توجه: برخی توضیحات ابتدایی در نوت بوک داده شده است.

درخت تصمیم [decision-tree-and-more/jupyter-titanic.ipynb]

مرحله اول| بررسی داده‌ها و انتخاب فیچرها و تبدیل فیچرهای غیرعددی به عددی

آنالیز داده‌ها و رسم نمودار

با توجه به نمودار رسم شده در **In [5]** با چشم مشخص می‌شود که زن بودن، در بقا خیلی موثر است؛ پس قطعا فیچر sex برای درخت تصمیم انتخاب می‌شود.

در **In [6]**، نمودار مربوط به مقایسه با توجه به سنین مختلف رسم شده که در همه‌ی رنج‌های سنی نزدیک پنجاه درصد است. به جز در حدود 20 تا 30 سالگی و پیرمردان و زنان لب گور. همانطور دیده می‌شود (در نوت بوک) با شرط پیر بودن، درخت تصمیم می‌تواند تصمیم قاطعی بگیرند.

این دو ویژگی بر اساس "اول زن‌ها و بچه‌ها" بررسی شدند. ویژگی بعدی که در **In [7]** نمودار آن رسم شده، بررسی می‌کند که آیا کسی که پول بیشتری داده، لیاقت بیشتری برای زندگی دارد یا نه؟ در این نمودار تجمعی شیب خط مربوط به کشته شدگان در ابتدا خیلی زیاد است، این نشان می‌دهد که افراد زیادی که کشته شده-اند (بیش از چهارصد نفر)، پول خیلی کمی برای بلیط داده‌اند. علاوه بر این کسانی که زنده مانده‌اند مجموع پولشان به پانصد رسیده و کسانی که کشته‌شده اند، به سیصد هم نرسیده، این درحالی است که تعداد زنده ماندگان حدود پانصد و کشته شدگان، سیصد است.

انتخاب ویژگی‌ها

در مرحله‌ی بعد در **In [8]** با ماتریس وابستگی بررسی کردم که آیا Pclass و Fare وابستگی زیادی دارند، تا یکی از آن‌ها را حذف کنم. نتیجه -0.57 بود، که به نظر من کافی نیست، پس هیچ فیچری در این مرحله حذف نشد.

درباره‌ی سایر ویژگی‌ها در نوت بوک توضیحات داده شده است.

پیش‌پردازش

در مرحله‌ی بعدی، شروع به حذف داده‌های NaN کردم. برای سن از میانگین استفاده کردم و برای Cabin از No room.

همچنین برای Embarked با کمک mode گرفتن از این ستون، مقادیر خالی، جایگزین شده‌اند. در نهایت در test.csv یک سطر از Fare خالی بود، که با میانگین مرتبط، جایگزین شد.

در **In [13]** شروع به تبدیل داده‌های عددی به غیر عددی کردم که به شکل زیر می‌باشد:

فیچر	متغیرهای قبلی	متغیرهای جدید
Sex	Female	-1
	male	1
Embarked	S	0
	C	1
	Q	2

دو ویژگی SibSp و Parch مربوط به خانواده هستند و می‌توانند با هم جمع شوند، برای این یک ویژگی FamSize تعریف شده که مجموع این دو فیچر و یک (خود طرف) است.

در نهایت با دسته‌بندی Cabin با توجه به حرف اولشان، می‌توان اطلاعات نسبی بدست آورد این مرحله به پایان می‌رسد. این دسته‌بندی در `In [16]` انجام شده که در آن کابین‌ها با شروع A تا G جایگزین 1 تا 7 می‌شوند و No room ها با صفر جایگزین می‌شوند. اسم این فیچر جدید cCabin است.

نتایج نهایی در `Out [16]` قابل مشاهده است.

مرحله دوم| آموزش، تست و توییک

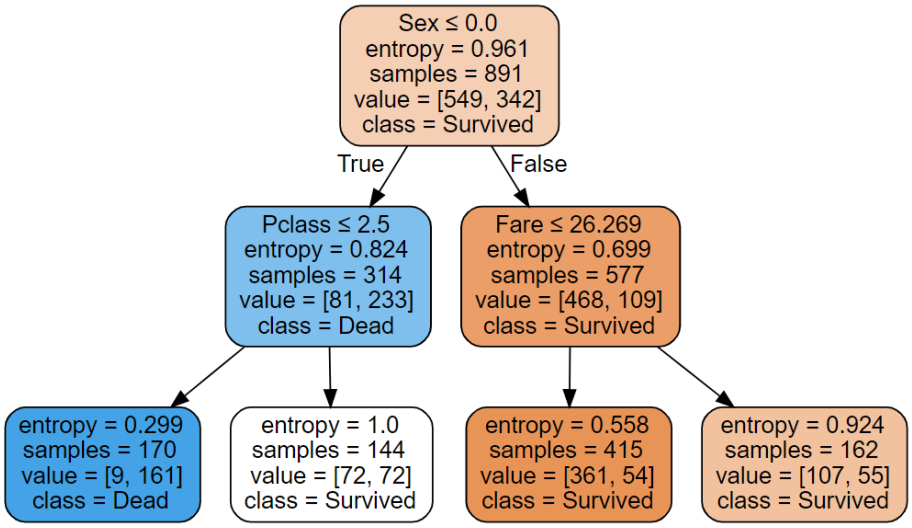
آموزش

در اینجا، داده‌ها را با نسبت ۸۰ به ۲۰ به داده‌های آموزش و تست اختصاص دادم، تا بتوانم با توجه به نتایج، خروجی نهایی را بهبود بدهم. این کار با کمک shuffle انجام شده است.

فیچرهای انتخاب شده بصورت زیر می‌باشند

- Age
- Sex
- Fare
- Pclass
- Embarked
- FamSize
- cCabin

برنامه با اجرا، درخت را نمایش می‌دهد و همینطور فایل پی دی اف آن را خروجی می‌دهد. برای نمونه درخت با عمق ۲ و با استفاده از انتروپی به شکل زیر است



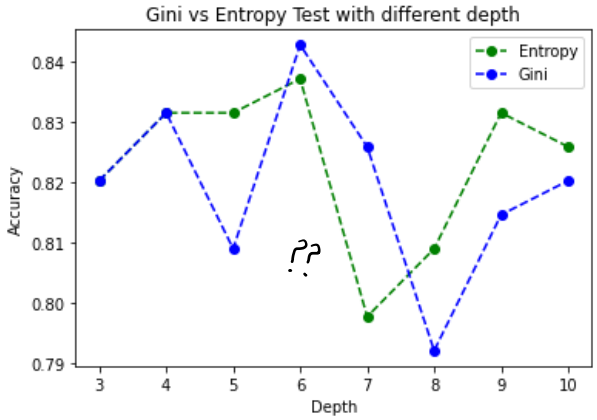
با تغییرات عمق و معیار تقسیم نتایج زیر حاصل شد

عمق	روش	میانگین آموزش	میانگین تست
۴	Entropy	۸۲	۸۲
۵	Gini	۸۳	۸۲
	Entropy	۸۳	۸۰
۶	Entropy	۸۴	۸۳
	Gini	۸۵	۸۲

در نهایت با بررسی میانگین‌ها، عمق ۶ با روش gini انتخاب شد. در حالت‌های خاصی، تست نتیجه‌ی زیر ۸۰ هم می‌دهد ولی در حالت کلی، بیش از ۸۰ است و همیشه بیش از ۷۵ است. تست‌های Kaggle هم نتوانست از نمره ۰.۷۸ بیشتر شود.

در نهایت در `In [24]` تاثیر عمق در معیارهای مختلف بررسی شده (که نسبتا رندوم هستند) و پس از آن تخمین نهایی از test.csv در فایل `output.csv` ذخیره می‌شود.

این نمودار با shuffle یکسان، random state یکسان و پنجره‌های یکسان رسم شده.



جنگل تصادفی [decision-tree-and-more/jupyter-titanic.ipynb]

در **In [15]** برای بهبود نتایج، کلاس شخصیتی را به فیچرها اضافه کردم. این کار تاثیر نسبی در نتایج نهایی داشت. مقایسه‌هایی که در این تمرین انجام شده با تمرین قبلی است و برای این مقایسه‌ها **In [15]** را اجرا نکردم تا مقایسه‌ها در شرایط یکسان باشد.

در **In [24]** جنگل تصادفی بهینه را با روش gini و با 500 کلاس‌بند تشکیل دادم. مقدار n_jobs=-1 تعیین می‌کند که عملیات بصورت موازی انجام شود تا به سرعت بیشتری برسیم. ساده بودن درخت‌های تصمیم موجود در این جنگل با min_samples_split = 10 حاصل شده که باعث می‌شود نودها با کمتر از 10 داده را بیشتر از این تقسیم نکنیم.

In [25] و **In [26]** به ترتیب دقت داده‌های آموزش و تست را چاپ می‌کنند.

در تمرین قبل، بهترین پاسخ برای درخت تصمیم، در حالت آموزش دقت ۸۷ و در حالت تست دقت ۸۱ بدست آمد. این مقادیر با پارامتر عمق ۶ و روش gini بدست آمدند. همانطور که در تمرین قبل ذکر شد، در حالت‌های خاصی، درخت تصمیم نتیجه‌ی زیر ۸۰ می‌دهد ولی هیچوقت کمتر از ۷۵ نیست و میانگین ۸۰ دارد.

در این تمرین بهترین نتیجه‌ی جنگل تصادفی، در حالت آموزش ۸۹ و در حالت تست ۸۴ شد. این حالت با ۵۰۰ کلاس‌بند با روش gini و عمق متغیر بر اساس تعداد اعضای هر نود، بدست آمد. میانگین جنگل تصادفی برای آموزش ۸۷ و برای تست ۸۳ بدست آمد.

نمودار رسم شده در **Out [27]** نشان‌دهنده‌ی اهمیت فیچرها در جداسازی داده‌های هستند.

جدول رسم شده در **Out [28]** مقایسه‌ی زمان انجام را برای درخت تصمیم و جنگل تصادفی انجام می‌دهد. برای مقایسه‌ی سرعت، با استفاده از کتابخانه زمان پایتون، به نتیجه‌ی زیر رسیدم:

زمان تست میلی‌ثانیه	زمان آموزش میلی‌ثانیه	
0.09	4.00	درخت تصمیم
104.13	444.20	جنگل تصادفی

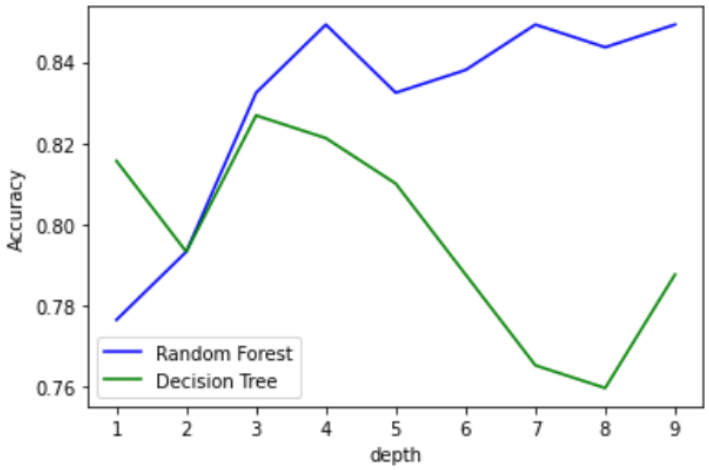
زمان‌های بدست آمده، زمانی است که در آن بهترین نتیجه‌ی درخت تصمیم و جنگل تصادفی بدست آمد.

با تغییر مقادیر عمق و روش جداسازی در **In [30]**، جدول زیر میانگین دقت‌ها را نشان می‌دهد.

تست	آموزش	روش	عمق
80	83	Gini	3
79	83	Entropy	
84	86	Gini	6

	entropy	86	84

برای مقایسه‌ی کلی بین این دو روش، دقت تست را بر اساس عمق‌های مختلف تست کردم و به نتیجه‌ی زیر رسیدم:



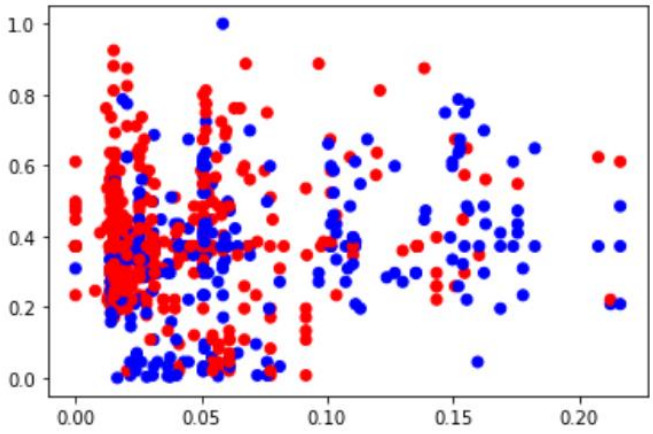
همانطور که مشخص است، به طور کلی جنگل تصادفی با صد کلاس‌بند از یک درخت تصمیم، بهتر عمل می-کند.

ماشین بردار پشتیبان [decision-tree-and-more/jupyter-titanic.ipynb]

برای پیش‌پردازش در **In [31]** با استفاده از نرمال‌سازی کتابخانه sklearn، داده‌ها را برای آموزش آماده کردم. در اینجا از روش minmax استفاده شده است.

مدل با کرنل خطی در **In [32]** آموزش داده شده است. دقت میانگین برای داده‌های آموزشی برابر 78 درصد بود و برای داده‌های تست به دقت 79 رسیدم. همانطور که مشخص است، SVM از روش جنگل تصادفی و درخت تصمیم بدتر عمل کرده است.

در **In [34]** اسکترپلات براساس age و fare رسم شده و مشخص است که با توجه به اینکه ابعاد بالا می‌باشد و پیچیدگی خطی نیست، نمی‌توان داده‌ها را بصورت خطی از هم جدا کرد و به همین دلیل است که کرنل خطی جواب نمی‌دهد و نیاز به جداساز پیچیده‌تری داریم.



در **In [33]** با کمک کرنل rbf و تنظیماتی مثل C=1 و gamma='scale' به نتایج به مراتب بهتری رسیدیم. به طور میانگین SVM با این کرنل به میانگین دقت آموزشی 81 و تست 83 می‌رسیم که از کرنل خطی بهتر است و تقریبا به دقت جنگل تصادفی می‌رسد.

دلیل برتری کرنل rbf این است که پیچیده‌تر است و در واقع داده‌ها بصورت خطی قابل جدا شدن نیستند. به عبارت بهتر مدل با کرنل خطی bias دارد و underfit می‌شود ولی rbf این مشکل را ندارد و با انعطاف بیشتری می‌تواند جداسازهای بهتری بسازد.

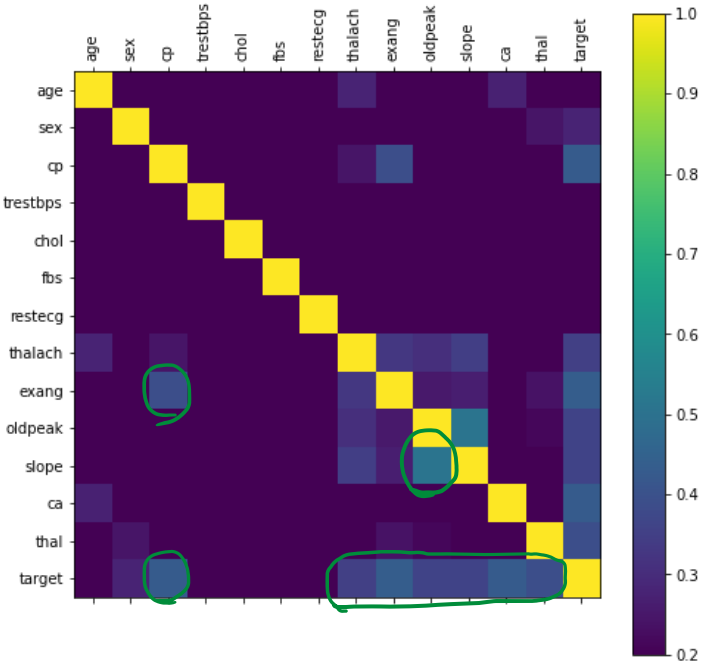
بیز ساده و k نزدیکترین همسایه `[knn-nb/jupyter-heart-disease.ipynb]`

مرحله اول| تحلیل و آماده‌سازی

تحلیل

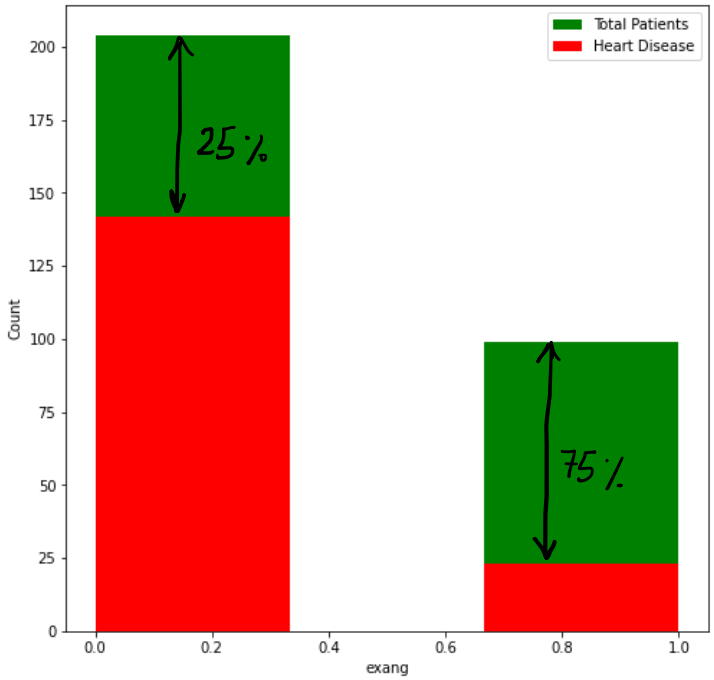
در ابتدا بررسی شده که تعداد داده‌ها در کلاس‌های مختلف، متناسب باشند تا در Naïve Bayes به مشکل برنخوریم. اینکار در **In [7]** انجام شده و نتیجه نشان می‌دهد که توزیع داده‌ها نسبتا یکسان است.

در **In [9]** ماتریس وابستگی فیچرها رسم شده است. در اینجا برای وضوح، وابستگی‌های کوچکتر است 0.2 را همرنگ صفر در نظر گرفته ام.

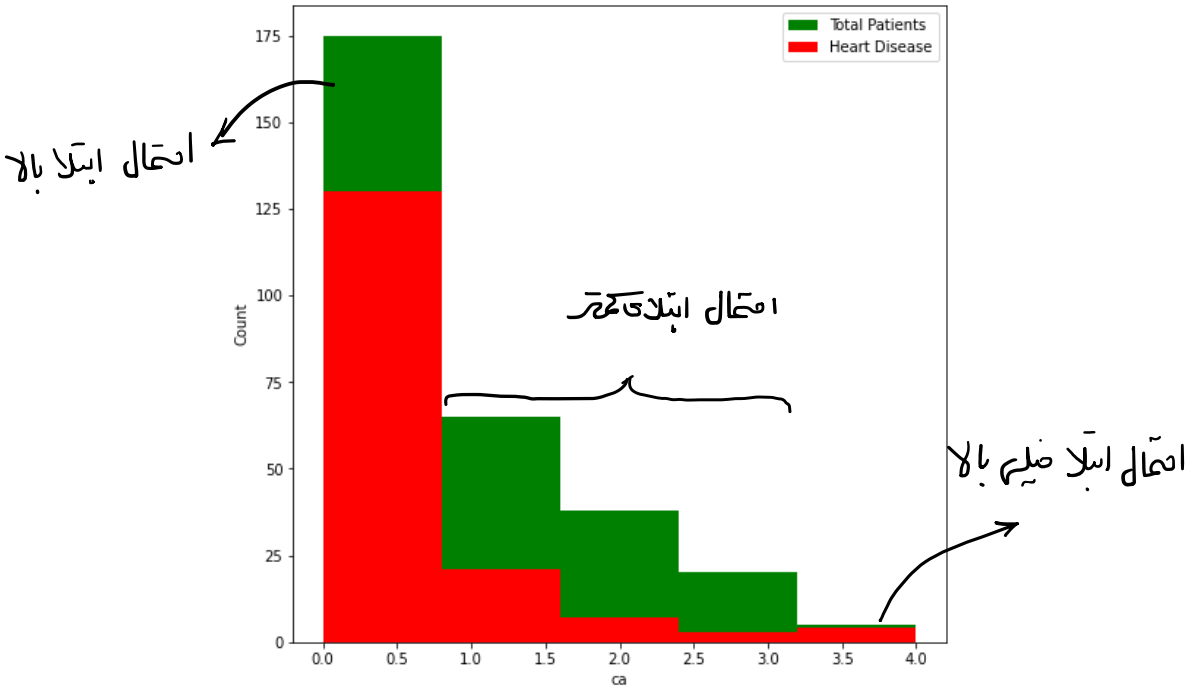


همانطور که در گوشه مشخص است، thal، ca، exang و cp با target ارتباط بیشتری دارند. در وسط تصویر هم slope با oldpeak وابستگی دارد که می‌توانیم یکی از آنها را حذف کنیم. در اینجا بدلیل محدود بودن داده-های آموزشی و همچنین بعد آنها curse of dimensionality دامنگیر ما نمی‌شود (علاوه بر این دانستن ارتباط این فیچرها نیاز به علم پزشکی دارد)، پس از feature selection می‌گذریم.

نمودارهای وابستگی ca، cp و exang در **In [10]** تا **In [12]** آمده.



در این شکل دیده می‌شود که exang داشتن، احتمال بیماری قلبی را کاهش می‌دهد و برعکس.



این شکل هم نشان دهنده این است که ca خیلی کم و خیلی زیاد، احتمال ابتلا به بیماری قبلی را افزایش می‌دهد.

پیش پردازش

همانطور که در **Out [15]** دیده می‌شود، در دیتاست داده NaN وجود ندارد، پس به مرحله‌ی بعدی می‌رویم.

در **In [16]** برای پیش پردازش، از آنجا که scale در نتیجه kNN موثر است، نرمال‌سازی صورت گرفته. این کار به کمک StandardScalar انجام شده. نتیجه‌ی نرمال‌سازی در **Out [16]** قابل مشاهده است.

قبل از شروع آموزش تقسیم داده‌ها صورت می‌گیرد، با شمارش نسبی داده‌های دو دسته در **In [19]** و **In [20]** از تابع شافل اطمینان پیدا می‌کنیم.

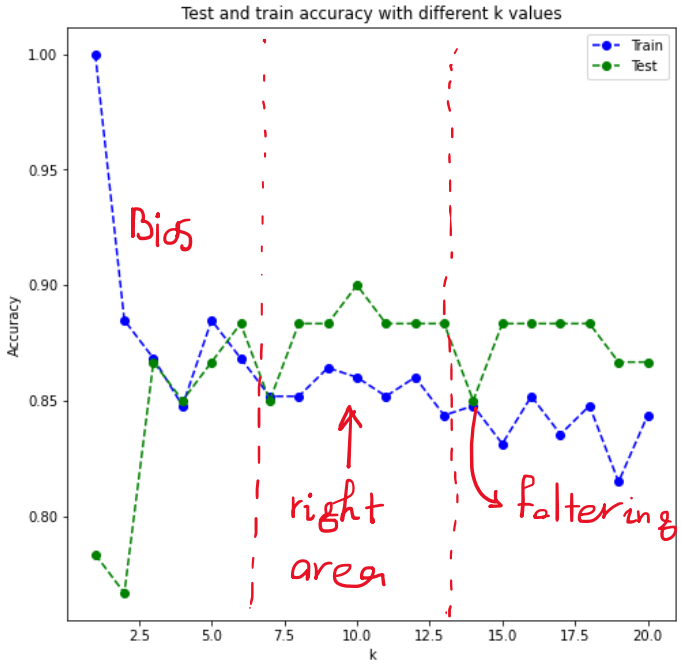
مرحله دوم| ساخت و آموزش

kNN

در روش kNN با استفاده از k = 12 به دقت 0.86 برای داده‌های آموزشی و 0.88 برای داده‌های تست رسیدم.

در **Out [22]** تنظیمات kNN خروجی داده شده است.

نمودار دقت برای داده‌های آموزشی و تست، با k ها مختلف به شکل زیر می‌باشد:



به نظر k بین 6 و 12 برای این مسئله مناسب می‌باشد.

Naïve Bayes

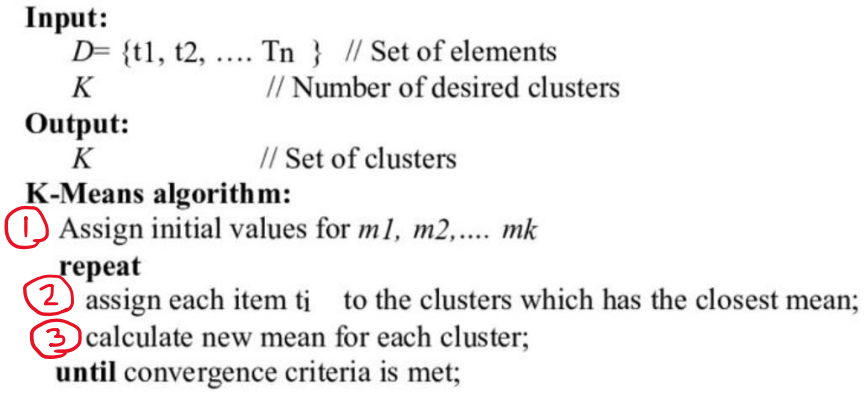
مانند kNN با استفاده از کتابخانه sklearn و با تنظیمات اولیه **Out [27]** به نتیجه دقت برای داده‌های آموزشی 0.83 و 0.90 برای داده‌های تست می‌رسم.

جدول مقایسه این دو روش در `Out [30]` به شکل زیر می‌باشد:

	Train	Test
kNN	86%	88%
Naïve Bayes	83%	90%

پیاده‌سازی kmeans دو بعدی [kmeans/jupyter-image-compression.ipynb]

مرحله اول| پیاده‌سازی



برای الگوریتم بالا، سه تابع تعریف شده است. در **In [2]** تابع pick_centroids برای مرحله‌ی یک است که با گرفتن کل نقاط و مقدار k، k مرکز اولیه بصورت رندوم از نقاط انتخاب می‌کند و پس می‌دهد.

در **In [3]** مرحله‌ی دو انجام شده که نزدیکترین مرکز به هر نقطه را حساب کرده و در یک numpy array شامل اندیس مرکز مرتبط بر می‌گرداند.

در **In [4]** مراکز آپدیت می‌شوند که مربوط به مرحله‌ی سه است. این کار با میانگین گرفتن از داده‌ها در خوشه‌ی مربوط انجام می‌شود.

الگوریتم kmeans در **In [5]** پیاده‌سازی شده است. این تابع داده‌ها و مقدار k و تعداد iteration ها تا پایان را از ورودی گرفته و شبه کد بالا را اجرا می‌کند

مرحله دوم| اجرای روی دیتاست

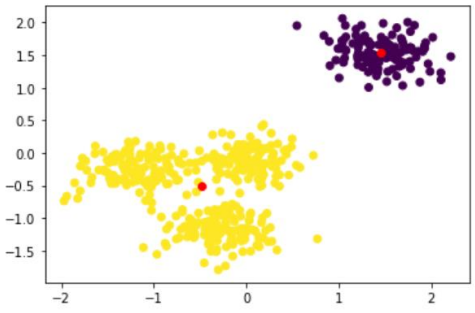
الف | در **In [7]** دو دیتاست داده شده، لود شده اند و scatter plot آن‌ها در **Out [7]** قابل مشاهده است.

*از آنجایی که در این تمرین با داده دو بعدی کار می‌کنیم، استفاده از pandas در محاسبه فاصله‌ها سربار خیلی زیادی دارد، پس بهتر است از numpy استفاده کنیم. علاوه بر این تصویر سوال بعد هم بصورت numpy لود می‌شود.

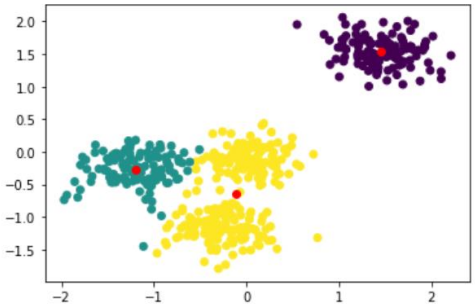
ب و ج | برای k = 2 در **In [9]** تابع kmeans نتایج نهایی را خروجی می‌دهد (کلاستر ها و مراکزشان). در **Out [10]** خروجی scatter plot رسم شده و در **Out [11]** نتایج خطای هر کلاستر و میانگین خطای کل خوشه‌بندی چاپ شده است.

این کار برای k = 3 و k = 4 هم انجام شده‌اند.

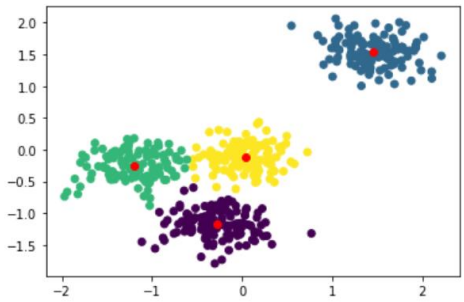
نتایج برای شروع ابتدایی ایده‌آل در شکل زیر قابل مشاهده هستند



Cluster 1 error: 0.31
Cluster 2 error: 0.74
Clustering Error: 0.53

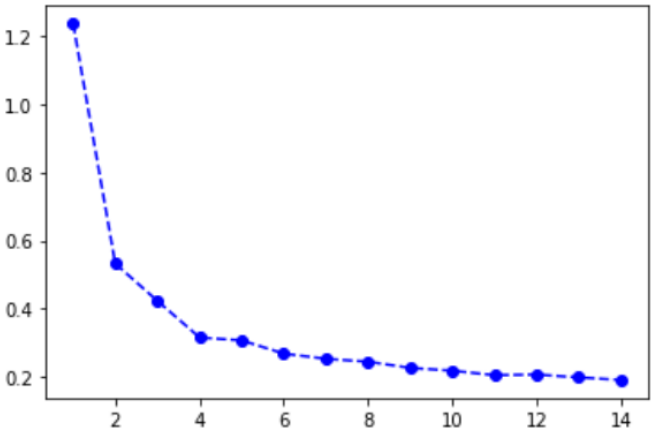


Cluster 1 error: 0.31
Cluster 2 error: 0.33
Cluster 3 error: 0.61
Clustering Error: 0.42



Cluster 1 error: 0.33
Cluster 2 error: 0.31
Cluster 3 error: 0.32
Cluster 4 error: 0.28
Clustering Error: 0.31

د و هـ | در `In [18]` با مقادیر مختلف k، خطای خوشه‌بندی بررسی شده و نمودار مربوطه در `Out [18]` قابل مشاهده است.

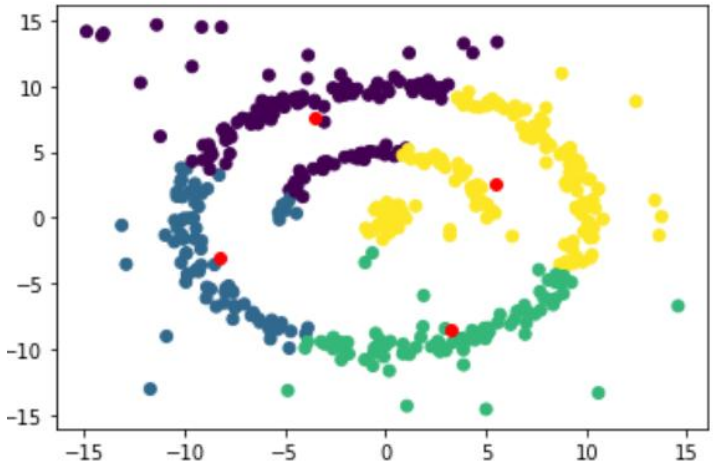


همانطور که دیده می‌شود بعد از k = 4 تقریبا بهبودی نداشتیم و به elbow رسیده‌ایم. پس در این صورت k = 4 برای k انتخاب ایده‌آلی است.

ی | برای dataset2.csv با kmeans به نتیجه‌ی `Out [19]` میرسیم که قابل قبول نیست. خطای خوشه‌بندی برای این دیتاست بیشتر از 8 است (با k = 4). البته این خطا با مرحله‌ی قبلی قابل مقایسه نیست.

دلیل این اتفاق این است که kmeans بر اساس فاصله خوشه‌بندی می‌کند و خوشه‌هایش شکل دایره‌ای دارند. در dataset2 خوشه‌ها به شکل پیوسته هستند و می‌توانند با روش‌های density based یا nearest neighbor به جواب برسند.

لازم به ذکر است الگوریتم kmeans به طور کلی برای خوشه‌های با تراکم متفاوت، سایز متفاوت و شکل غیر دایره‌ای به مشکل بر می‌خورد.


15
10
5
0
−5
−10
−15
−15
−10
−5
0
5
10
15

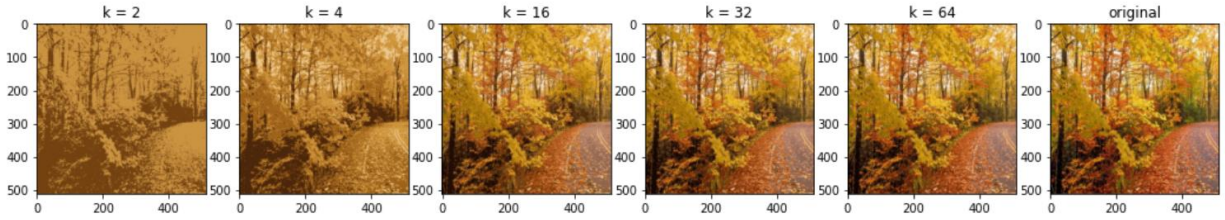
کاهش رنگ عکس [kmeans/jupyter-image-compression.ipynb -> part 3]

در **In [24]** برای هر یک از اعضای لیست colors، kmeans را انجام می‌دهیم. مراحل کلی این کار برای یک k خاص به شکل زیر است:

```
1  read image as img
2  reshape img into 2d array
3  run kmeans on reshaped image: result of kmeans is a list of k colors and
   index of what each pixel belongs to
4  replace each pixel with its' center (its' representing color)
5  reshape the result into original size
```

دو بعدی کردن تصویر به این شکل انجام شده که بجای طول و عرض پیکسل، از یک ایندکس خطی استفاده شده است؛ چرا که برای کاهش رنگ نیازی به مکان پیکسل‌های نزدیک بهم نداریم و "مکان" و نزدیکی در اینجا نزدیکی رنگ‌هاست. این درحالی است که در مسائلی مثل image segmentation مکان هر پیکسل اهمیت زیادی دارد.

در نهایت خروجی تصاویر در **Out [25]** نمایش داده شده است.



شناسایی نقاط پرتراکم کرونایی `[dbscan/jupyter-covid.ipynb]`

مرحله اول | نمایش داده‌های خام

با کمک دستورات ذکر شده در صورت تمرین، نقاط ابتدایی در `In [1]` تنظیم شده اند تا نقشه به ایران اشاره کند. سپس با خواندن `covid.csv`، نقاط در لیست dataset ذخیره شده‌اند.

در نهایت برای نمایش، هر داده با یک نقطه‌ی قرمز در آن مکان نمایش داده شده است.

مرحله دوم| پیاده‌سازی DBSCAN

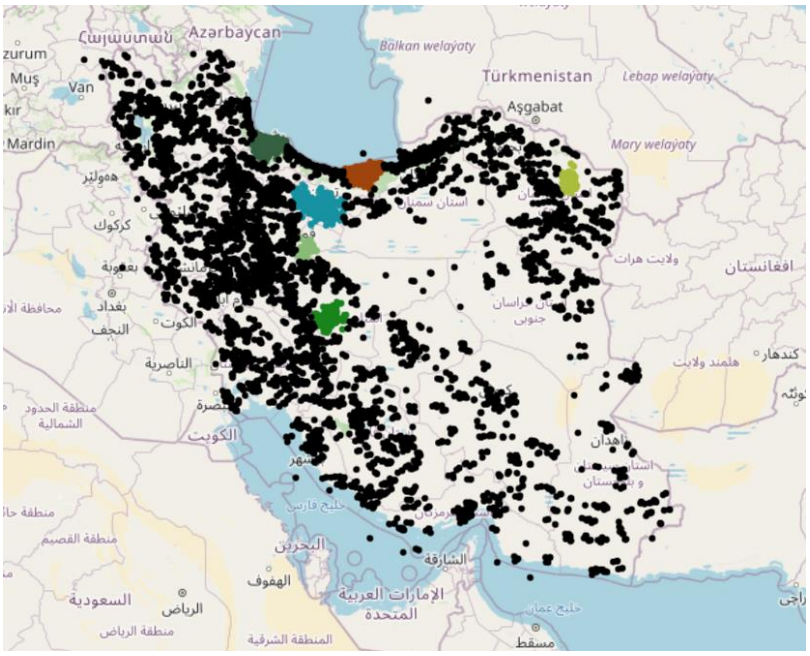
در `In [2]` الگوریتم DBSCAN با پارامترهای eps = 0.3 و minPts = 10 اجرا شده است و به 38 کلاستر و 341 داده پرت رسیدیم.

labels شامل برچسب مجازی(برای مشخص کردن داده‌های هم کلاستر) داده هاست. تعداد داده‌های پرت، تعداد داده‌هایی است که label برابر 1- دارند و سایز label های یونیک منهای یک (داده‌های پرت) تعداد کلاسترهای پیدا شده را نمایش می‌دهد.

مرحله سوم| پیدا کردن شهرهای پرتراکم

برای پیدا کردن ۶ شهر پرتراکم، با تغییر eps و minPts به مقدار ۰.۳ برای eps و ۳۳۰ برای minPts رسیدم. ۶ شهر پرتراکم شامل تهران، قم، مشهد، مازندران، رشت و یک شهر در جنوب قم (فکر می کنم کرمان باشد) است.

در نهایت خروجی نقشه در `Out [4]` نمایش داده شده است.



کاهش ابعاد بردار بازنمایی [pca/colab-gloves.ipynb]

مرحله اول | دانلود و خواندن فایل داده‌ها

این مرحله با `wget!` و استفاده از کتابخانه zipfile انجام شد و فایل‌ها ذخیره شدند. پس از دانلود و unpack کردن، باید داده‌ها را وارد پایتون بکنیم. برای این کار در `In [3]`، تک تک خطوط را خوانده و کلمات و بردار-های بازنمایی متناظر را در دو لیست ذخیره می‌کنیم. لیست شامل کلمات embedding_words و لیست شامل بردارهای بازنمایی embedding_vectors نام دارند. برای سادگی لیست دوم را به آرایه numpy تبدیل می-کنیم.

مرحله دوم | پیاده‌سازی PCA

تابع pca در `In [4]` تعریف شده در این مرحله با ورودی embedding_vectors و تعداد pc های موردنیاز، مهم‌ترین بردارهای اساسی را بازمیگرداند. برای این کار از فرایند زیر پیروی می‌شود:

```
1  normalize data
2  calculate covariance matrix
3  extract biggest r eigon vectors
4  map data onto biggest r eigon vectors
```

در نهایت با اجرای تابع بالا با ورودی embedding_vectors و r=2 خروجی مورد نظر، یعنی داده‌ها با ابعاد کمتر بدست می‌آیند.

مرحله سوم | نمایش داده‌های کاهش بعد داده شده

در این مرحله در `In [7]` با انتخاب کلمات قابل تمایز، به نتیجه‌ی زیر رسیدیم: