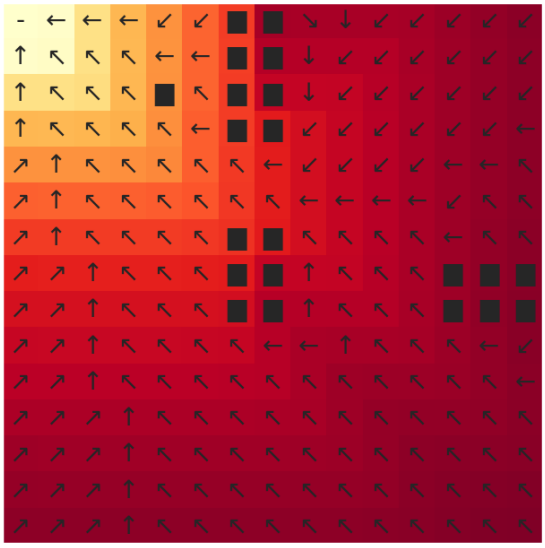
سعی شده تصاویر متناظر در هر صفحه در جای ثابتی قرار بگیرند بنابراین میتوانید با زدن دکمه های جهت روی کیبورد نتیجه مقادیر مختلف Discount Factor را مشاهده کنید.

Discount Factor = 0.85

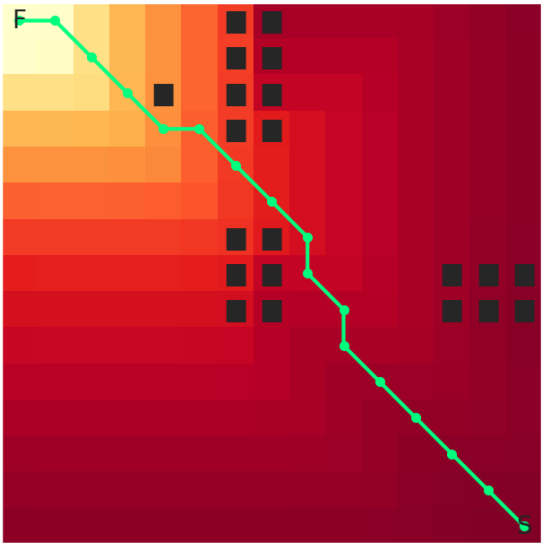
Value - Base Conditions



Policy - Base Conditions



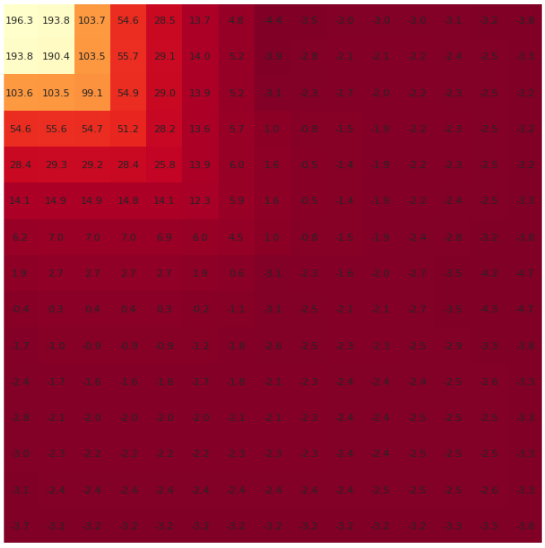
Path - Base Conditions



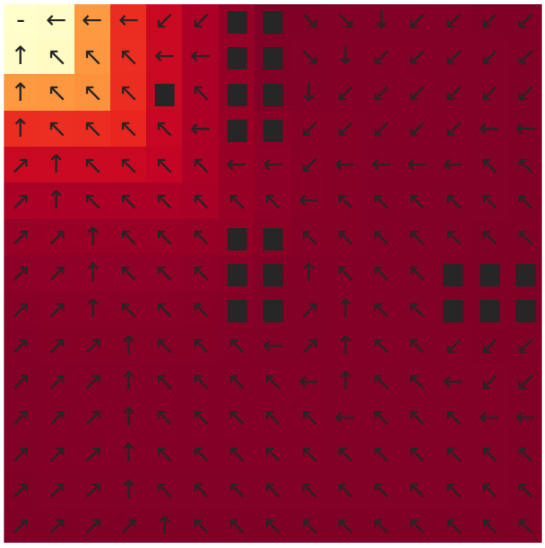
کاهش مقدار Discount Factor باعث میشود Agent آینده نگری کمتری داشته باشد. چون مقادیر استیت های همسایه با ضریب کمتری بر روی یک استیت تاثیر میگذارند و در نتیجه استیت های دور تر با ضریب بسیار کمتری بر روی استیت های دور تاثیر میگذارند که در نتیجه میتوان گفت ناحیه دید Agent کمتر میشود. طی آزمایش های انجام شده اگر مقدار Discount Factor را خیلی کم بگذاریم ممکن است مسیری برای رسیدن به Goal پیدا نشود!

Discount Factor = 0.6

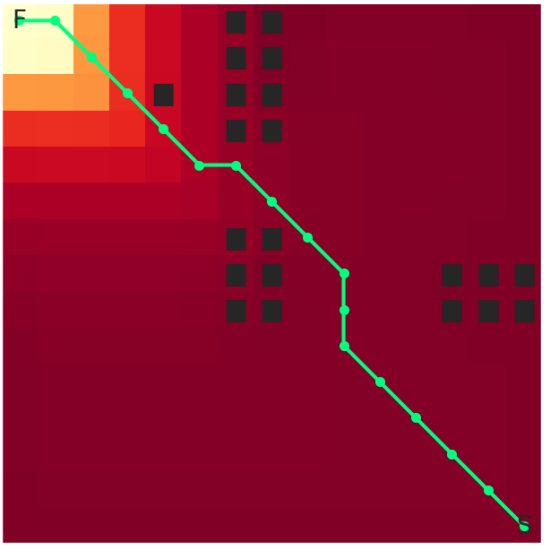
Value - Base Conditions



Policy - Base Conditions



Path - Base Conditions



بخش سوم) در نهایت میخواهیم کمی در مورد کار کتابخانه gym و ساختار Agent, Environment ای که پیاده سازی کردیم صحبت کنیم. طبق بررسی های انجام شده معمولا نحوه اجرای یک محیط و ایجنت ساخته شده با فریمورک gym به شکل زیر است.

```
for i in range(num_episodes):

    state = env.reset()
    print('Time Step 0 :')
    env.render()

    for t in range(num_timesteps):
        random_action = env.action_space.sample()

        new_state, reward, done, info = env.step(random_action)
        print ('Time Step {} :'.format(t+1))


        env.render()
        if done:
            break
```

به ازای تعدادی episode که هر episode تعدادی timestep دارد تابع step مربوط به محیط صدا زده میشود. این کار به هنگام برابر شدن مقدار done با True متوقف میشود و episode بعدی آغاز میشود. لازم به ذکر است که قبل از اجرای episode بعدی محیط reset میشود. همچنین مرسوم است در هر timestep وضعیت فعلی محیط را به صورت انیمیشن نمایش دهیم. مدل بر اساس reward ای که در هر مرحله میگیرد با کمک الگوریتم های مختلفی میتواندند آموزش ببیند.

کتابخانه amalearn که یک wrapper برای gym است یک سری تغییرات در نحوه کار ایجاد کرده. به عنوان مثال تابع step مربوط به محیط را در تابع take_action مربوط به Agent صدا میزند و ... .

نحوه پیاده سازی این سیستم در نوت بوک ما به این شکل بود که دفعه اولی که take_action صدا زده میشود با استفاده از algorithm انتخابی کاربر (که به Constructor پاس داده شده) Policy بهینه را محاسبه میکنیم.

```
# Hyperparameters
actionPrice=-0.01
goalReward=1000
punish=-1
theta=0.01
discount=0.9

# Model
environment1 = Environment(actionPrice=actionPrice, goalReward=goalReward, punish=punish, obstacles=obstacles)
agent1 = Agent(environment=environment1, theta=theta, discount=discount, algorithm='policy_iteration')
```

```
# Taking the first step (need to calculate the policy)
observation, reward, done, info =  agent1.take_action()
```
[14] ✓ 37.3s

```
Calculating The Optimal Policy ...
Algorithm: Policy Iteration
Iteration: 1, stable: False
Iteration: 2, stable: False
Iteration: 3, stable: False
Iteration: 4, stable: False
Iteration: 5, stable: False
Iteration: 6, stable: True
Policy Converged after 6 iterations
```

اما از دفعه های بعد نیازی به اجرای الگوریتم های Policy Iteration یا Value Iteration نیست و میتوانیم با تکیه بر Policy پیدا شده در هر step حالت محیط را عوض کنیم. روند کلی تابع take_action را مشاهده میکنید که قسمت اصلی مدل ماست و از Main برنامه صدا زده میشود. در بخش اول بر اساس Algorithm انتخابی مدل را در صورت نیاز آموزش میدهد. و در ادامه بر اساس action بهینه که از Policy گرفته میشود، تابع step محیط را صدا میزنیم. در نهایت محیط را نمایش میدهیم. ویدیوی مربوط به اجرای این بخش را میتوانید در فایل فشرده ارسالی ببینید.

```python
# Done✅
def take_action(self) -> tuple[object, float, bool, object]:
    # Calculate the Optimal Policy for the first time
    if not self.is_policy_calculated:
        print("Calculating The Optimal Policy ...")
        if self.algorithm == "value_iteration":
            print("Algorithm: Value Iteration")
            self.value_iteration(debug=True)
        elif self.algorithm == "policy_iteration":
            print("Algorithm: Policy Iteration")
            self.policy_iteration(debug=True)
        self.is_policy_calculated = True

    # Pass data from agent to environment
    self.environment.optimal_policy = self.policy
    self.environment.optimal_value = self.V

    next_action = self.policy[self.environment.state[0]-1, self.environment.state[1]-1]

    # Step method of the environment
    observation, reward, done, info = self.environment.step(next_action)

    # Render the environment after every action
    self.environment.render()

    return observation, reward, done, info
```

تعدادی از فریم های انیمیشن: (راست به چپ)