«*In The Name Of GOD*»

[HW-03-Report]

[MACHINE LEARNING]

Hasan Masroor | [403131030] | January 13, 2025

"فهرست مطالب تمرين 03"

Problem 1 3

A) 3

B) 3

C) 4

D) 5

E) 5

Problem 2 6

A) 6

B) 7

C) 7

D) 8

E) 8

F) 9

Problem 4 10

A) 10

B) 12

C) 14
Problem 5 15
A) 15
B) 21
C) 22
Problem 6 24

Problem 1: Breast Cancer Prevention using K-Means Algorithm

A.

در این تمرین از الگوریتم خوشه‌بندی K-Means جهت خوشه‌بندی استفاده شده است. برای این کار ابتدا کلاسی تحت عنوان KMeansCluster نوشته شده است که وظیفه آموزش مدل K-Means و خوشه‌بندی داده‌ها را بر عهده دارد. این کلاس از متدهای زیر ایجاد شده است:

- **init:** این متد ایجادکننده یک شی جدید از K-Means است که به عنوان ورودی سه پارامتر اصلی k که تعداد خوشه مدنظر، tol برای حد آستانه تغییرات مرکز خوشه‌ها و max_iter برای تعداد تکرار الگوریتم برای انجام خوشه‌بندی را دریافت می‌کند. علاوه بر آن، این متد یک متغیر خالی برای مرکز خوشه‌ها نیز ایجاد می‌کند.
- **predict:** این متد به ازای تمام نقاط درون دیتاست، فاصله اقلیدسی را با مرکز خوشه‌ها (دو مرکز) محاسبه می‌کند و هر نقطه را به نزدیک‌ترین خوشه نسبت می‌دهد.
- **fit:** این متد به عنوان ورودی نقاط درون دیتاست و مرکز خوشه‌ها را می‌گیرد و سپس به تعداد max_iter هربار نقاط را به نزدیک‌ترین مرکز خوشه لیبل‌دهی می‌کند و بعد از انجام لیبل‌دهی به تمام نقاط درون دیتاست، با میانگین‌گیری از نقاط آن خوشه، مرکز خوشه‌های جدید را می‌یابد. در ادامه بررسی می‌کند که مرکز خوشه‌های جدید میزان تغییراتشان با مرکز خوشه انتخاب شده قبلی از مقدار حد آستانه مشخص شده tol بیشتر شده است یا خیر. اگر مقدار آن کمتر باشد که دیگر الگوریتم را تکرار نمی‌کنیم و لیبل نقاط و مرکز خوشه‌ها نهایی شده‌اند؛ در غیر اینصورت به انجام دوباره الگوریتم تا رسیدن به تعداد تکرار max_iter ادامه می‌دهیم .
- **accuracy:** در اینجا لیبل‌های پیش‌بینی‌شده را با لیبل‌های درست داده شده مقایسه می‌کنیم تا ببینیم در کل در تعداد کل نقاط چند لیبل را به درستی پیش بینی کرده‌ایم. در اینجا چون ممکن است مقدار عددی لیبل‌های درون دیتاست اولیه با مقدار عددی ست‌شده برای هر خوشه در فرایند انجام الگوریتم متفاوت شده باشد، پس اگر مقدار صحت کمتر از 0.5 شده باشد؛ آن را منهای یک می‌کنیم تا مقدار اصلی صحت به‌دست آید.
- **sse:** در این متد فاصله اقلیدسی تمام نقاط درون دیتاست را با مرکز خوشه آن به‌دست می‌آوریم و جمع می‌کنیم. هر چه این مقدار کمتر باشد نشان می‌دهد که نقاط به مرکز خوشه خود نزدیک‌تر هستند و خوشه‌بندی بهتر انجام شده است.

B.

در اینجا در ابتدا نقاط درون دیتاست را استانداردسازی می‌کنیم و سپس ۵ بار مقادیر تصادفی به دو مرکز خوشه نسبت می‌دهیم و با الگوریتم K-Means نوشته شده با مقادیر ورودی دو خوشه، مقدار tol برابر با 0.5 و 1000 تکرار اجرا می‌کنیم. نتایج خروجی Accuracy و SSE (sum of squared error) به صورت زیر است:

```
Test 0:
Accuracy:  0.9242957746478874
SSE:  949275498.5759952

Test 1:
Accuracy:  0.8961267605633803
SSE:  949391266.6407444

Test 2:
Accuracy:  0.9119718309859155
SSE:  949279404.4161131

Test 3:
Accuracy:  0.9242957746478874
SSE:  949268615.750155

Test 4:
Accuracy:  0.903169014084507
SSE:  949183487.3352125
```

با مقادیر متفاوت و تصادفی مرکز خوشه‌ها، مقادیر صحت و SSE جواب‌ها همچنان نزدیک به هم هستند و تاثیر چندانی بر روی جواب آخر نداشته‌اند؛ دلیل آن این می‌تواند باشد که مراکز 5 خوشه‌بندی انجام شده به هم نزدیک هستند.

C.

با نقاط مرکز خوشه داده شده در سوال، الگوریتم را یک بار با max_iter = 1 و بار دیگر با مقدار 1000 اجرا کردیم (در هر دو مقدار tol برابر با 0.001 بوده است). دلیل این که یک بار داده‌ها با یک iteration آموزش داده شده‌اند این است که مقادیر آن قرار است در بخش D مورد استفاده قرار گیرد. نتایج آن در ادامه قرار دارد:

نتایج با max_iter = 1:

```
Accuracy:  0.5774647887323944
SSE:  949913991.5424824
```

نتایج با max_iter = 1000:

```
Accuracy:  0.9066901408450704
SSE:  949375956.5257089
```

.D

در این بخش از مراکز خوشه‌های به‌دست آمده از آموزش داده‌ها با max_iter برابر با 1000 در قسمت قبل برای آموزش دوباره داده‌ها استفاده شده است. در این بخش برای آموزش تعداد max_iter برابر با یک تعیین شده است. نتایج خروجی به شرح زیر است:

```
Accuracy:  0.9066901408450704
SSE:  949375956.5257089
```

با توجه به نتایج به دست آمده مشاهده می‌شود که فقط با یک iteration به جواب مشابه با max_iter برابر با 1000 در بخش قبل دست یافته است؛ یعنی این مراکز خوشه، مراکز بسیار مناسبی برای این خوشه‌ها بوده‌اند. در صورتی که این را با max_iter برابر با 1000 اجرا کنیم نتایج خروجی تغییری نمی‌کند. این موضوع نشان می‌دهد که الگوریتم در همان ابتدا همگرا شده و مراکز خوشه‌های بهتری پیدا نمی‌کند. این در حالی اتفاق افتاده است که با توجه به بخش قبل، با یک iteration به جواب در حدود 57.7464 درصد دست یافته بوده است.

.E

برای این بخش از دو الگوریتم Agglomerative و SVM که به ترتیب از الگوریتم‌های Unsupervised و Supervised هستند انتخاب کرده‌ایم. نتایج خروجی الگوریتم برای الگوریتم Agglomerative به صورت زیر است:

```
Accuracy:  0.9014084507042254
```

نتایج خروجی الگوریتم SVM نیز به صورت زیر است:

```
Accuracy:  0.9876760563380281
```

برای الگوریتم Supervised که از SVM استفاده کردیم به دقت خیلی خوبی نسبت به K_Means رسیدیم و نشان از عملکرد بهتر این الگوریتم است؛ اما از طرفی هم می‌دانیم که روش K_Means بدون دانستن لیبل‌ها به دقتی که بالاتر به‌دست آوردیم رسیده است و عملکرد خوب این الگوریتم را نشان می‌دهد. برای روش‌های نظارتی و بدون نظارت از روش های دیگری مثل جنگل تصادفی، DBSCAN و ... هم می‌توانستیم استفاده کنیم.

Problem 2: Airplane crash

.A

ابتدا دیتاست را آپلود می‌کنیم و نمونه‌ای از داده‌ها (به عنوان مثال 5 داده اول) را به نمایش می‌گذاریم:

	Unnamed: 0	PassengerId	Survived	Class	Name	Sex	Age	Ticket Price	Safety
0	0	1	Didn't Survive	Economy	Braund, Mr. Owen Harris	male	22.0	7.2500	0.336957
1	1	2	Survived	First Class	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	71.2833	0.553571
2	2	3	Survived	Economy	Heikkinen, Miss. Laina	female	26.0	7.9250	0.336957
3	3	4	Survived	First Class	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	53.1000	0.336957
4	4	5	Didn't Survive	Economy	Allen, Mr. William Henry	male	35.0	8.0500	0.336957

سپس بهتر است که پیش‌پردازش انجام دهیم و بررسی می‌کنیم که در ستون‌ها آیا مقادیر گمشده داریم یا خیر که در ستون‌های Age تعداد 177 مقدار گمشده و در Safety هم 2 مقدار گمشده داریم:

```
Missing values in each column:
Unnamed: 0         0
PassengerId        0
Survived           0
Class              0
Name               0
Sex                0
Age              177
Ticket Price       0
Safety             2
dtype: int64
```

حالا برای اصلاح مقادیر گمشده می‌توانیم سطرهای مربوط به این مقادیر را حذف کنیم اما راه بهتر این است که آنها را با میانگین یا میانه جایگزین کنیم که در این مسئله ما با میانگین این مقادیر را جایگزین می‌کنیم و سپس مجدد بررسی می‌کنیم که همه ستون‌ها درست باشند و شامل مقادیر گمشده نباشند.

```
Unnamed: 0       0
PassengerId      0
Survived         0
Class            0
Name             0
Sex              0
Age              0
Ticket Price     0
Safety           0
dtype: int64
```

.B

در این پارت باید داده‌ها را به دو مجموعه آموزش (80%) و تست (20%) تقسیم کنیم. برای این کار ابتدا می‌آییم هدف ما که "Survived" هست را در y ذخیره می‌کنیم و همچنین این ستون را به همراه ستون‌های Name, PassengerId و Unnamed که بی ربط هستند و اطلاعاتی نسبت به هدف به ما نمی‌دهند حذف می‌کنیم و بعد تقسیم‌بندی داده‌های آموزش و تست را انجام می‌دهیم:

```
(891, 9)
```

```
Training set size: (712, 5), Test set size: (179, 5)
```

.C

در این پارت بادید یک مدل SVM را پیاده‌سازی کنیم و اول می‌آییم ستون‌هایی که داده‌های کتگوریکال دارند را به مقادیر عددی تبدیل می‌کنیم و در نتیجه باید روی ستون‌های Class و Sex و Survived این کار را انجام دهیم و آنها را به داده‌های عددی تبدیل کنیم. حال مدل SVM را با کرنل خطی پیاده‌سازی می‌کنیم.

از درس با accuracy و precision و recall آشنا هستیم:

- $Accuracy = \frac{\#TP+\#TN}{\#P+\#N} = \frac{\#TP+\#TN}{\#TP+\#FN+\#TN+\#FP}$
- $Precision = \frac{TP}{TP+FP}$ و $TPR = \frac{TP}{TP+FN} = Sensitivity = Recall$

در نهایت هم مقادیر مربوط به هر هر کدام را تا چهار رقم اعشار در خروجی نمایش می‌دهیم:

```
Accuracy: 0.7821
Precision: 0.7536
Recall: 0.7027
```

.D

برای تنظیم هایپرپارامترها از روش گرید سرچ استفاده می‌کنیم و سه هایپرپارامتر C و gamma و kernel داریم که به صورت زیر تنظیم می‌کنیم که بهترین حالت را پیدا کنیم:

- 'C': [0.1, 1, 10, 100],
- 'gamma': [1, 0.1, 0.01, 0.001],
- 'kernel': ['linear', 'poly', 'rbf']

به دلیل محدودیت سیستمی و توان سخت‌افزاری مقادیر بالا مدت زمان طولانی برای اجرا روی سیستم سپری کرد اما اجرای سلول به اتمام نرسید و به عنوان نمونه و برای سادگی مقادیر بالا را به صورت زیر تنظیم کرده‌ایم که در اجرای آن مشکلی ایجاد نشود:

- 'C': [0.1, 1, 10],
- 'kernel': ['linear']

چون gamma برای کرنل‌های غیرخطی استفاده می‌شود برای اجرای کرنل linear نیازی به آوردن آن ندارم اما برای کرنل‌های غیرخطی حتما باید مقداردهی بالاتر که قرار داده شد استفاده کنیم تا بهترین هایپرپارامترها پیدا شوند؛ همچنین در این کد تعداد fold را برابر 5 قرار دادیم و سپس در خروجی دقت را تا چهار رقم اعشار نمایش داده‌ایم:

```
Best Parameters: {'C': 0.1, 'kernel': 'linear'}
Best Cross-Validation Score: 0.7879
Test Accuracy with Best Parameters: 0.7821
```

.E

در پارت قبلی از روش Grid Search با Cross-Validation برای تنظیم هایپرپارامترهای مدل SVM استفاده کردیم و برای عملکرد بهتر مدل باید کرنل‌های مختلف مثل rbf و poly به همراه مقادیر مختلف هایپرپارامترهای c و gamma در نظر گرفته شود که بهترین ترکیب که دقت بالایی را ایجاد کند را مدل بتواند پیدا کند؛ اما با توجه

به محدودیت سیستم و توان سخت‌افزار، مقادیر و مواردی که به آن اشاره کردیم در سلول کد در مدت طولانی در حال اجرا بود و به دلیل عدم به پایان رسیدن برای پیدا کردن بهترین ترکیب هایپرپارامترها؛ مجبور شدیم برای نمونه مقادیر پارامترها را به صورت زیر ساده‌سازی کنیم:

- 'C': [0.1, 1, 10],
- 'kernel': ['linear']

مقادیر gamma برای کرنل‌های غیرخطی مورد استفاده قرار می‌گیرند و به همین خاطر این پارامتر را در کد نمونه نیازی نبود که در نظر بگیریم، زیرا برای کرنل خطی تأثیری ندارند.

همچنین نتایج Grid Search نشان داد که دقت روی داده‌های تست برابر با 0.7821 به دست آمد و بهترین مقادیر هایپرپارامترها نیز به شرح زیر هستند:

- ✓ C = 0.1
- ✓ kernel = 'linear'

مقدار C=0.1 نشان می‌دهد که مدل جریمه کمی برای خطاهای طبقه‌بندی در نظر می‌گیرد. این انتخاب مناسب است، زیرا مقدار کم C باعث ایجاد یک مدل ساده‌تر و تعمیم‌پذیرتر می‌شود و ما در یادگیری ماشین بیشتر میل داریم که از مدل‌های ساده شروع کنیم و بعد به سمت مدل‌های پیچیده‌تر برویم و اصطلاحا پیچیدگی در سادگی است. (اگر مقدار C بیش از حد بالا انتخاب شود، ممکن است Overfitting رخ دهد.)

برای بهبود عملکرد و افزایش دقت، می‌توان با استفاده از کرنل‌های غیرخطی و مقادیر بیشتری از gamma را بررسی کرد و بهترین مقادیر هایپرپارامترها را بدست آورد.

.F

در این پارت باید با استفاده از هایپرپارامترهایی که در پارت d بدست آوردیم مدل SVM را مجدد آموزش دهیم و مقادیر accuracy و precision و recall را به دست آوریم.

ما در این کد هم با استفاده از مقادیر هایپرپارامترهای قبل یعنی کرنل خطی و c=0.1، SVM را آموزش دادیم و در نهایت هم تا چهار رقم اعشار دقت و ... را به دست آوردیم و در خروجی نمایش دادیم.

مطمئنا در صورت عدم مشکل سخت افزاری و مقداردهی هایپرپارامترهای بخش d با موارد گفته شده در قبل احتمال بهبود بیشتر با استفاده از کرنل‌های غیرخطی و یا مقادیر دیگر پارامتر gamma وجود داشت و در کل سعی شد کانسپت روش کار داده شود و یک نمونه ساده صرفا جهت نمایش خروجی برای دقت و بهترین ترکیب هایپرپارامترها نمایش داده شود.

Problem 4: Diabet

A.

قبل از پیاده‌سازی پارت اول با نگاهی به دیتاست متوجه می‌شویم که بعضی از ستون‌ها که شامل مقادیر 0 هستند از لحاظ منطقی غیرعادی هستند؛ به عنوان مثال مقدار 0 برای فشار خون، شاخص توده بدنی و ... نرمال نیستند و در نتیجه یا باید سطرهای حاوی 0 را حذف کنیم و یا راه بهتر این است که این مقادیر را با میانگین یا میانه هر ستون جایگزین کنیم که ما از میانه استفاده کرده‌ایم. ستون‌های گفته‌شده عبارتند از:

- diastolic_blood_pressure
- triceps_skinfold_thickness
- serum_insulin
- bmi

پس قبل از پیاده‌سازی پارت اول برای بهبود عملکرد مدل ابتدا پیش‌پردازش را انجام می‌دهیم و تمامی مقادیر صفر در ستون‌هایی که بالاتر گفته شد با مقدار NaN جایگزین شدند تا به عنوان مقادیر گم‌شده شناسایی شوند سپس برای پر کردن مقادیر گم‌شده، از میانه (Median) هر ستون استفاده شد.

برای استانداردسازی مقادیر و بهبود عملکرد مدل یک مرحله نرمال‌سازی هم انجام می‌دهیم و از Min-Max Scaling استفاده شد. این روش مقادیر هر ستون را در بازه ۰ تا ۱ مقیاس‌بندی می‌کند و از اهمیت بالای ویژگی‌های با مقادیر بزرگ جلوگیری می‌کند. همچنین با توجه به توضیحات کلاس درس می توان از روش‌های دیگر نرمال‌سازی هم مثل Z-Score و ... هم استفاده کرد

بعد از پیش‌پردازش چند نمونه از داده‌های هر ستون را نمایش می‌دهیم:

```
Preprocessed Data:
   time_pregnant_no  plasma_concentration  diastolic_blood_pressure  \
0          0.352941              0.743719                  0.489796
1          0.058824              0.427136                  0.428571
2          0.470588              0.919598                  0.408163
3          0.058824              0.447236                  0.428571
4          0.000000              0.688442                  0.163265

   triceps_skinfold_thickness  serum_insulin       bmi  diabetes_pedigree  \
0                    0.304348       0.133413  0.314928           0.234415
1                    0.239130       0.133413  0.171779           0.116567
2                    0.239130       0.133413  0.104294           0.253629
3                    0.173913       0.096154  0.202454           0.038002
4                    0.304348       0.185096  0.509202           0.943638

        age  class
0  0.483333      1
1  0.166667      0
2  0.183333      1
3  0.000000      0
4  0.200000      1
```

پس از پیش‌پردازش باید RandomForest را پیاده‌سازی کنیم و طبق گفته مسئله سه تا پارامتر داریم که الگوریتم ترکیب‌های مختلف آن‌ها را برای پیدا کردن بهترین مدل بررسی می‌کند؛ همچنین باید ستون Class را همبه عنوان ویژگی هدف در نظر بگیریم و آن را در y ذخیره می‌کنیم و از ویژگی‌های X هم حذف می‌کنیم. پارامترهای الگوریتم ما عبارتند از:

- **n_estimators:** تعداد درخت‌ها را نشان می‌دهد و ما سه مقدار 50 و 75 و 100 را برای این پارامتر در نظر گرفته‌ایم.
- **max_features:** حداکثر ویژگی‌های هر درخت را نشان می‌دهد و برای این پارامتر ما از sqrt و log2 که معمولا رایج است استفاده کرده‌ایم.
- **max_depth:** حداکثر عمق درخت را نشان می‌دهد و سه مقدار 10 و 20 و 30 را در نظر گرفته‌ایم.

(برای جلوگیری از پیچیده نشدن مدل حداکثر مقدار را None نگذاشتیم و از اعداد محدود استفاده کرده‌ایم.)

همچنین داده‌ها را به دو مجموعه آموزش (80%) و تست (20%) تقسیم کردیم و برای هر ترکیب از پارامترهای بالا، RandomForest آموزش داده شد و دقت مدل بر روی داده‌های تست را به دست آوردیم. برای نمایش بهتر خروجی را در غالب یک دیتافریم نمایش دادیم که خروجی زیر حاصل شده است:

	n_estimators	max_features	max_depth	accuracy
0	50	sqrt	10	0.733766
1	50	sqrt	20	0.746753
2	50	sqrt	30	0.746753
3	50	log2	10	0.733766
4	50	log2	20	0.740260
5	50	log2	30	0.740260
6	75	sqrt	10	0.759740
7	75	sqrt	20	0.753247
8	75	sqrt	30	0.759740
9	75	log2	10	0.759740
10	75	log2	20	0.733766
11	75	log2	30	0.733766
12	100	sqrt	10	0.746753
13	100	sqrt	20	0.746753
14	100	sqrt	30	0.746753
15	100	log2	10	0.759740
16	100	log2	20	0.720779
17	100	log2	30	0.720779

در نهایت بین ترکیب های مختلف پارامترها و دقت مربوط به هر کدام، بیشترین دقت را به همراه پارامترهای انتخاب شده آن در خروجی به نمایش می‌گذاریم:

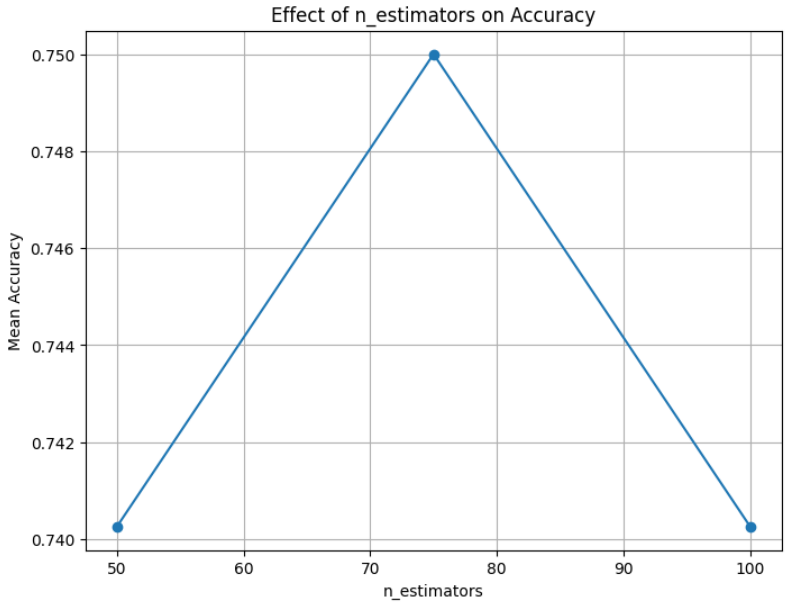
(n_estimators= 75, max_features= sqrt, max_depth= 10)

```
Best Model parameters:
n_estimators          75
max_features        sqrt
max_depth             10
accuracy         0.75974
Name: 6, dtype: object
```

.B

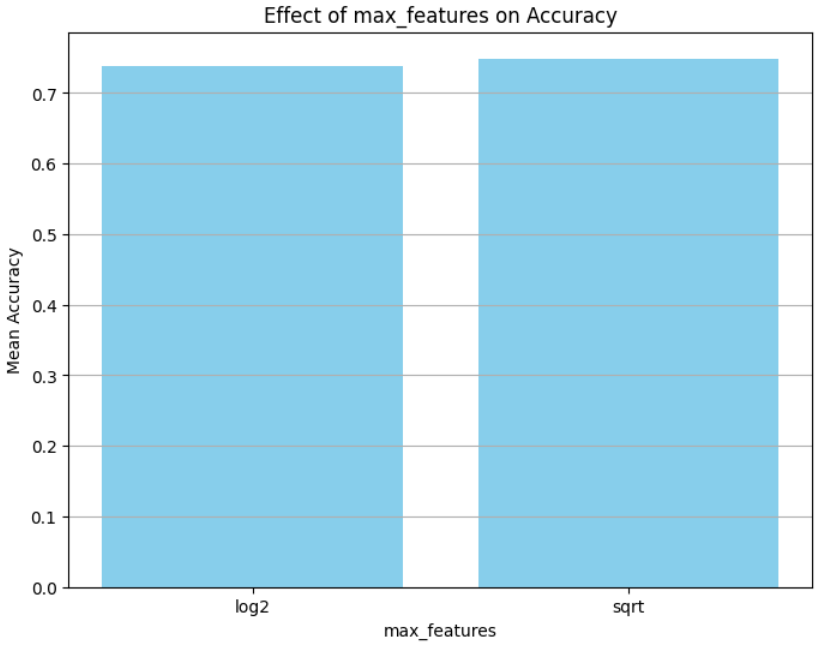
در این بخش برای نشان دادن تاثیر پارامترها روی دقت از نتایج قسمت قبل استفاده کرده‌ایم
با استفاده از تابع groupby() می‌آییم داده‌های هر پارامتر را گروه‌بندی می‌کنیم و فقط داده‌های مربوط به ستون دقت را برای هر گروه انتخاب می‌کنیم و سپس با mean() میانگین مقادیر دقت را برای هر گروه انتخاب می‌کنیم و در نهایت هر کدام از پارامترها را رسم می‌کنیم که تاثیر آنها روی دقت رو بررسی کنیم:

- **n_estimators:**



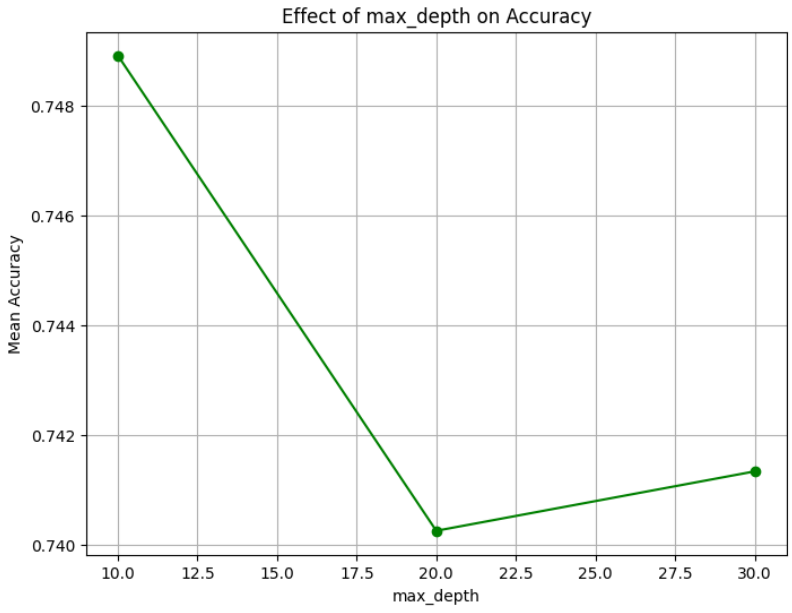
نمودار نشان می‌دهد که با افزایش تعداد درخت‌ها، دقت مدل بهبود می‌یابد، اما این افزایش تا یک حدی می‌تواند که دقت را بهبود بدهد و از یک جایی به بعد (75) روند نزولی می‌گیرد و نه تنها باعث بهبود نمی‌شود بلکه دقت را کم هم می‌کند.

- **max_features:**



نمودار دوم را به جای اینکه مثل قبل خطی رسم کنیم میله‌ای رسم کردیم چون این دو مقدار ارتباطی با هم ندارند و نمودار میله تاثیر هر کدارم را روی دقت بهتر نشان می‌دهد. عملکرد هر دو نزدیک به هم می‌باشد اما sqrt نسبت به log2 دقت بهتری داشته است.

- **max_depth:**



در نمودار آخر هم می‌بینیم که با افزایش عمق دقت دارد کاهش پیدا می‌کند اما از یک نقطه‌ای به بعد (20) مجدد دقت سیر صعودی به خود می‌گیرد و افزایش پیدا می‌کند و احتمالا بعد از نقطه 30 نیز این روند صعودی را حفظ می‌کند.

پس به این نتیجه رسیدیم که افزایش تعداد درخت‌ها تا یک نقطه‌ای دقت را افزایش می‌دهد و بعد از آن روند نزولی می‌گیرد. مقدار sqrt نسبت به log2 دقت بالاتری ایجاد کرد؛ همچنین عمق در ابتدا روند نزولی دارد اما از یک نقطه‌ای به بعد صعودی می‌شود و دقت را بهبود می‌دهد.

.C

در پارت آخر هم باید سه روش Ensemble را پیاده‌سازی کنیم و سعی کنیم که دقت را نسبت به بخش قبل بهبود بدهیم و ما از GradientBoosting, AdaBoost, ExtraTrees استفاده کرده‌ایم و برای Adaboost و Gradient Boosting از n_estimators=75 که در پارت اول محاسبه کردیم استفاده می‌کنیم و برای Extera Trees هم علاوه بر پارامتر قبلی، max_features='sqrt', max_depth=10 را هم در نظر می‌گیریم.

در نهایت هم خروجی ها را به یک دیتا فریم تبدیل می‌کنیم و سپس نمایش می‌دهیم:

```
Ensemble Models Results:
                Model  Test Accuracy
0  Gradient Boosting       0.733766
1           AdaBoost       0.740260
2        Extra Trees       0.779221
```

مدل Extera Trees دقت بهتری را نسبت به مدل اولیه RandomForest به دست آورد و در نتیجه توانستیم با استفاده از متدهای Ensemble مدلی ارائه دهیم که دقت را افزایش دهد و نسبت به مدل قبلی بهتر عمل کند.

Problem 5: More Into Clustering

A.

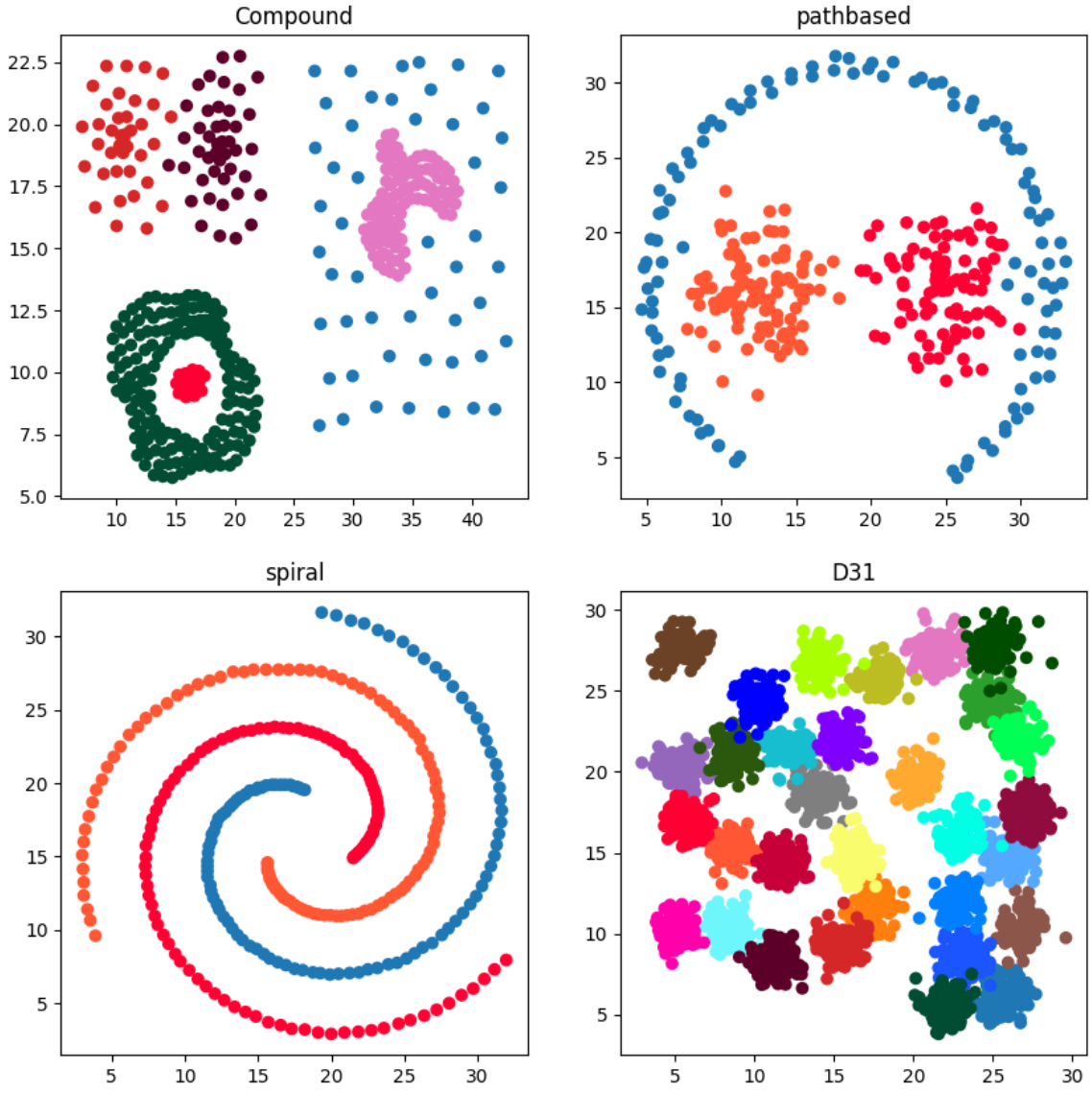
ابتدا داده ها را می‌خوانیم و فایل‌های دیتاست‌ها همانند فایل csv هستند با این تفاوت که ویژگی ها با tab از هم جدا شده‌اند. هر چهار دیتاست سه ویژگی دارند که اسم دو ویژگی اول آنها را x1 و x2 و اسم خوشه واقعی که به آن تعلق دارند را label می‌گذاریم. در نهایت به عنوان نمونه محتوای compound رو نمایش می‌دهیم:

	x1	x2	label
0	26.75	22.15	1
1	29.80	22.15	1
2	31.55	21.10	1
3	27.70	20.85	1
4	29.90	19.95	1
...	...	...	...
394	15.85	9.95	6
395	15.35	9.90	6
396	15.60	9.45	6
397	15.30	9.15	6
398	15.10	9.55	6

399 rows × 3 columns

در قسمت بعد برای اینکه هر خوشه تا جای امکان رنگ متفاوتی برای تمایز داشته باشد لیستی از ۳۱ رنگ متمایز در فرمت هگزادسیمال تعریف می‌کنیم. این رنگ‌ها برای نمایش داده‌ها در نمودارها استفاده خواهند شد؛ سپس با استفاده از تابع ListedColormap() لیست رنگ‌ها را به یک colormap سفارشی تبدیل می‌کنیم.

یک چیدمان subplot ایجاد می‌کنیم و سپس داده‌های هر دیتاست را با استفاده از نمودار scatter رسم می‌کنیم و هر خوشه با رنگ متفاوتی نشان داده شده است:



از آن‌جایی که DBSCAN الگوریتمی مبتنی بر فاصله داده‌ها است، scale کردن داده ها گام مهمی است و بدین ترتیب داده ها را scale می کنیم (نرمالایز می کنیم). ابتدا داده‌های ویژگی x1 و x2 برای هر دیتاست به کمک StandardScaler استانداردسازی می‌شوند و بعد از آن ویژگی‌های استاندارد شده به عنوان *_X (به عنوان مثال X_D31) و مقادیر لیبل‌ها به عنوان *_y (به عنوان مثال y_D31) برای هر دیتاست جداگانه ذخیره می‌شوند:

```
scaler = StandardScaler()
X_compound = scaler.fit_transform(df_compound[['x1', 'x2']])
y_compound = df_compound['label']

scaler = StandardScaler()
X_pathbased = scaler.fit_transform(df_pathbased[['x1', 'x2']])
y_pathbased = df_pathbased['label']

scaler = StandardScaler()
X_spiral = scaler.fit_transform(df_spiral[['x1', 'x2']])
y_spiral = df_spiral['label']

scaler = StandardScaler()
X_D31 = scaler.fit_transform(df_D31[['x1', 'x2']])
y_D31 = df_D31['label']
```

کلاس DBSCAN دو پارامتر مهم دارد که با توجه به دیتاست و دانش پیش‌زمینه باید تنظیم شوند: eps که شعاع همسایگی را مشخص می کند و min_samples که مینیمم نمونههایی که در همسایگی eps یک نقطه (با احتساب خودش) باید وجود داشته باشد تا به آن core point گفته شود. X که در تابع fit مقداردهی می‌شود دیتاست را ذخیره می کند:

```
class DBSCAN:
    Tabnine | Edit | Test | Explain | Document | Ask
    def __init__(self, eps=0.5, min_samples=5):
        self.eps = eps
        self.min_samples = min_samples
        self.X = None
```

تابع fit از دو تابع کمکی find_neighbors و find_connected_component استفاده می کند که در ادامه توضیح داده می‌شوند. در این تابع ابتدا دیتاست در X ذخیره می‌شود؛ سپس یک لیست labels ساخته می‌شود که در المان iام آن، خوشه‌ای که داده iام دیتاست به آن متعلق است مشخص خواهد شد. خوشه‌ها با یک عدد صحیح مشخص می‌شوند:

’-1‘: نشان‌دهنده نویز بودن داده است یعنی به خوشه‌ای تعلق ندارد. ممکن است در طی آموزش ابتدا یک داده نویز تشخیص داده شود اما بعداً به خوشه‌ای نسبت داده شود.

’0‘: نشان‌دهنده این است که داده هنوز بررسی نشده است. در انتها برچسب هیچ داده‌ای 0 نمی ماند.

سایر اعداد صحیح 1 تا k: k تعداد خوشه‌هایی است که در انتها توسط الگوریتم تعیین شده‌اند. اگر داده‌ای یکی از این اعداد صحیح را گرفت دیگر برچسب آن عوض نمی‌شود.

در ابتدا هیچ داده‌ای بررسی نشده است؛ پس همگی در لیست labels مقدار 0 دارند. همچنین متغیری به نام label تعریف می‌شود و عددی که خوشه کنونی را نشان می‌دهد را ذخیره می‌کند، ابتدا مقدار 1 دارد که نشان‌دهنده خوشه اول است.

طبق الگوریتم روی داده‌های X با متغیر p حرکت می‌کنیم؛ اگر p طبق تعریف یک core point باشد همه داده‌های قابل دسترسی از نظر چگالی (density reachable) از p که از نظر چگالی به هم متصل هستند، با برچسب خوشه جاری مشخص می‌شوند. اگر p در labels مقدار صحیح بزرگتر از صفر داشت دیگر بررسی نمی‌شود. همچنین اگر p در labels مقدار 1- داشته باشد، یعنی قبلا به عنوان نویز نشانه‌گذاری شده است یعنی کمتر min_samplesهمسایه داشته، پس نمی تواند core point باشد و بررسی نمی‌شود.

پس هنگام حرکت روی X تنها p را موقعی مورد بررسی قرار می‌دهیم که در labels مقدار صفر داشته باشد.

در صورتی که p در labels مقدار صفر داشت با استفاده از تابعfind_neighbors همسایه‌های p را می‌یابیم و در neighbors_p ذخیره می‌کنیم. در صورتی که سایز این لیست کمتر از min_samples باشد به عنوان نویز شناسایی می‌شود و برچسب آن 1- می شود (ممکن است بعداً به عنوان border point یک خوشه دیگر شناسایی شود و برچسب آن تغییر کند). اگر سایز این لیست برابر یا بیشتر از min_samples باشد، یعنی p یک core point است و با استفاده از تابع find_connected_component تمام داده‌های متعلق به خوشه جاری با برچسب label مشخص می‌شوند.. سپس label را یکی زیاد می‌کنیم تا خوشه بعدی را شناسایی کنیم. حال دوباره داده بعدی در X را می گیرد و به عنوان p مورد بررسی قرار می‌دهیم:

```
def fit(self, X):
    self.X = X

    self.labels = [0]*len(X)
    label = 1

    for p in range(len(self.X)):
        if self.labels[p] == 0:
            neighbors_p = self.find_neighbors(p)
            if len(neighbors_p) < self.min_samples:
                self.labels[p] = -1
            else:
                self.find_connected_component(p, neighbors_p, label)
                label += 1
```

تابع find_connected_component داده‌هایی از نقطه p که density reachable هستند را به عنوان label نشانه‌گذاری می‌کند. ابتدا خود p به عنوان label نشانه گذاری می‌شود. سپس روی همسایه‌های p با عنوان q حرکت می کنیم. ابتدا برچسب خود q را به label تغییر می‌دهیم، در ادامه اگر در labels:

1- نشانه‌گذاری شده باشد، یعنی کمتر از min_samples همسایه داشته و قبلا نویز شده است.

0 نشانه‌گذاری شده باشد، یعنی تا کنون بررسی نشده است. ابتدا خود q برچسب label را می گیرد؛ سپس همسایه‌های آن پیدا می‌شوند و در neighbors_q ذخیره می‌شوند . اگر سایز این لیست کمتر از min_samples بود نیازی به بررسی همسایه های q نیست، زیرا q یک border point است و ما را به سایر داده های این خوشه نمی‌رساند. اما اگر سایز neighbors_q بزرگتر مساوی min_samples بود، یعنی q خود یک core points است و باید همسایه‌هایش نیز مورد بررسی قرار گیرند؛ پس آن ها را به صف neighbors_p اضافه می کنیم:

با FIFO نقاط را مورد بررسی قرار داده‌ایم.

```
def find_connected_component(self, p, neighbors_p, label):
    self.labels[p] = label
    i = 0

    while i < len(neighbors_p):
        q = neighbors_p[i]
        if self.labels[q] == -1:
            self.labels[q] = label
        elif self.labels[q] == 0:
            self.labels[q] = label
            neighbors_q = self.find_neighbors(q)
            if len(neighbors_q) >= self.min_samples:
                neighbors_p = neighbors_p + neighbors_q
        i += 1
```

تابع find_neighbors روی همه داده های X با عنوان q حرکت می کند و فاصله اقلیدسی q با p را اندازه می‌گیرد. اگر این فاصله کمتر از eps شد q را به عنوان همسایه p در یک لیست اضافه می کند. در نهایت این لیست را به عنوان همسایه های p بر می گرداند:

```
def find_neighbors(self, p):
    neighbors = []

    for q in range(0, len(self.X)):
        if np.linalg.norm(self.X[p] - self.X[q]) < self.eps:
            neighbors.append(q)

    return neighbors
```

با استفاده از کلاس DBSCAN برای هر چهار دیتاست یک dbscan فیت می‌کنیم. مقدار eps و min_samples با آزمون و خطا به دست آمده است.

برای محاسبه تعداد خوشه ها، تابع calculate_n_clusters مقادیر منحصر به فرد درdbscan.labels را به جز مقدار 1- می شمارد:

```
def calculate_n_clusters(dbscan):
    n_clusters_ = len(set(dbscan.labels)) - (1 if -1 in dbscan.labels else 0)
    print('Estimated number of clusters: %d' % n_clusters_)
    return n_clusters_
```

تابع calculate_purity معیار purity برای خوشه‌بندی dbscan را مشخص می‌کند.

از آن‌جایی که عدد استفاده‌شده برای نشان دادن یک خوشه ممکن است در الگوریتم ما و برچسب‌های اصلی متفاوت باشد، بیشترین اشتراک هر خوشه که شناسایی کردیم را با خوشه‌های اصلی به دست می‌آوریم و آن خوشه در واقع خوشه مد نظر ما است. برای این کار اشتراک هر مقدار برچسب خودمان با هر مقدار برچسب اصلی را در یک ماتریس counts ذخیره می‌کنیم. خانه ijام این ماتریس نشان‌دهنده تعداد داده‌هایی است که به عنوان خوشه iام در الگوریتم ما مشخص شده‌اند؛ در حالی که خوشه اصلی آنها j نام‌گذاری شده بود. سپس از هر ستون این ماتریس ماکسیمم می‌گیریم و با هم جمع می‌کنیم. این مقدار برابر کل تعداد داده‌هایی است که به درستی خوشه‌بندی شده‌اند. سپس آن را بر تعداد کل داده‌ها تقسیم می‌کنیم تا purity به دست آید:

```
def calculate_purity(n_clusters_, dbscan, X, y):
    counts = np.zeros((n_clusters_, len(np.unique(y))))
    for i in range(len(dbscan.labels)):
        if dbscan.labels[i] != -1:  # Ignore noise
            counts[dbscan.labels[i]-1, y[i]-1] += 1
    purity = np.sum(np.amax(counts, axis=1)) / len(X)
    print('Purity: %.4f' % purity)
    return purity
```

برای همه دیتاست‌ها تعداد خوشه‌ها و purity را گزارش می‌کنیم و در ادامه این مقادیر را آورده‌ایم.

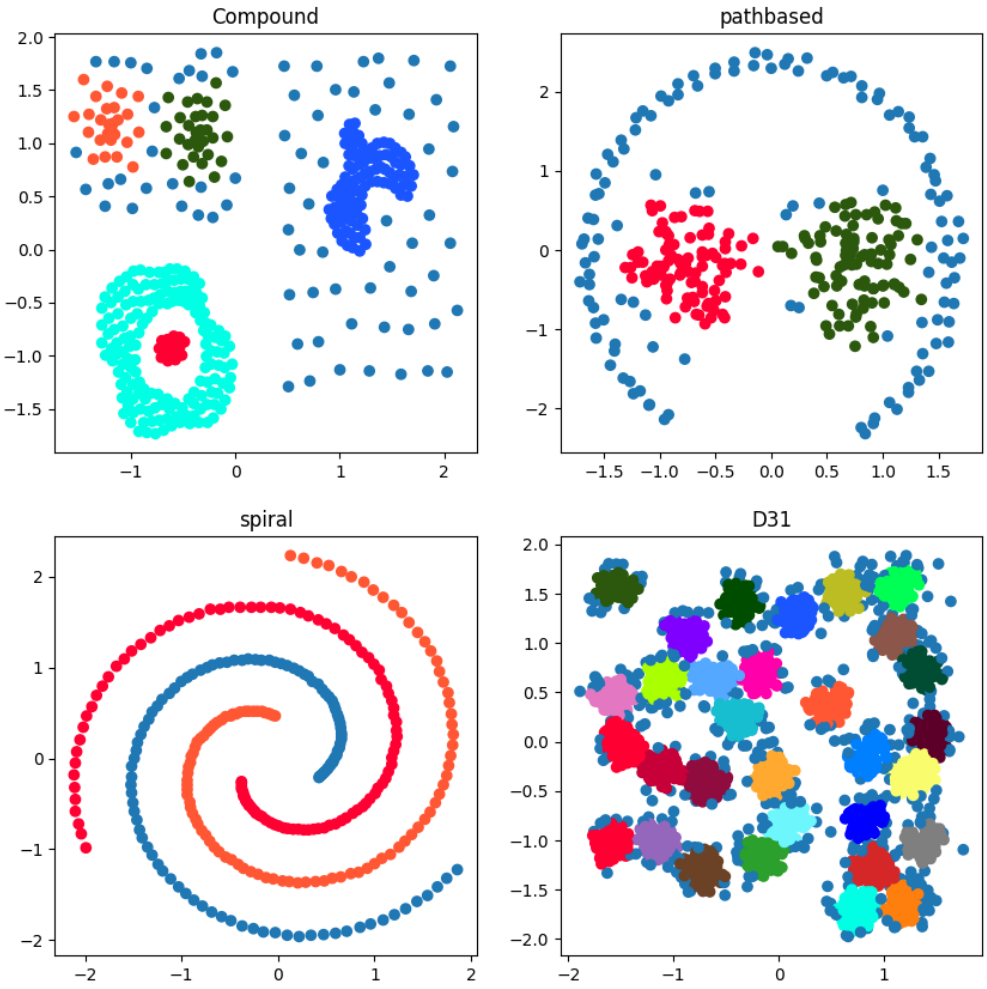
در دیتاست pathbased ضعیف‌ترین عملکرد را داشتیم؛ یکی از دلایل آن می‌تواند تفاوت چگالی خوشه به شکل path با دو خوشه دیگر باشد:

```
Estimated number of clusters: 5
Estimated number of clusters: 2
Estimated number of clusters: 3
Estimated number of clusters: 31

Purity: 0.8095
Purity: 0.5833
Purity: 1.0000
Purity: 0.8600
```

.B

در این قسمت هم نتایج خوشه‌بندی را رسم می‌کنیم و هر خوشه را با رنگ متفاوتی نشان می‌دهیم، همچنین نقاط نویز نیز با رنگ متفاوتی نمایش داده شده‌اند:



C.

الگوریتم DBSCAN به ویژگی‌های دیتاستی که روی آن اعمال می‌شود حساس است. انواع مختلف دیتاست‌ها می‌توانند اثرات متفاوتی بر عملکرد الگوریتم DBSCAN داشته باشند. در ادامه برخی از آنها ذکر شده‌اند:

- **تغییر چگالی**

- چگالی بالا: DBSCAN در خوشه‌هایی با چگالی بالا به خوبی کار می‌کند. اگر دیتاست شامل خوشه‌های متراکم باشد که توسط مناطقی با تراکم نقطه کمتر از هم جدا شده‌اند، DBSCAN در شناسایی این خوشه‌ها موثر عمل می‌کند.
- چگالی کم: در مناطق با چگالی نقطه پایین، DBSCAN ممکن است نقاط را به عنوان نویز برچسب‌گذاری کند، به خصوص اگر چگالی آنقدر کم باشد که معیارهای الگوریتم برای تشکیل خوشه‌ها را برآورده کند.

- **اشکال و اندازه‌های مختلف خوشه**

- اشکال نامنظم: DBSCAN قادر است خوشه‌هایی با اشکال نامنظم را شناسایی کند، زیرا خوشه‌ها را بر اساس اجزای متصل به چگالی به جای اشکال هندسی تعریف می‌کند.
- اندازه‌های متفاوت: الگوریتم می‌تواند خوشه‌هایی با اندازه‌های مختلف را مدیریت کند، زیرا برای تعریف یک خوشه به تعداد ثابتی از نقاط متکی نیست.

- **حساسیت به نویز**

- نقاط نویز/ پرت: DBSCAN به نقاط نویز و پرت حساس است. نقاط داده نویز یا پرت ممکن است به عنوان نویز برچسب‌گذاری شوند، اما اگر مقدار قابل توجهی نویز در دیتاست وجود داشته باشد، عملکرد الگوریتم می‌تواند تحت تاثیر قرار گیرد.

- **حساسیت پارامتر**

- پارامتر اپسیلون(eps/ε) و حداقل تعداد نقاط همسایه (min_samples): انتخاب پارامترها، به ویژه پارامتر فاصله ε و حداقل تعداد نقاط موردنیاز برای تشکیل یک منطقه متراکم (core point)، میتواند به طور قابل توجهی بر نتایج تاثیر بگذارد. این مقادیر باید بر اساس ویژگی‌های دیتاست انتخاب شوند.

- **مقیاس‌بندی ویژگی**

- مقیاس داده‌ها: عملکرد DBSCAN را می‌توان تحت تاثیر مقیاس ویژگی‌ها قرار داد. معمولا مقیاس‌بندی ویژگی‌ها برای اطمینان از این که الگوریتم نسبت به ویژگی‌هایی با مقیاس‌های بزرگتر سوگیری ندارد، سودمند است.

- **ابعاد**

- داده‌های با ابعاد بالا: در فضاهای با ابعاد بالا، مفهوم فاصله کمتر شهودی می‌شود و "معضل ابعاد" می‌تواند بر عملکرد DBSCAN تاثیر بگذارد. پیش‌پردازش یا تکنیک‌های کاهش بُعد ممکن است در چنین مواردی مفید باشد.

- **چگالی یکنواخت**

- تراکم یکنواخت: اگر دیتاست دارای تراکم نقاط نسبتا یکنواخت در کل فضا باشد، DBSCAN ممکن است به اندازه کافی موثر نباشد، زیرا برای شناسایی خوشه‌ها بر مفهوم تنوع چگالی متکی است.

به طور خلاصه، عملکرد الگوریتم DBSCAN تحت تاثیر چگالی و توزیع نقاط داده، شکل و اندازه خوشه‌ها، وجود نویز و انتخاب پارامترها و ... است. درک ویژگی‌های دیتاست و تنظیم پارامترهای الگوریتم براساس آن برای دستیابی به نتایج بهینه مهم است.

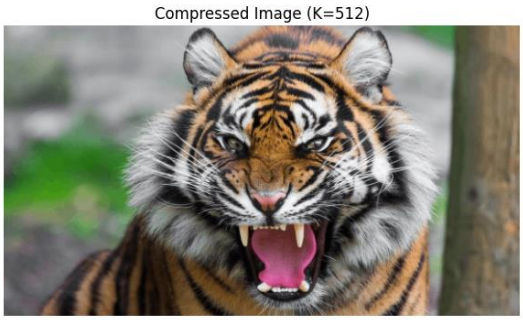
Problem 6: Image Compression

فشرده سازی عکس با KMeans، با کمک کاهش تعداد رنگ‌ها سایز عکس را کاهش می‌دهد. در ابتدا عکس را با کتابخانه OpenCV باز می‌کنیم. از آن‌جایی که فرمت اولیه عکس به صورت BGR است پس آن را به فرمتRGB تبدیل می‌کنیم. ابعاد عکس اولیه به صورت (328, 584, 3) است که دو مقدار اولیه عرض و طول پیکسل و مقدار سوم مقادیر RGB هر پیکسل است. با تغییر شکل دادن ابعاد عکس به فرمی که هر سطر نشان دهنده یک پیکسل و مقادیر RGB پیکسل ابعاد ویژگی‌های دیتاست باشد، آنها را به فرم ورودی مناسب الگوریتم تبدیل می‌کنیم. ابعاد جدید به صورت (191552, 3) است. در ادامه با الگوریتم KMeans از کتابخانه scikit-learn هر پیکسل را خوشه‌بندی می‌کنیم (مقادیر انتخاب شده برای مقدار تعداد خوشه یا k برابر با 4، 16، 64، 128، 256، 512 بوده است و می‌توان مقادیر دیگری را نیز استفاده نمود.) و سپس برای هر خوشه مقدار نقطه مرکزی (centroid) را به دست می‌آوریم.

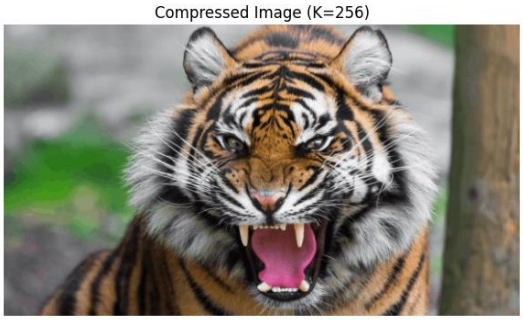
تعداد خوشه‌ها نشان دهنده تعداد رنگ‌هایی است که می‌خواهیم عکس پس از فشرده سازی داشته باشد.
همچنین داده‌هایی که در یک خوشه قرار می‌گیرند مقادیر RGB نزدیک به هم و در نتیجه رنگ مشابه دارند. در ادامه به ازای هر مقدار پیکسل درون هر خوشه، مقدار نقطه مرکزی آن خوشه را قرار می‌دهیم. این یعنی هر یک از داده‌ها پس از فشرده سازی رنگی مشابه مرکز خوشه خود خواهند داشت. در آخر دیتاست را به فرمت اولیه بر می‌گردانیم و آن را نمایش می‌دهیم.

شکل نهایی به دست آمده و عکس اولیه با مقادیر خوشه‌بندی گفته شده در ادامه قرار دارد:

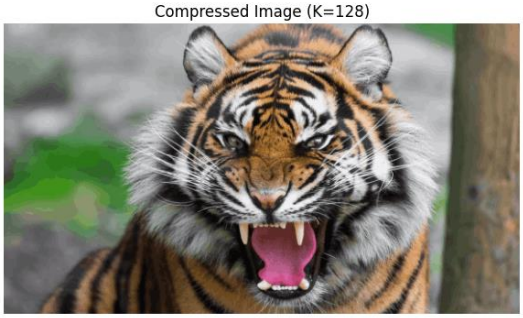
- **K= 512:**



- **K= 256:**



- **K= 128:**



- **K= 64:**

- **K= 16:**

- **K= 4:**

«. . . دی‌ماه ۱۴۰۳ . . .»