30-Days Machine Learning Project Challenge

A COMPILATION OF ALL MY ARTICLES

ABBAS ALI

30 Days Machine Learning Project Challenge

Abbas Ali

PREFACE

Hey there!

On the month of March 2024, I decided to take on a 30-Day Machine Learning Project Challenge. During the challenge I did some simple beginner level Machine Learning Projects consistently for 30 days, also after finishing the project each day I wrote an article on Medium explaining the project I did to the best of my ability.

After finishing my challenge, I had a very good grasp of Machine Learning.

Not only that many people got interested in my challenge. Every day at least 150 people visit my Medium page to read the articles I wrote during my challenge.

Many even started taking the challenge. So, I just thought of making their work easy by writing this E-book and publishing it for free.

In this E-book, I have compiled all the articles I wrote during my 30-Day Challenge.

If you been following my challenge, you might have known that the first 15 projects are supervised learning projects and the second 15 projects are unsupervised learning projects.

You can go through each project one by one.

No problem if you are not able to understand something. When I got started, I was not able to understand a few concepts but that's totally fine.

As a beginner you should only try to learn consistently.

Go one day at a time.

Enjoy the book.

Table Of Content

Supervised Learning Projects

Unsupervised Learning Projects

Day-16: Iris Dataset Clustering 🌷

Day-17: Mall Customers Clustering 🏬

Day-18: KMeans, DBSCAN, and PCA 🫧

Day-19: Apriori Algorithm For Market Basket Analysis 🍞 🥛 🧈

Day-20: Anomaly Detection Using Isolation Forest😂 😂 😱 😂

Day-21: Topic Modelling Using LDA 📄

Day-22: Dimensionality Reduction Using t-SNE and UMAP 🍞 ➡ 🥪

Day-23: Anomaly Detection Using Isolation Forest and Local Outlier Factor 1️⃣ 1️⃣ 0️⃣ 1️⃣ 1️⃣

Day-24: Kernel Density Estimation Using Seaborn 📊

Day-25: Customer Segmentation using Hierarchical Clustering 🏪 🛍️

Day-26: Anomaly Detection Using OneClassSVM😇 😇 😈 😇 😇

Day-27: Clustering Using GMM, KMeans, and DBSCAN🦹 🦹 🦹 🦹 🦹 🦹

Day-28: Association Rule Learning Using FP-Growth 🍞 🧈 🥛

Day-29: Clustering Using Bayesian Gaussian Mixture 🌻 🌼 🌷

Day-30: Dimensionality Reduction Using LDA and SVD ✊

Day-1: Customer Spending Score Prediction 🛍️

As it was day one I was searching for Datasets to work with, and then I found a really simple dataset on my laptop itself. It looks something like this👇

```
CustomerID, Gender, Age, Annual Income (k$), Spending Score (1-100)
0, 1, Male, 19, 15, 39
1, 2, Male, 21, 15, 81
2, 3, Female, 20, 16, 6
3, 4, Female, 23, 16, 77
4, 5, Female, 31, 17, 40
```

The goal is to predict the "Spending Score" using the other columns.

I should tell you this, I was not at all motivated to start the coding I was procrastinating, and then I said to myself "Do it even when you don't feel like it".

Then I started coding, After like 3 minutes I was on a flow that felt amazing.

So, I started by importing the libraries as always.

```
import numpy as np
import pandas as pd
```

```
import matplotlib.pyplot as plt
import seaborn as sns
```

I don't know why I import matplotlib and seaborn 😅.

Then I loaded the dataset.

```
df = pd.read_csv('Mall_Customers.csv')
```

After loading the dataset, I used different methods to understand the dataset more clearly, the methods I used are👇

```
df.head()

df.shape

df.isnull().any()
```

Then, I didn't like the names of a few columns so I changed them. Initially, I had no idea how to do that, but after a Google search, I learned it.

```
df.rename(columns = {'Spending Score (1-100)':'Spending Score'}, inplace=True)

df.rename(columns={"Annual Income (k$)": 'Annual Income'}, inplace=True)
```

As there were no null values in my dataset, I started separating the dataset into features(X) and labels(y).

```
y = df['Spending Score']
X = df.drop(columns='Spending Score')
```

After that, I imported the **train_test_split** method from the **sklearn.model_selection** module to split my data into training and testing sets. I made a 20%.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=45)

X_train.shape, y_train.shape # ((160, 4), (160,))

X_test.shape, y_test.shape # ((40, 4), (40,))
```

Yes, it is a really small dataset. Come on it's just day one. I am just learning don't judge me.

In the dataset, I had one categorical column — Gender(There are just 2😄).

Before feeding our data to a machine-learning model we need to convert the categorical column to a numerical column, so I used the Label Encoder.

```
labelencoder = LabelEncoder()
X_train['Gender'] = labelencoder.fit_transform(X_train['Gender'])
X_test['Gender'] = labelencoder.fit_transform(X_test['Gender'])
```

Now, my new data (X_train) looks something like this 👇,

```
CustomerID, Gender, Age, Annual Income
97, 98, 0, 27, 60
174, 175, 0, 52, 88
138, 139, 1, 19, 74
67, 68, 0, 68, 48
175, 176, 0, 30, 88
```

First, I used a Support Vector Regression Model.

```
from sklearn import svm

svm_model = svm.SVR()

svm_model.fit(X_train, y_train)

svm_prediction = svm_model.predict(X_test)

from sklearn.metrics import mean_absolute_error

svm_mae = mean_absolute_error(y_test, svm_prediction)

svm_mae # 21.66895201510343 (Not Good!)
```

Then, I used a Decision Tree Regressor Model.

```
from sklearn.tree import DecisionTreeRegressor

dt_model = DecisionTreeRegressor()

dt_model.fit(X_train, y_train)

dt_prediction = dt_model.predict(X_test)

dt_mae = mean_absolute_error(y_test, dt_prediction)

dt_mae # 20.15 (Bad!)
```

If you are someone who is good at machine learning please tell me in the comment if I made any mistake.

I know the mean absolute error was really bad in both models. Then I thought maybe I needed to standardize using **StandardScaler** so I did this👇.

```
from sklearn.preprocessing import StandardScaler

std_scaler = StandardScaler()

scaled_X_train = pd.DataFrame(std_scaler.fit_transform(X_train), columns=X_train.columns, index=X_train.index)
scaled_X_test = pd.DataFrame(std_scaler.fit_transform(X_test), columns=X_test.columns, index=X_test.index)
```

```
labelencoder = LabelEncoder()
scaled_X_train['Gender'] =
labelencoder.fit_transform(X_train['Gender'])
scaled_X_test['Gender'] =
labelencoder.fit_transform(X_test['Gender'])

svm_model.fit(scaled_X_train, y_train)

new_svm_prediction = svm_model.predict(scaled_X_test)

mean_absolute_error(y_test, new_svm_prediction) #
20.52389413223758
```

First I standardized all the columns including the gender column then I thought maybe I should not standardize the gender column so I again changed it. (Remember, I am just learning).

Then I used the SVM model again on the scaled dataset but no improvement😳.

At last, I used the Linear Regression Model.

```
"""# Linear Regression"""

from sklearn.linear_model import LinearRegression

lr_model = LinearRegression()

lr_model.fit(scaled_X_train, y_train)
```

```
lr_prediction = lr_model.predict(scaled_X_test)

lr_prediction

lr_mae = mean_absolute_error(y_test, lr_prediction)

lr_mae # 21.162694166396186
```

That's it guys. I learned a few things today, like how to rename a column, and more importantly, I am back coding, I have been not coding for so long and today felt like a refreshing restart.

Day — 1☑

Day-2: Wine Quality Prediction 🍷

Today, I worked with the wine 🍷 quality dataset. Below is a sample of what the dataset looks like.

type fixed acidity volatile acidity citric acid residual sugar chlorides free sulfur dioxide total sulfur dioxide density pH sulphates alcohol quality
0 white 7.0 0.27 0.36 20.7 0.045 45.0 170.0 1.0010 3.00 0.45 8.8 6
1 white 6.3 0.30 0.34 1.6 0.049 14.0 132.0 0.9940 3.30 0.49 9.5 6
2 white 8.1 0.28 0.40 6.9 0.050 30.0 97.0 0.9951 3.26 0.44 10.1 6
3 white 7.2 0.23 0.32 8.5 0.058 47.0 186.0 0.9956 3.19 0.40 9.9 6
4 white 7.2 0.23 0.32 8.5 0.058 47.0 186.0 0.9956 3.19 0.40 9.9 6

So, first, let me tell you where I found this, I studied this article from GeeksForGeeks 👉 Wine Quality Prediction.

But I did not just copy the code as it is, I read it and did the code in my way.

The dataset is available here👉Dataset.

Now let me tell you what I did today.

I started by importing the libraries that I always import.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Then I loaded the data and used a few methods like head, info, and describe to understand it.

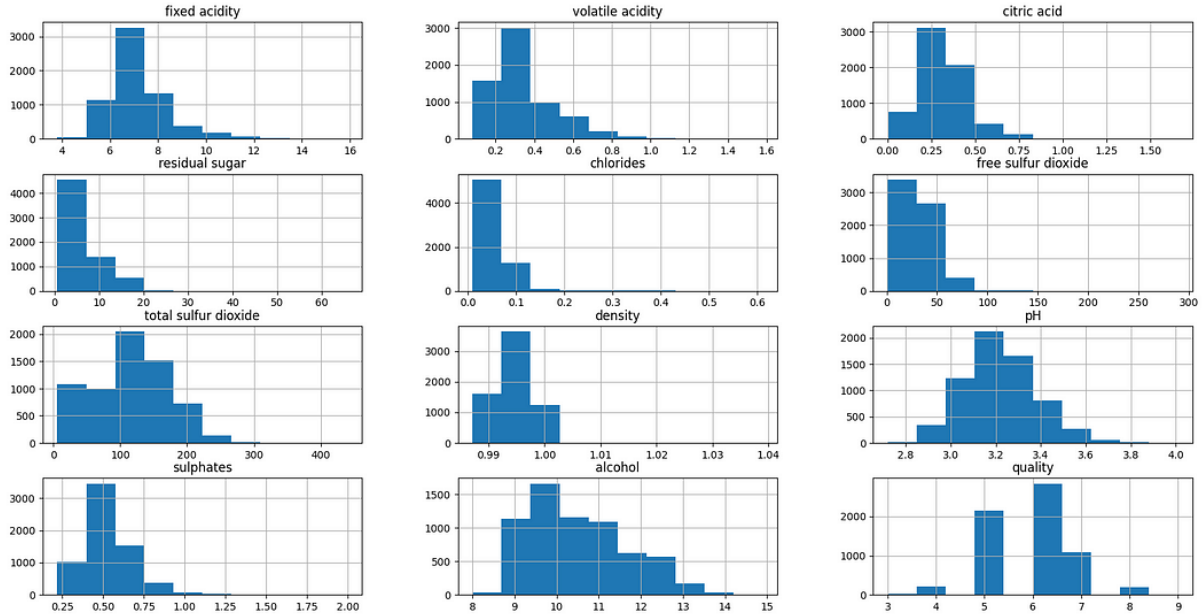
```
df = pd.read_csv('winequalityN.csv')


df.head()

df.tail()

df.info()

df.describe().T

df.isnull().sum()

df['type'].unique()
```

I found that in some columns there are a few null values, there was no null value in the categorical column(type). Only a few numerical columns had null values so, I used **fillna()** and filled the null values of each column with their mean value.

```
for col in df.columns:
  if df[col].isnull().sum() > 0:
    df[col] = df[col].fillna(df[col].mean())
```

And then to know the distribution of values of each column, I did this 👇.

```
df.hist(figsize=(20, 10))
```



This is what I got!

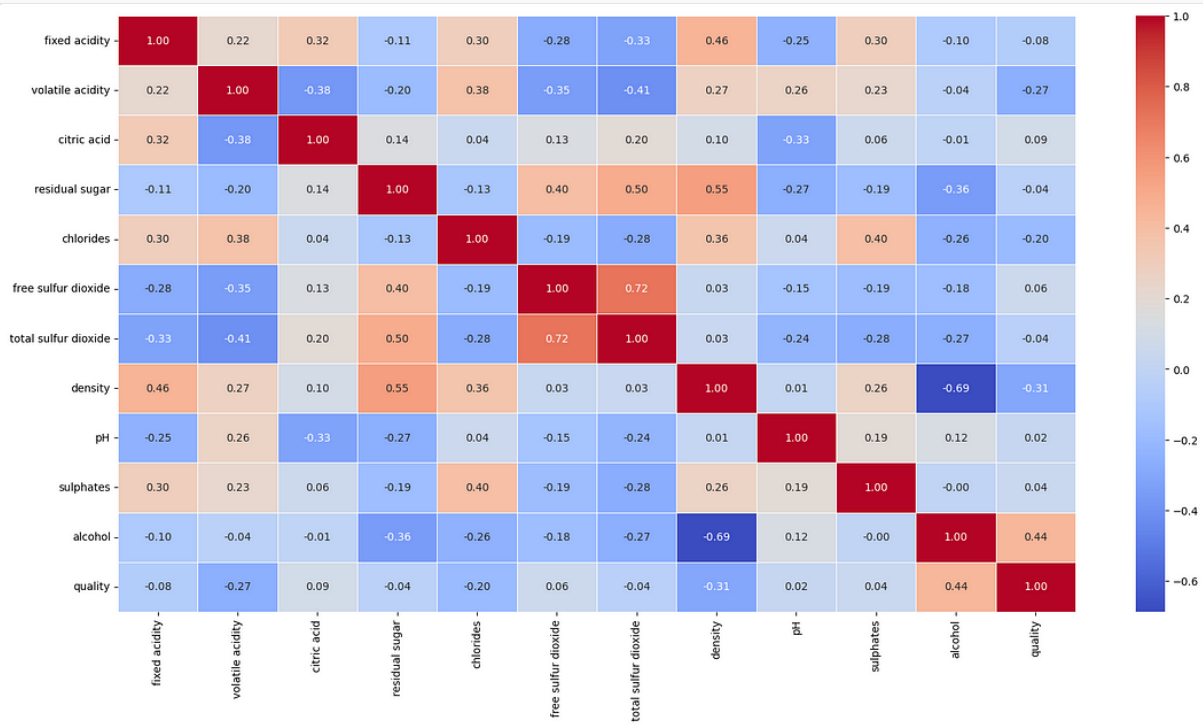
We definitely need to normalize this data, which I will eventually do but before that let's know the number of unique values present in the 'quality' columns — the label of our dataset.

Let's know the correlation between all the columns.

```
df.corr()
```

To make it visually appealing let's use heatmap.

```
plt.figure(figsize=(20, 10))
sns.heatmap(df.corr(), annot=True,  cmap='coolwarm', fmt=".2f", linewidths=0.5)
```



Noice!

It seems that only the 'alcohol' column has a 40% correlation with the quality column.

```
df.quality.unique() #array([6, 5, 7, 8, 4, 3, 9])
```

So, there are 7 unique values which is not good. So, what we can do is, we can consider 1 if quality is above 5 and 0 if it is below 5. It's a classification problem.

```
df['best quality'] = [1 if x > 5 else 0 for x in df['quality']]
```

Now we can remove the 'quality' column.

```
df = df.drop(columns='quality')
```

To handle the categorical column I did this 👇.

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

df['type'] = le.fit_transform(df['type'])
```

I think now we can make the splitting.

```
y = df['best quality']
X = df.drop(columns='best quality')

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
```

```
random_state=89)

X_train.shape, X_test.shape # (5197, 12), (1300, 12))
```

As we already know the distribution of values in each column is not in our favor so we need to normalize that.

```
from sklearn.preprocessing import MinMaxScaler

norm = MinMaxScaler()
X_train = norm.fit_transform(X_train)
X_test = norm.fit_transform(X_test)
```

It's time to create the models and make predictions.

I started with the Logistic Regression Model. This is the only Regression model used for Classification tasks.

```
from sklearn.linear_model import LogisticRegression

lr_model = LogisticRegression()

lr_model.fit(X_train, y_train)

lr_prediction = lr_model.predict(X_test)

from sklearn.metrics import accuracy_score

lr_prediction
```

```
lr_accuracy = accuracy_score(y_test, lr_prediction)
lr_accuracy # 0.7453846153846154 (Not Bad!)
```

I never used this model, but in the article, they used this model so I gave it a try.

```
from xgboost import XGBClassifier

XGB_model = XGBClassifier()

XGB_model.fit(X_train, y_train)

XGB_prediction = XGB_model.predict(X_test)
XGB_prediction

XGB_accuracy = accuracy_score(y_test, XGB_prediction)

XGB_accuracy # 0.6746153846153846
```

Finally, I used the Support Vector Classifier.

```
from sklearn.svm import SVC

svc_model = SVC()

svc_model.fit(X_train, y_train)
svc_prediction = svc_model.predict(X_test)
svc_accuracy = accuracy_score(y_test, svc_prediction)
```

```
svc_accuracy # 0.7407692307692307
```

The accuracy of the Logistic Regression Model and the Support Vector Classifier Model is better than XGBoost.

I need to learn more about hyperparameter tuning. Also, I need to learn more data preprocessing techniques.

That's it guys!

Day — 2☑

Day-3: Spam Mail Prediction📩

Today I worked with a spam mail dataset.

```
Category Message
0 ham Go until jurong point, crazy.. Available only ...
1 ham Ok lar... Joking wif u oni...
2 spam Free entry in 2 a wkly comp to win FA Cup fina...
3 ham U dun say so early hor... U c already then say...
4 ham Nah I don't think he goes to usf, he lives aro...
```

As usual, imported the necessary libraries(Hoof, I am tired of saying this😅) and loaded the dataset.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split

df = pd.read_csv('/content/mail_data.csv')
```

Then to understand the dataset more clearly, I used some methods like👇,

```
df.isnull().sum()
"""
Category    0
Message     0
dtype: int64
"""

df.shape # (5572, 2)

df['Category'].value_counts()

"""
ham     4825
spam     747
Name: Category, dtype: int64
"""
```

Thank god there was no null value in my dataset. But the dataset was too biased it had more **ham** data I mean 4825😳 and just 747 **spam** data.

I didn't know what to do, then asked my only friend ChatGPT, and he introduced me to the concept of under-sampling and over-sampling.

Under-sampling decreases the number of rows where ham is present and makes it equal to spam. Over-sampling is just the opposite. So, I under-sampled my dataset.

```
ham_messages = df[df['Category'] == 'ham']
spam_messages = df[df['Category'] == 'spam']

len(undersampled_ham) # 747 (Now both spam and ham have equal number of rows in the the dataset)

new_df = pd.concat([undersampled_ham, spam_messages])

print(new_df)

"""OUTPUT:
 Category Message
3714 ham If i not meeting ü all rite then i'll go home ...
1311 ham I.ll always be there, even if its just in spir...
548 ham Sorry that took so long, omw now
1324 ham I thk 50 shd be ok he said plus minus 10.. Did...
3184 ham Dunno i juz askin cos i got a card got 20% off...
... ... ...
5537 spam Want explicit SEX in 30 secs? Ring 02073162414...
5540 spam ASKED 3MOBILE IF 0870 CHATLINES INCLU IN FREE ...
5547 spam Had your contract mobile 11 Mnths? Latest Moto...
5566 spam REMINDER FROM O2: To get 2.50 pounds free call...
5567 spam This is the 2nd time we have tried 2 contact u...
1494 rows × 2 columns

"""
```

Then I changed the category column values from ham to 0 and spam to 1 using the label encoder.

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
```

```
new_df['Category'] = le.fit_transform(new_df['Category'])
print(new_df.head())

"""OUTPUT:
 Category Message
3714 0 If i not meeting ü all rite then i'll go home ...
1311 0 I.ll always be there, even if its just in spir...
548 0 Sorry that took so long, omw now
1324 0 I thk 50 shd be ok he said plus minus 10.. Did...
3184 0 Dunno i juz askin cos i got a card got 20% off...
"""
```

I found that my dataset is not balanced, because I concatenated the dataset after under-sampling which made all the ham rows on top and spam rows on the bottom. So, to balance that I did this👇.

```
balanced_df = new_df.sample(frac=1,
random_state=42).reset_index(drop=True)

balanced_df.head(10)

"""OUTPUT:
 Category Message
0 1 URGENT, IMPORTANT INFORMATION FOR O2 USER. TOD...
1 1 Panasonic & BluetoothHdset FREE. Nokia FREE. M...
2 1 Do you want a new Video handset? 750 any time ...
3 1 Hi if ur lookin 4 saucy daytime fun wiv busty ...
4 1 09066362231 URGENT! Your mobile No 07xxxxxxxxx...
5 0 Will do. Was exhausted on train this morning. ...
6 0 Oh Howda gud gud.. Mathe en samachara chikku:-)
7 0 Dear Hero,i am leaving to qatar tonite for an ...
```

```
8 0 Great! How is the office today?
9 0 Oooh bed ridden ey? What are YOU thinking of?
"""
```

Need to learn more about the sampling method.

Then I divided the dataset into features and labels.

```
X = balanced_df['Message']
y = balanced_df['Category']

X_train.shape, X_test.shape #((1120,), (374,))
y_train.shape, y_test.shape #((1120,), (374,))
```

As we are dealing with text data, we need to do something to convert that text data into a numerical value. Initially, I don't know what to do. Then I Googled and found the sklearn documentation inside I found the TfidfVectorizer.

Below is a sample code of how TfidfVectorizer converts text to numbers.

```
from sklearn.feature_extraction.text import TfidfVectorizer
corpus = [
    'This is the first document.',
    'This document is the second document.',
    'And this is the third one.',
    'Is this the first document?',
]
```

```
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)
print(vectorizer.get_feature_names_out())

"""
OUTPUT:
array(['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third',
       'this'], dtype=object)
"""

X.toarray()

"""
OUTPUT:
array([[0.        , 0.46979139, 0.58028582, 0.38408524, 0.        ,
        0.        , 0.38408524, 0.        , 0.38408524],
       [0.        , 0.6876236 , 0.        , 0.28108867, 0.        ,
        0.53864762, 0.28108867, 0.        , 0.28108867],
       [0.51184851, 0.        , 0.        , 0.26710379, 0.51184851,
        0.        , 0.26710379, 0.51184851, 0.26710379],
       [0.        , 0.46979139, 0.58028582, 0.38408524, 0.        ,
        0.        , 0.38408524, 0.        , 0.38408524]])
"""
```

As the TfidfVectorizer returns a sparse matrix I converted it into an array to get a glance of how my sample corpus looks in numerical form.

Then I applied this vectorizer in my dataset.

```
vectorizer = TfidfVectorizer(min_df=1, stop_words='english', lowercase=True)
```

I added a few parameters, I don't know much about this vectorizer, need to learn. But let me tell you how the parameters I mentioned works the ***min_df=1*** ignores words that just appear once, ***stop_words='english'*** ignores all the common English words like ('the', 'a', 'and', 'is' and more) and ***lowercase=True*** makes all the words in the dataset into lowercase.

```
new_X_train = vectorizer.fit_transform(X_train)
new_X_test = vectorizer.transform(X_test)
```

Used the vectorizer on both train and test sets. Then,

I used the Logistic Regression Model.

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression()
lr.fit(new_X_train, y_train)
lr_prediction = lr.predict(new_X_test)
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_test, lr_prediction)
print(accuracy) # 0.9331550802139037😄
```

93%.

I know there are a lot of things I need to learn.

I am planning to speed up my learning by spending 1 hour each day reading Machine Learning Books. I am going to do it from tomorrow. If you have any good book recommendations comment below.

Day — 3☑

Day-4: Melbourne Housing Prediction 🏠

In Kaggle there are some amazing free courses available for beginners who are just getting started with Machine Learning or Data Science.

Long back, I completed the "Intro To Machine Learning" course and "Intermediate Machine Learning" course.

But after that, I had a very long gap, I was not in touch with Machine Learning for a while.

So, I forgot a lot of concepts, but no big deal, I just did the course again. This time it was really easy and I was able to recall all the basic concepts that I learned.

When I was learning from that course on the side I had my Google Colab opened and I was coding along. Also, it offers an interactive exercise-like environment where you need to write code for the questions after learning a section. It's an amazing course I recommend it if you are just getting started with machine learning.

Here is the link to the course: Intro To Machine Learning (It's not a video course)

I used the **Melbourne Housing Snapshot** dataset. I was not copying all the code from the course, I was coding on my own. Actually, my code was even better😁.

Here is my code,

```
import pandas as p
```

I know we usually import pandas as **pd** but I thought of doing it differently.

```
df = p.read_csv('/content/melb_data.csv')
p.set_option('display.max_column', 100)
```

As the dataset had 21 rows I used the ***set_option()*** method to increase the display of the number of columns, if you don't use ***set_option()*** Google Colab will show you only a few columns.

```
df.head()

"""OUTPUT:
Suburb Address Rooms Type Price Method SellerG Date Distance
Postcode Bedroom2 Bathroom Car Landsize BuildingArea YearBuilt
CouncilArea Lattitude Longtitude Regionname Propertycount
0 Abbotsford 85 Turner St 2 h 1480000.0 S Biggin 3/12/2016 2.5
3067.0 2.0 1.0 1.0 202.0 NaN NaN Yarra -37.7996 144.9984 Northern
Metropolitan 4019.0
1 Abbotsford 25 Bloomburg St 2 h 1035000.0 S Biggin 4/02/2016 2.5
3067.0 2.0 1.0 0.0 156.0 79.0 1900.0 Yarra -37.8079 144.9934
```

```
Northern Metropolitan 4019.0
2 Abbotsford 5 Charles St 3 h 1465000.0 SP Biggin 4/03/2017 2.5
3067.0 3.0 2.0 0.0 134.0 150.0 1900.0 Yarra -37.8093 144.9944
Northern Metropolitan 4019.0
3 Abbotsford 40 Federation La 3 h 850000.0 PI Biggin 4/03/2017 2.5
3067.0 3.0 2.0 1.0 94.0 NaN NaN Yarra -37.7969 144.9969 Northern
Metropolitan 4019.0
4 Abbotsford 55a Park St 4 h 1600000.0 VB Nelson 4/06/2016 2.5
3067.0 3.0 1.0 2.0 120.0 142.0 2014.0 Yarra -37.8072 144.9941
Northern Metropolitan 4019.0
"""

df.shape #(13580, 21)
df.isnull().sum()
"""OUTPUT:
Suburb           0
Address          0
Rooms            0
Type             0
Price            0
Method           0
SellerG          0
Date             0
Distance         0
Postcode         0
Bedroom2         0
Bathroom         0
Car             62
Landsize         0
BuildingArea    6450
YearBuilt       5375
CouncilArea     1369
Lattitude        0
Longtitude       0
```

```
Regionname          0
Propertycount       0
"""
```

I was performing some operations to understand the dataset. The 3 columns **"BuildingArea"**, **"YearBuilt"**, and **"CouncilArea"** has a lot of null values I will drop them all in a while.

```
df.describe().T

"""OUTPUT:
 count mean std min 25% 50% 75% max
Rooms 13580.0 2.937997e+00 0.955748 1.00000 2.000000 3.000000
3.000000e+00 1.000000e+01
Price 13580.0 1.075684e+06 639310.724296 85000.00000
650000.000000 903000.000000 1.330000e+06 9.000000e+06
Distance 13580.0 1.013778e+01 5.868725 0.00000 6.100000 9.200000
1.300000e+01 4.810000e+01
Postcode 13580.0 3.105302e+03 90.676964 3000.00000 3044.000000
3084.000000 3.148000e+03 3.977000e+03
Bedroom2 13580.0 2.914728e+00 0.965921 0.00000 2.000000
3.000000 3.000000e+00 2.000000e+01
Bathroom 13580.0 1.534242e+00 0.691712 0.00000 1.000000
1.000000 2.000000e+00 8.000000e+00
Car 13518.0 1.610075e+00 0.962634 0.00000 1.000000 2.000000
2.000000e+00 1.000000e+01
Landsize 13580.0 5.584161e+02 3990.669241 0.00000 177.000000
440.000000 6.510000e+02 4.330140e+05
BuildingArea 7130.0 1.519676e+02 541.014538 0.00000 93.000000
126.000000 1.740000e+02 4.451500e+04
YearBuilt 8205.0 1.964684e+03 37.273762 1196.00000 1940.000000
```

```
1970.000000 1.999000e+03 2.018000e+03
Lattitude 13580.0 -3.780920e+01 0.079260 -38.18255 -37.856822 -
37.802355 -3.775640e+01 -3.740853e+01
Longtitude 13580.0 1.449952e+02 0.103916 144.43181 144.929600
145.000100 1.450583e+02
"""
```

I don't know why I used this even though I didn't want to know anything from this😅(Muscle Memory).

```
df.info()
"""OUTPUT:
class 'pandas.core.frame.DataFrame'>
RangeIndex: 13580 entries, 0 to 13579
Data columns (total 21 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   Suburb        13580 non-null  object
 1   Address       13580 non-null  object
 2   Rooms         13580 non-null  int64
 3   Type          13580 non-null  object
 4   Price         13580 non-null  float64
 5   Method        13580 non-null  object
 6   SellerG       13580 non-null  object
 7   Date          13580 non-null  object
 8   Distance      13580 non-null  float64
 9   Postcode      13580 non-null  float64
 10  Bedroom2      13580 non-null  float64
 11  Bathroom      13580 non-null  float64
 12  Car           13518 non-null  float64
 13  Landsize      13580 non-null  float64
 14  BuildingArea  7130 non-null   float64
```

```
 15  YearBuilt      8205 non-null   float64
 16  CouncilArea    12211 non-null  object
 17  Lattitude      13580 non-null  float64
 18  Longtitude     13580 non-null  float64
 19  Regionname     13580 non-null  object
 20  Propertycount  13580 non-null  float64
dtypes: float64(12), int64(1), object(8)
memory usage: 2.2+ MB
"""
```

Used the ***info()*** method to know how many categorical columns I have. I got 8.

```
df.nunique()
"""OUTPUT:
Suburb          314
Address       13378
Rooms             9
Type              3
Price          2204
Method            5
SellerG         268
Date             58
Distance        202
Postcode        198
Bedroom2         12
Bathroom          9
Car              11
Landsize       1448
BuildingArea    602
YearBuilt       144
```

```
CouncilArea       33
Lattitude       6503
Longtitude      7063
Regionname         8
Propertycount    311
dtype: int64
"""
```

Using the ***nunique()*** to know the number of unique values present in every column. (I love pandas ♡).

```
df = df.drop(columns=['BuildingArea', 'YearBuilt', 'CouncilArea'])
```

I dropped the columns that I mentioned earlier.

```
df.isnull().sum()
"""OUTPUT:
Suburb          0
Address         0
Rooms           0
Type            0
Price           0
Method          0
SellerG         0
Date            0
Distance        0
Postcode        0
Bedroom2        0
Bathroom        0
Car            62
```

```
Landsize        0
Lattitude       0
Longtitude      0
Regionname      0
Propertycount   0
dtype: int64
"""
```

Still, there is one column left that has null values, I tried to replace the null values with other values as it had only a few null values but then I realized it doesn't have a very good correlation with the price(which is label). So, I dropped it.

```
df = df.drop(columns=['Car'])
```

Now we have a data frame with zero null value columns. Missing values are handled. It's time to handle the categorical columns. To do that, first, let's filter the categorical columns with low cardinality values.

Low cardinality means less number of unique values.

```
low_cardinality_col = [col for col in df.columns if df[col].dtype ==
object and df[col].nunique() < 10]
num_col = [col for col in df.columns if df[col].dtype in [int, float]]
```

Now, we can concatenate the above two columns. Drop the "Price" column as it is the label(y) and create our features(X) set.

```
X = df[low_cardinality_col + num_col].drop(columns='Price')
X.tail()
"""OUTPUT:

Type Method Regionname Rooms Distance Postcode Bedroom2
Bathroom Landsize Lattitude Longtitude Propertycount
13575 h S South-Eastern Metropolitan 4 16.7 3150.0 4.0 2.0 652.0 -
37.90562 145.16761 7392.0
13576 h SP Western Metropolitan 3 6.8 3016.0 3.0 2.0 333.0 -
37.85927 144.87904 6380.0
13577 h S Western Metropolitan 3 6.8 3016.0 3.0 2.0 436.0 -37.85274
144.88738 6380.0
13578 h PI Western Metropolitan 4 6.8 3016.0 4.0 1.0 866.0 -
37.85908 144.89299 6380.0
13579 h SP Western Metropolitan 4 6.3 3013.0 4.0 1.0 362.0 -37.81188
144.88449 6543.0
"""
```

The above output is our features set.

```
X[low_cardinality_col].nunique()
"""OUTPUT:
Type         3
Method       5
Regionname   8
dtype: int64
"""
```

As you can see from the above code, we have only 3 categorical columns in our feature we need to convert them into some numerical value before feeding the feature set to the model.

I was confused at this stage about what encoder to use. Whether to use One-hot-encoder or Label Encoder, obviously we cannot use the Ordinal Encoder as the 3 columns don't have any particular ordering.

I used the Label Encoder.

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
X['Type'] = le.fit_transform(df['Type'])
X['Method'] = le.fit_transform(df['Method'])
X['Regionname'] = le.fit_transform(df['Regionname'])
X.head()
"""OUTPUT:

Type Method Regionname Rooms Distance Postcode Bedroom2
Bathroom Landsize Lattitude Longtitude Propertycount
0 0 1 2 2 2.5 3067.0 2.0 1.0 202.0 -37.7996 144.9984 4019.0
1 0 1 2 2 2.5 3067.0 2.0 1.0 156.0 -37.8079 144.9934 4019.0
2 0 3 2 3 2.5 3067.0 3.0 2.0 134.0 -37.8093 144.9944 4019.0
3 0 0 2 3 2.5 3067.0 3.0 2.0 94.0 -37.7969 144.9969 4019.0
4 0 4 2 4 2.5 3067.0 3.0 1.0 120.0 -37.8072 144.9941 4019.0
"""
```

Now our features set is all numerical. We can proceed with the model creation. You may think "But what about scaling?". I had the same

question but I asked ChatGPT should I needed to scale this dataset, it said "It depends on what model you are going to use".

I was going to use the DecisionTreeRegressor 🌳 and RandomForestRegroessor model, so I don't have to scale. If I was to use other models I would have to.

Before moving on with the model creation we need to do a train test split 🔨.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=56)
X_train.shape, X_test.shape #((10864, 12), (2716, 12))
y_train.shape, y_test.shape #((10864,), (2716,))
```

Let's first create our Decision Tree Regressor Model and Check it's performance using the mean absolute error.

```
from sklearn.tree import DecisionTreeRegressor
dtr = DecisionTreeRegressor()
dtr.fit(X_train, y_train)
dtr_prediction = dtr.predict(X_test)
from sklearn.metrics import mean_absolute_error
mae = mean_absolute_error(y_test, dtr_prediction)
mae #223163.49926362297
```

The mae is not good. This means that If the actual price is 1000000. The model will predict 1000000-223163.49926362297=something wrong.

Let's try Random Forest Regressor.

```
from sklearn.ensemble import RandomForestRegressor
def rf_mae(X_train, X_test, y_train, y_test):
  rfr = RandomForestRegressor()
  rfr.fit(X_train, y_train)
  prediction = rfr.predict(X_test)
  rfr_mae = mean_absolute_error(y_test, prediction)
  return rfr_mae
```

I created a function to make the process simple.

```
rf_mae(X_train, X_test, y_train, y_test)
# 169436.44666403672
```

Much better than Decision Tree Regressor Model.

Today, was nice.

I was in a flow state man.

I was coding + writing this blog for like 2 hours straight, without any distractions.

I feel like, I can finish my Day — 5 project now itself😉.

Day — 4☑

Day-5: House Price Prediction 🏠

I am really getting good at Data Preprocessing and Regression tasks.

Like yesterday, today I chose a Regression problem but this time with more columns.

Let me get into the code straight away.

```
import pandas as p
p.set_option('display.max_columns', 100)
p.set_option('display.max_rows', 100)
```

I found this amazing method called ***set_option()*** which allows us to see all the columns. This will help you in situations when your Google Colab is not showing all the columns and instead, it shows the first few columns then (...) then the last few columns, you know what I am trying to say right.

In the past, I used to import all the libraries that I could remember even if I was not using them, but nowadays I only import the library when I need them.

```
housing_train = p.read_csv('HousingPriceTrain.csv')
housing_test = p.read_csv("HousingPriceTest.csv")
```

I had 2 separate datasets this time. One is for training and one is for testing the label column was only available in the training set.

```
housing_train.shape, housing_test.shape
# ((1460, 81), (1459, 80))
housing_train.head()
"""
OUTPUT:
Id MSSubClass MSZoning LotFrontage LotArea Street Alley LotShape
LandContour Utilities LotConfig LandSlope Neighborhood Condition1
Condition2 BldgType HouseStyle OverallQual OverallCond YearBuilt
YearRemodAdd RoofStyle RoofMatl Exterior1st Exterior2nd
MasVnrType MasVnrArea ExterQual ExterCond Foundation BsmtQual
BsmtCond BsmtExposure BsmtFinType1 BsmtFinSF1 BsmtFinType2
BsmtFinSF2 BsmtUnfSF TotalBsmtSF Heating HeatingQC CentralAir
Electrical 1stFlrSF 2ndFlrSF LowQualFinSF GrLivArea BsmtFullBath
BsmtHalfBath FullBath HalfBath BedroomAbvGr KitchenAbvGr
KitchenQual TotRmsAbvGrd Functional Fireplaces FireplaceQu
GarageType GarageYrBlt GarageFinish GarageCars GarageArea
GarageQual GarageCond PavedDrive WoodDeckSF OpenPorchSF
EnclosedPorch 3SsnPorch ScreenPorch PoolArea PoolQC Fence
MiscFeature MiscVal MoSold YrSold SaleType SaleCondition SalePrice
0 1 60 RL 65.0 8450 Pave NaN Reg Lvl AllPub Inside Gtl CollgCr Norm
Norm 1Fam 2Story 7 5 2003 2003 Gable CompShg VinylSd VinylSd
BrkFace 196.0 Gd TA PConc Gd TA No GLQ 706 Unf 0 150 856 GasA
Ex Y SBrkr 856 854 0 1710 1 0 2 1 3 1 Gd 8 Typ 0 NaN Attchd 2003.0
RFn 2 548 TA TA Y 0 61 0 0 0 0 NaN NaN NaN 0 2 2008 WD Normal
208500
1 2 20 RL 80.0 9600 Pave NaN Reg Lvl AllPub FR2 Gtl Veenker Feedr
Norm 1Fam 1Story 6 8 1976 1976 Gable CompShg MetalSd MetalSd
None 0.0 TA TA CBlock Gd TA Gd ALQ 978 Unf 0 284 1262 GasA Ex Y
SBrkr 1262 0 0 1262 0 1 2 0 3 1 TA 6 Typ 1 TA Attchd 1976.0 RFn 2
460 TA TA Y 298 0 0 0 0 0 NaN NaN NaN 0 5 2007 WD Normal
```

```
181500
2 3 60 RL 68.0 11250 Pave NaN IR1 Lvl AllPub Inside Gtl CollgCr Norm
Norm 1Fam 2Story 7 5 2001 2002 Gable CompShg VinylSd VinylSd
BrkFace 162.0 Gd TA PConc Gd TA Mn GLQ 486 Unf 0 434 920 GasA
Ex Y SBrkr 920 866 0 1786 1 0 2 1 3 1 Gd 6 Typ 1 TA Attchd 2001.0
RFn 2 608 TA TA Y 0 42 0 0 0 0 NaN NaN NaN 0 9 2008 WD Normal
223500
3 4 70 RL 60.0 9550 Pave NaN IR1 Lvl AllPub Corner Gtl Crawfor Norm
Norm 1Fam 2Story 7 5 1915 1970 Gable CompShg Wd Sdng Wd Shng
None 0.0 TA TA BrkTil TA Gd No ALQ 216 Unf 0 540 756 GasA Gd Y
SBrkr 961 756 0 1717 1 0 1 0 3 1 Gd 7 Typ 1 Gd Detchd 1998.0 Unf 3
642 TA TA Y 0 35 272 0 0 0 NaN NaN NaN 0 2 2006 WD Abnorml
140000
4 5 60 RL 84.0 14260 Pave NaN IR1 Lvl AllPub FR2 Gtl NoRidge Norm
Norm 1Fam 2Story 8 5 2000 2000 Gable CompShg VinylSd VinylSd
BrkFace 350.0 Gd TA PConc Gd TA Av GLQ 655 Unf 0 490 1145 GasA
Ex Y SBrkr 1145 1053 0 2198 1 0 2 1 4 1 Gd 9 Typ 1 TA Attchd 2000.0
RFn 3 836 TA TA Y 192 84 0 0 0 0 NaN NaN NaN 0 12 2008 WD
Normal 250000
"""
```

It's a big dataset not by rows but by columns.

```
null_value_cols = [col for col in housing_train.columns if
housing_train[col].isnull().any()]
```

I just wanted to know all the columns which are containing null values so I wrote the above code.

```
housing_train[null_value_cols].isnull().sum()
"""OUTPUT:
LotFrontage     259
Alley          1369
MasVnrType        8
MasVnrArea        8
BsmtQual         37
BsmtCond         37
BsmtExposure     38
BsmtFinType1     37
BsmtFinType2     38
Electrical        1
FireplaceQu     690
GarageType       81
GarageYrBlt      81
GarageFinish     81
GarageQual       81
GarageCond       81
PoolQC         1453
Fence          1179
MiscFeature    1406
dtype: int64
"""
```

As you can see from the above output columns **PoolQC, Fence, and MiscFeature** are just useless and also the **FireplaceQu** column.

```
housing_train = housing_train.drop(columns=['Alley', 'FireplaceQu',
'PoolQC', 'Fence', 'MiscFeature'])
```

```
housing_test = housing_test.drop(columns=['Alley', 'FireplaceQu',
'PoolQC', 'Fence', 'MiscFeature'])
```

After dropping those useless columns. I again checked the null value columns, to decide which columns to drop and which columns to encode the null value with other values.

```
housing_train[null_value_cols].isnull().sum()
"""OUTPUT:
LotFrontage    259
MasVnrType       8
MasVnrArea       8
BsmtQual        37
BsmtCond        37
BsmtExposure    38
BsmtFinType1    37
BsmtFinType2    38
Electrical       1
GarageType      81
GarageYrBlt     81
GarageFinish    81
GarageQual      81
GarageCond      81
"""
```

Before that, we need to check the datatype of all the null value columns because if it's a categorical column we will be using a different strategy, and if it's a numerical column we will be using a different strategy. So, let's know the datatypes.

```
housing_train[null_value_cols].dtypes
"""OUTPUT:
LotFrontage    float64
MasVnrType      object
MasVnrArea     float64
BsmtQual        object
BsmtCond        object
BsmtExposure    object
BsmtFinType1    object
BsmtFinType2    object
Electrical      object
GarageType      object
GarageYrBlt    float64
GarageFinish    object
GarageQual      object
GarageCond      object
dtype: object
"""
housing_train = housing_train.drop(columns='LotFrontage')
```

I decided to drop the **LotFrontage** column as it had more null values. Also, I am going to drop the rows that contain null values if I keep the **LotFrontage** columns I will lose a lot of rows but If I just remove the LotFrontage column and then remove the rows containing null values I will only lose 140 rows which is fine.

```
fresh_housing_train = housing_train.dropna()
fresh_housing_train.shape #(1338, 75)
```

As you can see the shape of our train set from 1460 it became 1338 which is not that bad. And we are free from null values or missing values if you will.

```
low_cardinality_col = [col for col in fresh_housing_train.columns if
fresh_housing_train[col].dtype == object and
fresh_housing_train[col].nunique() < 10]
high_cardinality_col = [col for col in fresh_housing_train.columns if
fresh_housing_train[col].dtype == object and
fresh_housing_train[col].nunique() >= 10]
num_col = [col for col in fresh_housing_train.columns if
fresh_housing_train[col].dtype in [int, float]]
```

Then I separated the columns in the dataset into 3 categories. Because for categorical columns the preprocessing will be different as compared to numerical columns.

```
new_housing_train =
p.concat([fresh_housing_train[low_cardinality_col],
fresh_housing_train[high_cardinality_col],
fresh_housing_train[num_col]], axis=1)
```

Then I again combined them😅. And also removed the **Id** column which is of no use. And removed the **LotFrontage** and **Id** columns from the test set as well.

```
new_housing_train = new_housing_train.drop(columns='Id')
housing_test = housing_test.drop(columns=['LotFrontage', 'Id'])
```

I just did all the preprocessing I did for the train set to the test set.

```
null_value_cols = [col for col in housing_test.columns if
housing_test[col].isnull().any()]
housing_test[housing_test.isnull().any(axis=1)][null_value_cols] #Prints
all the null value columns
fresh_housing_test = housing_test.dropna()
low_cardinality_col = [col for col in fresh_housing_test.columns if
fresh_housing_test[col].dtype == object and
fresh_housing_test[col].nunique() < 10]
high_cardinality_col = [col for col in fresh_housing_test.columns if
fresh_housing_test[col].dtype == object and
fresh_housing_test[col].nunique() >= 10]
num_col = [col for col in fresh_housing_test.columns if
fresh_housing_test[col].dtype in [int, float]]
new_housing_test = p.concat([fresh_housing_test[low_cardinality_col],
fresh_housing_test[high_cardinality_col],
fresh_housing_test[num_col]], axis=1)
```

And before moving on to further preprocessing I made a copy of both the train and test set, in case I made a wrong move.

```
copy_of_housing_train = new_housing_train.copy()
copy_of_housing_test = new_housing_test.copy()
```

Now it's time to handle the categorical values or we can say like, it's time to encode the categorical columns.

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
cat_col = low_cardinality_col + high_cardinality_col
for col in cat_col:
  new_housing_train[col] = le.fit_transform(new_housing_train[col])

for col in cat_col:
  new_housing_test[col] = le.fit_transform(new_housing_test[col])
```

Now we have a perfect train set and a test set. Let's split the train set into train set and validation set. And use the test set for just testing.

I named the validation sets as **X_test** and **y_test** don't get confused.

```
from sklearn.model_selection import train_test_split
y = new_housing_train['SalePrice']
X = new_housing_train.drop(columns='SalePrice')
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=766)
X_train.shape, X_test.shape # ((1070, 73), (268, 73))
```

I didn't standardize or normalize the numerical columns because I am going to use the DecisionTreeRegressor and RandomForestRegressor as they work well with these types of datasets.

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor

dtr = DecisionTreeRegressor()
rfr = RandomForestRegressor()

dtr.fit(X_train, y_train)
dtr_prediction = dtr.predict(X_test)

rfr.fit(X_train, y_train)
rfr_prediction = rfr.predict(X_test)
```

Now let's see which model works best.

```
from sklearn.metrics import mean_absolute_error,
mean_absolute_percentage_error
dtr_error = mean_absolute_error(y_test, dtr_prediction)
dtr_error_percentage = mean_absolute_percentage_error(y_test,
dtr_prediction)
dtr_error, dtr_error_percentage*100 #(27336.078358208953,
14.489400761489787)
```

Nice!

That error is not much. Actually, it is saying the model is producing just 14% error which is not that bad.

Let's try RandomForestRegressor.

```
rfr_error = mean_absolute_error(y_test, rfr_prediction)
rfr_error_percentage = mean_absolute_percentage_error(y_test,
rfr_prediction)
rfr_error, rfr_error_percentage*100 # (18731.851417910446,
10.288181221800267)
```

That's even better!

Without using any parameter both the models are performing well which means we did a great job in the data preprocessing step.

That's it guy the project is done. Let's talk about other stuff.

I realized that I need to focus more on EDA. I am not doing any visualization. I should start doing that. Then…

I should also focus on hyperparameter tuning. Then…

I should learn to use more machine learning algorithms. Then…

I should learn more metrics and methods to evaluate my models.

Yeah. That's it those are the things.

1. EDA
2. Hyperparameter Tuning.

3. More Machine Learning Algorithms.

4. More Metrics and Methods to evaluate my Model.

I feel like this challenge is working. I can see myself thinking on my own feet. Also, I started to write better than before, don't you think😉.

I will see you tomorrow!👩🏻.

Day — 5☑

Day-6: California Housing Price Prediction 🏠

There is a one-must-read book for all the people who want to get into the world of Machine Learning and Artificial Intelligence.

It's a very practical book with hands-on projects and examples. It covers all the machine learning topics in-depth with coding.

Okay enough of the suspense, it's called "***Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow***" by Aurélien Géron.

My today's project is from the 2nd chapter of that book, which is called "***End-to-End Machine Learning Project***". I read the chapter completely and was exposed to a lot of different concepts.

I will share all the concepts with you. I don't know them in-depth for now, but I will learn them in depth in the coming days.

Even if you are not interested in reading books to learn Machine Learning you must read chapters 1 and 2. Chapter 1 is full of theory but when reading Chapter 2 you should have your Google Colab or Jupyter Notebook opened.

Okay, let me get into the code. If you want to follow along which is something I would recommend you to do because I am going to cover

a lot of machine learning concepts. You can read the article and in the second tab open your Google Colab and follow along.

Get the data:

Get the dataset from here: California Housing Prices Dataset

Load the data:

Let's start by importing pandas🐼.

```
import pandas as p
df = p.read_csv('california_housing.csv')
```

Take a quick look at the dataset

Let's understand the dataset using the methods we always use.

```
df.head() #Gives the first 5 rows
df.tail() #Gives the last 5 rows
df.shape #Gives the shape of the dataset (rows, columns) -> (20640, 10)
df.isnull().sum() #Gives the count of null values present in each column (Output below)
"""
OUTPUT:
longitude             0
latitude              0
housing_median_age    0
total_rooms           0
total_bedrooms      207
population            0
```

```
households            0
median_income         0
median_house_value    0
ocean_proximity       0
dtype: int64
"""
```

So, we just have null values in one column "***total_bedrooms***". And we have 20640 rows and 10 columns in our dataset.

Let's know the type of data present in each of the columns using the **info()** method and also know the statistics of the dataset like max, min, std, and more using the **describe()** method.

```
df.info()
"""
OUTPUT:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15210 entries, 0 to 15209
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           15210 non-null  float64
 1   latitude            15210 non-null  float64
 2   housing_median_age  15210 non-null  float64
 3   total_rooms         15210 non-null  float64
 4   total_bedrooms      15062 non-null  float64
 5   population          15210 non-null  float64
 6   households          15210 non-null  float64
 7   median_income       15210 non-null  float64
 8   median_house_value  15210 non-null  float64
 9   ocean_proximity     15209 non-null  object
```

```
dtypes: float64(9), object(1)
memory usage: 1.2+ MB
"""
df.describe() # Run and see the output yourself because it's too big.
```

I hope you are running the codes on your Google Colab along with me.

We can see that we only have one categorical column called "***ocean_proximity***", let us know how many unique values are there in it.

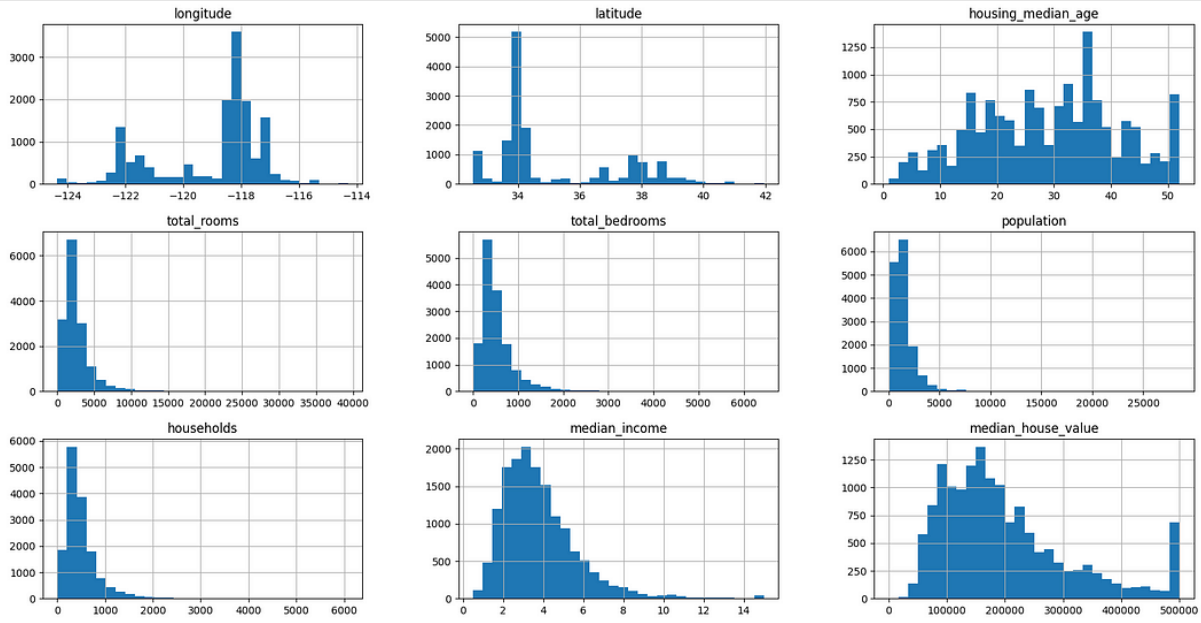
And also know how many times each value occurs in the dataset.

```
df['ocean_proximity'].unique()
"""
OUTPUT:
df['ocean_proximity'].unique()
array(['NEAR BAY', '<1H OCEAN', 'INLAND', 'NEAR OCEAN',
'ISLAND', nan],
      dtype=object)
"""


df['ocean_proximity'].value_counts()
"""
OUTPUT:
<1H OCEAN     7299
INLAND        4929
NEAR OCEAN    1489
NEAR BAY      1487
```

```
ISLAND          5
Name: ocean_proximity, dtype: int64
"""
```

To know the value spread of each numerical column we are going to plot a histogram using the matplotlib library.

```
import matplotlib.pyplot as plt
df.hist(bins=30, figsize=(20, 10))
```



The bins parameter is the number of bars you want the histogram to have. I gave 30. You can understand bins by counting the "***housing_median_age***" histogram in the above graph (can you see there are 30 buildings, try 40 in your code).

We can see that most of our numerical columns are not standardized soon we will standardize them after completing a few more steps.

Before moving on any further, we first need to set aside a test set. Let's do that.

Create a test set

```
from sklearn.model_selection import train_test_split
train_set, test_set = train_test_split(df, test_size=0.2,
random_state=8)
train_set.shape, test_set.shape #((16512, 10), (4128, 10))
```

I know what you are thinking. You usually use train_test_split to split the feature and label into **X_train, X_test, y_train,** and **y_test**. But it is also possible to split a single dataset into two.

Now you need to understand a concept called sampling. Let me explain that using an example, say there are 400 girls and 600 boys in a school and the school principal wants to select a sample of 100 students from the school to give a survey, the sample selected by the school principal must represent the whole population, the principal cannot select 100 boys for the survey (it's called sampling bias if the principal does that)because the girls are left out, the principal must select 60 boys and 40 girls for the sample.

I hope you understood the concept of sampling.

Now we are going to again divide the dataset into a train set and test set but this time, when dividing the dataset we are going to stratify it using the "***median_income***" column. You may ask why the

"***median_income***" column, it is because it has the highest correlation with the target column which is "***median_house_value***".

```
df.corr()['median_house_value'].sort_values(ascending=False)
"""OUTPUT:
median_house_value    1.000000
median_income         0.688075
total_rooms           0.134153
housing_median_age    0.105623
households            0.065843
total_bedrooms        0.049686
population           -0.024650
longitude            -0.045967
latitude             -0.144160
"""
```

Let me try to explain to you what stratify means, in the median_income columns the income ranges from 👇

```
df['median_income'].min(), df['median_income'].max()
(0.4999, 15.0001)
```

Actually, 0.4999 means $5000, and 15.0001 means $150000. So when diving the dataset into train(80%) and test set(20%) if the train set has 80 values between the range $5000(0.5)-$30000(3.0) then the test set must have 20 values between the same range. Similarly, if the train set has 200 values between the range of $30000-$60000 then the test set must have 40 values between that range. That's called stratifying.

(I was thinking so deeply to explain this to you. I know the concept but I don't know how to put it into words. I hope the above example is clear.)

So, to do that we first need to create a new column named "***income_cat***" which will assign the value 1 to median_income ranging from $5000(0.5)–$15000(3.0), 2 to median_income ranging from $30000–$45000 and so on.

To do that we are going to use the **cut()** method from pandas.

```
import numpy as n
df['income_cat'] = p.cut(df['median_income'], bins=[0., 1.5, 3.0, 4.5,
6.0, n.inf], labels=[1, 2, 3, 4, 5])
df.head()
"""
OUTPUT:

longitude latitude housing_median_age total_rooms total_bedrooms
population households median_income median_house_value
ocean_proximity income_cat
0 -122.23 37.88 41.0 880.0 129.0 322.0 126.0 8.3252 452600.0 NEAR
BAY 5
1 -122.22 37.86 21.0 7099.0 1106.0 2401.0 1138.0 8.3014 358500.0
NEAR BAY 5
2 -122.24 37.85 52.0 1467.0 190.0 496.0 177.0 7.2574 352100.0 NEAR
BAY 5
3 -122.25 37.85 52.0 1274.0 235.0 558.0 219.0 5.6431 341300.0 NEAR
BAY 4
4 -122.25 37.85 52.0 1627.0 280.0 565.0 259.0 3.8462 342200.0
```

```
NEAR BAY 3
"""
```

As you can see from the above output values ranging from 6.0 and more are assigned 5 in the "***income_cat***" column.

Now let's split it. Stratified by "***median_income***" we stratify using the stratify parameter. See the below code fully.

```
strat_train_set, strat_test_set = train_test_split(df, test_size=0.2,
stratify=df['income_cat'], random_state=8)
strat_train_set['income_cat'].value_counts()
"""
OUTPUT:
3    5789
2    5265
4    2911
5    1890
1     657
"""

strat_test_set['income_cat'].value_counts()
"""
OUTPUT:
3    1447
2    1316
4     728
5     472
1     165
"""
```

You can now easily understand now by seeing the above output.

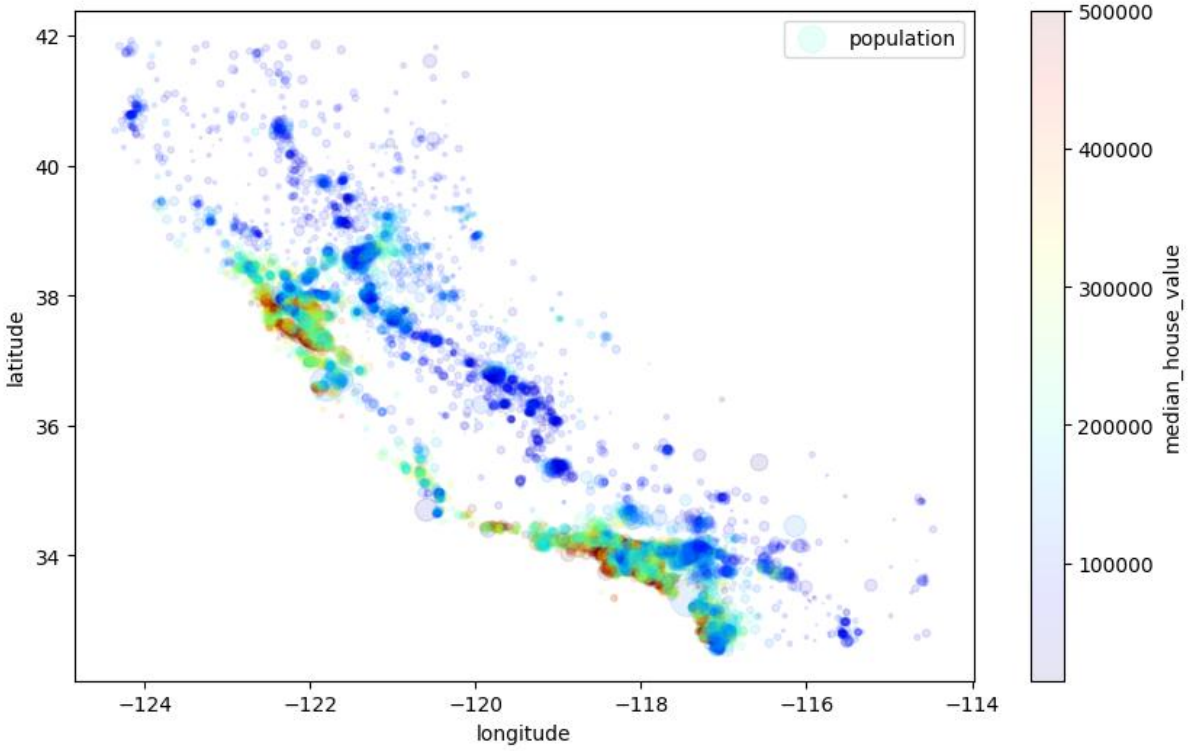
Now there is no need for the "***income_cat***" column any more let's drop it.

```
strat_train_set.drop(columns='income_cat', inplace=True)
strat_test_set.drop(columns='income_cat', inplace=True)
```

Let's plot California.

Visualize the Data to gain insights

```
housing.plot(kind='scatter', x='longitude', y='latitude', figsize=(10, 6),
s=housing['population']/100, label='population', alpha=0.1, cmap='jet',
c='median_house_value', colorbar='True')
```



It's California!

Prepare the data for machine learning algorithms

Now let's divide the dataset into features and labels.

```
housing = strat_train_set.drop(columns='median_house_value')
housing_label = strat_train_set['median_house_value'].copy()
```

Clean and Preprocess the dataset

The first step in data preprocessing is handling missing values let's do that.

In our dataset we only have one column with missing values which is "***total_bedrooms***" as you can see below.

```
housing.isnull().sum()
"""
OUTPUT:
longitude             0
latitude              0
housing_median_age    0
total_rooms           0
total_bedrooms      120
population            0
households            0
median_income         0
ocean_proximity       0
dtype: int64
```

```
''''''
```

You can handle this using 3 different ways.

1. Drop the whole columns (using **drop()**)
2. Drop the null value rows (using **dropna()**)
3. Replace the null values with the mean, median, or mode of the column. (You can simply use fillna() or there is a Imputer called SimpleImputer).

I am going to use the SimpleImputer and fill the null values of "***total_bedrooms***" with the median value of the column.

But I am not going to do that like we always do I am going to use something called Pipeline.

Using Pipeline we can combine more preprocessing steps together I will show you how.

So, we have decided that we are going to use Pipeline then let's also complete one more preprocessing step which is Standardizations.

As we saw earlier that the values in our numerical columns are not standard, to make them standard we are going to use StandardScaler.

Let's see how the pipeline looks.

```
housing_num = housing.drop(columns='ocean_proximity')

from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
   ('impute', SimpleImputer(strategy='median')),
   ('standarize', StandardScaler())
])

housing_num_tr = num_pipeline.fit_transform(housing_num)

housing_num_tr
"""
OUTPUT:
array([[-0.81067237,  1.03452915, -0.72069548, ...,  2.4034797 ,
        1.2857664 , -1.09294375],
       [ 1.87267473, -0.62650195, -0.63972359, ..., -0.62867088,
        -0.80880713, -0.32985752],
       [-1.50192   ,  1.22450045, -0.96361113, ..., -0.04680458,
        0.448446  , -0.41642476],
       ...,
       [ 0.64258053, -0.5144676 , -0.96361113, ...,  0.40527417,
        0.29828823,  0.59066784],
       [-1.22977526,  1.7408327 ,  0.33193903, ...,  0.28681038,
        0.7258561 ,  0.13792923],
       [-1.30597579,  1.61905623, -0.96361113, ...,  5.04278343,
        4.66049849, -0.23261094]])
"""
```

That's it our numerical columns are clean now. Let's clean and transform our categorical colum. Actually, it's already clean we just need to transform let's do that.

But we are not going to do that like we always do, I am again going to introduce you to a new concept called ColumnTransform (If you are already familiar with this, good).

```
num_attributes = list(housing_num)
cat_attributes = ['ocean_proximity']

from sklearn.compose import ColumnTransformer

full_pipeline = ColumnTransformer([
    ('num', num_pipeline, num_attributes),
    ('cat', OneHotEncoder(), cat_attributes)
])
```

I know we already transformed our numerical column. But the above **full_pipeline** will allow us to transform more new data. Now we made a full pipeline to take care of our preprocessing.

Now let's completely clean and preprocess the original dataset using our pipeline.

```
housing_prepared = full_pipeline.fit_transform(housing)
```

Select and train a Model

I feel like this article is too long. I will quickly complete the models part, bare with me.

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
lr.fit(housing_prepared, housing_label)
some_data = housing.iloc[:5]
some_label = housing_label.iloc[:5]
some_prepared_data = full_pipeline.transform(some_data)
some_prediction = lr.predict(some_prepared_data)

from sklearn.metrics import mean_squared_error,
mean_absolute_error
error = mean_squared_error(some_label, some_prediction)
n.sqrt(error) #44910.39810668252
```

Now let's try decision tree.

```
from sklearn.tree import DecisionTreeRegressor
dtr = DecisionTreeRegressor()
dtr.fit(housing_prepared, housing_label)
dtr_prediction = dtr.predict(some_prepared_data)
dtr_error = mean_squared_error(some_label,dtr_prediction)
n.sqrt(dtr_error) #0.0
```

Man, have you ever met Overfitting. Here it is.

Now let's evaluate the model using cross validation.

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(dtr, housing_prepared, housing_label,
scoring='neg_mean_squared_error', cv=10)
n.sqrt(-scores)
'''''''
OUTPUT:
array([69740.80471327, 70734.99564574, 73755.52852368,
70040.82810658,
       67227.25562064, 67951.38661292, 70667.18481112,
66094.49515777,
       71296.61173991, 68635.00515019])
''''''
n.sqrt(-scores).mean() #69614.40960818216
```

Let's try Random Forest.

```
from sklearn.ensemble import RandomForestRegressor
rfr = RandomForestRegressor()
rfr.fit(housing_prepared, housing_label)
rfr_scores = cross_val_score(rfr, housing_prepared, housing_label,
scoring='neg_mean_squared_error', cv=10)
n.sqrt(-rfr_scores), n.sqrt(-rfr_scores).mean()
''''''
OUTPUT:
(array([44499.93441923, 46591.08193298, 47141.51224975,
49985.21257502,
       43002.94210925, 47355.26799223, 42809.13137683,
50503.69111973,
       45274.34750506, 48175.34812435]),
```

```
 46533.84694044267)
"""
```

The Random forest model is better than the other two model.

Do you know how to download your model. Here is how.

```
import joblib

joblib.dump(rfr, 'rfr_model.pkl')
```

Using Joblib you can download your model and then do whatever you want.

That's it guys.

See you tomorrow.

Day — 6☑

Day-7: Heart Disease Prediction 💗

Today, I found this dataset called Heart Disease Prediction it is a small dataset with just 240 rows and 14 columns.

Get the dataset from here👉Heart Disease Prediction Dataset.

Let's get into the code.

Load the dataset

```
import pandas as p
df = p.read_csv('/content/Heart_Disease_Prediction.csv')
```

Understand the dataset

```
df.head()

"""
OUTPUT:

Age Sex Chest pain type BP Cholesterol FBS over 120 EKG results Max HR Exercise angina ST depression Slope of ST Number of vessels fluro Thallium Heart Disease
0 70 1 4 130 322 0 2 109 0 2.4 2 3 3 Presence
1 67 0 3 115 564 0 2 160 0 1.6 2 0 7 Absence
2 57 1 2 124 261 0 0 141 0 0.3 1 0 7 Presence
3 64 1 4 128 263 0 0 105 1 0.2 2 1 7 Absence
```

```
4 74 0 2 120 269 0 2 121 1 0.2 1 1 3 Absence
"""
df.shape # (270, 14) It's a really really small dataset.
df.info()
"""
OUTPUT:

0s
# It's a really really small dataset.
df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 270 entries, 0 to 269
Data columns (total 14 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Age                   270 non-null    int64
 1   Sex                   270 non-null    int64
 2   Chest pain type       270 non-null    int64
 3   BP                    270 non-null    int64
 4   Cholesterol           270 non-null    int64
 5   FBS over 120          270 non-null    int64
 6   EKG results           270 non-null    int64
 7   Max HR                270 non-null    int64
 8   Exercise angina       270 non-null    int64
 9   ST depression         270 non-null    float64
 10  Slope of ST           270 non-null    int64
 11  Number of vessels fluro  270 non-null    int64
 12  Thallium              270 non-null    int64
 13  Heart Disease         270 non-null    object
dtypes: float64(1), int64(12), object(1)
"""



df.isnull().sum()
```

```
"""
OUTPUT:
9]
0s
# It has just 1 categorical column and that is our Label/Target. I think it is going to be really easy.
df.isnull().sum()
Age                     0
Sex                     0
Chest pain type         0
BP                      0
Cholesterol             0
FBS over 120            0
EKG results             0
Max HR                  0
Exercise angina         0
ST depression           0
Slope of ST             0
Number of vessels fluro    0
Thallium                0
Heart Disease           0
"""
```

What!!! there are no null values. It's an easy day😀.

Also, there is no categorical column it's all numerical. Just the "***Heart Disease***" Column is categorical and that's our target column.

Let's replace the values(Presence and Absence) in the "Heart Disease" column with numbers(0s and 1s).

```
df['Heart Disease'] = df['Heart Disease'].replace({'Presence':1,
'Absence':0})
```

I am going to use machine learning algorithms that do not require scaling and standardizing the features. So, I am straight away moving to the splitting step 🔨.

```
y = df['Heart Disease']
X = df.drop(columns='Heart Disease')
```

Now let's import the models.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
```

Let's start with the Decision Tree Classifier. 🌳

I am going to do Cross-Validation. We do Cross-Validation using the **cross_val_score()** method.

In the **cross_val_score()** method I need to pass 5 main things.

1. Model
2. Features

3. Target/Label
4. Number of folds (For example, if you have 100 rows in your dataset in the first round the first 80 rows will be used for training and the remaining 20 will be used for testing. In the second round, the first 60(1-60) and the last 20(81-100) will be used for training and the rest will be used for training and so on. This happens 5 times(as I mentioned cv=5) and returns us 5 different scores we can get the mean of it using the mean() method).
5. Scoring (In the scoring I need to choose an evaluation metric which can be accuracy, precision score, recall score, f1 score, and others. In this case, I used accuracy.

```
from sklearn.model_selection import cross_val_score

dtc_scores = cross_val_score(DecisionTreeClassifier(), X, y, cv=5, scoring='accuracy')

print(dtc_scores) #[0.68518519 0.68518519 0.81481481 0.72222222 0.81481481]

print(dtc_scores.mean()) #0.7444444444444445
```

Well, that's not a bad score but let's try other models.

```
models = [DecisionTreeClassifier(), RandomForestClassifier(), XGBClassifier()]
```

```
for model in models:
  scores = cross_val_score(model, X, y, cv=5, scoring='accuracy')
  print(model, scores.mean())

"""
OUTPUT:

DecisionTreeClassifier() 0.7592592592592593

RandomForestClassifier() 0.8333333333333333

XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None,
early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None,
feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None,
max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan,
monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None, random_state=None, ...)
0.7814814814814814

"""
```

The Random Forest Classifier model is giving me good accuracy maybe I should go with that.

Let's try to improve our accuracy by adding some hyperparameters. To do this we are going to use something called **GridSearchCV**.

Before that let's split the data using train test split to see how our new model will work on new unseen data.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=9)

from sklearn.model_selection import GridSearchCV
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 20]}

grid_search = GridSearchCV(RandomForestClassifier(), param_grid, cv=5, scoring='accuracy')

grid_search.fit(X_train, y_train)

grid_search.best_estimator_ #
RandomForestClassifier(max_depth=10)
```

It is telling me to set ***max_depth=10*** for better accuracy let's try it.

```
rfc = RandomForestClassifier(max_depth=10)

rfc.fit(X_train, y_train)

rfc_prediction = rfc.predict(X_test)
```

```
from sklearn.metrics import accuracy_score

rfc_score = accuracy_score(y_test, rfc_prediction)

rfc_score #0.8703703703703703😉
```

That's good.

Today was the easiest day. Tomorrow I will find a more challenging dataset than this one.

I can see this challenge is working. I would really recommend you to take up this challenge if you want to get into machine learning and you are a beginner.

How this challenge has helped me so far,

1. I am being exposed to a lot of new concepts and techniques.
2. I am getting clear on what to learn.
3. I am getting to know a little about what algorithm to use for a particular dataset.
4. Last, but not least, I am forming a habit of coding every day.

If you are someone who knows more than me, I would love to get feedback from you on my coding. You can also tell me if there is something that I am not doing correctly.

If you are like me(Beginner), it would be great if you take up this challenge along with me. I will share what I am learning with you.

That's it guys.

See yaa...

Day — 7☑

Day-8: Titanic Survival Prediction 🚢

Today I used Titanic Survival Prediction Dataset. The most used dataset in the history of Kaggle.

If you are interested in taking this 30-day challenge start with this dataset. There is also a step-by-step tutorial available to help you do this project. Below are the links for both the step-by-tutorial and the Titanic dataset.

Titanic Tutorial

Titanic Dataset

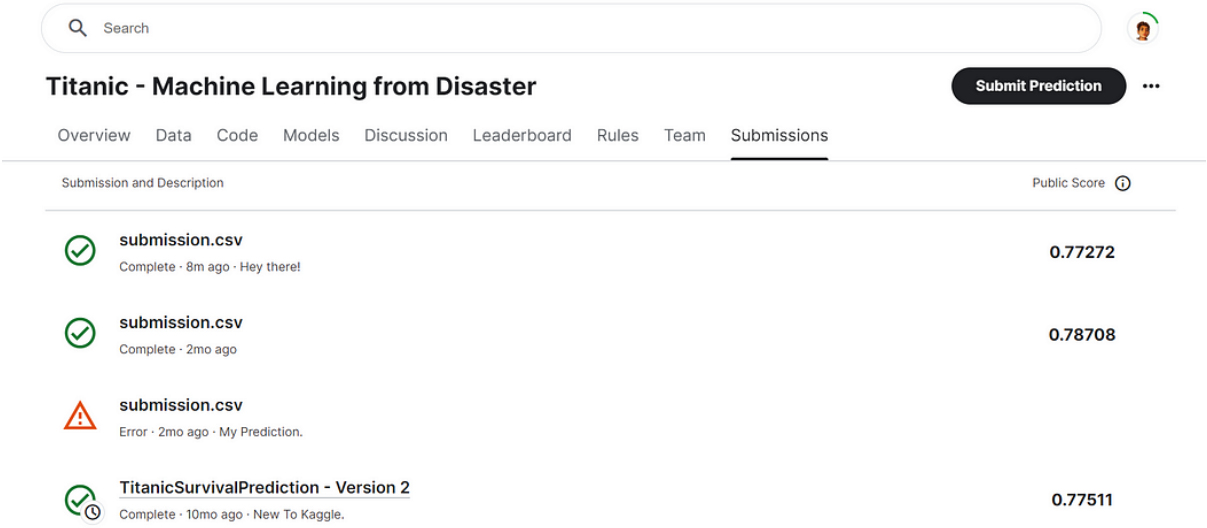
Let's get into the code.

I started by importing 🐼🐼🐼🐼.

```
import pandas as p
```

You will get two datasets, one named "train" and the other named "test". In the train set, we will have both features and the label but in the test set, we will only have the features.

It's because after creating a model using the train set you need to check the performance of your model with the test data and then you can submit your prediction to Kaggle. Like I did 👇



I got 77% accuracy today!

(I will also tell you how to submit your prediction to Kaggle in this article).

Then I imported both the train and test sets. Copied the "**PassengerId**" column in the test set to a variable called *passengerid*. You will know why I did this in the end.

```
train = p.read_csv('TitanicTrain.csv')
test = p.read_csv('TitanicTest.csv')
passengerid = test['PassengerId']
```

Then, as usual, to understand the dataset more clearly.

```
train.head() # Gives the first 5 rows of the dataset
train.shape # (891, 12)
train.isnull().sum()
"""
OUTPUT:
0s
train.isnull().sum()
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
"""
```

There are 3 columns with null values in the train set. Let's take a look at the test set.

```
test.shape # (418, 11)
test.isnull().sum()
"""
OUTPUT:
PassengerId      0
Pclass           0
```

```
Name          0
Sex           0
Age          86
SibSp         0
Parch         0
Ticket        0
Fare          1
Cabin       327
Embarked      0
"""
```

In both cases the "**Cabin**" column seems to have a lot of null values, So I thought it would be better to drop it. Also, the "**Name**" column is not going to be useful let's drop that as well.

```
train = train.drop(columns=['Name', 'Cabin'])
test = test.drop(columns=['Name', 'Cabin'])
```

To deal with the null values present in the "**Age**" column, I used the **fillna()** method and replaced the null values with the mean value of the column.

```
train_age_mean = train['Age'].mean()
test_age_mean = test['Age'].mean()

train['Age'].fillna(train_age_mean, inplace=True)
test['Age'].fillna(test_age_mean, inplace=True)
```

The “Embarked” column in the train set has 2 null values in it. So it is better to replace them with the mode value(most present value in the column).

```
train['Embarked'].fillna(train['Embarked'].mode()[0], inplace=True)
```

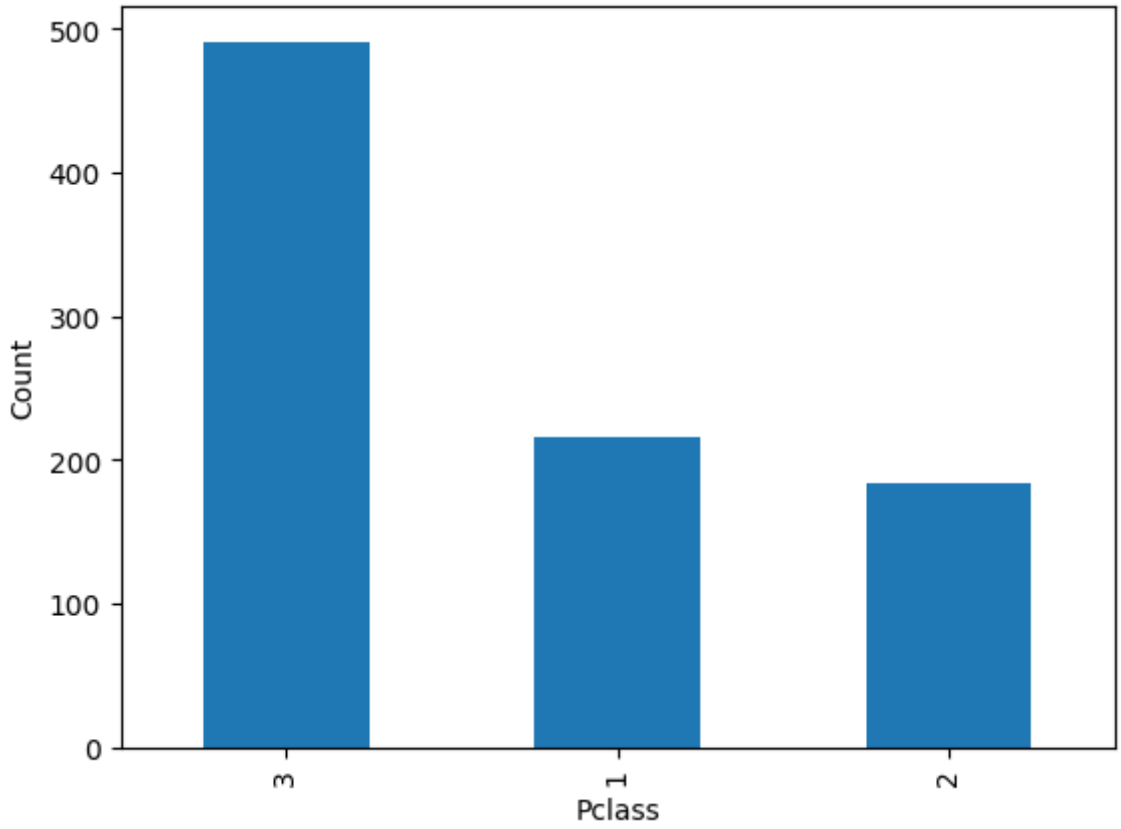
The “**PassengerId**” column is also useless.

```
train.drop(columns='PassengerId', inplace=True)
test.drop(columns='PassengerId', inplace=True)
```

Now, I thought of doing some visualization because, in the last 7 projects, I was not doing any visualization. So, I just plotted some bar charts using Matplotlib and Seaborn libraries.

EDA

```
Pclass_count = train.Pclass.value_counts()
Pclass_count.plot(kind='bar', xlabel='Pclass', ylabel='Count')
```

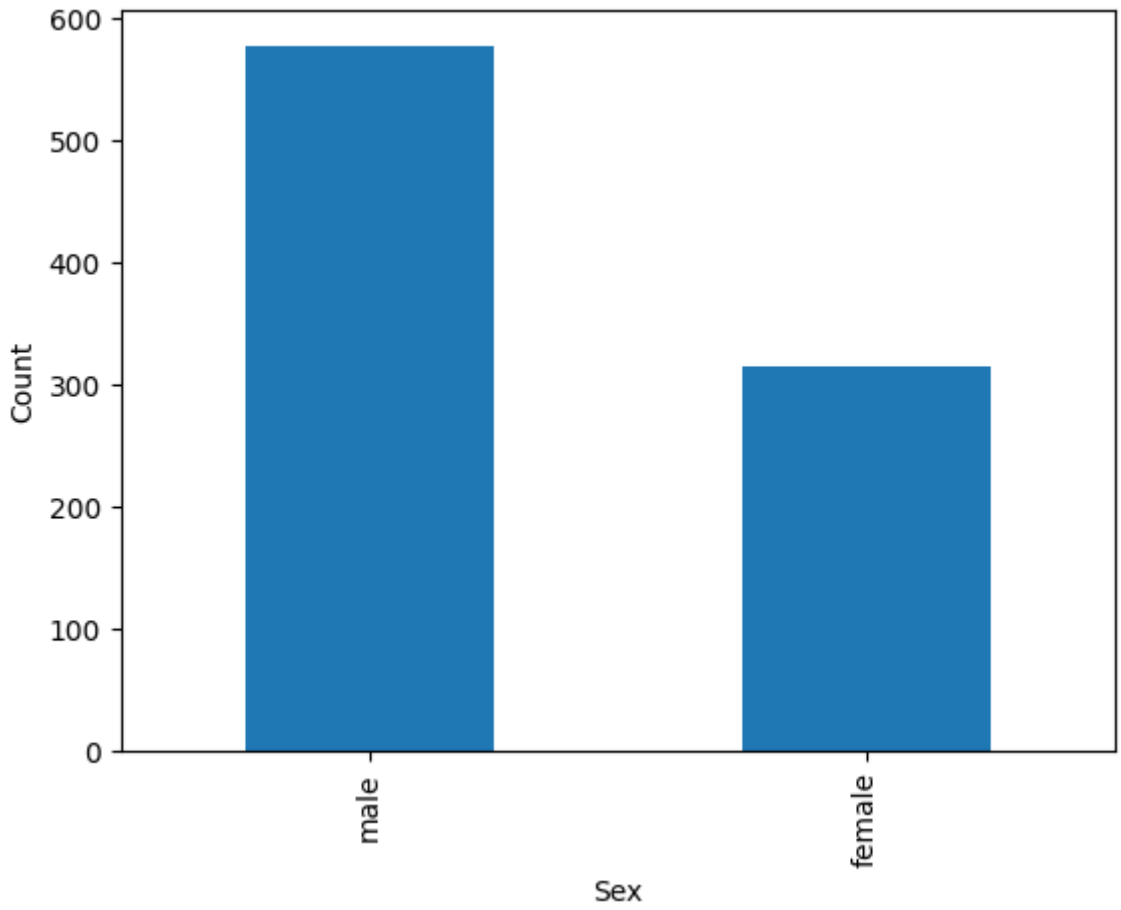


3rd class

Most of the people on the Titanic were traveling in 3rd class.

Let's see how many men and women were there in the Titanic.

```
sex_count = train.Sex.value_counts()
sex_count.plot(kind='bar', xlabel='Sex', ylabel='Count')
```

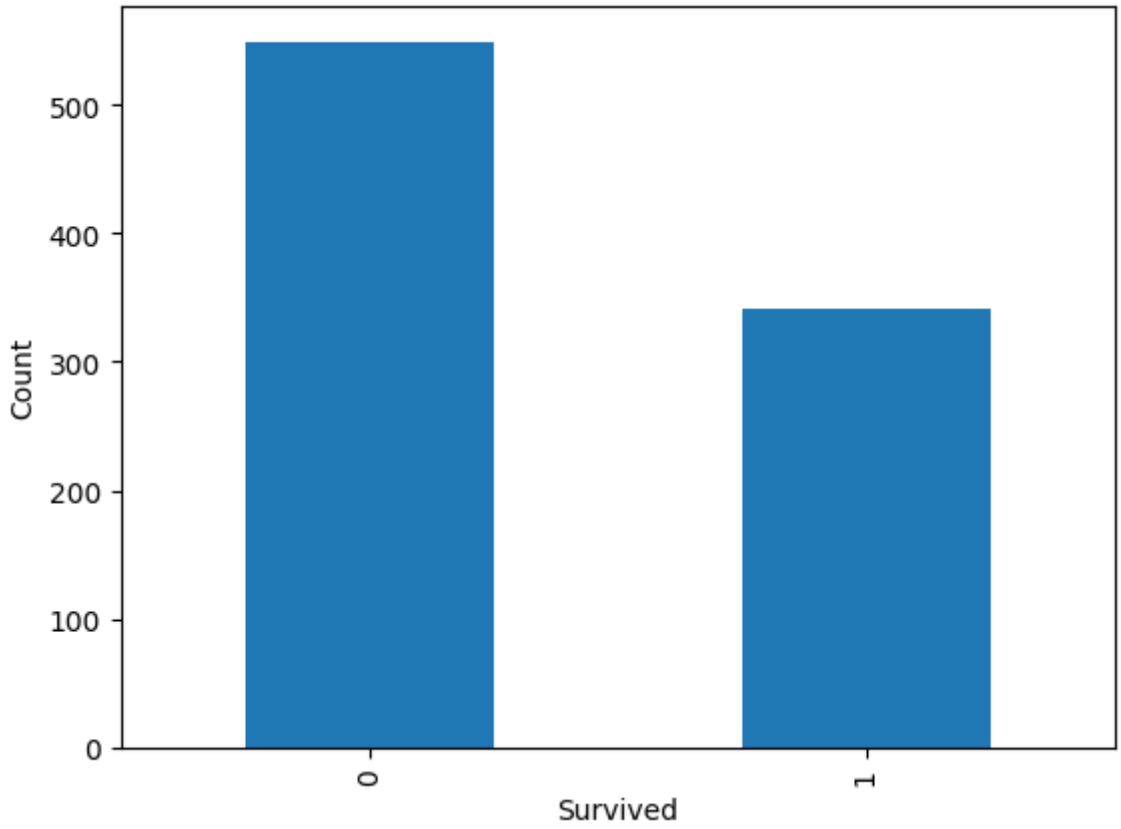


More men

There was double the number of men traveling than women.

Let's see how many people survived and how many people died.

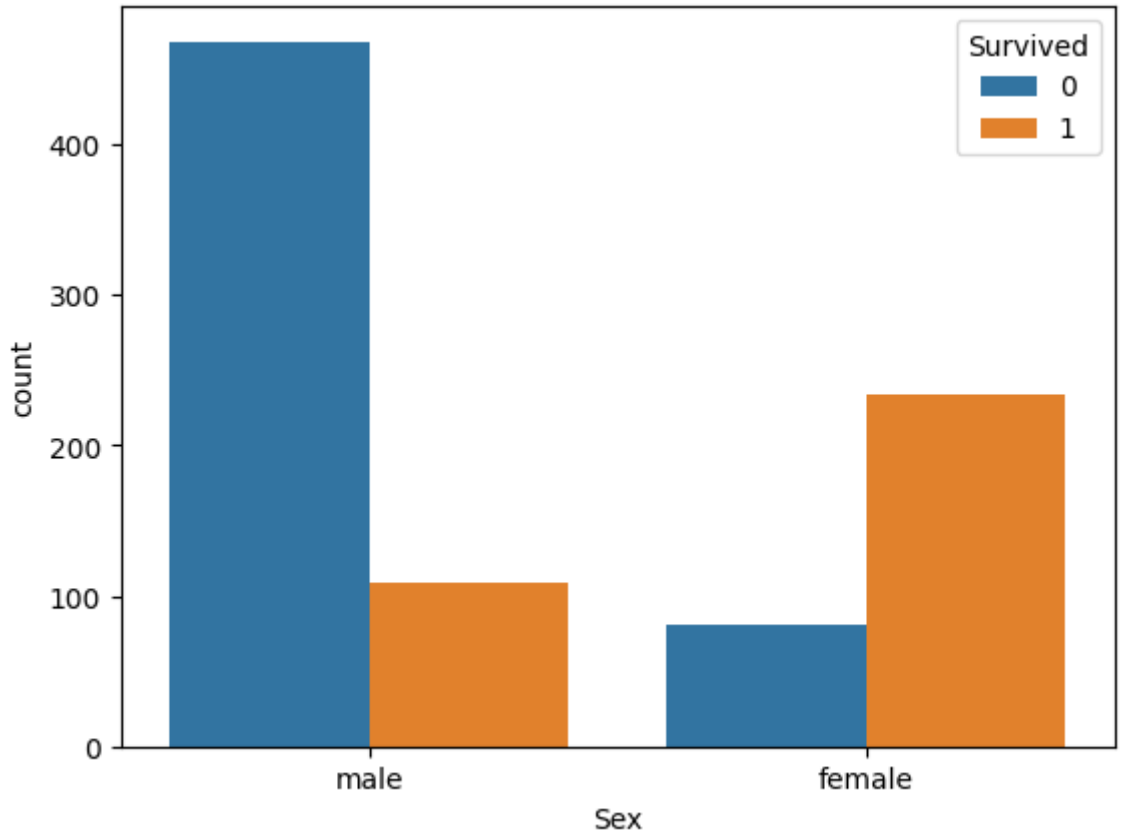
```
Survived_count = train.Survived.value_counts()
Survived_count.plot(kind='bar', xlabel='Survived', ylabel='Count')
```



Most People died!

Now using seaborn let's know how may men died and how many females died.

```
import seaborn as s
s.countplot(train, x='Sex', hue='Survived')
```

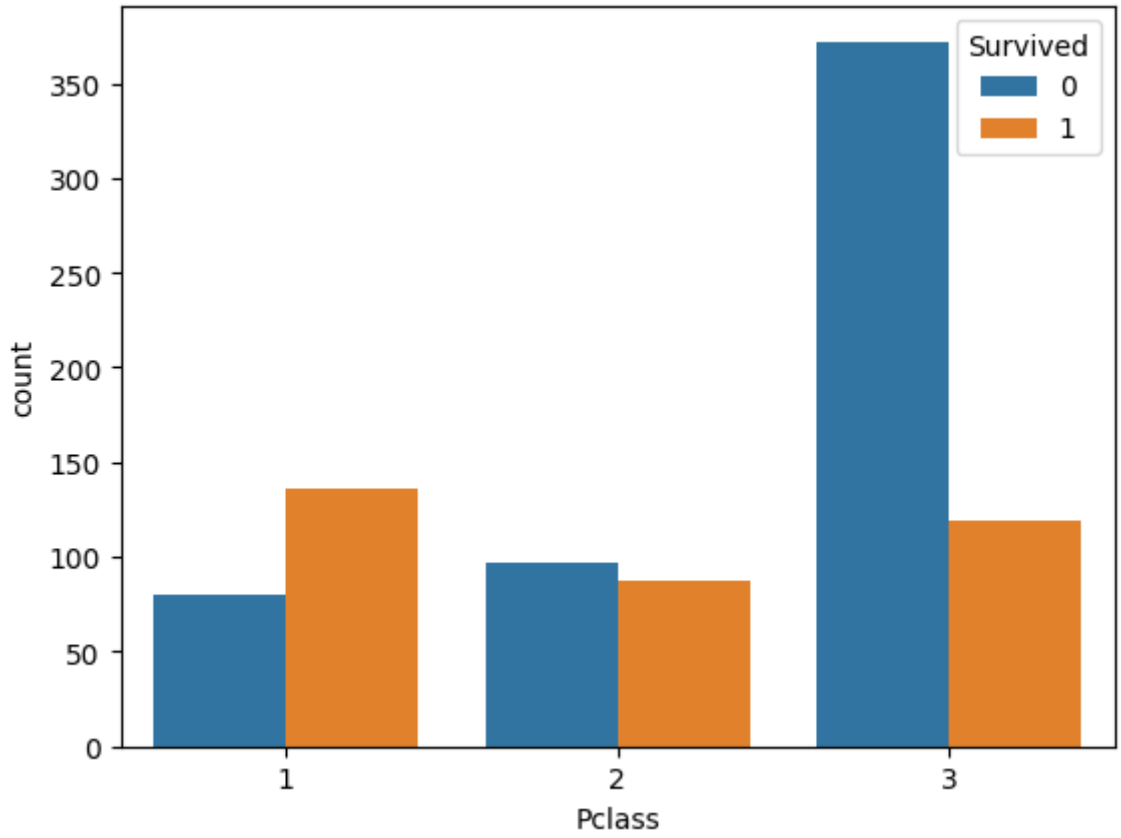


Men are great!

So, after hitting the iceberg, the men on the Titanic decided to use the lifeboats to first save the women and children. Respect 🧐. (Now they are talking about feminism, what a world😒).

Let's also see how which class of people survived the most.

```
s.countplot(train, x='Pclass', hue='Survived')
```



That's obvious!

Being rich has some benefits. So, most people who died in the Titanic Crash was males that to 3rd class males. Come on boys get rich😂.

Enough of the visualization, let's convert the categorical column values into numbers. And also drop the "**Ticket**" column.

```
train['Sex'].replace({'male':1, 'female': 0}, inplace=True)
test['Sex'].replace({'male':1, 'female': 0}, inplace=True)

train['Embarked'].replace({'S':2, 'C': 1, 'Q':0}, inplace=True)
test['Embarked'].replace({'S':2, 'C': 1, 'Q':0}, inplace=True)
```

```
train.drop(columns='Ticket', inplace=True)
test.drop(columns='Ticket', inplace=True)
```

Now let's split the train set into features and labels.

```
y = train.Survived.copy()
X = train.drop(columns='Survived').copy()
```

We need some data to test our models performance so, let's split this further into training and testing sets.

```
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=0.2,
random_state=786)
```

I am going to try 3 different models.

1. Decision Tree Classifier
2. Random Forest Classifier
3. Logistic Regression

Let's see which one is going to give me good accuracy.

Decision Tree Classifier

```
from sklearn.tree import DecisionTreeClassifier
dtc = DecisionTreeClassifier()
dtc.fit(Xtrain, ytrain)
dtc_prediction = dtc.predict(Xtest)

from sklearn.metrics import accuracy_score
dtc_score = accuracy_score(ytest, dtc_prediction)
dtc_score # 0.7486033519553073 (74% Not bad)
```

Random Forest Classifier

```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier(max_depth=15)
rfc.fit(Xtrain, ytrain)
rfc_prediction = rfc.predict(Xtest)
rfc_score = accuracy_score(ytest, rfc_prediction)
rfc_score # 0.7932960893854749 (That's good)
```

Logistic Regression

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression()
lr.fit(Xtrain, ytrain)
lr_prediction = lr.predict(Xtest)
lr_score = accuracy_score(ytest, lr_prediction)
lr_score # 0.7374301675977654 (Nah!)
```

Okay the Random Forest Classifier seems to perform well let's try to improve it further.

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [10, 15, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

rfc = RandomForestClassifier()


grid_search = GridSearchCV(estimator=rfc, param_grid=param_grid,
cv=5, n_jobs=-1)
grid_search.fit(Xtrain, ytrain)

best_params = grid_search.best_params_
print("Best Hyperparameters:", best_params)

best_rfc = RandomForestClassifier(**best_params)
best_rfc.fit(Xtrain, ytrain)

rfc_prediction = best_rfc.predict(Xtest)


rfc_score = accuracy_score(ytest, rfc_prediction)
print("Accuracy Score:", rfc_score)

"""
OUTPUT:
```

```
Best Hyperparameters: {'max_depth': 10, 'min_samples_leaf': 1,
'min_samples_split': 2, 'n_estimators': 150}
Accuracy Score: 0.7988826815642458
"""
```

There is a very sight improvement from 79.3% to 79.8%.

Good.

Let's use this model and get the predictions for our test data and then submit it to Kaggle.

```
test['Fare'].fillna(test['Fare'].mean(), inplace=True) # Forgot to do
this before


my_prediction = best_rfc.predict(test)

submission_df = p.DataFrame({'PassengerId':passengerid,
'Survived':my_prediction})
submission_df.to_csv('submission.csv', index=False)
```

Now, you will have a new file named "submission.csv" in your current working directory. You can submit that to Kaggle.

That's it.

Day — 8☑

Day-9: Movie Recommendation System 🎞️

Today, instead of doing the same the same classification and regression projects I thought of trying something different. So, I decided to do a project related to the recommendation system.

I was searching for it on YouTube and found a movie recommendation system project. Below is the link to the video.

https://youtu.be/7rEagFH9tQg

You can also find the dataset in the description of the video. I learned a few things that I didn't know previously from this video.

I also feel like within a few days from now I will become good at working on classification and regression tasks. So, instead of spending too much time on them. I decided to work on other tasks that I had never worked on before.

I may do projects on supervised learning in between this challenge but I will also make sure that I am focusing on unsupervised learning topics such as clustering, association, anomaly detection, dimensionality reduction, and more.

I am sharing all this because if you are like me it will give you an idea of what to do and how to approach machine learning as a beginner.

You start by first understanding the fundamentals of machine learning then immediately you should move to coding and building projects you will understand more from doing than by just studying.

You should make mistakes, it's okay if you are not able to understand some concepts they will start to make sense to you slowly but surely if you just learn consistently.

Let me share with you my code.

First import 🐼 and load the dataset.

```
import pandas as p
movies = p.read_csv('movies.csv')
```

Then to understand the data you can use the below lines one by one.

```
movies.head()
movies.shape
movies.isnull().sum()
```

We are not going to use all the columns in this dataset. We are only going to select a few columns, they are,

```
needed_cols = ['genres', 'keywords', 'tagline', 'cast', 'director']
df = movies[needed_cols]
```

```
df.isnull().sum()
"""
OUTPUT:
genres      28
keywords    412
tagline     844
cast        43
director    30
"""
```

All the columns we choose are categorical we will be converting them to vectors(numbers) in a while using **TfidfVectorizer()** method.

Before that, we need to replace the null value with an empty string.

```
for col in needed_cols:
  df[col] = df[col].fillna('')

df.isnull().sum()
"""
OUTPUT:
genres      0
keywords    0
tagline     0
cast        0
director    0
"""
```

Now, we are going to combine all the features into one. Also, you need to understand in this project we are not going to make any predictions.

What we are going to do is. We will give the user a prompt like **"Enter your favourite movie name: "**. After the user enters a movie name he/she will be recommended other movies similar to that of the movie they entered.

We are going to find the similarity between the movies using something called cosine similarity you will that how in a while.

Here is how you combine the features 👇.

```
combined_cols = df['genres'] + " " + df['keywords'] + " " + df['tagline']
+ " " + df['cast'] + " " + df['director']
combined_cols
"""
OUTPUT:
0       Action Adventure Fantasy Science Fiction cultu...
1       Adventure Fantasy Action ocean drug abuse exot...
2       Action Adventure Crime spy based on novel secr...
3       Action Crime Drama Thriller dc comics crime fi...
4       Action Adventure Science Fiction based on nove...
                            ...
4798    Action Crime Thriller united states\u2013mexic...
4799    Comedy Romance  A newlywed couple's honeymoon ...
4800    Comedy Drama Romance TV Movie date love at fir...
4801     A New Yorker in Shanghai Daniel Henney Eliza...
4802    Documentary obsession camcorder crush dream gi...
```

```
"""
```

Let's see fully what is on the first row.

```
combined_cols[0]
"""
OUTPUT:
Action Adventure Fantasy Science Fiction culture clash future space
war space colony society Enter the World of Pandora. Sam
Worthington Zoe Saldana Sigourney Weaver Stephen Lang Michelle
Rodriguez James Cameron
"""
```

It's a single string consisting of all the details of the movie from its genres, taglines, keywords, cast, and directors.

As you know computers are dumb they can't understand words we need to convert them into numbers using the **TfidfVectorizer()** as I told you before.

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
vectorized_combined_cols = vectorizer.fit_transform(combined_cols)
print(vectorizer_combined_cols)
```

It will return a sparse matrix. You can also check how the array looks like using,

```
import numpy as n
n.set_printoptions(threshold=n.inf)
vectorized_combined_cols.toarray()[0]
```

It will give a big array.

It's time to use the ***cosine_similarity()***. It will give us a matrix(like a correlation matrix) It carries the percentage of similarity each movie has with each other.

For example, the movie Iron Man-1 will have a similarity score of 1 with movies like Iron Man-2 and Iron man-3. And it may have a similarity score of 0.8(80%) with movies like Avengers, Avengers: Age of Ultron, and Avengers: End-Game.

I hope you can understand. Below is the code.

```
from sklearn.metrics.pairwise import cosine_similarity
similarity = cosine_similarity(vectorized_combined_cols)
n.set_printoptions(threshold=n.10) #This will print only the first 10 rows from the numpy array
print(similarity)
"""
OUTPUT:
[[1.         0.07219487 0.037733   ... 0.         0.         0.        ]
```

```
 [0.07219487 1.         0.03281499 ... 0.03575545 0.         0.        ]
 [0.037733   0.03281499 1.         ... 0.         0.05389661 0.        ]
 ...
 [0.         0.03575545 0.         ... 1.         0.         0.02651502]
 [0.         0.         0.05389661 ... 0.         1.         0.        ]
 [0.         0.         0.         ... 0.02651502 0.         1.        ]]
"""

similarity.shape # (4803, 4803)
```

Now we are going to create a list that will have all the movie names in it.

```
movie_titles = movies['title'].tolist()
movie_titles
"""
OUTPUT:
['Avatar',
 "Pirates of the Caribbean: At World's End",
 'Spectre',
 'The Dark Knight Rises',
 'John Carter',
 'Spider-Man 3',
 'Tangled',
 'Avengers: Age of Ultron',
 'Harry Potter and the Half-Blood Prince',
 'Batman v Superman: Dawn of Justice',
 'Superman Returns',
 'Quantum of Solace',
 "Pirates of the Caribbean: Dead Man's Chest",
 'The Lone Ranger',
 'Man of Steel',
```

'The Chronicles of Narnia: Prince Caspian',
'The Avengers',
'Pirates of the Caribbean: On Stranger Tides',
'Men in Black 3',
'The Hobbit: The Battle of the Five Armies',
'The Amazing Spider-Man',
'Robin Hood',
'The Hobbit: The Desolation of Smaug',
'The Golden Compass',
'King Kong',
'Titanic',
'Captain America: Civil War',
'Battleship',
'Jurassic World',
'Skyfall',
'Spider-Man 2',
'Iron Man 3',
'Alice in Wonderland',
'X-Men: The Last Stand',
'Monsters University',
'Transformers: Revenge of the Fallen',
'Transformers: Age of Extinction',
'Oz: The Great and Powerful',
'The Amazing Spider-Man 2',
'TRON: Legacy',
'Cars 2',
'Green Lantern',
'Toy Story 3',
'Terminator Salvation',
'Furious 7',
'World War Z',
'X-Men: Days of Future Past',
'Star Trek Into Darkness',
'Jack the Giant Slayer',

```
 'The Great Gatsby',
 'Prince of Persia: The Sands of Time',
 'Pacific Rim',
 'Transformers: Dark of the Moon',
 'Indiana Jones and the Kingdom of the Crystal Skull',
 'The Good Dinosaur',
 'Brave',
 'Star Trek Beyond',
 'WALL·E',
 'Rush Hour 3',
 '2012',
 'A Christmas Carol',
 'Jupiter Ascending',
 'The Legend of Tarzan' ...]
"""
```

We are going to import a library that you might have never known yet(even I don't know, but now I do).

It's called **difflib,** this library will help us find similar words in a list. This is how it works. Say there is a list like ['Iron Man', 'Avengers', 'Spider-Man', 'Iron Man-2, 'Captain America']. Now if the user enters something like, **"ironman2"** it will find the more similar words from the list and prints them.

Below is the code.

```
import difflib
user_input = input("Enter your favorite movie to get similar
```

```
recommendations: ")
close_movie_names = difflib.get_close_matches(user_input,
movie_titles)
print(close_movie_names)

"""
OUTPUT:
Enter your favorite movie to get similar recommendations: ironman2
['Iron Man 2', 'Birdman', 'Iron Man']
"""
```

We don't need all the names in the list, we are only going to take out the first name from the list because the first name will be more relavant.

```
close_match = close_movie_names[0]
print(close_match) #'Iron Man 2'
```

Now we need to find the index value of the movie.

```
index_of_the_movie = movies[movies.title ==
close_match]['index'].values[0]
print(index_of_the_movie) #79
```

Let's get all the similarity scores this movie(Iron Man 2) has with all other movies using the "*similarity*" we created before (**similarity = cosine_similarity(vectorized_combined_cols)**).

```
similarity_score = list(enumerate(similarity[index_of_the_movie]))
print(similarity_score)
```

After that, we are going to sort the movies in descending order so that more similar movies are at the top.

```
sorted_similar_movies = sorted(similarity_score, key = lambda x:x[1], reverse = True)
print(sorted_similar_movies)
```

Now we are going to use a for loop and get all the titles of the movies and print them out. The below code will print us 30 movies similar to that of 'Iron Man 2'.

I love RDJ. Do you?

```
print('Movies suggested for you : \n')

i = 1

for movie in sorted_similar_movies:
  index = movie[0]
  title_from_index = movies[movies.index==index]['title'].values[0]
  if (i<30):
    print(i, '.',title_from_index)
    i+=1
```

"""

OUTPUT:

Movies suggested for you :

1 . Iron Man 2

2 . Iron Man 3

3 . Avengers: Age of Ultron

4 . Iron Man

5 . The Avengers

6 . Captain America: Civil War

7 . Ant-Man

8 . X-Men: Apocalypse

9 . X-Men

10 . Captain America: The Winter Soldier

11 . Deadpool

12 . X2

13 . X-Men: Days of Future Past

14 . Thor: The Dark World

15 . The Incredible Hulk

16 . X-Men: The Last Stand

17 . Man of Steel

18 . The Amazing Spider-Man 2

19 . The Image Revolution

20 . Superman II

21 . X-Men: First Class

22 . Batman v Superman: Dawn of Justice

23 . Sin City

24 . The Jungle Book

25 . The Spirit

26 . Made

27 . X-Men Origins: Wolverine

28 . Spawn

29 . Red Sonja

"""

That’s it guys.

See you tomorrow. With another project.

Day — 9☑

Day-10: Diabetes Prediction 🍫

I was a little busy today, but that doesn't mean I will break my streak.

Today, I picked up a really…really easy dataset and worked with that. I usually don't like working with easy datasets but today, unfortunately, I didn't have the time and energy to work on hard datasets.

So, this is what I did today.

```
import pandas as p
```

I got two datasets this time, one for training and the other one for testing.

You can download the dataset from here: Diabetes Dataset

```
diabetesTrain = p.read_csv('DiabetesTrain.csv')
diabetesTest = p.read_csv('DiabetesTest.csv')

diabetesTrain.shape, diabetesTest.shape # ((2460, 9), (308, 9))

df = diabetesTrain.copy()
```

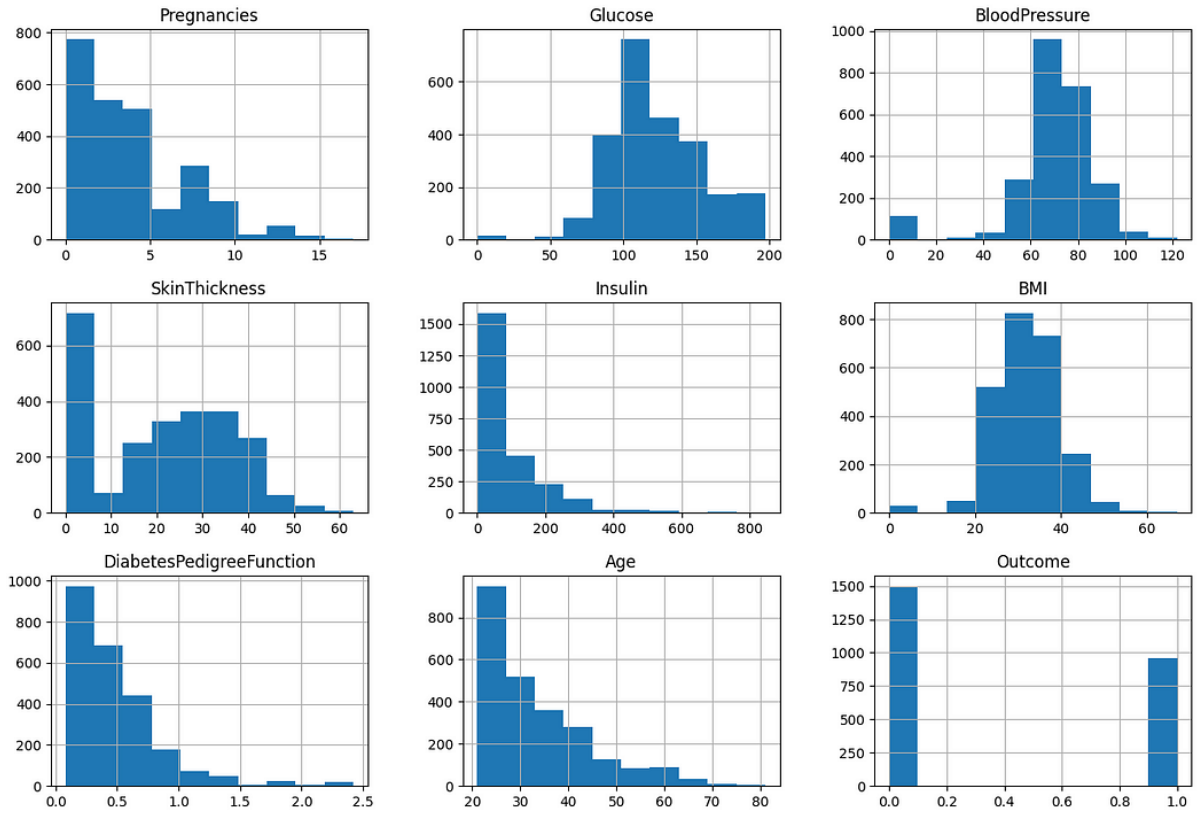
```
df.isnull().sum()
"""
OUTPUT:
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
"""


df.info()
"""
OUTPUT:
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               2460 non-null   int64
 1   Glucose                   2460 non-null   int64
 2   BloodPressure             2460 non-null   int64
 3   SkinThickness             2460 non-null   int64
 4   Insulin                   2460 non-null   int64
 5   BMI                       2460 non-null   float64
 6   DiabetesPedigreeFunction  2460 non-null   float64
 7   Age                       2460 non-null   int64
 8   Outcome                   2460 non-null   int64
"""
```

I loaded the datasets and then checked their shape, it's a small one. Then I copied the training set into a data frame called **df**.

Then I checked some information about the data frame it seems to have no null values and all the values are integers, my work is really easy now.

After that, I wanted to know how the values in each column are spread out, so I plotted a histogram📊.

```
df.hist(figsize=(15, 10))
```



We have values with different ranges for every column in the dataset.

Let's now first focus on splitting the data frame **df** into train and test.

```
sample_y = df['Outcome']
sample_X = df.drop(columns='Outcome')
from sklearn.model_selection import train_test_split
sampleXtrain, sampleXtest, sampleytrain, sampleytest =
train_test_split(sample_X, sample_y, random_state=84)
```

Now, let's create a Decision Tree Classifier. We don't have to scale or normalize the features because we are going to use Decision Tree.

```
from sklearn.tree import DecisionTreeClassifier
dtc = DecisionTreeClassifier()
dtc.fit(sampleXtrain, sampleytrain)
dtc_prediction = dtc.predict(sampleXtest)

from sklearn.metrics import accuracy_score
dtc_accuracy = accuracy_score(sampleytest, dtc_prediction)
dtc_accuracy #1.0
```

What the heck! I got 100% accuracy. I don't know how something must be wrong.

```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier()
rfc.fit(sampleXtrain, sampleytrain)
rfc_prediction = rfc.predict(X)
```

```
rfc_score = accuracy_score(y, rfc_prediction)
rfc_score #0.7954545454545454
```

Random Forest is giving me 79% Accuracy.

Let's try to improve it.

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

# Define the hyperparameter grid
param_grid = {
    'n_estimators': [100, 200, 300],  # Number of trees in the forest
    'max_depth': [None, 10, 20],      # Maximum depth of the trees
    'min_samples_split': [2, 5, 10],  # Minimum number of samples
required to split an internal node
    'min_samples_leaf': [1, 2, 4]     # Minimum number of samples
required to be at a leaf node
}

# Instantiate the random forest classifier
rf = RandomForestClassifier()

# Instantiate the grid search
grid_search = GridSearchCV(estimator = rf, param_grid = param_grid,
                           cv = 3, n_jobs = -1, verbose = 2)

# Fit the grid search to the data
grid_search.fit(sampleXtrain, sampleytrain)
```

```
# Get the best parameters
best_params = grid_search.best_params_
```

Don't get too confused. I don't know much about the GridSearchCV I am just learning about it. I got the above code from ChatGPT. The GridSearchCV will try all the different combinations of parameters I mentioned above in the **param_grid** and will give me the best one.

Let's see what are all the best parameters.

```
best_params
"""
OUTPUT:
{'max_depth': 20,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'n_estimators': 300}
"""
```

So, the best parameters are the above, let's try them.

```
from sklearn.ensemble import RandomForestClassifier
rfc_updated =
RandomForestClassifier(max_depth=20,min_samples_leaf=1,min_sampl
es_split=2,n_estimators=300)
rfc_updated.fit(sampleXtrain, sampleytrain)
rfc_updated_prediction = rfc_updated.predict(X)
rfc_updated_score = accuracy_score(y, rfc_updated_prediction)
rfc_updated_score #0.788961038961039
```

Nah!

The random forest without the parameter was far better.

I need to learn to tune the models well.

That's it.

Day — 10☑

Day-11: Parkinson's Diseases Prediction🧠

So, today I worked with the Parkinson's Disease Dataset, you can get the below dataset and follow along.

Dataset: *Parkinson's Disease Dataset*

Come on don't just read, if you reading on a mobile phone open your laptop, if you are already reading this on your laptop open Google Colab on a new tab and code with me.

In this article, I going to explain what each line of code is doing with you.

Let's get into the code.

Import 🐼s with its alias **pd** or you can name it whatever you want I named it **p**.

```
import pandas as p
```

After downloading the dataset from the link I provided you can upload that dataset to your Colab and then loaded into a data-frame.

```
df = p.read_csv('ParkinssonDisease.csv')
```

Now let's understand the dataset using some attributes and methods.

```
df.shape # (195, 24)
df.head()
```

The dataset has 24 rows but I am not able to see all of them. So, if I want to see all the columns present in the dataset, I can do something like this👇.

```
p.set_option('display.max_columns', 25)
df.head()
```

The **p.set_option('display.max_columns', 25)** will allow us to see all the columns present in our dataset. If you worked with datasets you know this, if a dataset has more columns let's say 60 and you try to view it using **df.head()** it shows the first five rows but when it comes to columns it will only show the first few columns then (...) then the last few column.

You will not be able to see all the columns that is why we need to use the **set_option()** method.

Let's get some more information about our dataset.

```
df.info()
"""
```

```
OUTPUT:
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   name             195 non-null    object
 1   MDVP:Fo(Hz)      195 non-null    float64
 2   MDVP:Fhi(Hz)     195 non-null    float64
 3   MDVP:Flo(Hz)     195 non-null    float64
 4   MDVP:Jitter(%)   195 non-null    float64
 5   MDVP:Jitter(Abs) 195 non-null    float64
 6   MDVP:RAP         195 non-null    float64
 7   MDVP:PPQ         195 non-null    float64
 8   Jitter:DDP       195 non-null    float64
 9   MDVP:Shimmer     195 non-null    float64
 10  MDVP:Shimmer(dB) 195 non-null    float64
 11  Shimmer:APQ3     195 non-null    float64
 12  Shimmer:APQ5     195 non-null    float64
 13  MDVP:APQ         195 non-null    float64
 14  Shimmer:DDA      195 non-null    float64
 15  NHR              195 non-null    float64
 16  HNR              195 non-null    float64
 17  status           195 non-null    int64
 18  RPDE             195 non-null    float64
 19  DFA              195 non-null    float64
 20  spread1          195 non-null    float64
 21  spread2          195 non-null    float64
 22  D2               195 non-null    float64
 23  PPE              195 non-null    float64
"""
```

Amazing, all we have is numerical values, ignore the "name" column we will drop that, there is no use for that column.

Now, let's check if there are any null values present in any of the columns in the dataset.

```
df.isnull().sum()
"""
OUTPUT:
name              0
MDVP:Fo(Hz)       0
MDVP:Fhi(Hz)      0
MDVP:Flo(Hz)      0
MDVP:Jitter(%)    0
MDVP:Jitter(Abs)  0
MDVP:RAP          0
MDVP:PPQ          0
Jitter:DDP        0
MDVP:Shimmer      0
MDVP:Shimmer(dB)  0
Shimmer:APQ3      0
Shimmer:APQ5      0
MDVP:APQ          0
Shimmer:DDA       0
NHR               0
HNR               0
status            0
RPDE              0
DFA               0
spread1           0
spread2           0
D2                0
PPE               0
"""
```

Thank god.

Our work is easy now.

We don't have to handle any null values and we don't have to handle any categorical values. If you want to know how to handle null values and categorical values you can check the other projects I did in this 30-day challenge.

In each of my projects in this 30-day challenge, I am working with different datasets and learning different concepts and techniques to preprocess, feature engineer, visualize, and choose and build models.

I highly recommend you take up this challenge if you are serious about building a career around AI.

Let's get into the topic.

As there is no use of the "name" column lets drop it.

```
df.drop(columns='name', inplace=True)
```

Now, we can divide our dataset into features(X) and label(y), in our the label column is the "status" column.

```
y = df['status']
X = df.drop(columns='status')
```

If you see the feature(X) columns the values in each column are in different ranges. For example, the "NHR" column has values ranging from (0.00065, 0.31482), and the "MDVP:Flo(Hz)" column has values ranging from (65.476, 239.17). See the below code.

```
X['NHR'].min(), X['NHR'].max() # (0.00065, 0.31482)
X['MDVP:Flo(Hz)'].min(), X['MDVP:Flo(Hz)'].min() # (65.476, 239.17)
```

Not only these two columns every other column is also unique in their values range. This is not good. We cannot feed a machine learning algorithm unstandardized features.

So, let's standardize it using the Standard Scaler.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X = scaler.fit_transform(X)
X

"""
OUTPUT:
array([[-0.82929965, -0.43616456, -0.95203729, ...,  0.48047686,
        -0.21053082,  0.86888575],
       [-0.77097169, -0.53097409, -0.05772056, ...,  1.31118546,
         0.27507712,  1.80360503],
       [-0.90947638, -0.7231683 , -0.10987483, ...,  1.01768236,
        -0.10362861,  1.40266141],
       ...,
       [ 0.49557839,  0.47010361, -0.96839309, ..., -0.81807931,
         0.78033848, -0.83241014],
```

```
       [ 1.07876114,  2.19004398, -0.95417967, ..., -0.22906571,
        -0.63700298, -0.92610456],
       [ 1.45481664,  0.69224632, -0.88348115, ..., -0.43085284,
         0.45480231, -0.64505466]])
"""
```

As you can see after standardizing our features(X) are now standardized. Also understand this, after standardizing our data frame will get converted into a NumPy array.

It's fine to feed NumPy array-type data to our machine-learning model.

But if you want to see your standardized features(X) like a data frame you can do something like this 👇

```
X_columns = df.drop(columns='status').columns
p.DataFrame(X, columns=X_columns)
"""
OUTPUT:

MDVP:Fo(Hz) MDVP:Fhi(Hz) MDVP:Flo(Hz) MDVP:Jitter(%)
MDVP:Jitter(Abs) MDVP:RAP MDVP:PPQ Jitter:DDP MDVP:Shimmer
MDVP:Shimmer(dB) Shimmer:APQ3 Shimmer:APQ5 MDVP:APQ
Shimmer:DDA NHR HNR RPDE DFA spread1 spread2 D2 PPE
0 -0.829300 -0.436165 -0.952037 0.334914 0.749759 0.132963
0.760800 0.131755 0.745985 0.739536 0.607859 1.119147 0.332985
0.607532 -0.067893 -0.193225 -0.807838 1.760814 0.801323
0.480477 -0.210531 0.868886
```

```
1 -0.770972 -0.530974 -0.057721 0.715418 1.037674 0.453892
1.276809 0.452684 1.681731 1.768464 1.547912 2.276504 1.159454
1.548254 -0.137843 -0.634508 -0.387524 1.837562 1.479853
1.311185 0.275077 1.803605
2 -0.909476 -0.723168 -0.109875 0.884991 1.325589 0.720770
1.585687 0.721813 1.202693 1.027636 1.175643 1.726176 0.699187
1.175323 -0.291633 -0.279760 -0.662075 1.942048 1.141445 1.017682
-0.103629 1.402661
3 -0.909622 -0.649092 -0.114229 0.775389 1.325589 0.578885
1.284076 0.577677 1.340396 1.207698 1.340547 1.848749 0.806859
1.340229 -0.280719 -0.281346 -0.613134 1.832380 1.440945
1.293840 0.062145 1.806954
4 -0.925657 -0.606245 -0.130608 1.368893 1.901418 1.095750
2.047187 1.09
"""
```

Now, our features look good. Let's move to the next step.

It's time to split 🔧 our features and labels into training sets and testing sets. We can do that using the **train_test_split()** method in the sci-kit learn library.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
stratify=y, random_state=34)
X_train.shape, X_test.shape # ((156, 22), (39, 22))
y_train.shape, y_test.shape # ((156,), (39,))
```

Splitting is done.

If you have been following all my projects from day 1 you know by now, I have been using the decision tree algorithm and the random forest algorithm a lot.

But this time I am going to use Support Vector Machine.

Let's build a Support Vector Classifier model and see how much accuracy it can give me.

```
from sklearn.svm import SVC
svc = SVC()
svc.fit(X_train, y_train)
svc_prediction = svc.predict(X_test)

from sklearn.metrics import accuracy_score
svc_score = accuracy_score(y_test, svc_prediction)
svc_score # 0.8974358974358975
```

89% is Good.

So that's it.

Day — 11☑

Day-12: MNIST Hand-Written Digits Recognition 🔢

Today I worked with a very famous dataset. It's called MNIST (Digits).

Below is a simple definition of MNIST Datasets.

The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning.

This is the first dataset that everyone will use when getting started with image-related tasks like image classification, image detection, and more.

Working with this dataset was unique compared to other datasets though the process was the same.

I learned this from the 3rd chapter of "**Hands-on Machine Learning with Scikit-learn, Keras, and TensorFlow**". I would highly recommend you read that chapter, it will make you a beast in solving classification problems.

Okay enough.

Let me share my code with you.

You don't have to download the dataset from some website for this. We can get the dataset directly from the **sklearn** library.

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
```

Now our MNIST Hand-Written digits dataset is fetched and stored in a variable called **mnist**.

Let's see what we have in it.

```
mnist.keys()
"""
OUTPUT:
dict_keys(['data', 'target', 'frame', 'categories', 'feature_names',
'target_names', 'DESCR', 'details', 'url'])
"""
```

We got some information about the **mnist**. Let's explore them one by one.

```
mnist.data
"""
OUTPUT:

pixel1 pixel2 pixel3 pixel4 pixel5 pixel6 pixel7 pixel8 pixel9 pixel10 ...
pixel775 pixel776 pixel777 pixel778 pixel779 pixel780 pixel781
```

```
pixel782 pixel783 pixel784
0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
2 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
3 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
"""
```

The **mnist.data** is a data frame containing all the feature column.

As we are working with images the features are just pixel values. There are 784 columns it's because we are working with 28x28 dimensional images (28 * 28 = 784).

Let's know how many hand-written digits images we have.

```
mnist.data.shape # (70000, 784)
```

We got 70000 handwritten digit images. Amazing, we can train our model very well with this many data points. But when fitting the model to our dataset it is going to take a lot of time. (I hope you are coding along).

```
mnist.target
"""
OUTPUT:

0       5
1       0
2       4
3       1
4       9
       ..
69995   2
69996   3
69997   4
69998   5
69999   6
"""
```

In the **mnist.target** we got our labels we can also call it target.

```
mnist.frame
"""
OUTPUT:

pixel1 pixel2 pixel3 pixel4 pixel5 pixel6 pixel7 pixel8 pixel9 pixel10 ...
pixel776 pixel777 pixel778 pixel779 pixel780 pixel781 pixel782
pixel783 pixel784 class
0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 5
1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0
```

```
2 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 4
3 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 1
4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
"""
```

mnist.frame is the complete data frame with both features and values, it's a combination of **mnist.data** and **mnist.target**.

```
mnist.feature_names #This will give us all the feature names there is no use in running this
mnist.DESCR
```

The above code will gives us all the information about the dataset such as who are the authors of the dataset, how many features are there, what is the dimension, and more. You can run the above code and see it yourself.

```
mnist.details
"""
OUTPUT:
{'id': '554',
 'name': 'mnist_784',
 'version': '1',
 'description_version': '2',
 'format': 'ARFF',
 'creator': ['Yann LeCun', 'Corinna Cortes', 'Christopher J.C. Burges'],
```

```
 'upload_date': '2014-09-29T03:28:38',
 'language': 'English',
 'licence': 'Public',
 'url':
'https://api.openml.org/data/v1/download/52667/mnist_784.arff',
 'parquet_url':
'https://openml1.win.tue.nl/datasets/0000/0554/dataset_554.pq',
 'file_id': '52667',
 'default_target_attribute': 'class',
 'tag': ['AzurePilot',
  'OpenML-CC18',
  'OpenML100',
  'study_1',
  'study_123',
  'study_41',
  'study_99',
  'vision'],
 'visibility': 'public',
 'minio_url':
'https://openml1.win.tue.nl/datasets/0000/0554/dataset_554.pq',
 'status': 'active',
 'processing_date': '2020-11-20 20:12:09',
 'md5_checksum': '0298d579eb1b86163de7723944c7e495'}
"""
```

There is no use of all this information, but you can use them if you want to know more details about the dataset.

Finally,

```
mnist.url
"""
OUTPUT:
https://www.openml.org/d/554
"""
```

You can also download the dataset using the above URL, but there is no need for that when we can directly load it from the **sklearn** library.

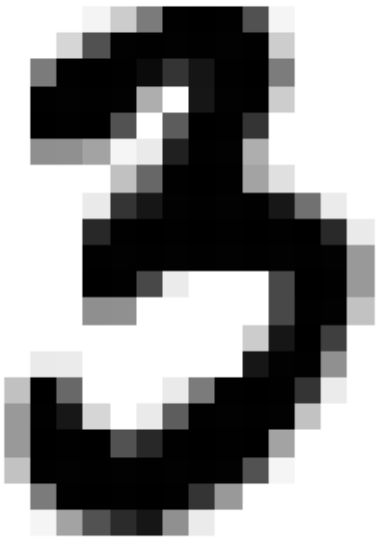
We are done exploring, let's do the real stuff.

I am going to put the features in X and target in y.

```
X = mnist.data.copy()
y = mnist.target.copy()
```

Let's first see our data visually using matplotlib.

```
import matplotlib.pyplot as mp
import numpy as n

first_digit = n.array(X.iloc[890]).reshape(28, 28)
mp.imshow(first_digit, cmap='binary')
mp.axis(False)
print(y[890])
```



3

In the above code, I first grabbed the 890th row from the dataset then converted it to an NumPy array then reshaped it to 28x28 because that is the dimension of our image.

After that using the **imshow()** method in the matplotlib library I displayed the NumPy array values like an image.

So, the features in 890th rows represents **3.** You can try different row values.

I forgot to tell you this, we got no null values.

Now, it's time to split 🔨.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=786)
X_train.shape, X_test.shape # ((56000, 784), (14000, 784))
y_train.shape, y_test.shape # ((56000,), (14000,))
```

I am going to use SVC, you can try different classifier if you want.

(When running the below code it's going to take some time, so be patient)

```
from sklearn.svm import SVC
svc = SVC()
svc.fit(X_train, y_train)
svc_prediction = svc.predict(X_test)

from sklearn.metrics import accuracy_score, confusion_matrix
svc_accuracy = accuracy_score(y_test, svc_prediction)
svc_accuracy # 0.9764285714285714
```

I told you, because the amount of data points is huge, our accuracy is awesome.

It's 97% baby. That's amazing.

But's let's also know the confusion matrix.

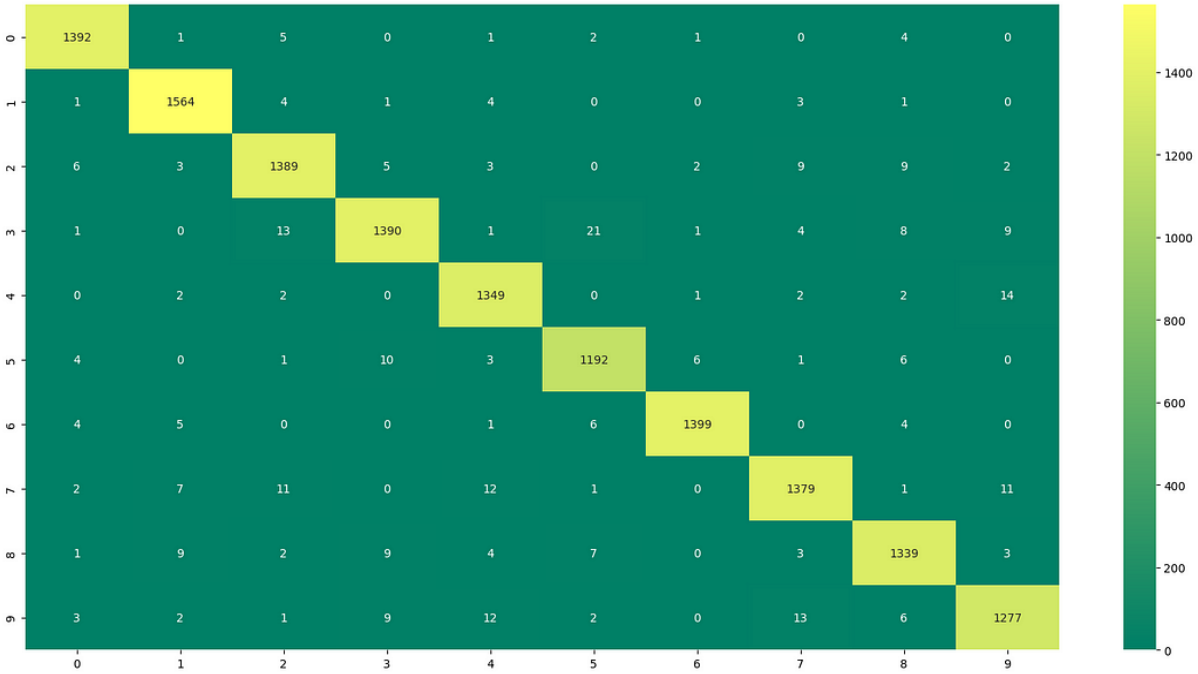
Confusion matrix will tell us how many times the model correctly predicted 1 as 1, 2 as 2, 3 as 3, and so on. It also tell as how may times it predicted 5 as 3, 3 as 5, 1 as 7, 7 as 1 and so on.

Am I confusing you, see the below code and heatmap you will understand what I am trying to say.

```
svc_confusion_matrix = confusion_matrix(y_test, svc_prediction)
svc_confusion_matrix
"""OUTPUT:
array([[1392,   1,   5,   0,   1,   2,   1,   0,   4,   0],
       [   1, 1564,   4,   1,   4,   0,   0,   3,   1,   0],
       [   6,   3, 1389,   5,   3,   0,   2,   9,   9,   2],
       [   1,   0,  13, 1390,   1,  21,   1,   4,   8,   9],
       [   0,   2,   2,   0, 1349,   0,   1,   2,   2,  14],
       [   4,   0,   1,  10,   3, 1192,   6,   1,   6,   0],
       [   4,   5,   0,   0,   1,   6, 1399,   0,   4,   0],
       [   2,   7,  11,   0,  12,   1,   0, 1379,   1,  11],
       [   1,   9,   2,   9,   4,   7,   0,   3, 1339,   3],
       [   3,   2,   1,   9,  12,   2,   0,  13,   6, 1277]])
"""
```

Heatmap 🔥 👇

```
import seaborn as s
mp.figure(figsize=(20, 10))
s.heatmap(svc_confusion_matrix, annot=True,  fmt='d',
cmap='summer')
```



Looks good!

See the first column. It says 0 has been correct predicted as 0, 1392 times. 0 has been predicted as 1, 1 times. 0 has been predicted as 2, 6 times. Now I hope you can understand confusion matrix(They gave the right name, it's confusing sometimes).

I also tried another model called **SGDClassifier**.

```
from sklearn.linear_model import SGDClassifier
sgdc = SGDClassifier()
sgdc.fit(X_train, y_train)
sgdc_prediction = sgdc.predict(X_test)
sgdc_accuracy = accuracy_score(y_test, sgdc_prediction)
sgdc_accuracy #0.843
```

84% not bad.

Then I tried the **SGDClassifier** model again after standardizing the features. But this time I used cross validation instead of train test split.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit_transform(X_train)

from sklearn.model_selection import cross_val_score

sgdc_scores = cross_val_score(sgdc, X_train, y_train, cv=3,
scoring="accuracy")
sgdc_scores # array([0.87009161, 0.86318101, 0.86762027])
```

It worked.

Before I got an accuracy of 84% now it's 86% approximately.

That's it.

Day — 12☑

Day-13: Space Ship Titanic 🚀

Today was a little tough but learned a few things that I didn't know previously.

I was working with the Spaceship Titanic Dataset from Kaggle. You can get the dataset using the below link.

Spaceship Titanic Dataset

You have to download both the train set and the test set. After downloading them and loading them onto your Google Colab you can just code along with me.

Let's get started.

Loading the dataset

Import the pandas 🐼 library as usual.

```
import pandas as p
```

Then let's load both the train set and the test set that we have downloaded. Also, we will store the "**PassengerId**" column in the test set in a separate variable which will come in handy when we want to submit our prediction to Kaggle.

```
train = p.read_csv('SpaceshipTitanicTrain.csv')
test = p.read_csv('SpaceshipTitanicTest.csv')
passengerid = test["PassengerId"]
```

Understanding the dataset

Now let's understand the dataset like we always do.

```
train.shape, test.shape # ((8693, 14), (4277, 13))
train.head()
"""
OUTPUT:

PassengerId HomePlanet CryoSleep Cabin Destination Age VIP
RoomService FoodCourt ShoppingMall Spa VRDeck Name Transported
0 0001_01 Europa False B/0/P TRAPPIST-1e 39.0 False 0.0 0.0 0.0 0.0
0.0 Maham Ofracculy False
1 0002_01 Earth False F/0/S TRAPPIST-1e 24.0 False 109.0 9.0 25.0
549.0 44.0 Juanna Vines True
2 0003_01 Europa False A/0/S TRAPPIST-1e 58.0 True 43.0 3576.0
0.0 6715.0 49.0 Altark Susent False
3 0003_02 Europa False A/0/S TRAPPIST-1e 33.0 False 0.0 1283.0
371.0 3329.0 193.0 Solam Susent False
4 0004_01 Earth False F/1/S TRAPPIST-1e 16.0 False 303.0 70.0
151.0 565.0 2.0 Willy Santantines True
"""
```

It's a complex dataset. Let's check if there are any null values present.

```
train.isnull().sum()
"""
OUTPUT:
PassengerId       0
HomePlanet      201
CryoSleep       217
Cabin           199
Destination     182
Age             179
VIP             203
RoomService     181
FoodCourt       183
ShoppingMall    208
Spa             183
VRDeck          188
Name            200
Transported       0
"""

test.isnull().sum()
"""
OUTPUT:
HomePlanet       87
CryoSleep        93
Cabin           100
Destination      92
Age              91
VIP              93
RoomService      82
FoodCourt       106
ShoppingMall     98
Spa             101
VRDeck           80
```

```
"""
```

Yep! We got null values in almost all the columns. We will handle them one by one, but before that let's remove the unwanted columns. In this case, the "PassengerId" column and the "Name" column are of no use.

Handling Null values

```
train.drop(columns=['Name', 'PassengerId'], inplace=True)
test.drop(columns=['Name', 'PassengerId'], inplace=True)
```

I was checking the dataset
using **train.head(30)** and **train.tail(30)** and found that most of the rows are repeating the same previous value. To give you an example, in the "**HomePlanet**" column the value in row 5 is "**Earth"** so does in row 4. Again, the value in row 8670 is "**Europa**" so does in row 8669.

So, for some columns we can replace the null values with it's previous value using the **ffill()** method.

```
train[['HomePlanet', 'CryoSleep', 'Destination', 'VIP']] =
train[['HomePlanet', 'CryoSleep', 'Destination',
'VIP']].fillna(method='ffill')
test[['HomePlanet', 'CryoSleep', 'Destination', 'VIP']] =
test[['HomePlanet', 'CryoSleep', 'Destination',
'VIP']].fillna(method='ffill')
```

```
train.isnull().sum()
"""
OUTPUT:
HomePlanet      0
CryoSleep       0
Cabin         199
Destination     0
Age           179
VIP             0
RoomService     181
FoodCourt       183
ShoppingMall    208
Spa           183
VRDeck          188
Transported     0
"""
```

We handled the missing values for few column, but still there are plenty of columns with null values. I would like to first handle all the null values in the categorical column before moving on to the numerical column.

Right now the only categorical column with null value is the "**Cabin**" column. And it is also a complex column with 3 different values, read below to understand clearly

Cabin - The cabin number where the passenger is staying. Takes the form deck/num/side, where side can be either P for Port or S for Starboard.

So, we got deck/num/side. I am going to split the "Cabin" column into 3 separate columns namely, "**CabinDeck**", "**CabinNum**", and "**CabinSide**".

```
train[['CabinDeck', 'CabinNum', 'CabinSize']] =
train['Cabin'].str.split('/', expand=True)
test[['CabinDeck', 'CabinNum', 'CabinSize']] =
train['Cabin'].str.split('/', expand=True)
```

Let's check the number of unique values present in all the columns.

```
HomePlanet       3
CryoSleep        2
Cabin         6560
Destination      3
Age             80
VIP              2
RoomService    1273
FoodCourt      1507
ShoppingMall   1115
Spa            1327
VRDeck         1306
Transported      2
CabinDeck        8
CabinNum      1817
CabinSize        2
```

After looking at this, I felt there is no need for the "**Cabin**" and "**CabinNum**" columns they are useless. Let's remove them,

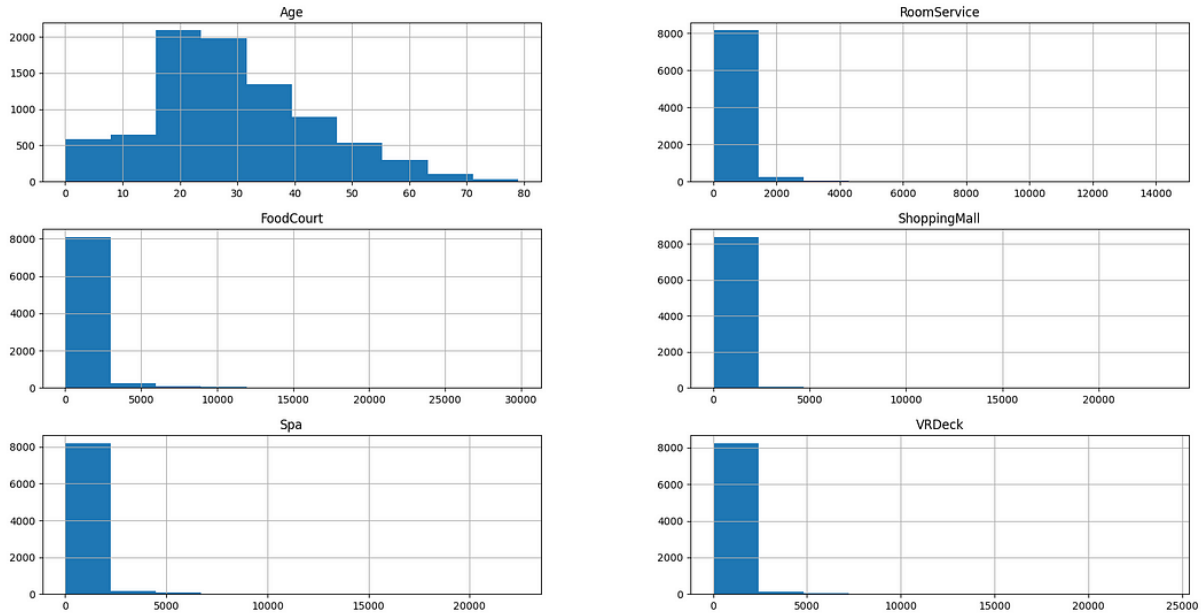
```
train.drop(columns=['Cabin', 'CabinNum'], inplace=True)
test.drop(columns=['Cabin', 'CabinNum'], inplace=True)
```

Even the "**CabinDeck**" and the "**CabinSize**" columns tend to repeat their previous values. Let's use **ffill()** method on them as well.

```
train['CabinDeck'] = train['CabinDeck'].fillna(method='ffill')
train['CabinSize'] = train['CabinSize'].fillna(method='ffill')

test['CabinDeck'] = test['CabinDeck'].fillna(method='ffill')
test['CabinSize'] = test['CabinSize'].fillna(method='ffill')
```

Null values in the categorical columns are handled. Let's move on to the numerical column. Before that let's know the value spread of all the numerical columns using a histogram.

```
train.hist(figsize=(20, 10))
```



Okay!

So, only the "**Age**" column has values spreading across a wide range. Other columns are having a lot of 0s in them.

After looking at this histogram. I have decided to fill the null values in the "**Age**" column with it's **mean** value and the fill the null values of other numerical columns with it's **mode** value.

```
train['Age'] = train['Age'].fillna(train['Age'].mean())
test['Age'] = test['Age'].fillna(test['Age'].mean())
train['RoomService'] =
train['RoomService'].fillna(train['RoomService'].mode()[0])
test['RoomService'] =
test['RoomService'].fillna(test['RoomService'].mode()[0])

train['FoodCourt'] =
train['FoodCourt'].fillna(train['FoodCourt'].mode()[0])
test['FoodCourt'] =
```

```
test['FoodCourt'].fillna(test['FoodCourt'].mode()[0])

train['ShoppingMall'] =
train['ShoppingMall'].fillna(train['ShoppingMall'].mode()[0])
test['ShoppingMall'] =
test['ShoppingMall'].fillna(test['ShoppingMall'].mode()[0])

train['Spa'] = train['Spa'].fillna(train['Spa'].mode()[0])
test['Spa'] = test['Spa'].fillna(test['Spa'].mode()[0])

train['VRDeck'] = train['VRDeck'].fillna(train['VRDeck'].mode()[0])
test['VRDeck'] = test['VRDeck'].fillna(test['VRDeck'].mode()[0])
```

Done!

Now, we have a clean dataset with zero null values.

```
train.isnull().sum()
"""
OUTPUT:

HomePlanet      0
CryoSleep       0
Destination     0
Age             0
VIP             0
RoomService     0
FoodCourt       0
ShoppingMall    0
Spa             0
VRDeck          0
```

```
Transported     0
CabinDeck       0
CabinSize       0
"""

test.isnull().sum()
"""
OUTPUT;
HomePlanet      0
CryoSleep       0
Destination     0
Age             0
VIP             0
RoomService     0
FoodCourt       0
ShoppingMall    0
Spa             0
VRDeck          0
CabinDeck       0
CabinSize       0
"""
```

That looks neat 👀.

Converting Categorical columns and other non-numerical columns

As you already know computers can only understand numbers so, now let's handle the data types and convert them to numbers.

```
train.dtypes
"""
```

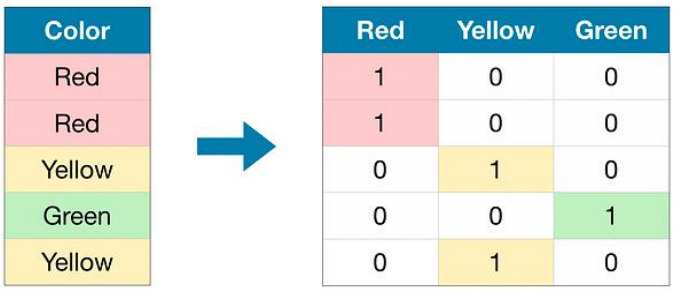
```
OUTPUT:
HomePlanet      object
CryoSleep       bool
Destination     object
Age             float64
VIP             bool
RoomService     float64
FoodCourt       float64
ShoppingMall    float64
Spa             float64
VRDeck          float64
Transported     bool
CabinDeck       object
CabinSize       object
dtype: object
"""
```

Let's start with the Boolean values. It's convert "**True**" to 1 and "**False**" to 0.

```
train[['CryoSleep', 'VIP', 'Transported']] = train[['CryoSleep', 'VIP', 'Transported']].astype(int)
test[['CryoSleep', 'VIP']] = test[['CryoSleep', 'VIP']].astype(int)
```

Now it's time to convert the categorical columns, I am going to use a technique called "**One-Hot Encoding**".

You can easily understand what "One-Hot Encoding" will do to our dataset seeing the below image.



Source: Kaggle

I will also show how the data-frame will look after performing one-hot encoding. Below is the code to do it.

We can simply use the **get_dummies()** method.

```
df = p.get_dummies(train, columns=['HomePlanet', 'Destination', 'CabinDeck', 'CabinSize'])
p.set_option('display.max_columns', 100)
df
"""
OUTPUT:
 CryoSleep Age VIP RoomService FoodCourt ShoppingMall Spa VRDeck Transported HomePlanet_Earth HomePlanet_Europa HomePlanet_Mars Destination_55 Cancri e Destination_PSO J318.5-22 Destination_TRAPPIST-1e CabinDeck_A CabinDeck_B CabinDeck_C CabinDeck_D CabinDeck_E CabinDeck_F CabinDeck_G CabinDeck_T CabinSize_P CabinSize_S
0 0 39.0 0 0.0 0.0 0.0 0.0 0.0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 1 0
1 0 24.0 0 109.0 9.0 25.0 549.0 44.0 1 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1
2 0 58.0 1 43.0 3576.0 0.0 6715.0 49.0 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 1
3 0 33.0 0 0.0 1283.0 371.0 3329.0 193.0 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 1
4 0 16.0 0 303.0 70.0 151.0 565.0 2.0 1
```

```
"""
```

(See the last few columns shown in the above code's output section)

Building Models and making Predictions

It's time to handle the numerical columns. But you know what we don't have to because we are going to use **DecisionTreeClassifier**, **RandomForestClassifier**, and **XGBClassifier**.

When using these classifier there is no need for standardization.

Let's move on to features and label split then to train and test split.

```
y = df['Transported'].copy()
X = df.drop(columns='Transported').copy()
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=786)
```

Decision Tree Classifier

```
from sklearn.tree import DecisionTreeClassifier
dtc = DecisionTreeClassifier()
dtc.fit(X_train, y_train)
dtc_prediction = dtc.predict(X_test)

from sklearn.metrics import accuracy_score, confusion_matrix
```

```
dtc_accuracy = accuracy_score(y_test, dtc_prediction)
dtc_accuracy # 0.7372052903967797
dtc_confusion_matrix = confusion_matrix(y_test, dtc_prediction)
dtc_confusion_matrix
"""
OUTPUT:
array([[614, 250],
       [207, 668]])
"""
```

73% Not bad. What do you think?

Random Forest Classifier

```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier(random_state=786)
rfc.fit(X_train, y_train)
rfc_prediction = rfc.predict(X_test)

rfc_accuracy = accuracy_score(y_test, rfc_prediction)
rfc_accuracy # 0.7987349051178838
rfc_confusion_matrix = confusion_matrix(y_test, rfc_prediction)
rfc_confusion_matrix
"""
OUTPUT:
array([[722, 142],
       [207, 668]])
"""
```

Better than DTC. 79% is good. Let's try XGB.

XGB Classifier

```
from xgboost import XGBClassifier
xgbc = XGBClassifier(random_state=786)
xgbc.fit(X_train, y_train)
xgbc_prediction = xgbc.predict(X_test)

xgbc_accuracy = accuracy_score(y_test, xgbc_prediction)
xgbc_accuracy #0.7952846463484762
```

Let's stick with RFC.

That's it.

Day — 13☑

Day-14: Tesla Stock Price Prediction 🚘

For the last few days, I have been working on a lot of classification tasks so today I decided to work on a regression task and started my search for a dataset on Kaggle.

After searching for a while, I found the "**Tesla Stock Price Dataset**". I don't know much about the stock market but I do know how to create a model that can predict the stock price.

Don't just read this article open your laptop and code along. You can get the dataset using the below link.

Click this 👉 Tesla Stock Price Dataset.

Let me walk you through my code.

I don't know how many times I am going to write this "Import 🐼🐼". Then load the dataset.

```
import pandas as p
df = p.read_csv('/content/TeslaStockDataset.csv')
```

Let's take a look at the dataset.

```
df.head()
"""
OUTPUT:

date open high low close volume rsi_7 rsi_14 cci_7 cci_14 sma_50
ema_50 sma_100 ema_100 macd bollinger TrueRange atr_7 atr_14
next_day_close
0 2014-01-02 9.986667 10.165333 9.770000 10.006667 92826000
55.344071 54.440118 -37.373644 15.213422 9.682107 9.820167
10.494240 9.674284 0.169472 9.740800 0.395333 0.402641
0.447550 9.970667
1 2014-01-03 10.000000 10.146000 9.906667 9.970667 70425000
53.742629 53.821521 -81.304471 17.481130 9.652800 9.826069
10.495693 9.680190 0.162623 9.776167 0.239333 0.379311 0.432677
9.800000
2 2014-01-06 10.000000 10.026667 9.682667 9.800000 80416500
46.328174 50.870410 -123.427544 -37.824708 9.629467 9.825047
10.496740 9.682577 0.141790 9.797900 0.344000 0.374267
0.426343 9.957333
3 2014-01-07 9.841333 10.026667 9.683333 9.957333 75511500
53.263037 53.406750 -84.784651 -20.779431 9.597747 9.830235
10.503407 9.688051 0.136402 9.837900 0.343334 0.369848
0.420414 10.085333
4 2014-01-08 9.923333 10.246667 9.917333 10.085333 92448000
58.368660 55.423026 60.799662 43.570559 9.573240 9.840239
10.511147 9.695964 0.140837 9.870167 0.329334 0.364060 0.413908
9.835333

"""
```

The dataset is full of numerical values.

```
df.isnull().sum()
"""
OUTPUT:
date             0
open             0
high             0
low              0
close            0
volume           0
rsi_7            0
rsi_14           0
cci_7            0
cci_14           0
sma_50           0
ema_50           0
sma_100          0
ema_100          0
macd             0
bollinger        0
TrueRange        0
atr_7            0
atr_14           0
next_day_close   0
"""
```

Cool! Our work is easy now. This dataset is already clean, we don't have to waste our time cleaning this. Let's know the shape of the dataset.

```
df.shape # (2516, 20)
```

It has 2516 rows and 20 columns.

```
df.info()
"""
OUTPUT:
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   date          2516 non-null   object
 1   open          2516 non-null   float64
 2   high          2516 non-null   float64
 3   low           2516 non-null   float64
 4   close         2516 non-null   float64
 5   volume        2516 non-null   int64
 6   rsi_7         2516 non-null   float64
 7   rsi_14        2516 non-null   float64
 8   cci_7         2516 non-null   float64
 9   cci_14        2516 non-null   float64
 10  sma_50        2516 non-null   float64
 11  ema_50        2516 non-null   float64
 12  sma_100       2516 non-null   float64
 13  ema_100       2516 non-null   float64
 14  macd          2516 non-null   float64
 15  bollinger     2516 non-null   float64
 16  TrueRange     2516 non-null   float64
 17  atr_7         2516 non-null   float64
 18  atr_14        2516 non-null   float64
 19  next_day_close  2516 non-null   float64
```

```
"""
```

As you can see from the above output, the "**date**" column is of type *object*. We need to convert the datatype of the "date" column from *object* to *datetime*.

Also, we can remove the index from the dataset and set the "**date**" column as the index.

```
df['date'] = p.to_datetime(df['date'])
df.set_index('date', inplace=True)

df.head()
"""
OUTPUT:

open high low close volume rsi_7 rsi_14 cci_7 cci_14 sma_50 ema_50
sma_100 ema_100 macd bollinger TrueRange atr_7 atr_14
next_day_close
date
2014-01-02 9.986667 10.165333 9.770000 10.006667 92826000
55.344071 54.440118 -37.373644 15.213422 9.682107 9.820167
10.494240 9.674284 0.169472 9.740800 0.395333 0.402641
0.447550 9.970667
2014-01-03 10.000000 10.146000 9.906667 9.970667 70425000
53.742629 53.821521 -81.304471 17.481130 9.652800 9.826069
10.495693 9.680190 0.162623 9.776167 0.239333 0.379311 0.432677
9.800000
2014-01-06 10.000000 10.026667 9.682667 9.800000 80416500
46.328174 50.870410 -123.427544 -37.824708 9.629467 9.825047
```

```
10.496740 9.682577 0.141790 9.797900 0.344000 0.374267
0.426343 9.957333
2014-01-07 9.841333 10.026667 9.683333 9.957333 75511500
53.263037 53.406750 -84.784651 -20.779431 9.597747 9.830235
10.503407 9.688051 0.136402 9.837900 0.343334 0.369848
0.420414 10.085333
2014-01-08 9.923333 10.246667 9.917333 10.085333 92448000
58.368660 55.423026 60.799662 43.570559 9.573240 9.840239
10.511147 9.695964 0.140837 9.870167 0.329334 0.364060 0.413908
9.835333
"""
```

Now it's perfect.

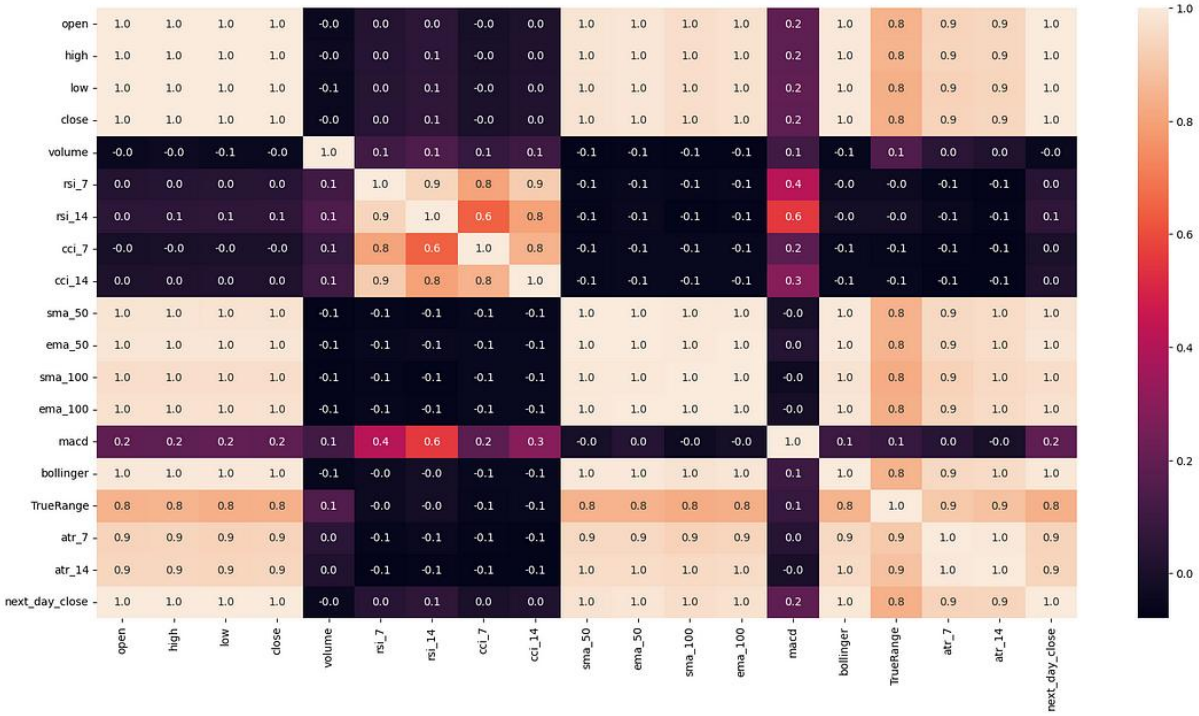
Let's visualize and see how well the stocks of Tesla are performing from year 2014 to 2023. (I Forgot to mention. This dataset contains the Tesla stock price information from the year 2014 to 2023).

```
import matplotlib.pyplot as plt
plt.figure(figsize=(20, 15))
plt.plot(df.index, df['open'], label='Open')
plt.plot(df.index, df['close'], label='Close')
plt.plot(df.index, df['high'], label='High')
plt.plot(df.index, df['low'], label='Low')
plt.legend()
```

Tesla was at its peak between 2021 and 2022

Now, let's know the correlation of every column present in the dataset. Before that, the target column in this dataset is the "**next_day_close**" column.

```
df_corr = df.corr()

import seaborn as sns
plt.figure(figsize=(20, 10))
sns.heatmap(df_corr, annot=True, fmt='.1f')
```



Looks good.

Just focus on the last row "**next_day_close**".

The "**next_day_close**" column has a 100% percent correlation with "**open**", "**close**", "**low**", "**high**", "**sma_50**", and so on. This means if values in those columns increase the values in the "**next_day_close**" column will also increase.

It's time for splitting the dataset into features and labels.

```
y = df['next_day_close']
X = df.drop(columns='next_day_close')
```

Now, let's further split these into training and testing sets.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=777)
```

Before moving on to creating models, we need to standardize our dataset. If you just check the dataset that we are currently working you can clearly see every column has values in different ranges.

That's not good. We cannot feed unstandardized data to our model. We are going to standardize our dataset using something called **StandardScaler**.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.fit_transform(X_test)
X_train_scaled.shape, X_test_scaled.shape # ((2012, 18), (504, 18))
```

We are almost done.

Its Model creation time.

Linear Regression

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
```

```
lr.fit(X_train, y_train)
lr_prediction = lr.predict(X_test)

from sklearn.metrics import mean_absolute_error
lr_mae = mean_absolute_error(y_test, lr_prediction)
lr_mae # 2.3867493377843387
```

Wow!

Our model is performing amazingly well. Our linear regression model is just giving a very low error.

$2.38 means that, if the actual value is $12.38 our prediction would be $10.

It's not that bad. Let's check other models' performance as well.

Decision Tree Regressor

```
from sklearn.tree import DecisionTreeRegressor
dtr = DecisionTreeRegressor()
dtr.fit(X_train, y_train)
dtr_prediction = dtr.predict(X_test)
dtr_mae = mean_absolute_error(y_test, dtr_prediction)
dtr_mae #3.588850648809524
```

3.5!

Random Forest Regressor

```
from sklearn.ensemble import RandomForestRegressor
rfr = RandomForestRegressor()
rfr.fit(X_train, y_train)
rfr_prediction = rfr.predict(X_test)
rfr_mae = mean_absolute_error(y_test, rfr_prediction)
rfr_mae #2.7267115297023783
```

2.7!

Support Vector Regressor

```
from sklearn.svm import SVR
svr = SVR()
svr.fit(X_train, y_train)
svr_prediction = svr.predict(X_test)
svr_mae = mean_absolute_error(y_test, svr_prediction)
svr_mae #74.63800067519O6
```

74! (Really bad)

We can just stick with the Linear Regression model.

That's it.

Day — 14☑

Day-15: Students Performance Prediction 💯

Day 15. My challenge is 50% completed today😎.

So, this is what I did today.

You can follow along with me but before that go and get the dataset using the below link.

Click here: [Student Performance Dataset]

Let's import 🐼🐼

```
import pandas as p
df = p.read_csv("Student_Performance.csv")

df.head()
"""
OUTPUT:
Hours Studied Previous Scores Extracurricular Activities Sleep Hours
Sample Question Papers Practiced Performance Index
0 7 99 Yes 9 1 91.0
1 4 82 No 4 2 65.0
2 8 51 Yes 7 2 45.0
3 5 52 Yes 5 2 36.0
4 7 75 No 8 5 66.0
"""
```

Are there any null values in this dataset? Let check.

```
df.isnull().sum()
"""
OUTPUT:
Hours Studied                     0
Previous Scores                   0
Extracurricular Activities        0
Sleep Hours                       0
Sample Question Papers Practiced  0
Performance Index                 0
"""
```

This dataset is a piece of cake🍰.

Let's learn more about this dataset before processing moving on.

```
df.shape # (10000, 6)

df.info()
"""
OUTPUT:
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   Hours Studied                   10000 non-null  int64
 1   Previous Scores                 10000 non-null  int64
 2   Extracurricular Activities      10000 non-null  object
 3   Sleep Hours                     10000 non-null  int64
 4   Sample Question Papers Practiced  10000 non-null  int64
 5   Performance Index               10000 non-null  float64
```

```
"""


df.describe()
"""
OUTPUT:
Hours Studied Previous Scores Sleep Hours Sample Question Papers
Practiced Performance Index
count 10000.000000 10000.000000 10000.000000 10000.000000
10000.000000
mean 4.992900 69.445700 6.530600 4.583300 55.224800
std 2.589309 17.343152 1.695863 2.867348 19.212558
min 1.000000 40.000000 4.000000 0.000000 10.000000
25% 3.000000 54.000000 5.000000 2.000000 40.000000
50% 5.000000 69.000000 7.000000 5.000000 55.000000
75% 7.000000 85.000000 8.000000 7.000000 71.000000
max 9.000000 99.000000 9.000000 9.000000 100.000000
"""
```

There is nothing for us to do. Without wasting any time let's split the dataset into features and labels.

```
y = df['Performance Index'].copy()
X = df.drop(columns='Performance Index').copy()

X.head()
"""
OUTPUT:
Hours Studied Previous Scores Extracurricular Activities Sleep Hours
Sample Question Papers Practiced
0 7 99 Yes 9 1
```

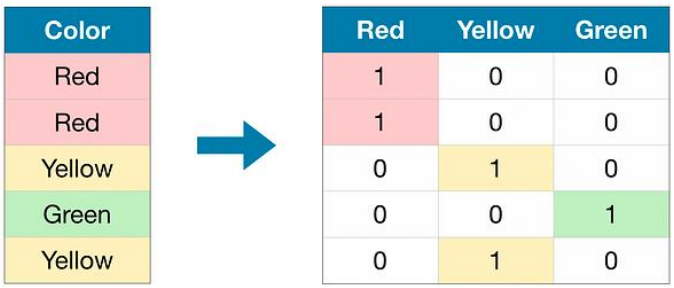
```
1 4 82 No 4 2
2 8 51 Yes 7 2
3 5 52 Yes 5 2
4 7 75 No 8 5
"""
```

We have one categorical column in the features(X). I am going to use a technique called One-Hot encoding to handle that.

```
X = p.get_dummies(X, columns=['Extracurricular Activities'])

X.head()
"""
OUTPUT:
Hours Studied Previous Scores Sleep Hours Sample Question Papers
Practiced Extracurricular Activities_No Extracurricular Activities_Yes
0 7 99 9 1 0 1
1 4 82 4 2 1 0
2 8 51 7 2 0 1
3 5 52 5 2 0 1
4 7 75 8 5 1 0
"""
```

Compare the outputs before using ***p.get_dummies*** and after using ***p.get_dummies*** you will understand how one-hot encoding works.

See the below image.



It's time for a train-test split 🔨.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=7)
```

Model building time!

Support Vector Regressor(SVR)

```
from sklearn.svm import SVR
svr = SVR()
svr.fit(X_train, y_train)
svr_prediction = svr.predict(X_test)

from sklearn.metrics import mean_absolute_error
svr_mae = mean_absolute_error(y_test, svr_prediction)
svr_mae # 1.8384445448641034
```

Nice! Our error is very low. Let's try some other model.

Linear Regression

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
lr.fit(X_train, y_train)
lr_prediction = lr.predict(X_test)
lr_mae = mean_absolute_error(y_test, lr_prediction)
lr_mae # 1.6313121603176193
```

It's good. But what if I standardize the data before feeding it to the model? Let's try that by creating a pipeline that will first standardize the data using **StandardScaler** and then passes the standardized data to train the model.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

lr_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('model', LinearRegression())
])

lr_pipeline.fit(X_train, y_train)
lr_pipeline_prediction = lr_pipeline.predict(X_test)
lr_pipeline_mae = mean_absolute_error(y_test, lr_pipeline_prediction)
lr_pipeline_mae #1.6313121603176188
```

What is the difference,

Non-standardized Data: 1.6313121603176193

Standardized Data: 1.6313121603176188

See the last 2 numbers there is a slight reduction in the error. 😄

That's it guys.

This is what I did today. (Easy day).

As I already told you from tomorrow on I will be focusing more on Unsupervised Learning tasks.

The next 15 days are going to be tough because I don't have any prior knowledge of Unsupervised learning like I do of Supervised Learning.

But I know one thing, I will be learning a lot more in the next 15 days than I did in the last 15 days.

Day — 15☑

Day-16: Iris Dataset Clustering 🌷

Enough of supervised learning problems. From today on I will be working with unsupervised learning problems.

So, today because I was just getting started in unsupervised learning I chose a very simple task of clustering the iris dataset.

This is what I did today.

I imported the necessary libraries first.

```
from sklearn.cluster import KMeans
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
```

Then I loaded the iris dataset.

```
iris = load_iris()
```

Then I stored the features of the iris dataset into a variable called X. Because it is an unsupervised learning task we should not teach the label or target.

```
X = iris.data
```

I immediately created a KMean clustering model.

```
kmeans = KMeans(n_clusters=3, random_state=56)
```

I fitted the feature into the model. And created a data frame using the features.

```
kmeans.fit(X)
iris_df = pd.DataFrame(iris.data, columns=iris.feature_names)
kmeans.labels_
"""
OUTPUT:
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 2, 2, 2, 2, 0, 2, 2, 2,
       2, 2, 2, 0, 0, 2, 2, 2, 2, 0, 2, 0, 2, 0, 2, 2, 0, 0, 2, 2, 2, 2,
       2, 0, 2, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2, 0], dtype=int32)
"""
```

The ***kmeans.labels_*** gives us all the values based on which the features are clustered by the KMeans model.

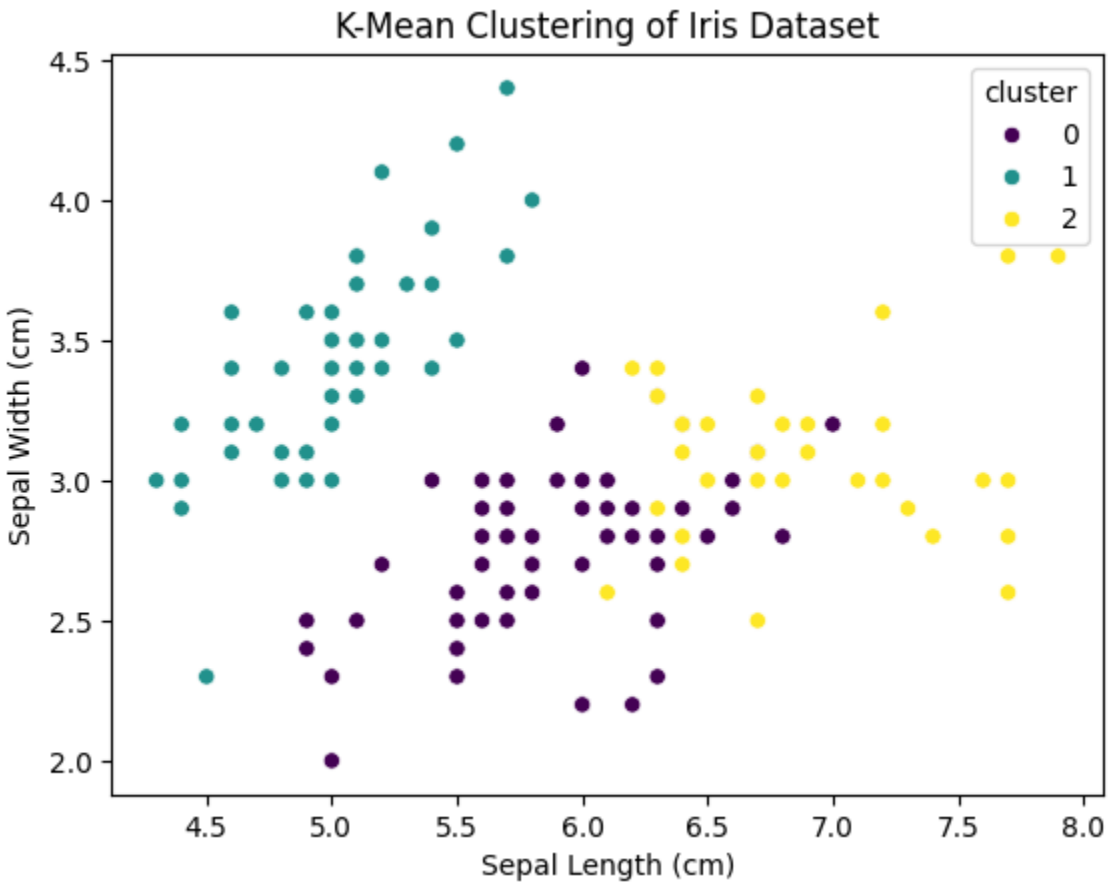
Then, I included the labels in the iris data frame.

```
iris_df['cluster'] = kmeans.labels_
```

Finally, we can not skip visualization when it comes to clustering.

Below is the code for visualization.

```
import matplotlib.pyplot as plt
import seaborn as sns

sns.scatterplot(x=iris_df['sepal length (cm)'], y=iris_df['sepal width
(cm)'], hue=iris_df['cluster'], palette='viridis')
plt.xlabel('Sepal Length (cm)')
plt.ylabel('Sepal Width (cm)')
plt.title('K-Mean Clustering of Iris Dataset')
plt.show()
```



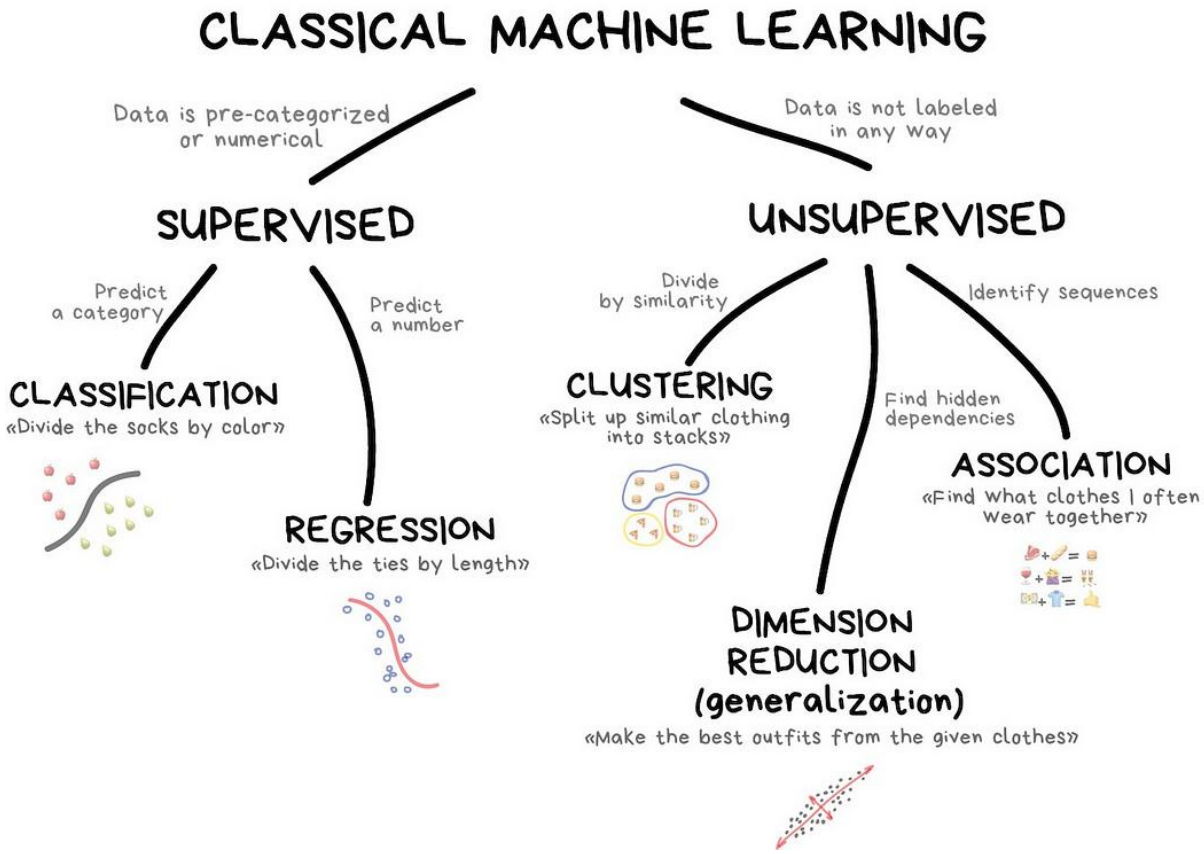
This is the shortest project in my whole 30-day challenge till now.

That's it.

Day — 16☑

Day-17: Mall Customers Clustering 🏬

As you already know after day 15 it's time to work on unsupervised learning tasks.



Yesterday, I started learning about "Clustering" and did a simple, actually very simple project using the "*Iris Dataset*". Today I picked up a different dataset and performed clustering using the **KMeans** algorithm.

Today I learned and understood a lot than yesterday.

I will try my best to write this article in a way that you can easily understand the concepts.

Let's begin.

Click here to get the dataset👉: Mall Customers Dataset

First import 🐼🐼 and load the dataset.

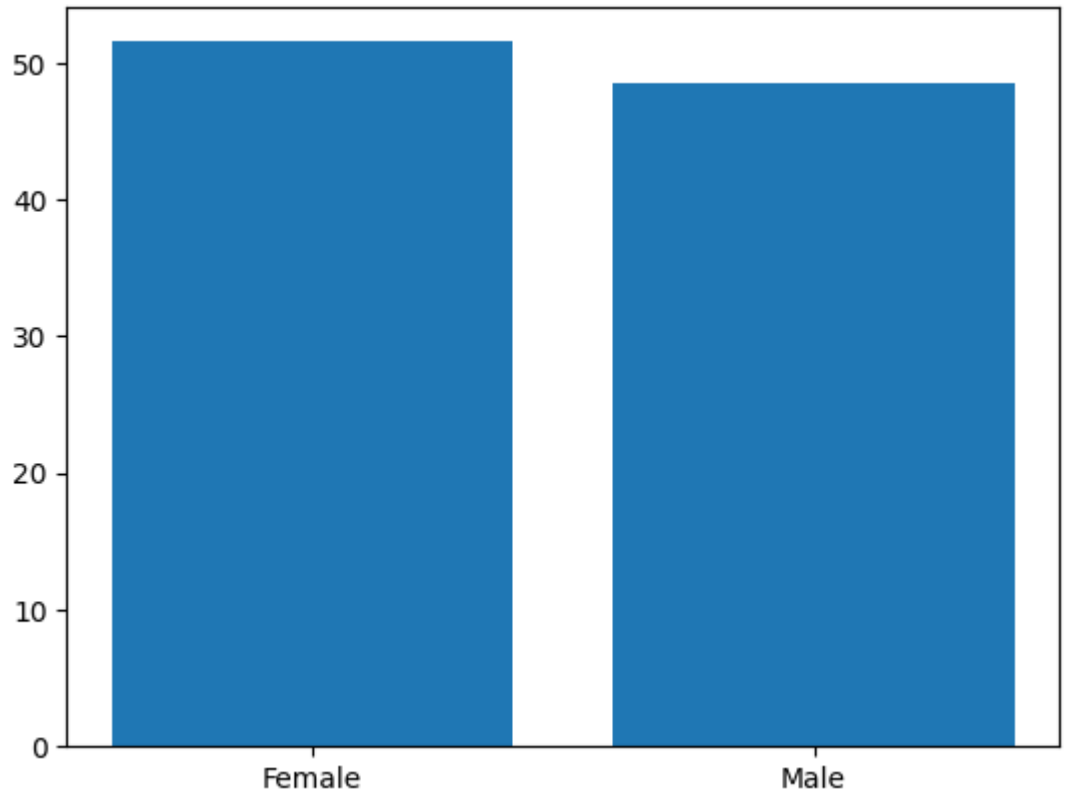
```
import pandas as p
df = p.read_csv('Mall_Customers.csv')

df.head()
"""
OUTPUT:
CustomerID Gender Age Annual Income (k$) Spending Score (1-100)
0 1 Male 19 15 39
1 2 Male 21 15 81
2 3 Female 20 16 6
3 4 Female 23 16 77
4 5 Female 31 17 40
"""
```

There are no null values in this dataset. As this is a clustering task we will be focusing more on visualization.

Our goal with this dataset is to find different groups of people visiting the mall.

First, let's try to know who spends the most male or female by plotting a bar chart.

```
spending_by_gender = df.groupby('Gender')['Spending Score (1-100)'].mean()
spending_by_gender
"""
OUTPUT:
Gender
Female    51.526786
Male      48.511364
"""
import matplotlib.pyplot as plt
plt.bar(spending_by_gender.index, spending_by_gender.values)
```

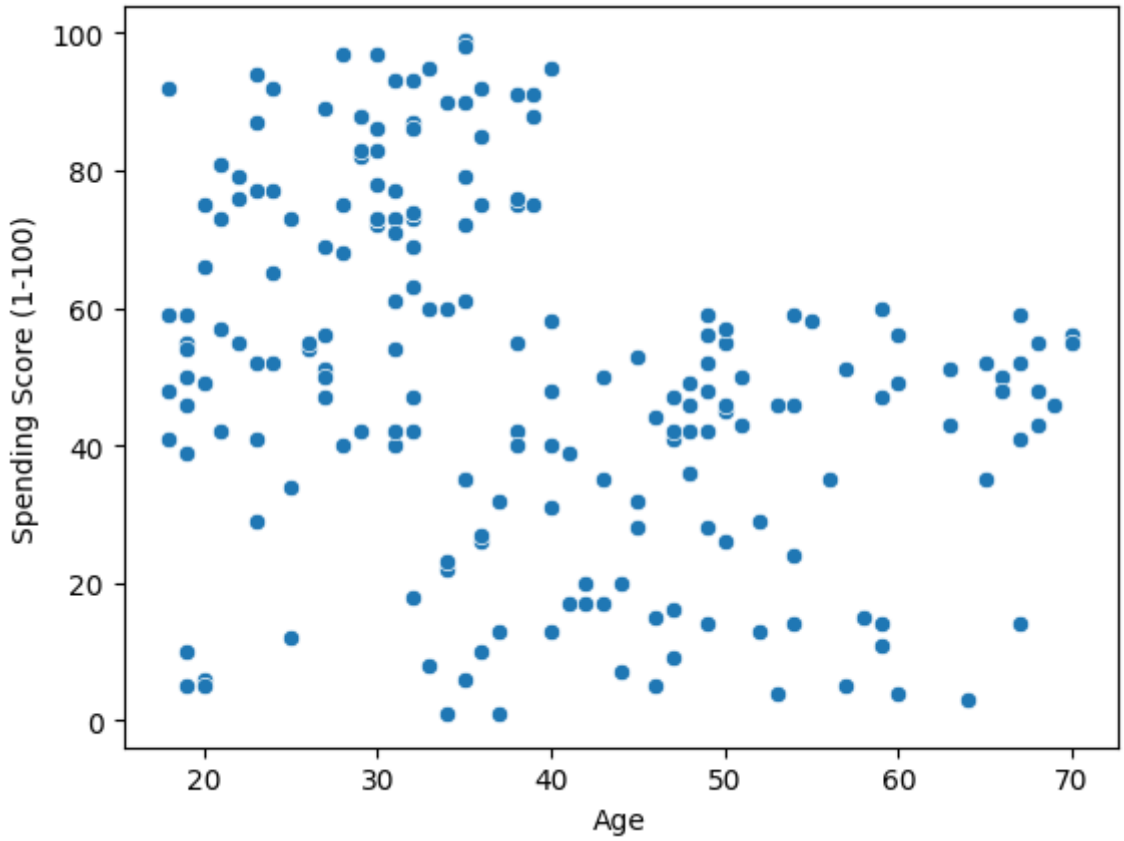


Really!

There is not much difference (But we know the truth).

Also, let's try a scatter plot using the age column on the x-axis and the spending score column on the y-axis.

```
import seaborn as sns
sns.scatterplot(df, x='Age', y='Spending Score (1-100)')
```

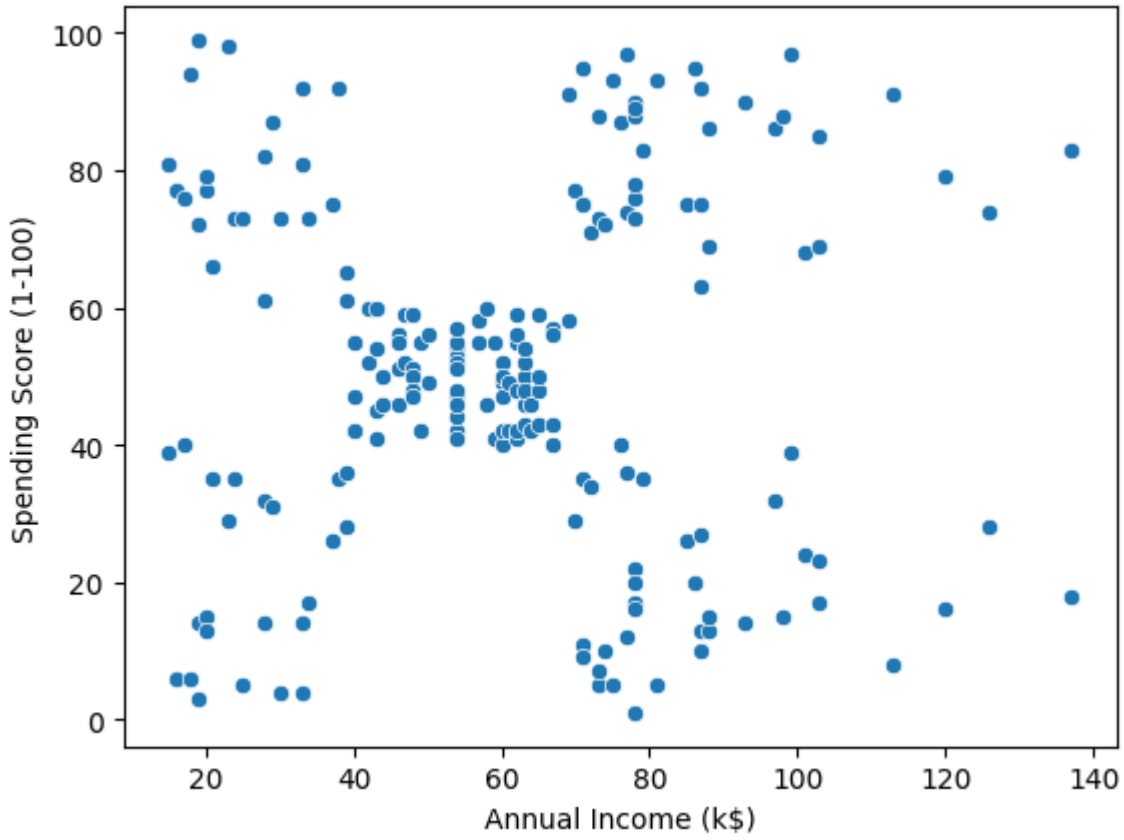


What do you see?

From the above graph, we can clearly see people between the ages of 20–40 have higher spending scores compared to people between the ages of 50–70 (That's called maturity).

Let's plot another scatter plot keeping the annual income on the x-axis and the spending scores on the y-axis.

```
sns.scatterplot(df, x='Annual Income (k$)', y='Spending Score (1-100)')
```



There are some clusters.

In the above graph, we can find some clusters. There are some visible groups. So, these are the two features(Annual Income and Spending Score) we are going to use to do our clustering.

Before that, we need to find out what is the right number of clusters we can make from this graph.

Should we cluster this into 3 or 4 or 5 or what?

To answer the above question there is a method called the "Elbow Method", which we will learn in a while but before that let's create a variable **X**(also don't forget to follow me on X) and store the two features(Annual Income and Spending Score).

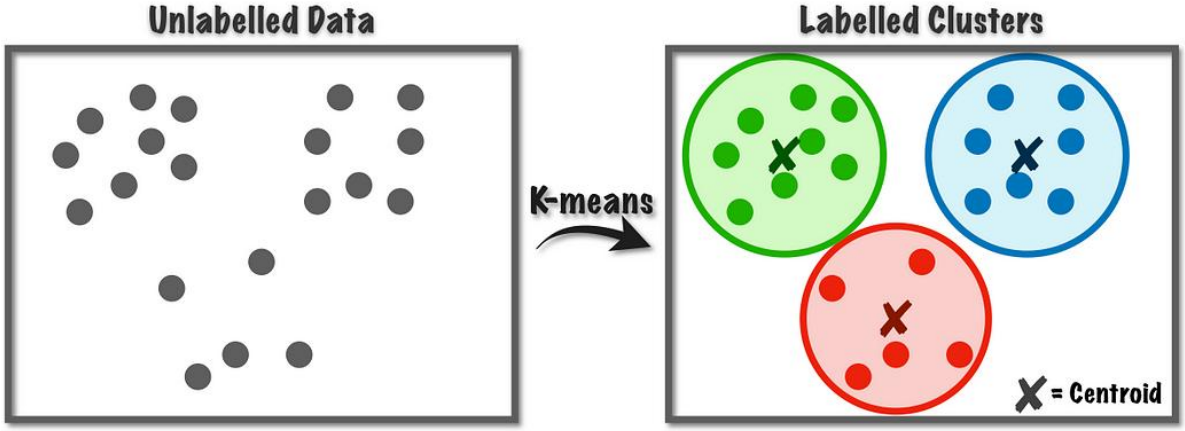
```
X = df.drop(columns=['CustomerID', 'Gender', 'Age']).values
X
"""
OUTPUT:
Os
X
array([[ 15,  39],
       [ 15,  81],
       [ 16,   6],
       [ 16,  77],
       [ 17,  40],
       [ 17,  76],
       [ 18,   6],
       [ 18,  94],
       [ 19,   3],
       [ 19,  72],
       [ 19,  14],
       [ 19,  99],
       [ 20,  15],
       [ 20,  77],
       [ 20,  13],
       [ 20,  79],
       .....
       .....
```

```
        .........
''''''
```

Now let me explain to you the 'Elbow Method" but before knowing that you need to know something called "inertia" in KMeans.

You can think of "inertia" as a metric to evaluate the KMeans model. If the inertia is low then good if the inertia is high then bad as simple as that. Also, it should not be that low.

Inertia is just the distance of all the points from its "centroid". If you don't know what "centroid" means. Here is a diagram.



When we are using KMeans to cluster features into 3 the KMeans try to find 3 centroids and the points near those centroids are clustered together. I hope you can understand.

Let's create the KMeans model and perform the "Elbow Method" to find the best number of clusters.

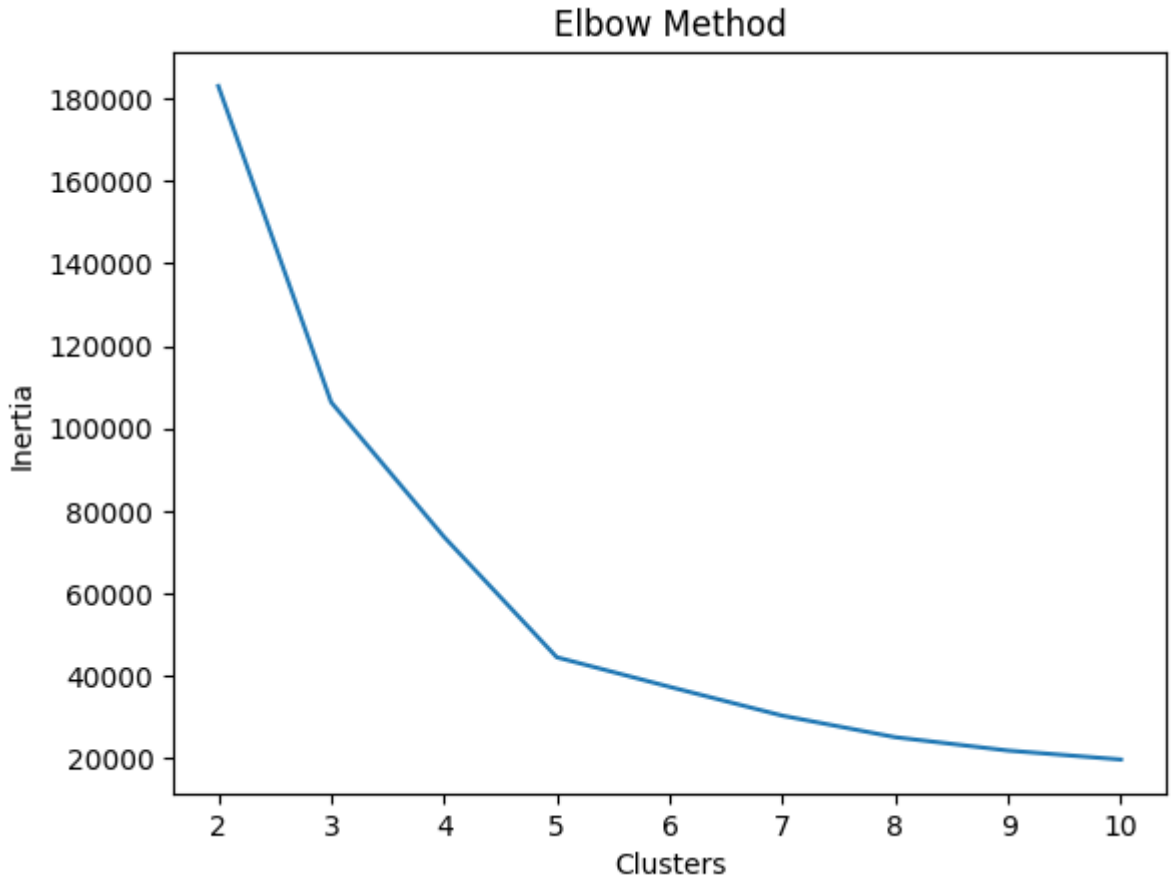
```
from sklearn.cluster import KMeans
inertia = []

for i in range(2, 11):
  model = KMeans(n_clusters=i, random_state=7)
  model.fit(X)
  inertia.append(model.inertia_)
```

The for loop above will create a KMeans model with different ***n_clusters*** from 2 to 10. In each loop, the model's "inertia" will be appended to a list called **inertia**.

```
inertia
"""
OUTPUT:
[183069.17582751298,
 106348.37306211122,
 73679.78903948836,
 44448.4554479337,
 37271.88623658949,
 30273.394312070042,
 25050.832307547527,
 21806.81299869546,
 19634.554629349976]
"""
```

As you can see as the number of clusters increases the inertia keeps getting low. You will soon understand why they named this method "Elbow".

```
plt.plot(range(2, 11), inertia)
plt.title('Elbow Method')
plt.xlabel('Clusters')
plt.ylabel('Inertia')
```

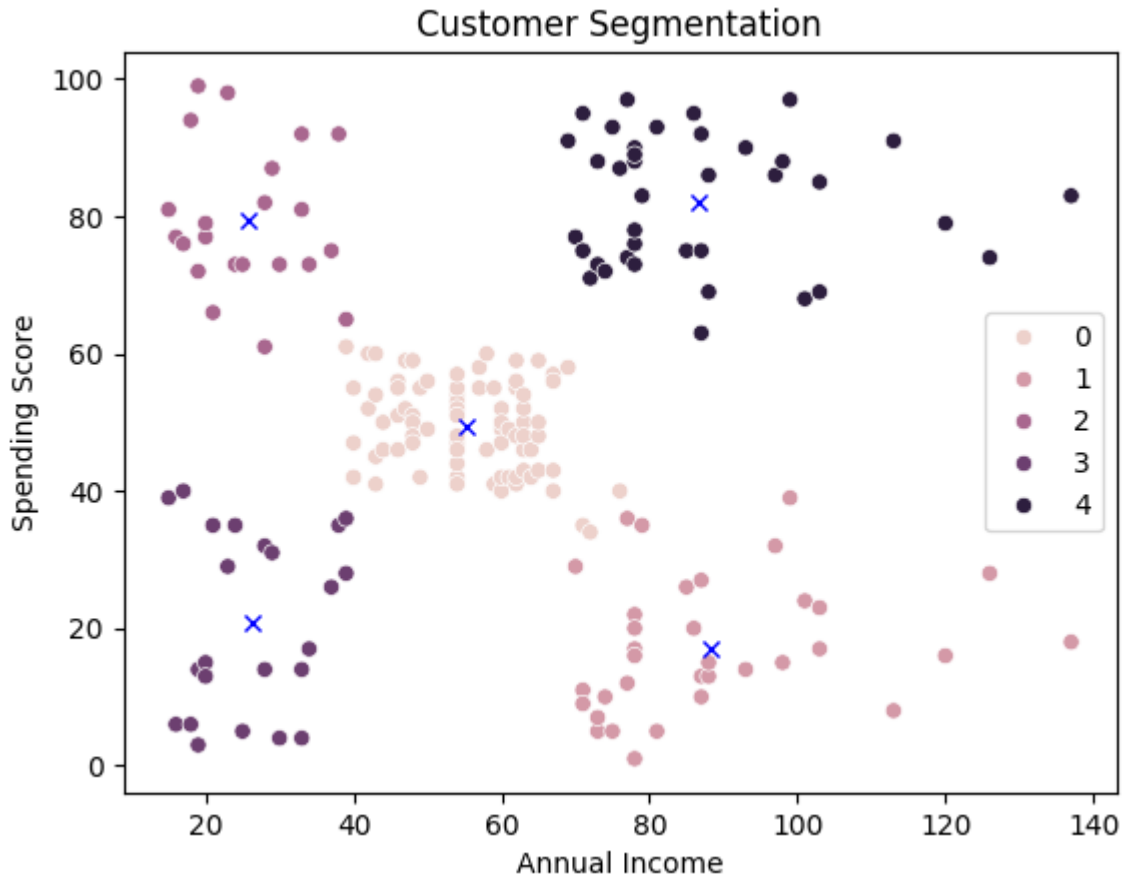


Elbow!

See the above chart, you should not choose the lowest point which is 10, we should choose 5 because that's where the inertia is getting down and that's the elbow😄.

Let's create our model using 5 clusters then plot and see how our model is grouping the people visiting the mall.

```
KMeansModel = KMeans(n_clusters=5, random_state=786)
KMeansModel.fit(X)

centers = KMeansModel.cluster_centers_
centers
"""
OUTPUT:
array([[55.2962963 , 49.51851852],
       [88.2       , 17.11428571],
       [25.72727273, 79.36363636],
       [26.30434783, 20.91304348],
       [86.53846154, 82.12820513]])
"""
center_x = centers[:, 0]
center_y = centers[:, 1]

sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=KMeansModel.predict(X))
plt.plot(center_x, center_y, 'xb')
plt.title('Customer Segmentation')
plt.xlabel('Annual Income')
plt.ylabel('Spending Score')
plt.show()
```



Nice!

So, we can divide the people into 5 groups or clusters based on their annual income and spending scores.

I have also plotted the centroids so that you can understand why they are grouped the way they are grouped.

The ***KMeansModel.cluster_centers_*** gives us the centers and ***KMeansModel.predict(X)*** gives us the labels by which the features are clustered, see below code.

```
KMeansModel.predict(X)
"""
OUTPUT:
array([3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2,
       3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 0,
       3, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 1, 4, 0, 4, 1, 4, 1, 4,
       0, 4, 1, 4, 1, 4, 1, 4, 1, 4, 0, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4,
       1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4,
       1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4, 1, 4,
       1, 4], dtype=int32)
"""
```

That's it.

Day — 17☑

Day-18: KMeans, DBSCAN, and PCA 🫧

This Unsupervised Learning is so confusing but understandable. In unsupervised learning, there are mostly clustering-type projects.

Today, I was learning about the 2 most used clustering algorithms (KMeans and DBSCAN) and a dimensional reduction technique called Principle Component Analysis (PCA).

Let me share with you what I learned today.

(I continued the code from day 16. You don't have to go there, I will start from the beginning here).

Here is my code.

I started by importing the two visualization libraries.

```
import matplotlib.pyplot as plt
import seaborn as sns
```

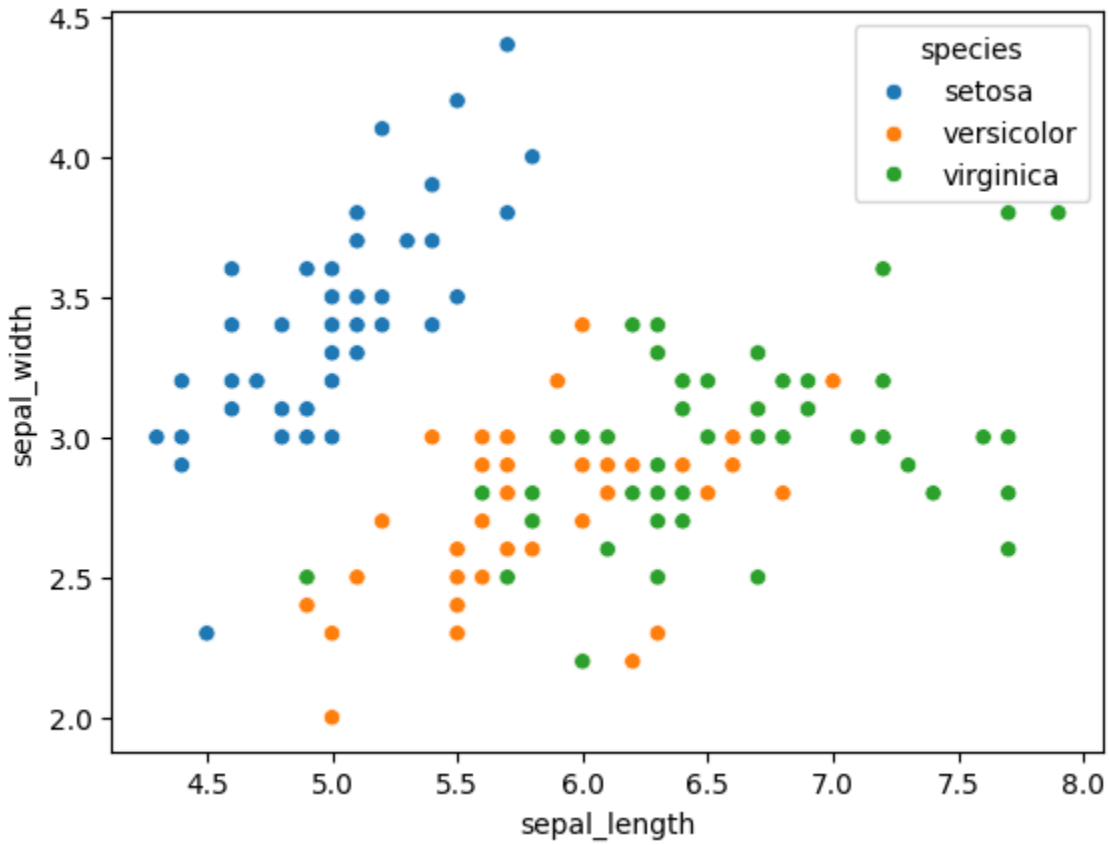
Do you know what, we can access the "***Iris***" dataset from the Seaborn library itself? Here is how you do it.

```
iris = sns.load_dataset('iris')
iris.head()
'''''
```

```
OUTPUT:
sepal_length sepal_width petal_length petal_width species
0 5.1 3.5 1.4 0.2 setosa
1 4.9 3.0 1.4 0.2 setosa
2 4.7 3.2 1.3 0.2 setosa
3 4.6 3.1 1.5 0.2 setosa
4 5.0 3.6 1.4 0.2 setosa
"""
```

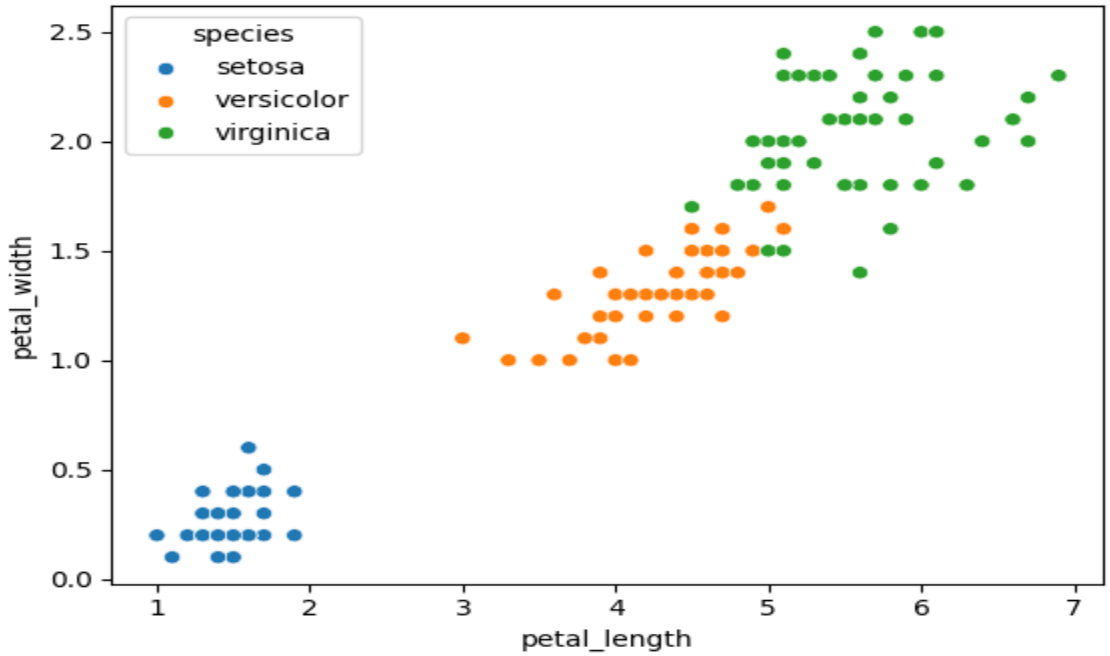
Then I visualized the data points using different features in scatter plot.

```
sns.scatterplot(iris, x='sepal_length', y='sepal_width', hue='species')
```



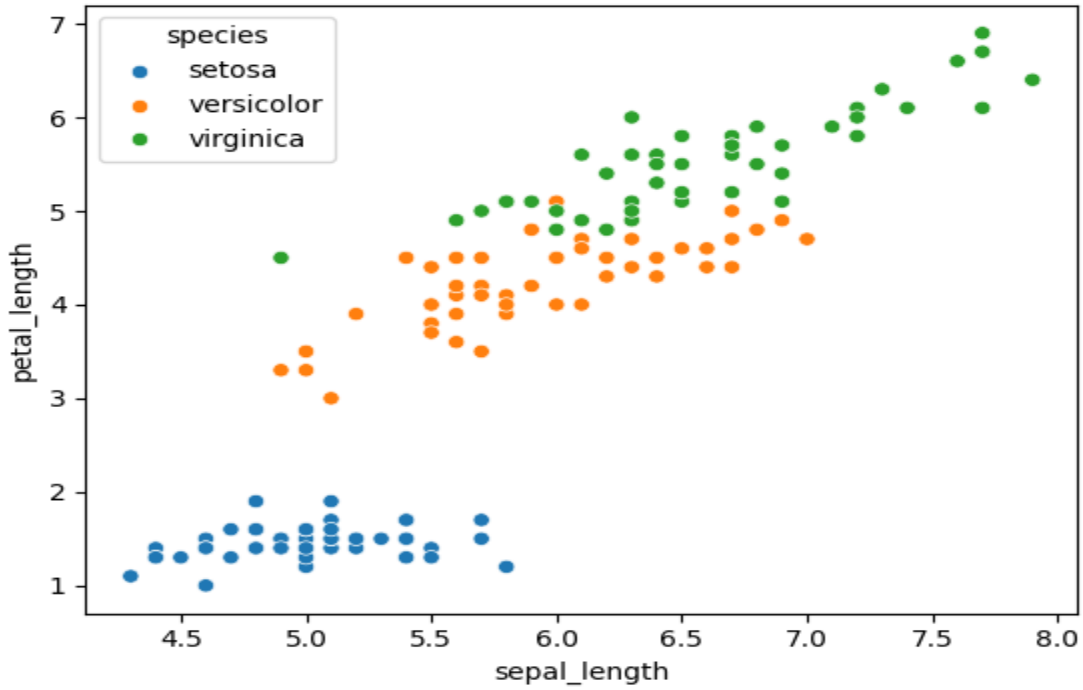
Combination — 1

```
sns.scatterplot(iris, x='petal_length', y='petal_width', hue='species')
```



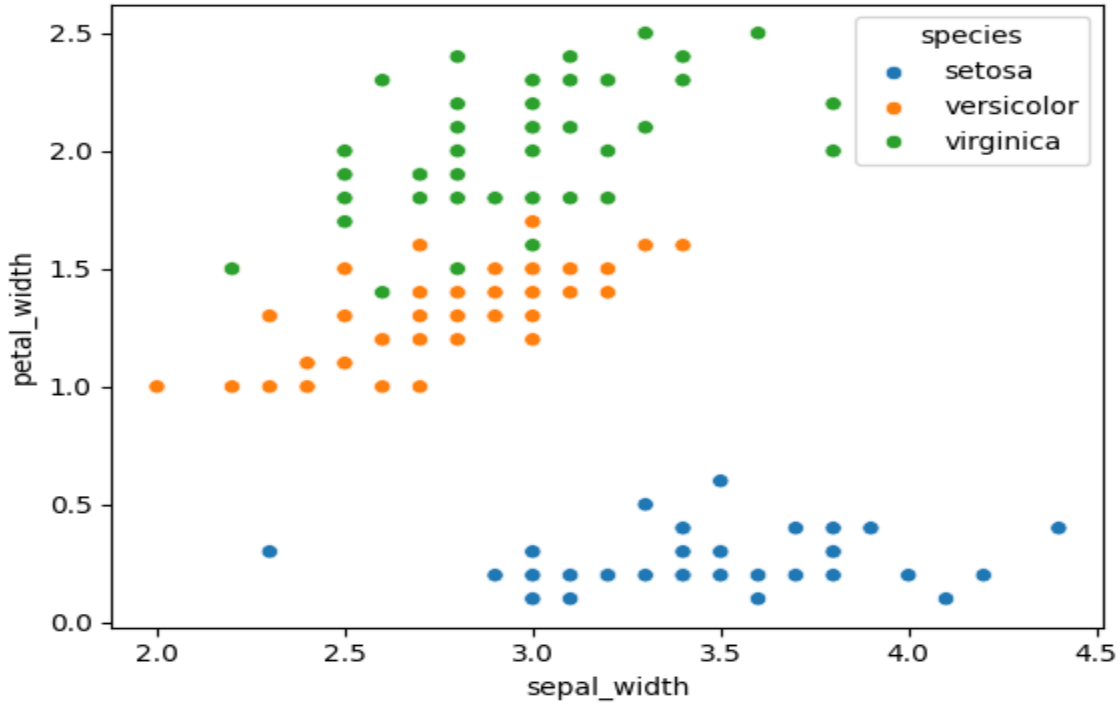
Combination — 2

```
sns.scatterplot(iris, x='sepal_length', y='petal_length', hue='species')
```



Combination — 3

```
sns.scatterplot(iris, x='sepal_width', y='petal_width', hue='species')
```



Combination — 4

Now, as you can see from the above visualizations there are 3 different clusters in the dataset (*setosa*, *versicolor*, and *virginica*).

To understand the concept of KMeans and DBSCAN(*Density-based spatial clustering of applications with noise*) I picked up two features from the "Iris" dataset and trained a KMeans model giving the *n_clusters* parameter the value of 3.

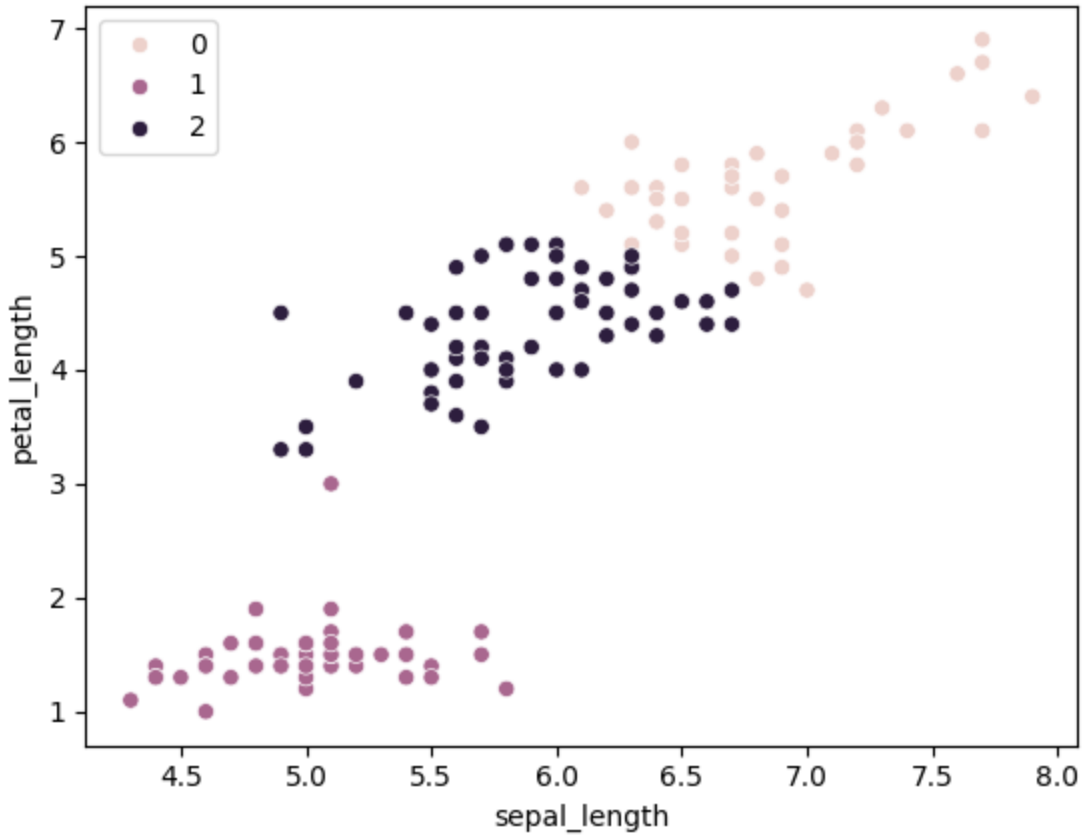
```
X = iris[['sepal_length', 'petal_length']].values
from sklearn.cluster import KMeans, DBSCAN
km = KMeans(n_clusters=3)
km.fit(X)
```

```
km.predict(X)
"""
OUTPUT:
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 0, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 0, 2, 0, 0, 0, 0, 2, 0, 0, 0,
       0, 0, 0, 2, 2, 0, 0, 0, 0, 2, 0, 2, 0, 2, 0, 0, 2, 2, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 2], dtype=int32)
"""
```

The *km.predict()* will give the clusters made by our KMeans model. Now instead of giving the *hue* parameter the "*species*" columns from the "Iris" dataset we are going to give it the *km.predict(X)*.

Let's see the clusters made by our KMeans model.

```
sns.scatterplot(iris, x='sepal_length', y='petal_length',
hue=km.predict(X))
```

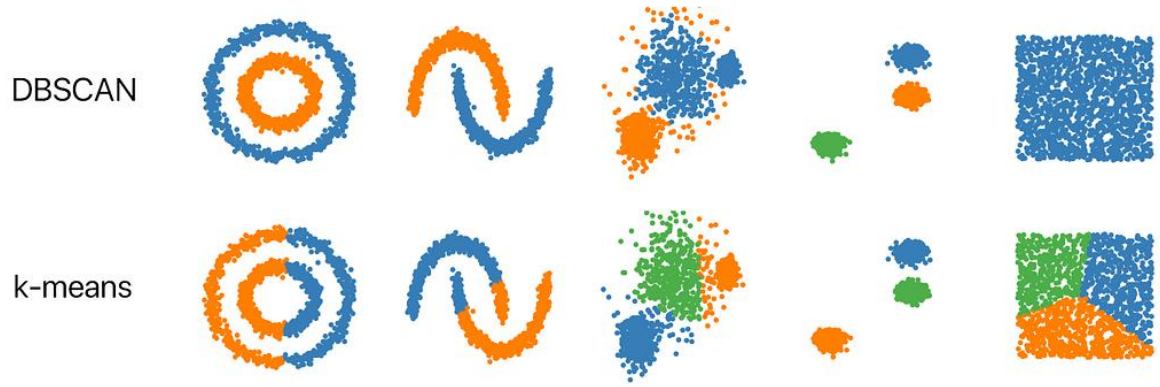


Compare it with the original

Our model has clustered very well, there are few incorrect points but still the clustering is good.

After doing this I moved on to the DBSCAN. DBSCAN works totally different when compared to KMeans.

Below is a image to make you understand when to use KMeans and when to use DBSCAN.



Can you understand?

So, when we have features like in the first(Donut), second(Horse-shoe), and the last(square) example from the above image, using DBSCAN would be the right choice. But when we have features separated like in the third example KMeans should be preferred.

For the fourth example either of them can be used.

I don't know how to explain the working of DBSCAN in words, I will make a video (I don't know when) about it.

Let's get into the code.

As you have already guessed for our example KMeans is the right choice but still let's try DBSCAN and see what type of visualization it will give.

```
dbscan = DBSCAN(eps=1, min_samples=6)
dbscan.fit(X)
```

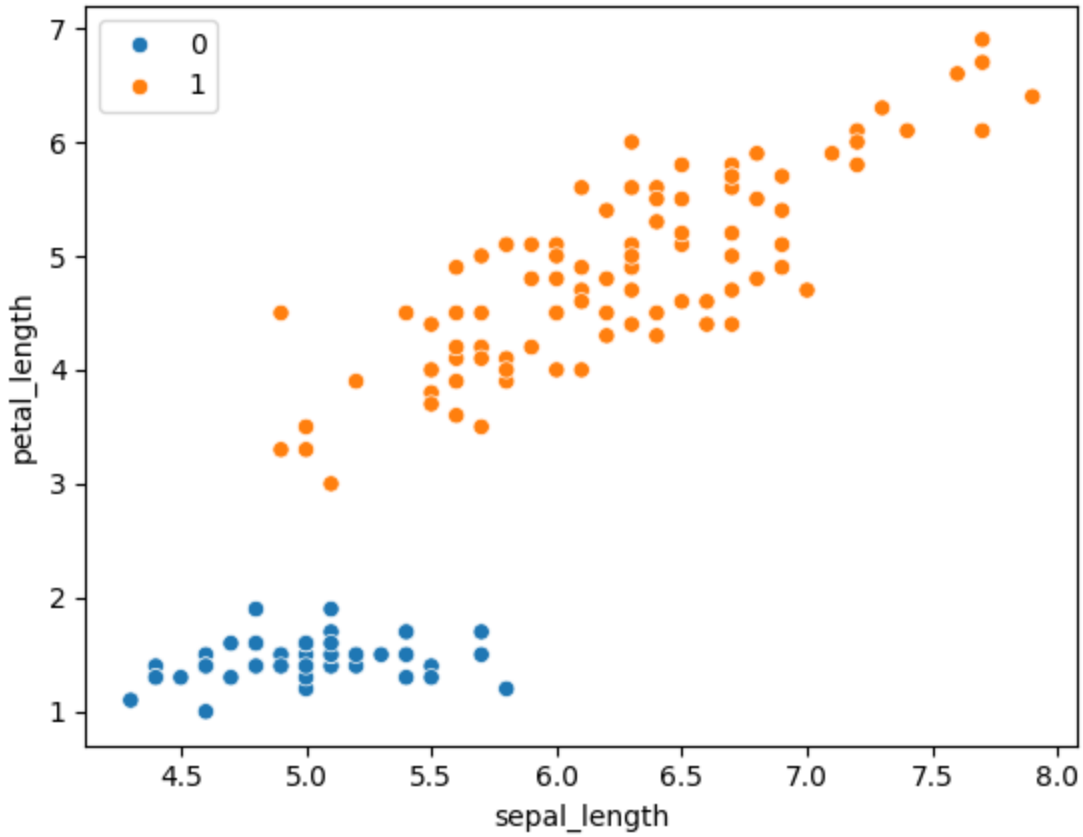
```
dbscan.labels_
"""
OUTPUT:
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
"""
```

Don't worry if you are not able understand what the above two parameters are for (*eps* and *min_samples*). When I make the video I will explain that as well.

As you can see from the above output, DBSCAN has clustered our features into just 2 clustered.

Let's visualize it.

```
sns.scatterplot(iris, x='sepal_length', y='petal_length',
hue=dbscan.labels_)
```



I hope you got the point.

Let me try my best to make you understand how DBSCAN works with my words.

So, first we will chose a random point in the plane then considering that random point as center we will draw a 1cm circle (that is the value of our ***eps*** see the parameter in the DBSCAN()).

After drawing the circle we will check how many points are there within the circle if there are 6 points within the circle (that is the value of our ***min_samples*** see the parameter in the DBSCAN()) Those 6 points forms a cluster. And it will not stop there not we will choose one of the 6 points in the circle and repeat the same process again and again until we cluster all the points that are close to each other.

I hope you can understand something now.

Now, let's move on to PCA.

I didn't learn PCA completely, I just got started with it today. I will learn it in-depth in the upcoming day.

But here is what I did today with PCA.

I imported the necessary library.

```
from sklearn.datasets import fetch_openml
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
```

Then, I used the MNIST dataset. Because the MNIST has more dimensions.

As our goal with PCA is to reduce the dimension choosing a dataset with more features would be the right choice.

```
mnist = fetch_openml('mnist_784', version=1)


X = mnist.data / 255.0  # Scale pixel values to between 0 and 1
y = mnist.target

# Reduce dimensionality using PCA
```

```
pca = PCA(n_components=2)  # Reduce to 2 principal components
X_pca = pca.fit_transform(X)

X_pca
"""
OUTPUT:
array([[ 0.47948276, -1.24013763],
       [ 3.96270444, -1.13712478],
       [-0.23134165,  1.54389908],
       ...,
       [-1.06472486,  2.3140395 ],
       [-1.21658619, -0.45773902],
       [ 4.15240765, -0.32702944]])
"""
```

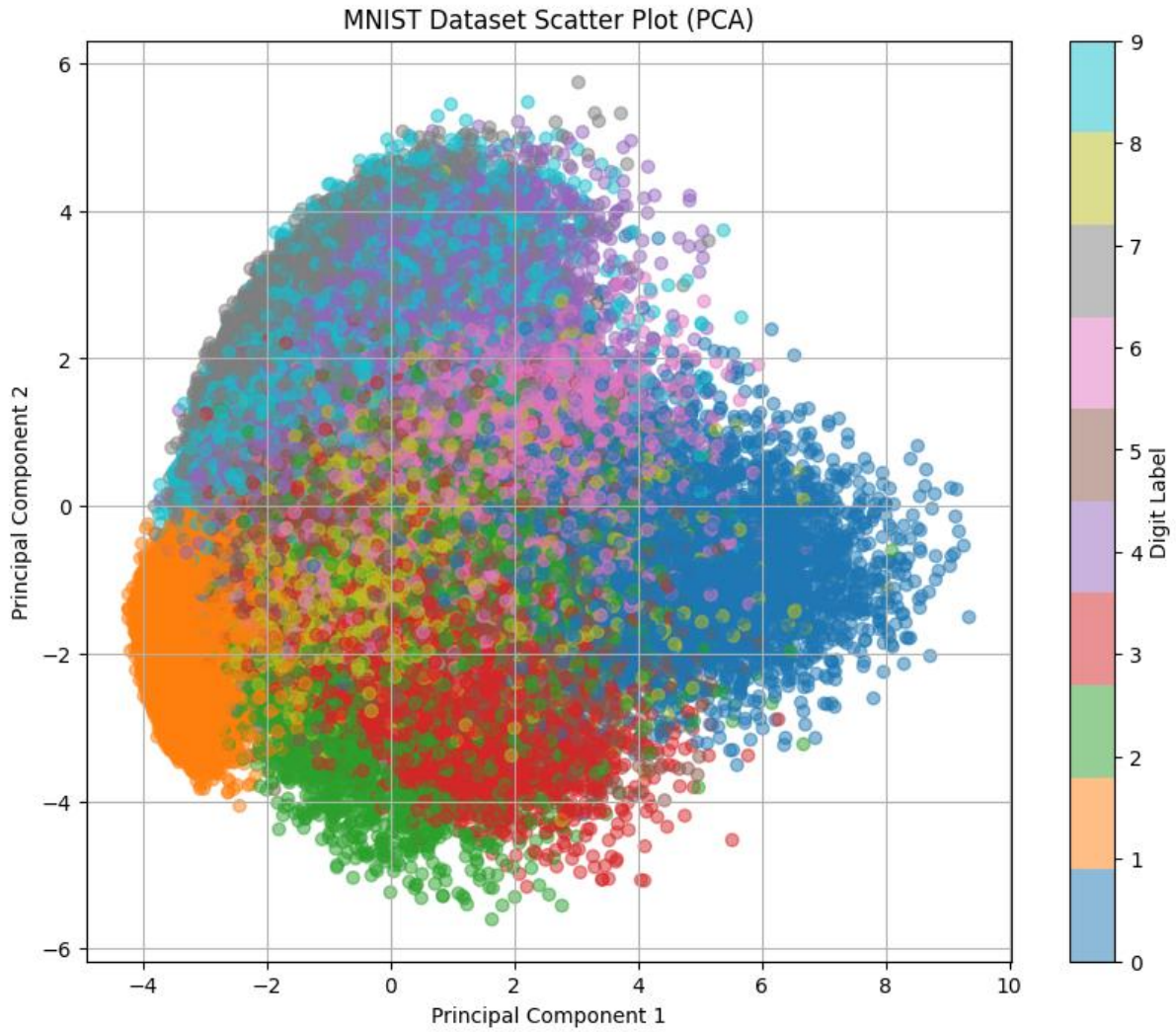
Then, I stored the features into variable **X** (also scaled it) and stored the label or target to **y**.

Then I created the PCA model and give the *n_components* parameter the value 2, which means after transformation we will have only two features.

Let plot and see how it looks.

```
plt.figure(figsize=(10, 8))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y.astype(int), cmap='tab10',
alpha=0.5)
plt.colorbar(label='Digit Label')
plt.xlabel('Principal Component 1')
```

```
plt.ylabel('Principal Component 2')
plt.title('MNIST Dataset Scatter Plot (PCA)')
plt.grid(True)
plt.show()
```



So the above is the MNIST features visualized in 2 dimensions.

As you can see the color blue(9) and gray(7) are overlapping, it's because 9 and 7 looks somewhat similar when written in hand.

Also, see 0 and 8.

That's it.

Day — 18☑

Day-19: Apriori Algorithm For Market Basket Analysis🍞🥛🧈

Today, I was learning a topic for the first time. If you know a little bit about Machine Learning you might know this, in unsupervised learning, we have something called Association Algorithm.

This Association Algorithm is very much different from clustering algorithms.

Here is a very simple example for you to understand how the Association.

Imagine yourself shopping in a supermarket, you are buying bread, and after putting the bread in your basket you move a little bit and you see butter just beside the bread counter, then you pick up the butter as well, then beside the butter counter you see mustard, now you will pick up the mustard and put it in your basket.

If you just start noticing the placement of products in the supermarket you will come to know that, they will place products like shampoo and conditioner together, similarly, they will place bread and butter together it is because they have analyzed the purchasing patterns of customers to make people buy more product and increase their profit. This type of analysis is known as **Market Basket Analysis**.

If you want to classify dog and cat pictures you will use classification algorithms, right? Similarly, to do a market basket analysis, we will use the association algorithm.

Association algorithms do not just help with market basket analysis, they can be used for other applications as well to name a few.

1. Recommendations Systems.
2. Customer Segmentation (Association algorithms can be used to segment customers based on their purchasing behavior.)
3. Fraud Detection (Association algorithms can help detect fraudulent activities by analyzing patterns of behavior in financial transactions, insurance claims, or online activities.)

and there is more.

This is the first time I am learning about Association algorithms, specifically, Apriori Algorithm.

I will try my best to share with you what I have learned today, In the future after learning more about this topic and building more projects related to this topic, I will write a comprehensive article. But for now, let me share what I did today.

Unlike other algorithms in machine learning the Apriori algorithm is not available in the **Scikit-Learn** library. It is in a library called ***mlxtend***.

```
from mlxtend.frequent_patterns import apriori
from mlxtend.frequent_patterns import association_rules
import pandas as pd
```

As a beginner in this topic I don't what to work on some big dataset so I asked ChatGPT to create a very simple dataset.

```
# Define the dataset
data = {
    'Transaction ID': [1, 2, 3, 4, 5],
    'Items': [['bread', 'milk'],
              ['bread', 'diaper', 'beer', 'eggs'],
              ['milk', 'diaper', 'beer', 'cola'],
              ['bread', 'milk', 'diaper', 'beer'],
              ['bread', 'milk', 'diaper', 'cola']]
}

# Convert the dataset into a pandas DataFrame
df = pd.DataFrame(data)
df

"""
OUTPUT:

Transaction ID Items
0 1 [bread, milk]
1 2 [bread, diaper, beer, eggs]
2 3 [milk, diaper, beer, cola]
3 4 [bread, milk, diaper, beer]
4 5 [bread, milk, diaper, cola]
```

```
"""
```

So, there are 2 columns in our data frame, the first column is transactions, and the second column is the items purchased.

You can think of the above data like this,

Say, 5 people are entering a supermarket to buy a few things,

the first person buys bread then milk (Bread -> Milk)

the second person buys bread, diaper, beer, and eggs (Bread → Diaper → Beer → Eggs) (ChatGPT is getting witty nowadays)

the third person buys milk, diaper, beer, and cola (Milk -> Diaper -> Beer -> Cola).

and so on.

Now our goal is to find which patterns occur the most, that's where Apriori is going to help us.

But moving on to that, let's do some preprocessing.

```
# Convert the list of items in each transaction into a string
df['Items'] = df['Items'].apply(lambda x: ','.join(x))
df
```

```
"""
OUTPUT:

Transaction ID Items
0 1 bread,milk
1 2 bread,diaper,beer,eggs
2 3 milk,diaper,beer,cola
3 4 bread,milk,diaper,beer
4 5 bread,milk,diaper,cola
"""

# Apply one-hot encoding to convert the transaction data into a binary format
onehot = df['Items'].str.get_dummies(sep=',')
onehot

"""
OUTPUT:
beer bread cola diaper eggs milk
0 0 1 0 0 0 1
1 1 1 0 1 1 0
2 1 0 1 1 0 1
3 1 1 0 1 0 1
4 0 1 1 1 0 1
"""
```

Now let's know the frequently occurring buying patterns and there score(which is known as Support in Association).

```
# Apply the Apriori algorithm to find frequent itemsets
frequent_itemsets = apriori(onehot, min_support=0.4,
```

```
use_colnames=True)
frequent_itemsets

"""
OUTPUT:


support itemsets
0 0.6 (beer)
1 0.8 (bread)
2 0.4 (cola)
3 0.8 (diaper)
4 0.8 (milk)
5 0.4 (beer, bread)
6 0.6 (beer, diaper)
7 0.4 (milk, beer)
8 0.6 (bread, diaper)
9 0.6 (milk, bread)
10 0.4 (cola, diaper)
11 0.4 (milk, cola)
12 0.6 (milk, diaper)
13 0.4 (beer, bread, diaper)
14 0.4 (milk, beer, diaper)
15 0.4 (milk, bread, diaper)
16 0.4 (milk, cola, diaper)
"""
```

In the above code, the value in the ***min_support*** parameter sets the threshold the buying pattern should have in order to be included in the frequent itemset.

See index 1, it says bread is being purchased 80% of the time. See index 9, it says milk and bread are being purchased together 60% of the time.

```
# Generate association rules
rules = association_rules(frequent_itemsets, metric='confidence',
min_threshold=0.7)
rules

"""
OUTPUT:

antecedents consequents antecedent support consequent support
support confidence lift leverage conviction zhangs_metric
0 (beer) (diaper) 0.6 0.8 0.6 1.00 1.250000 0.12 inf 0.500000
1 (diaper) (beer) 0.8 0.6 0.6 0.75 1.250000 0.12 1.6 1.000000
2 (bread) (diaper) 0.8 0.8 0.6 0.75 0.937500 -0.04 0.8 -0.250000
3 (diaper) (bread) 0.8 0.8 0.6 0.75 0.937500 -0.04 0.8 -0.250000
4 (milk) (bread) 0.8 0.8 0.6 0.75 0.937500 -0.04 0.8 -0.250000
5 (bread) (milk) 0.8 0.8 0.6 0.75 0.937500 -0.04 0.8 -0.250000
6 (cola) (diaper) 0.4 0.8 0.4 1.00 1.250000 0.08 inf 0.333333
7 (cola) (milk) 0.4 0.8 0.4 1.00 1.250000 0.08 inf 0.333333
8 (milk) (diaper) 0.8 0.8 0.6 0.75 0.937500 -0.04 0.8 -0.250000
9 (diaper) (milk) 0.8 0.8 0.6 0.75 0.937500 -0.04 0.8 -0.250000
10 (beer, bread) (diaper) 0.4 0.8 0.4 1.00 1.250000 0.08 inf 0.333333
11 (milk, beer) (diaper) 0.4 0.8 0.4 1.00 1.250000 0.08 inf 0.333333
12 (milk, cola) (diaper) 0.4 0.8 0.4 1.00 1.250000 0.08 inf 0.333333
13 (cola, diaper) (milk) 0.4 0.8 0.4 1.00 1.250000 0.08 inf 0.333333
14 (cola) (milk, diaper) 0.4 0.6 0.4 1.00 1.666667 0.16 inf 0.666667
"""
```

Now let me explain to you what each column in the above output means one by one.

Antecedents — This column says what Item is being purchased first. See index 0, it says "beer" is being purchased first.

Consequents — This column says what Item is being purchased next after the first item(Antecedents), again see index 0, after purchasing "beer", the "diaper" is being brought.

Antecedent support — This column gives the probability of how many times the antecedent is purchased, in index 0, it says "beer" is being purchased 60% of the time.

Consequent support — Similarly, this will give the probability of how many times the consequent is purchased.

Support — This will give the probability of how many times both the Antecedent and Consequent are purchased together. See index 0, it says "beer" and "diaper" are purchased together 60% of the time.

Confidence — This column indicates the confidence of the association rule, which is the probability of finding the consequent item(s) in a transaction given that the antecedent item(s) are present. For example, the confidence of the first rule is 1.00, indicating that 'diaper' is always purchased when 'beer' is purchased.

I don’t know much about the other columns. As I already told you I am a beginner in this topic.

So, that’s it.

Day — 19☑

Day-20: Anomaly Detection Using Isolation Forest😂😂😭😂

Today, I was learning about Anomaly Detection or Outlier Detection.

If you don't know what an anomaly or outliner is, here is a simple example. Say in a class there are 15 students, 14 students in that class are average students who score marks between 50 to 70 and there is that 1 student who always scores above 97 he is the outlier.

I hope you can understand now.

To put it simply, a weird or unusual thing in a group of similar things.

This one is going to be easy, let's straight away get into the code.

Import pandas as **p**, yes you read it right "**p**".

```
import pandas as p
```

Then, to make things simple, I created a data frame myself using 15 instances and 2 columns (StudentId and Marks).

```
df = p.DataFrame({'StudentId':[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15],
```

```
                'Marks': [67, 56, 66, 74, 70, 45, 55, 59, 69, 99, 68, 51, 60,
69, 50]})

df.head()
"""
OUTPUT:

StudentId Marks
0 1 67
1 2 56
2 3 66
3 4 74
4 5 70
"""
```

Now, let's use the boxplot to see the outlier(that topper geek😳).

```
import seaborn as sns
sns.boxplot(df.Marks)
```

All the average students are represented by the below rectangle blue box

See the topper at the top.

Let's create an Isolation Forest Model to detect the outlier in the dataset.

```
from sklearn.ensemble import IsolationForest
isolation_forest = IsolationForest(contamination=0.01)  # Set contamination to the expected proportion of outliers
isolation_forest.fit(df[['Marks']])
```

You can play with the contamination parameter. I have been playing with it for a while and found that 0.01 is the value that correctly tells me the outlier.

```
# Predict outliers
outlier_preds = isolation_forest.predict(df[['Marks']])

# Add outlier predictions to DataFrame
df['Outlier'] = outlier_preds
```

So, now I added a new column in our data frame which will tell which instance in the data frame is an outlier. If an instance is an outlier then it will have -1 in the Outlier column else it will have 1. Below is the output.

```
# Print DataFrame with outlier predictions
print(df)

"""
OUTPUT:
 StudentId  Marks  Outlier
0         1    67        1
1         2    56        1
2         3    66        1
3         4    74        1
4         5    70        1
5         6    45        1
6         7    55        1
7         8    59        1
8         9    69        1
9        10    99       -1
10       11    68        1
11       12    51        1
12       13    60        1
13       14    69        1
14       15    50        1
"""
```

We found the black sheep.

That's it.

Day — 20☑

Day-21: Topic Modelling Using LDA 📄

Today, was tough. I didn't learn the topic I was supposed to learn completely, I got some help from ChatGPT to complete this day.

So, here is what I did.

I asked ChatGPT to create some sample text.

```
# Updated sample text data with "technology" and "sports" themes
documents = [
    "Smartphones are an essential piece of technology in today's digital age.",
    "Professional athletes train rigorously to excel in their respective sports.",
    "Electric vehicles represent a significant advancement in automotive technology.",
    "Watching sports events live offers an exhilarating experience for fans.",
    "Virtual reality technology immerses users in simulated environments for gaming and entertainment.",
    "Sports equipment manufacturers continually innovate to improve performance and safety.",
    "Artificial intelligence is being integrated into various aspects of modern technology.",
    "Participating in sports promotes physical fitness and overall well-being."
]
```

Then I imported the CountVectorizer and LatentDirichletAllocation.

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import LatentDirichletAllocation
```

Using the CountVectorizer, I converted the text into numerical values.

```
# Convert text data into numerical feature vectors using
CountVectorizer
vectorizer = CountVectorizer(stop_words='english',
max_features=1000)
X = vectorizer.fit_transform(documents)
```

Hoof, I forgot to main thing, Our goal with this code is to classify the text in the documents into different topics. We can accomplish that using LatentDirichletAllocation.

```
# Apply LDA
lda = LatentDirichletAllocation(n_components=2, random_state=42)  #
Assuming there are 2 topics
lda.fit(X)
# Assign each document to the topic with the highest probability
topic_assignments = lda.transform(X).argmax(axis=1)

# Group documents by their assigned topics
topic_documents = {'Technology': [], 'Sports': []}
for i, topic_idx in enumerate(topic_assignments):
    topic = 'Technology' if topic_idx == 0 else 'Sports'
```

```
    topic_documents[topic].append(documents[i])

# Print documents grouped by topics
for topic, docs in topic_documents.items():
    print(f"{topic}:")
    for doc in docs:
        print("-", doc)
    print()
```

Assigning Documents to Topics:

```
topic_assignments = lda.transform(X).argmax(axis=1)
```

- `lda.transform(X)` computes the topic distribution for each document in the corpus `X`. Each row in the resulting matrix represents a document, and each column represents a topic, with values indicating the probability of each document belonging to each topic.
- `argmax(axis=1)` returns the index of the maximum value along each row, effectively assigning each document to the topic with the highest probability.

Grouping Documents by Topics:

```
topic_documents = {'Technology': [], 'Sports': []}
for i, topic_idx in enumerate(topic_assignments):
    topic = 'Technology' if topic_idx == 0 else 'Sports'
    topic_documents[topic].append(documents[i])
```

- We initialize a dictionary `topic_documents` with two keys: `'Technology'` and `'Sports'`, each associated with an empty list.
- We iterate through the `topic_assignments`, which contains the index of the assigned topic for each document.
- For each document, we determine its assigned topic based on the index (`topic_idx`) obtained from `topic_assignments`.
- We append the document to the corresponding list in the `topic_documents` dictionary based on its assigned topic.

Printing Documents Grouped by Topics:

```
for topic, docs in topic_documents.items():
    print(f"{topic}:")
    for doc in docs:
        print("-", doc)
    print()
```

- We iterate through the `topic_documents` dictionary, which contains lists of documents grouped by topics.
- For each topic, we print the topic label (`topic`) followed by a colon (`:`).
- We iterate through the list of documents (`docs`) associated with the current topic and print each document preceded by a hyphen (`-`).

- We print a blank line after printing all documents for a given topic to separate topics in the output.

That's it.

Day — 21☑

Day-22: Dimensionality Reduction Using t-SNE and UMAP 🍞 ➞ 🥪

Today, I was learning about Dimensionality Reduction. If you don't know what Dimensionality reduction is, here is a simple definition.

When dealing with large datasets we tend to have more features(Columns) which makes it difficult for us to make sense of the data through visualization. That's when dimensional reduction techniques come to the rescue.

Using dimensionality reduction techniques like Principal Component Analysis(PCA), t-Distributed Stochastic Neighbor Embedding(t-SNE), Singular Value Decomposition(SVD), Uniform Manifold Approximation and Projection(UMAP), and a few more techniques we can reduce the higher dimensions or features of our data into 2 or 3 dimensions.

As humans, it is not possible for us to see and understand graphs that are more than 3 dimensions. To be honest, understanding the 3D visualizations in and of itself is difficult, but we are good with 2D visualization.

Enough of the theory, let's get into the code.

Below is the video I used to learn UMAP. The below video is code-oriented, it doesn't cover any theory, but all you need to understand is that dimensionality reduction techniques are used to reduce the dimension(or features) of our dataset so that we can visualize and understand our data more clearly.

https://youtu.be/015vL0cJfA0

You will start to understand this concept more clearly once we get into the code.

I want you to code along with me. Open your Google Colab now.

We are going to use the "Iris" Dataset.

First, import a library called **Plotly**. If you don't know what Plotly is, it is an amazing visualization library. It is one of the competitors of Matplotlib and Seaborn.

```
import plotly.express as px
```

We can import the "Iris" Dataset directly from the Plotly library. Using the below code.

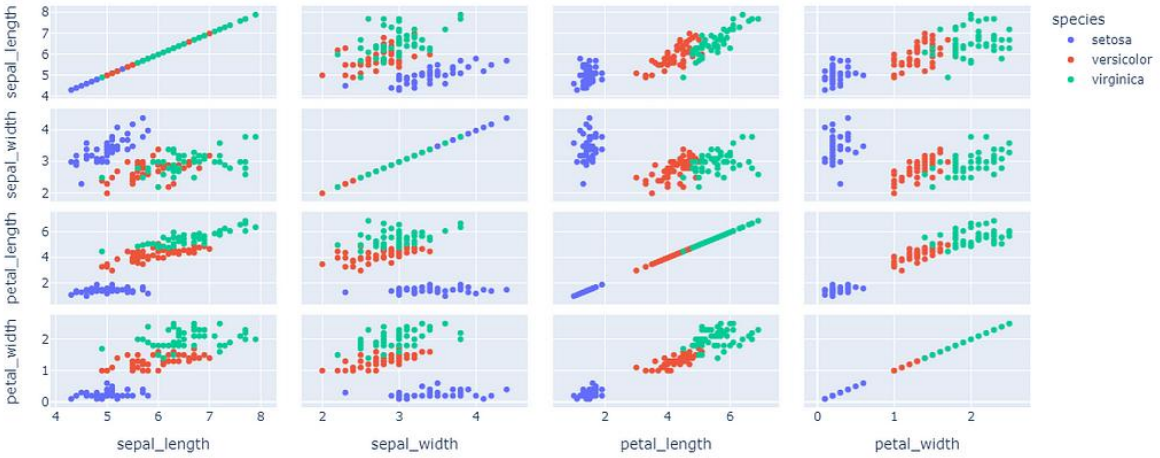
```
df = px.data.iris()

df.head()
```

```
"""
OUTPUT:

sepal_length sepal_width petal_length petal_width species species_id
0 5.1 3.5 1.4 0.2 setosa 1
1 4.9 3.0 1.4 0.2 setosa 1
2 4.7 3.2 1.3 0.2 setosa 1
3 4.6 3.1 1.5 0.2 setosa 1
4 5.0 3.6 1.4 0.2 setosa 1
"""
```

We have an extra column named "***species_id***" which we don't want to include during visualization. So, let's create a variable named **features** and store only the features we need.

Then I am going to create a variable **fig** and score my **scatter_matrix** inside it. A scatter_matrix will give us the scatter plot of all the features.

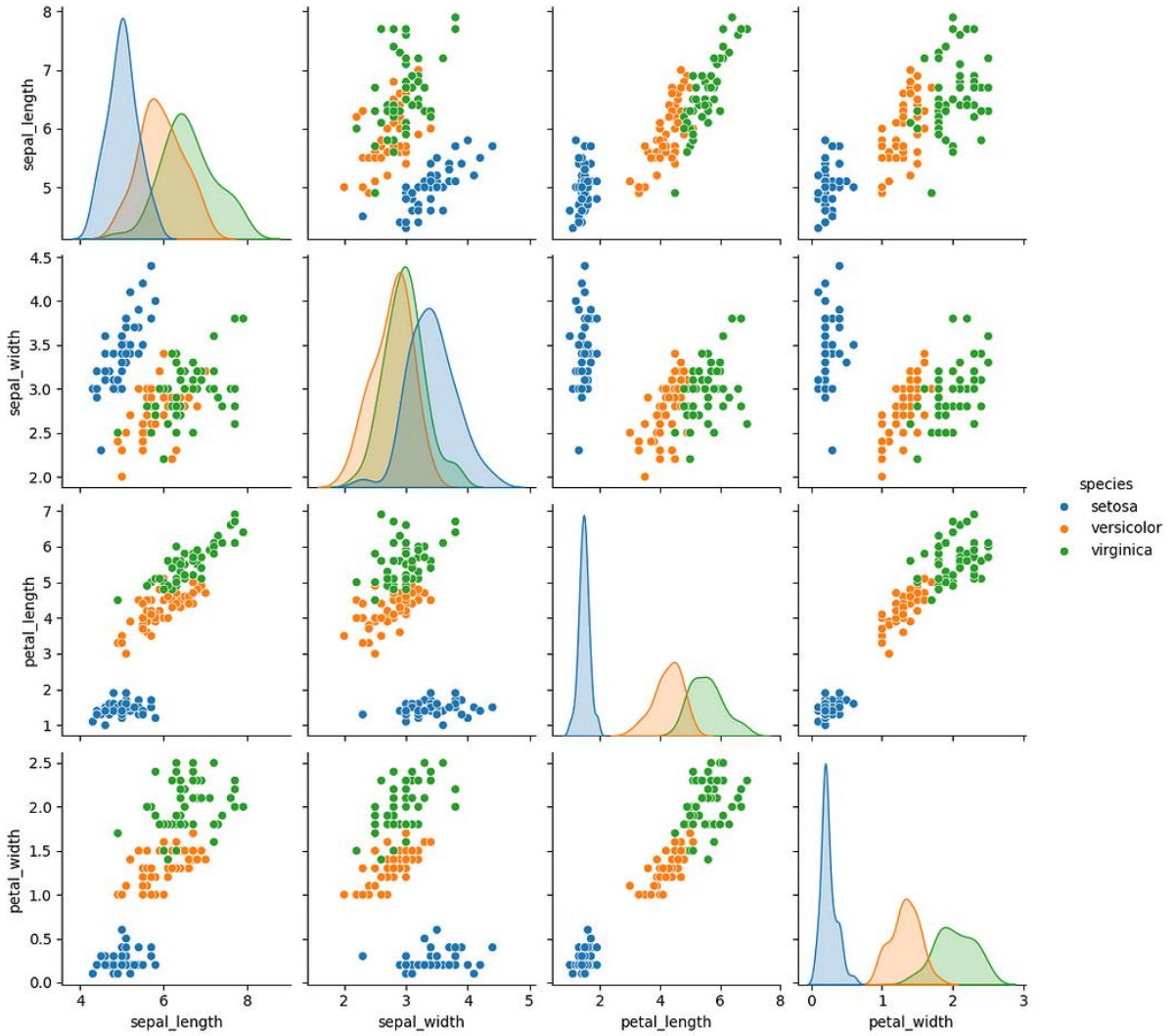
```
features = ['sepal_length', 'sepal_width', 'petal_length',
'petal_width']
fig = px.scatter_matrix(df, dimensions=features, color='species')
fig.show()
```



Looks good.

We can do the same thing using Seaborn as well. It's named **pairplot**.

```
import seaborn as sns
df2 = df.drop(columns='species_id')
sns.pairplot(df2, hue='species')
```



Good.

But the difference between these two is when you hover over any point in the graph produced by the Plotly library you will be able to see its values, not only that there are zoom-in and zoom-out options and plenty of other options as well.

If you are coding along you will understand what I am trying to say.

Now we are going to create a t-SNE model which will reduce the dimension of our "Iris" dataset into to 2D.

```
from sklearn.manifold import TSNE
features = df.loc[:, :'petal_width']
tsne = TSNE(n_components=2, random_state=7)
projections = tsne.fit_transform(features)
fig = px.scatter(projections, x=0, y=1, color=df.species,
labels={'color':'species'})
fig.show()
```

First we imported the **TSNE** function from the sklearn library/manifold module.

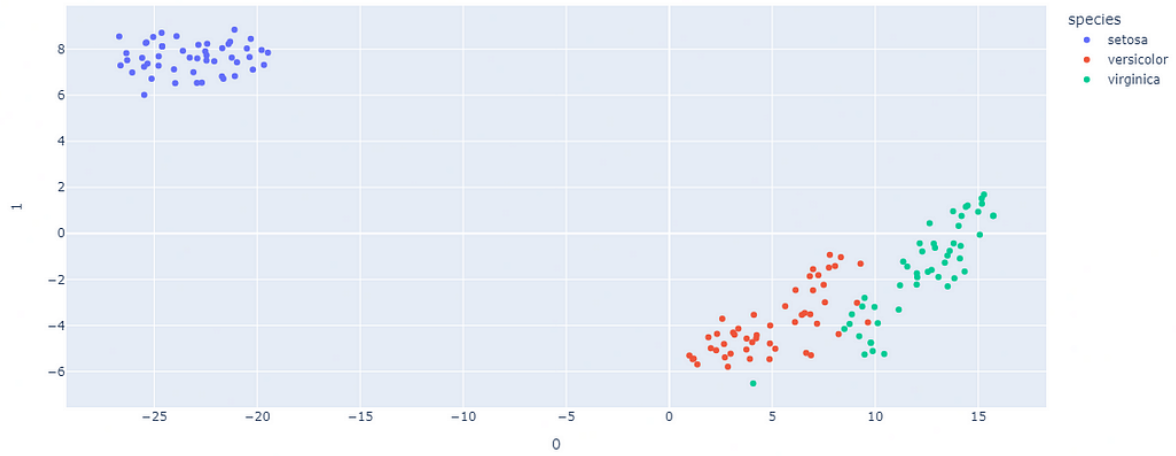
Then we created a variable features which will take 4 main columns('***sepal_length***', '***sepal_width***', '***petal_length***', '***petal_width***') columns ignoring the "***species***" and "***species_id***".

After that, we created our TSNE model which will transform any dataset we pass into 2-dimensional ones because we have set the **n_components** parameter as 2.

Then we passed the **features** to the TSNE model and stored the transformed 2D features into a variable called **projections**.

Finally, Using the Plotly library we created a scatter plot of the 2D features (projections).

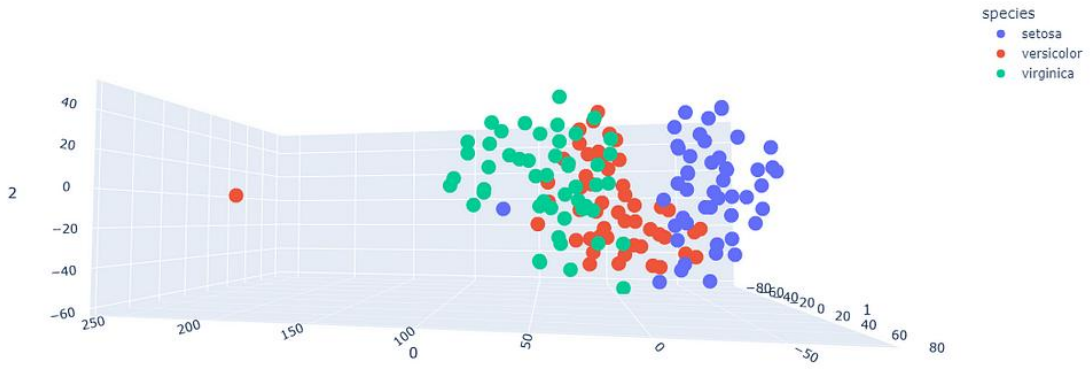
Below is the output of the code.



Very clear!

Similarly, now we are going to create 3D features by just changing the **n_components** parameter.

```
tsne = TSNE(n_components=3, random_state=7)
projections = tsne.fit_transform(features)

fig = px.scatter_3d(projections, x=0, y=1, z=2, color=df.species,
labels={'color':'species'})
fig.show()
```



Looks cool!

If you are coding along, you can play with the above 3D graph.

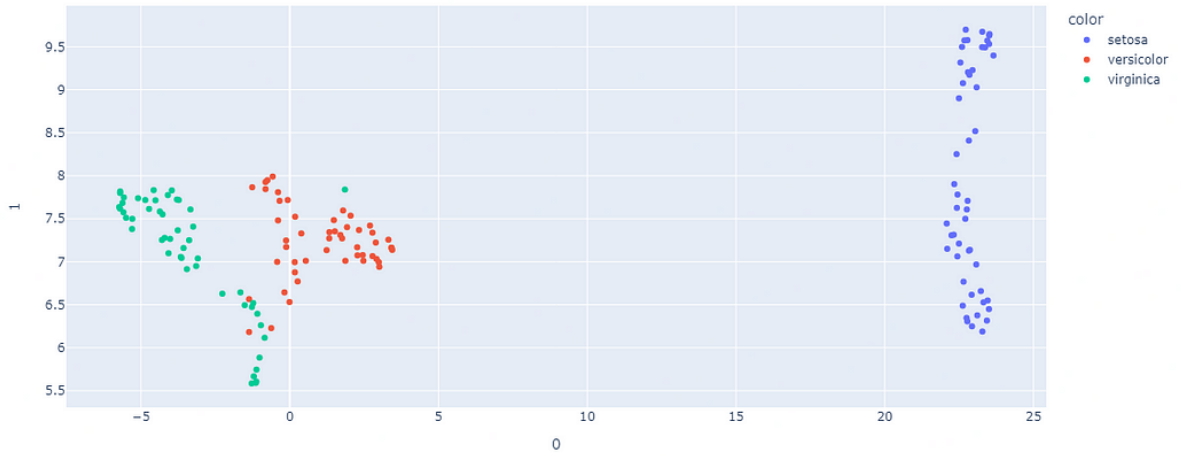
Now, we have created 2D and 3D features using t-SNE. Let's do the same using UMAP.

But before that, you need to install it, use the below to install the library.

```
!pip install umap-learn
```

Let's import and reduce the features using UMAP and see the visualization in both 2D and 3D.

```
umap_2d = UMAP(n_components=2, random_state=56)
umap_3d = UMAP(n_components=3, random_state=43)

proj_2d = umap_2d.fit_transform(features)
proj_3d = umap_3d.fit_transform(features)

fig_2d = px.scatter(proj_2d, x=0, y=1, color=df.species)
fig_3d = px.scatter_3d(proj_3d, x=0, y=1, z=2, color=df.species)
fig_2d.show()
```

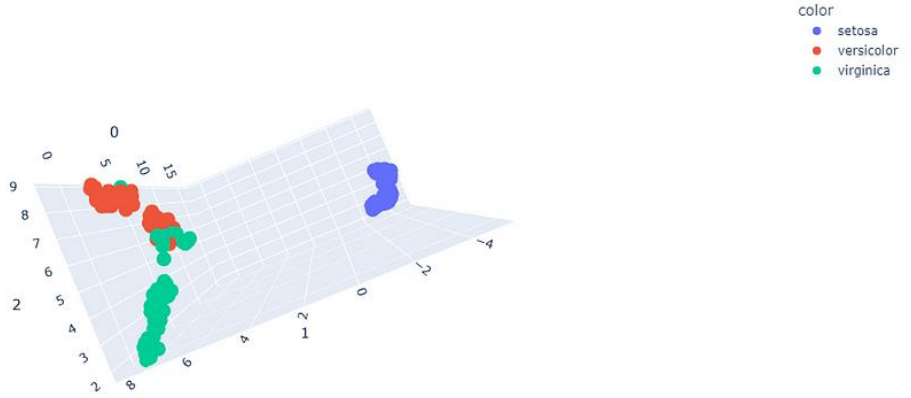


Compare this with the 2D figure of t-SNE

Each dimensionality technique works differently. In the future, I will learn the workings of these techniques and will write a comprehensive article about them. But for now, let's know them practically.

Below is the 3D visualization.

```
fig_3d.show()
```



Play with it.

That's it.

Day — 22☑

Day-23: Anomaly Detection Using Isolation Forest and Local Outlier Factor 1️⃣ 1️⃣ 0️⃣ 1️⃣ 1️⃣

Today, I was again learning about outliers.

I used two different Outlier detection techniques to identify outliers from a sample NumPy array. This article aims to make you understand how to implement the outlier detection techniques to identify the outlier in our data in a very simple way.

I am not gonna go into depth about the techniques, because I don't know that myself. I am just gonna show you the code. You will be able to understand it very easily.

Let's get into the code.

First, let's import the necessary libraries.

```
import numpy as np
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor
import matplotlib.pyplot as plt
```

As you can see from the above code, the two techniques that we are going to use to detect the outliers are **Isolation Forest** and **Local**

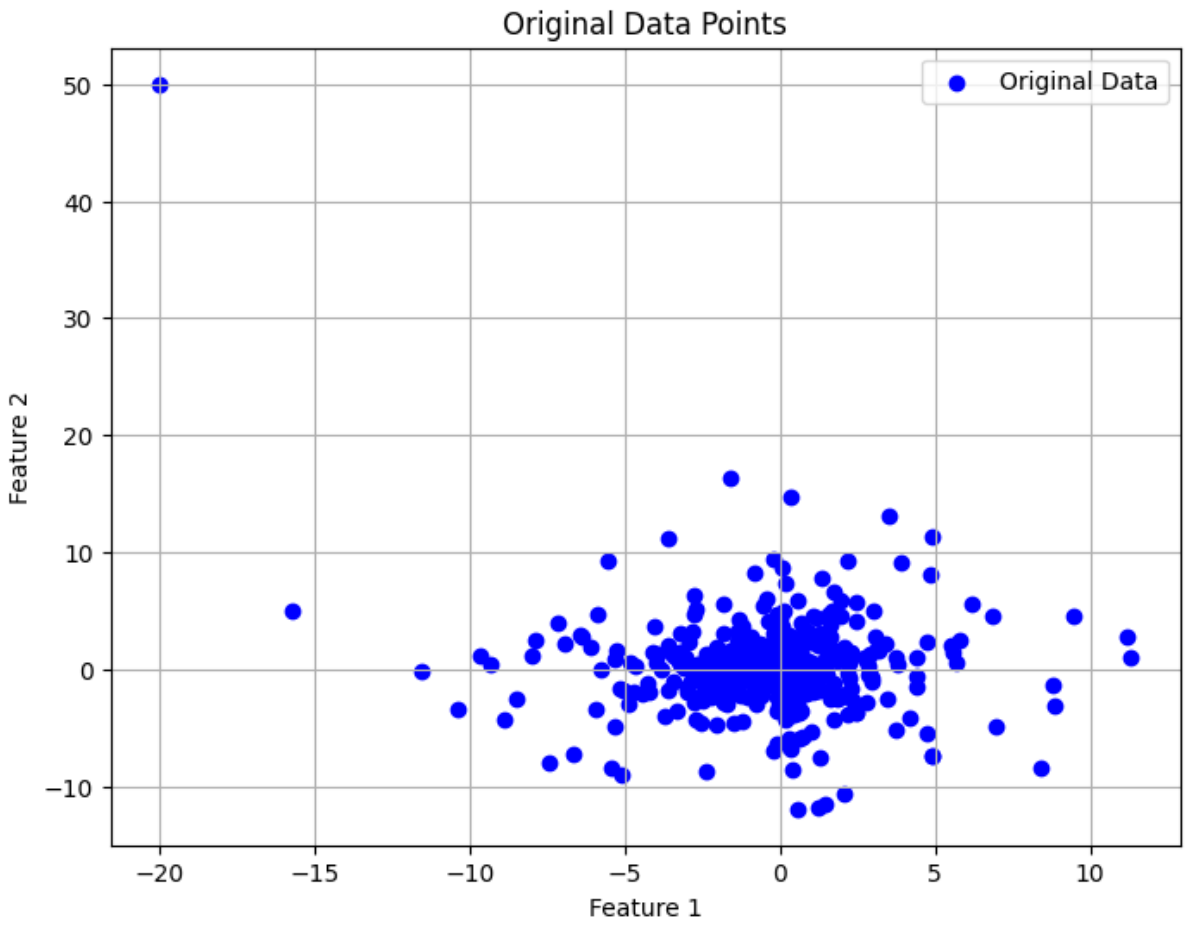
Outlier Factor. We need NumPy to create sample data points, and we need Matplotlib to visualize our data points.

```
# Generate more data points
np.random.seed(42)
X = np.concatenate([X, np.random.normal(loc=0, scale=3, size=(100, 2))])
```

The above code will create 100 random data points in 2 dimensions. Just print **X** and check out how it looks.

Now, let's see how our data looks like in scatter plot.

```
# Plot the original data points
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c='blue', label='Original Data')
plt.title('Original Data Points')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.grid(True)
plt.show()
```



Can you see the Anomaly(Outlier)

So, these are our data points.

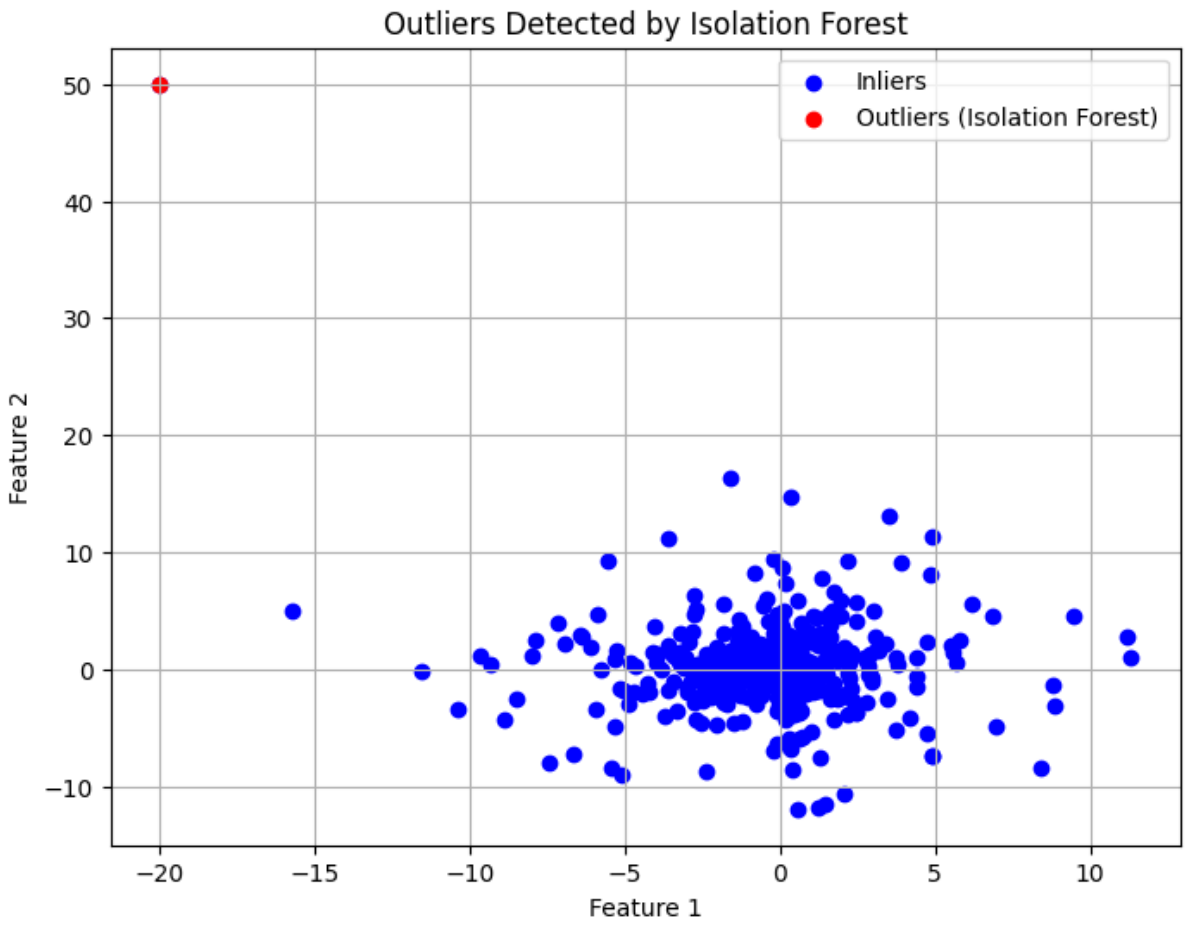
Now, let's use the first technique(Isolation Forest) to identify the outlier and also visualize it.

```
# Apply Isolation Forest
clf = IsolationForest(n_estimators=20, contamination=0.001)
clf.fit(X)
outliers_isf = clf.predict(X)

# Plot outliers detected by Isolation Forest
plt.figure(figsize=(8, 6))
```

```
plt.scatter(X[:, 0], X[:, 1], c='blue', label='Inliers')
plt.scatter(X[outliers_isf == -1, 0], X[outliers_isf == -1, 1], c='red',
label='Outliers (Isolation Forest)')
plt.title('Outliers Detected by Isolation Forest')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.grid(True)
plt.show()
```

The ***n_estimators*** parameter will create 20 decision trees, and the ***contamination*** parameter is used to control the threshold value. If you increase the ***contamination*** parameter value to 0.01 the model will find more outliers, just play with it you will start to understand how it works.

Let's see the scatter plot produced by the Isolation Forest technique.



Red is outlier!

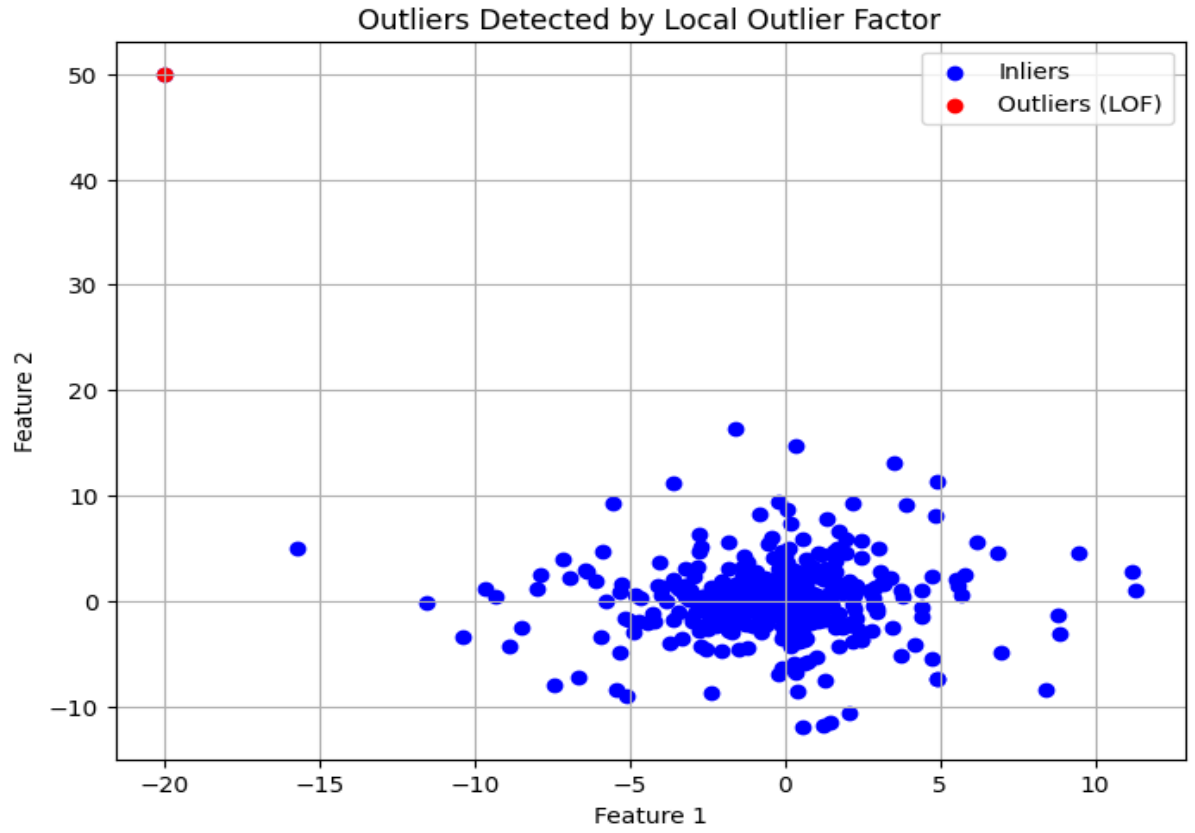
Nice!

If you increase the contamination parameter value few blue dots which are spread out will also be considered as outliers.

Let's see how Local Outlier Factor performs.

```
# Apply Local Outlier Factor
lof = LocalOutlierFactor(n_neighbors=20, contamination=0.001)
outliers_lof = lof.fit_predict(X)

# Plot outliers detected by Local Outlier Factor
```

```
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c='blue', label='Inliers')
plt.scatter(X[outliers_lof == -1, 0], X[outliers_lof == -1, 1], c='red',
label='Outliers (LOF)')
plt.title('Outliers Detected by Local Outlier Factor')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.grid(True)
plt.show()
```



Same result!

That's it.

Day — 23☑

Day-24: Kernel Density Estimation Using Seaborn 📊

Today's topic is **Kernel Density Estimation(KDE)**. Before moving on to the topic, you need to know what is a **Probability Density Function(PDF)**.

Say, there are some data points like this,

[3, 4, 5, 1, 2, 3, 4, 5, 6, 3, 3, 3, 5, 6, 7, 1, 1, 1, 3, 3, 3, 5]

From the above data points can you tell me which number has more density?

If you said 3, then you already know what PDF is.

It's really simple — when there are many continuous values or data points, we can group them into some range of values, for example, [45, 67, 12, 10, 45, 78, 90, 87, 100, 34, 56, 89, 90, 15, 20, 27] the values [10, 12, 15, 20] can be grouped as one in the range 10 to 20, the values [27, 34, 45, 45] can be grouped as one in the range 30 to 50 and so on.

I hope you can understand what PDF is. It's just knowing how dense the data points are in our dataset.

Now let's come to KDE, KDE is a density estimator. But in this article, I am not gonna confuse you by creating a model using KDE, we are just going to visualize our data points using KDE.

I hope you know about Histograms. A histogram is also a Density estimator. It helps us understand how many data points are there in a particular range.

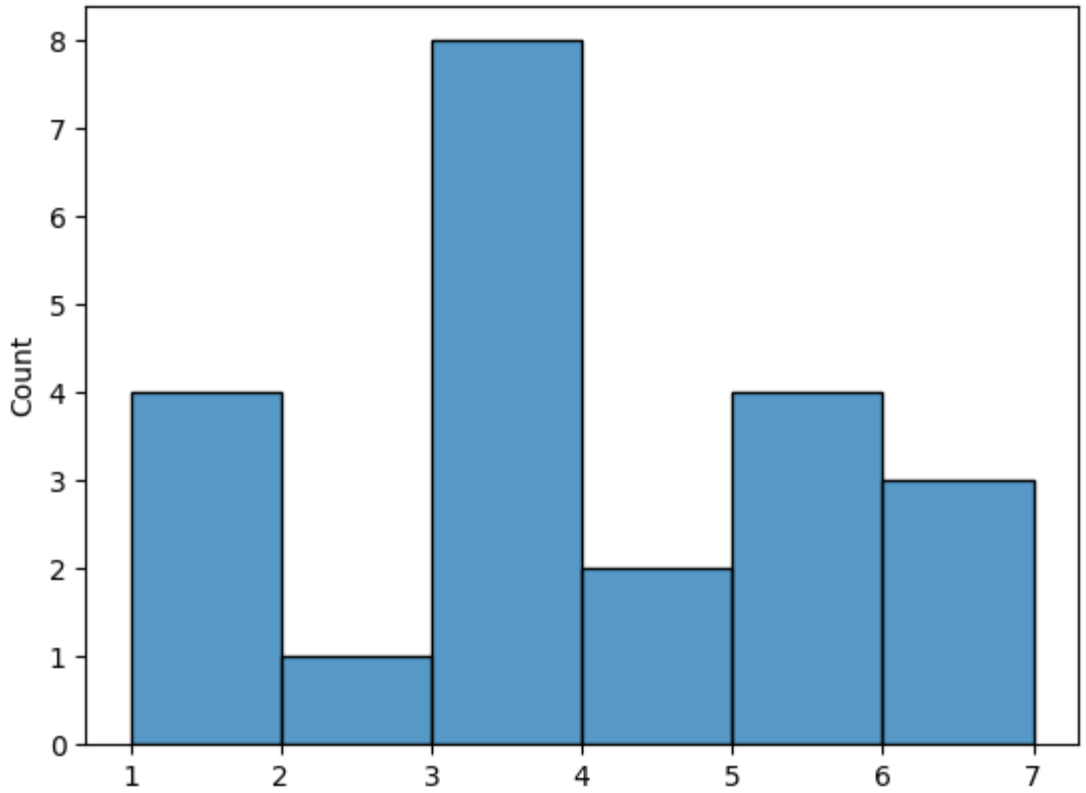
If you still haven't understood anything, no problem, you will understand it now. Let's get into the code.

Import the necessary libraries.

```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

Now, let's use a histogram and plot the sample data points that we saw in the beginning of this article.

```
sample = np.array([3, 4, 5, 1, 2, 3, 4, 5, 6, 3, 3, 3, 5, 6, 7, 1, 1, 1, 3, 3, 3, 5])
sns.histplot(sample)
```

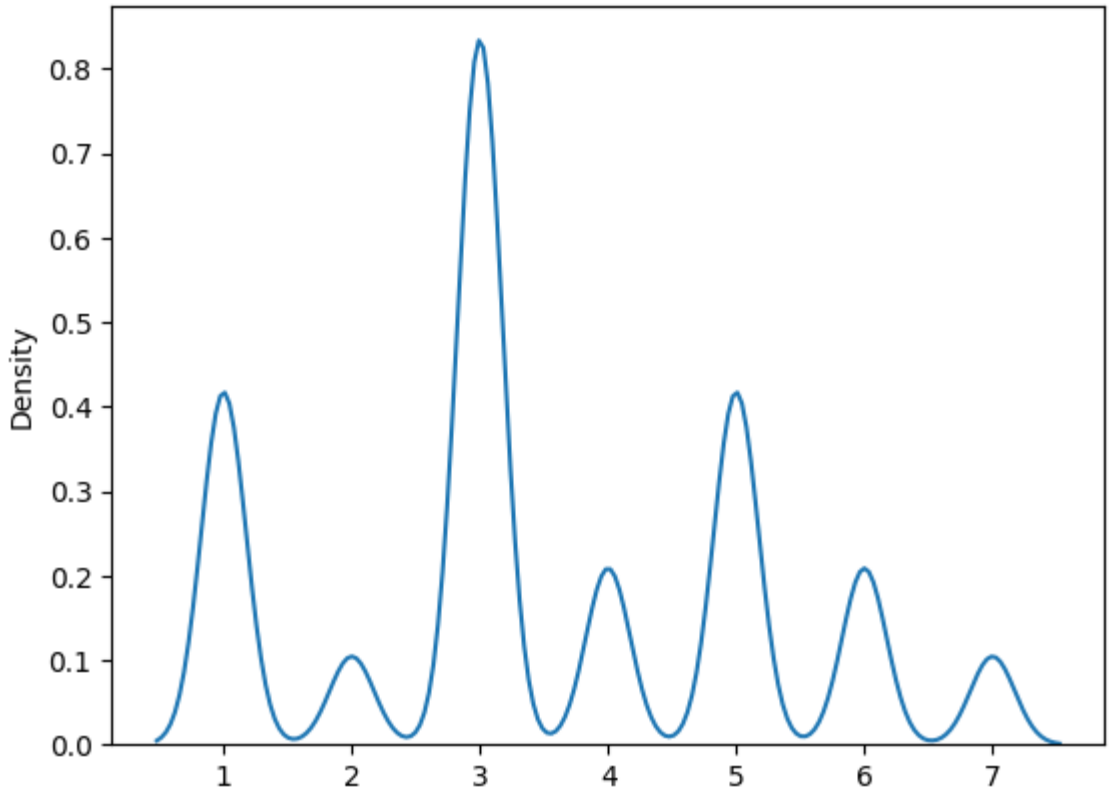


Understood?

From the histogram, we can conclude that the number 3 has a huge density.

Now, KDE is not so different from the histogram, instead of creating bars to know the density we are going to create curves that's it.
Below is the code.

```
sample = np.array([3, 4, 5, 1, 2, 3, 4, 5, 6, 3, 3, 3, 5, 6, 7, 1, 1, 1, 3, 3, 3, 5])
sns.kdeplot(sample, bw=0.1)
```



Curves

The **bw** parameter is called the Band-Width. If you increase the bandwidth the curves will become more smooth. As the number of data points in our sample is very small, if need to set the bandwidth as low as possible.

You can change the value in the **bw** parameter and see how the curves change.

Enough of the example, let's import a dataset and explore the KDE some more.

In this article, First we are going to plot a Univariate KDE Plot and then a Bivariate KDE plot using the Seaborn Library.

First, import the dataset. We are going to use a dataset that is already available in the Seaborn library. This dataset contains information about cars.

```
car = sns.load_dataset('mpg')
car.head()
mpg cylinders displacement horsepower weight acceleration
model_year origin name
0 18.0 8 307.0 130.0 3504 12.0 70 usa chevrolet chevelle malibu
1 15.0 8 350.0 165.0 3693 11.5 70 usa buick skylark 320
2 18.0 8 318.0 150.0 3436 11.0 70 usa plymouth satellite
3 16.0 8 304.0 150.0 3433 12.0 70 usa amc rebel sst
4 17.0 8 302.0 140.0 3449 10.5 70 usa ford torino
```

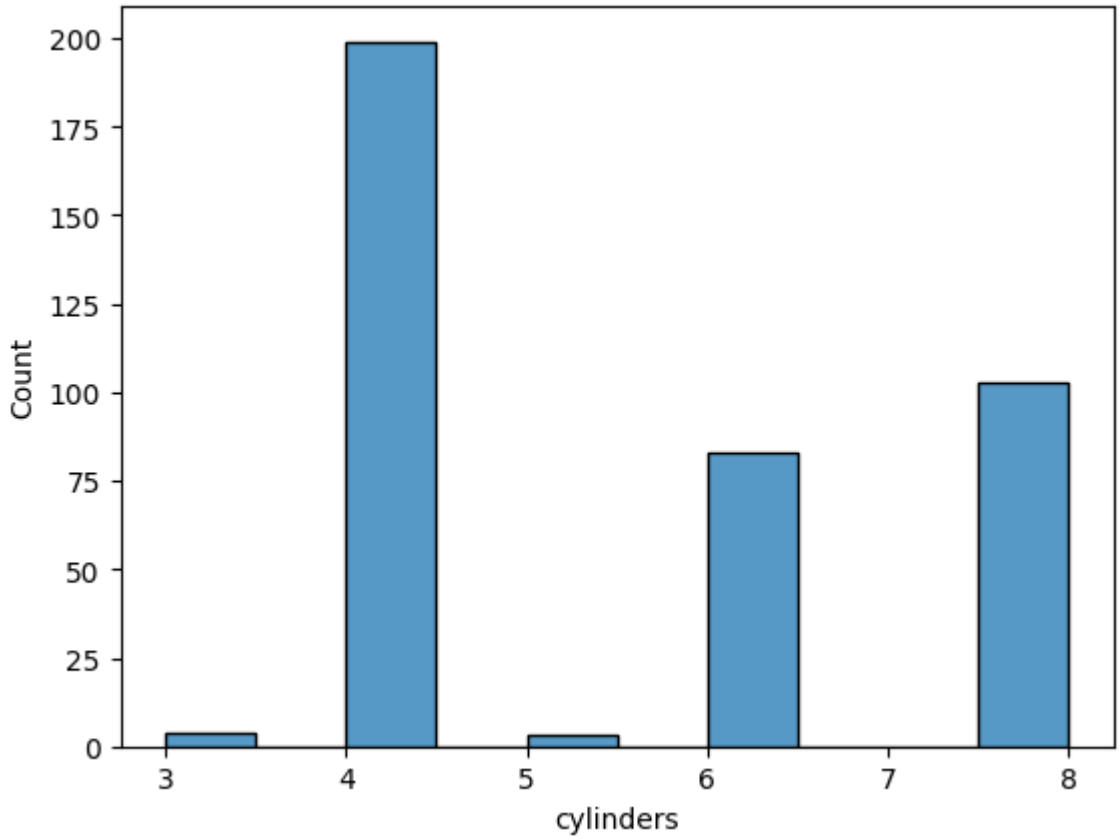
mpg stands for mileage per gallon.

So, this is the dataset we are going to use. Let's explore the dataset to understand it better.

```
car.cylinders.nunique() #5
car.cylinders.unique() #array([8, 4, 6, 3, 5])
```

There are 5 unique values present in the "***cylinders***" column of the dataset. If we want to know the density of each of the values, we know what we should do. Let's do that.

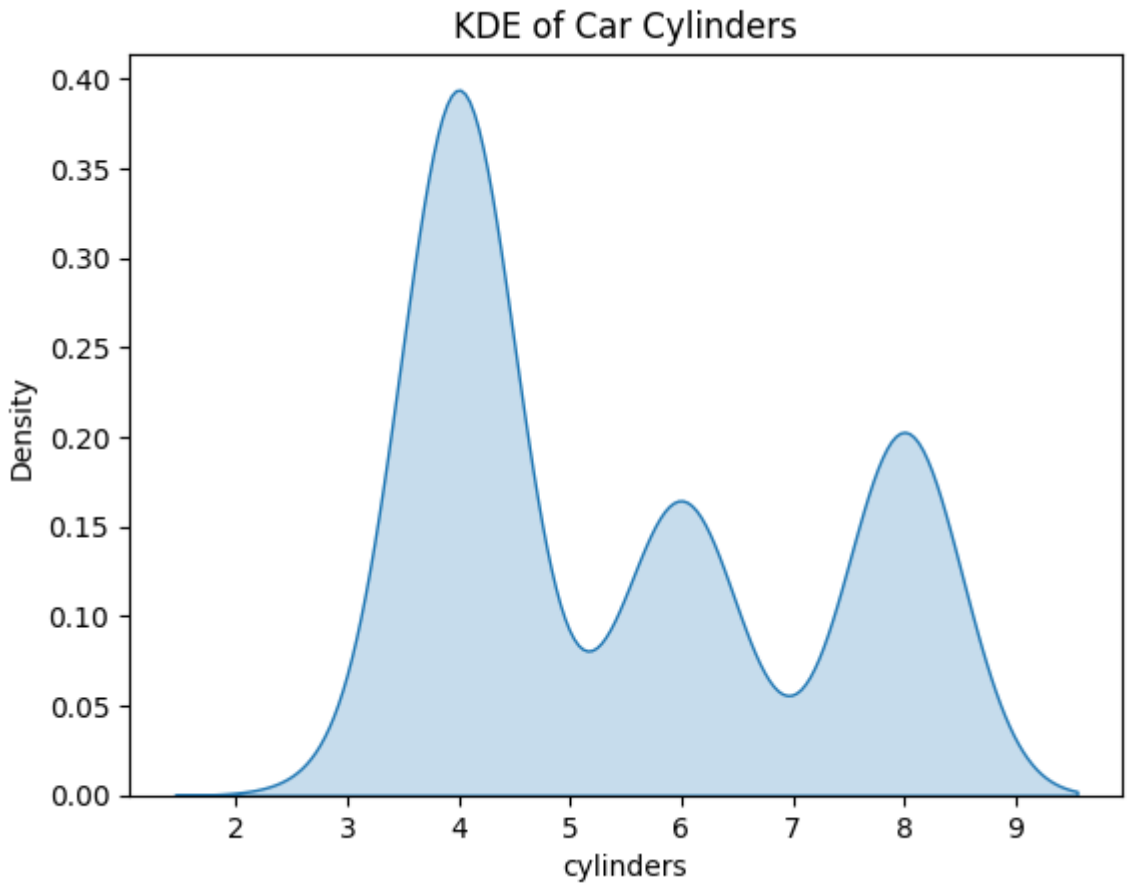
```
sns.histplot(car.cylinders)
```



Nice!

So, there are more 4-cylinder cars in our dataset. Let's KDE.

```
sns.kdeplot(car.cylinders, shade=True)
plt.title('KDE of Car Cylinders')
```



Curves!

The **shade** parameter just shades the inner parts of the curve, just remove the parameter and run the code you will understand its purpose.

Let's try some other columns.

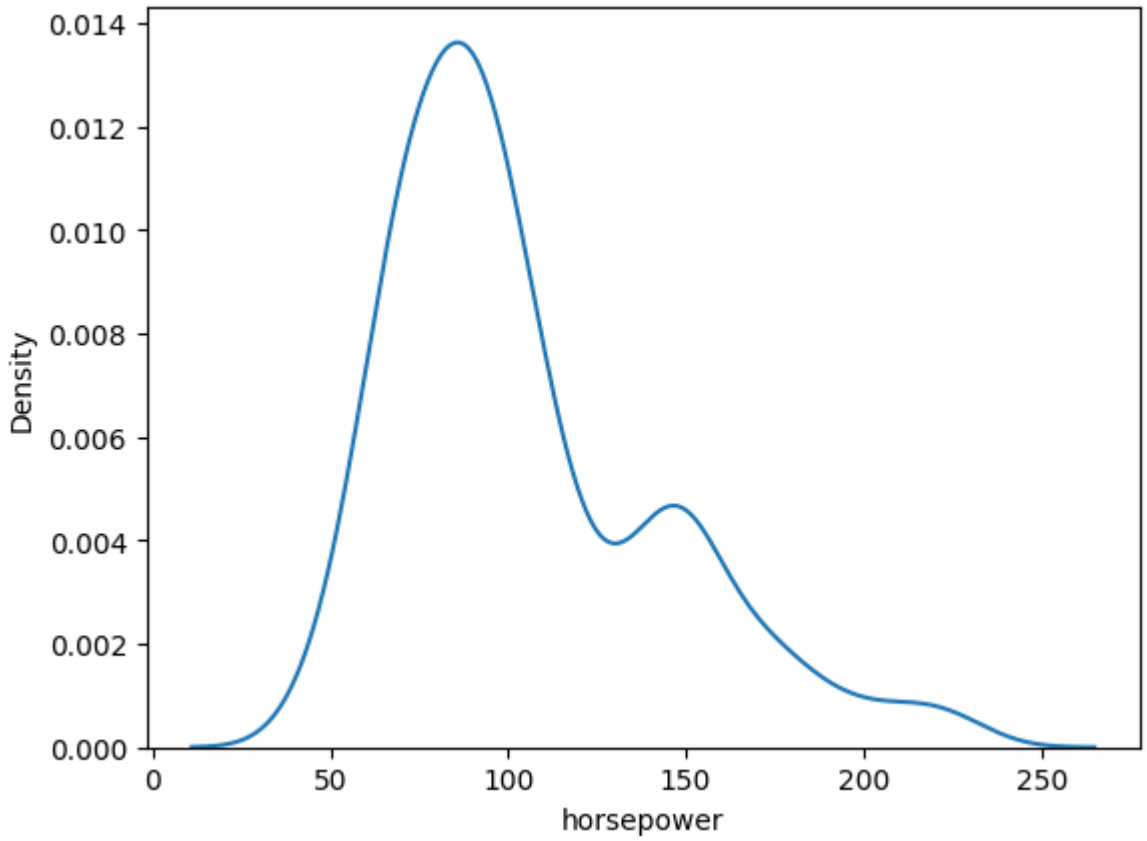
Before that, remove the null values.

```
car.isnull().sum()
"""
OUTPUT:
mpg          0
```

```
cylinders      0
displacement   0
horsepower     6
weight         0
acceleration   0
model_year     0
origin         0
name           0
"""

car = car.dropna()
```

Now, let's KDE plot the "horsepower" column.

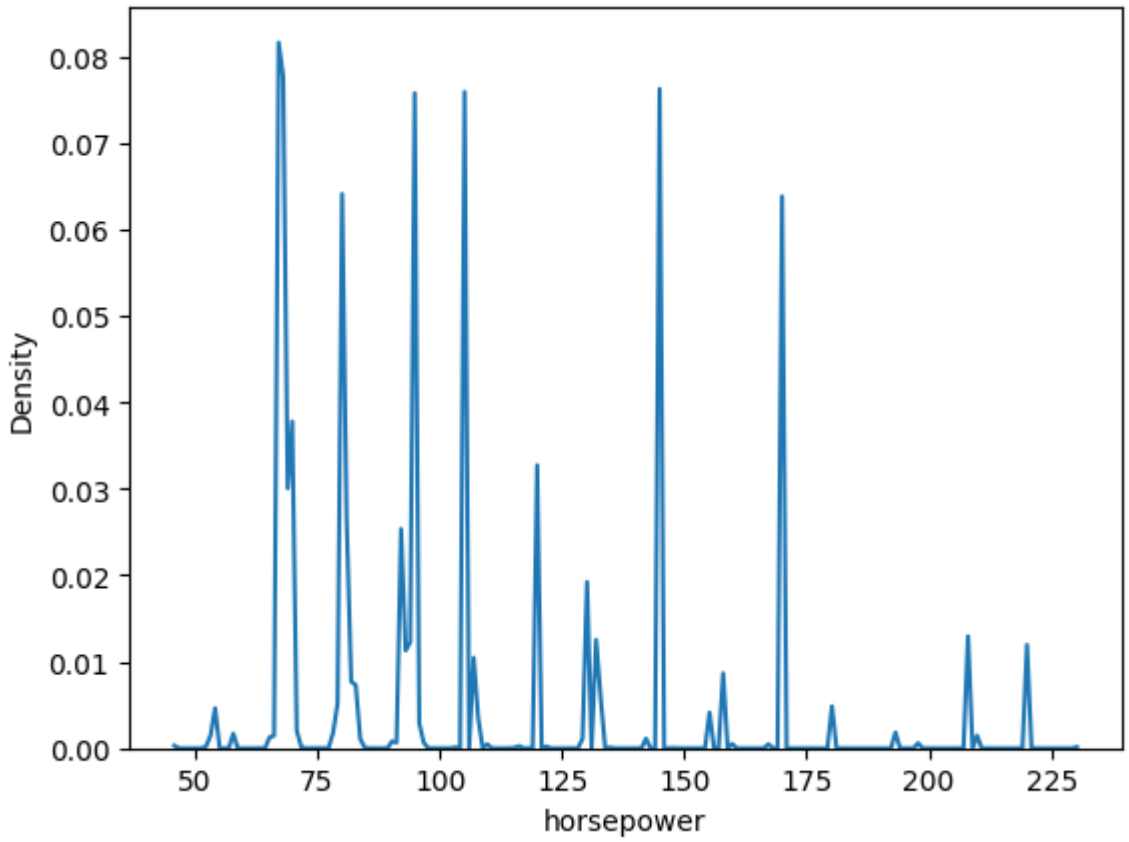
```
sns.kdeplot(car.horsepower)
```



Most car has horsepoer between 50–100

If you want to know more precise information, we can decrease the bandwidth.

```
sns.kdeplot(car.horsepower, bw=0.002)
```

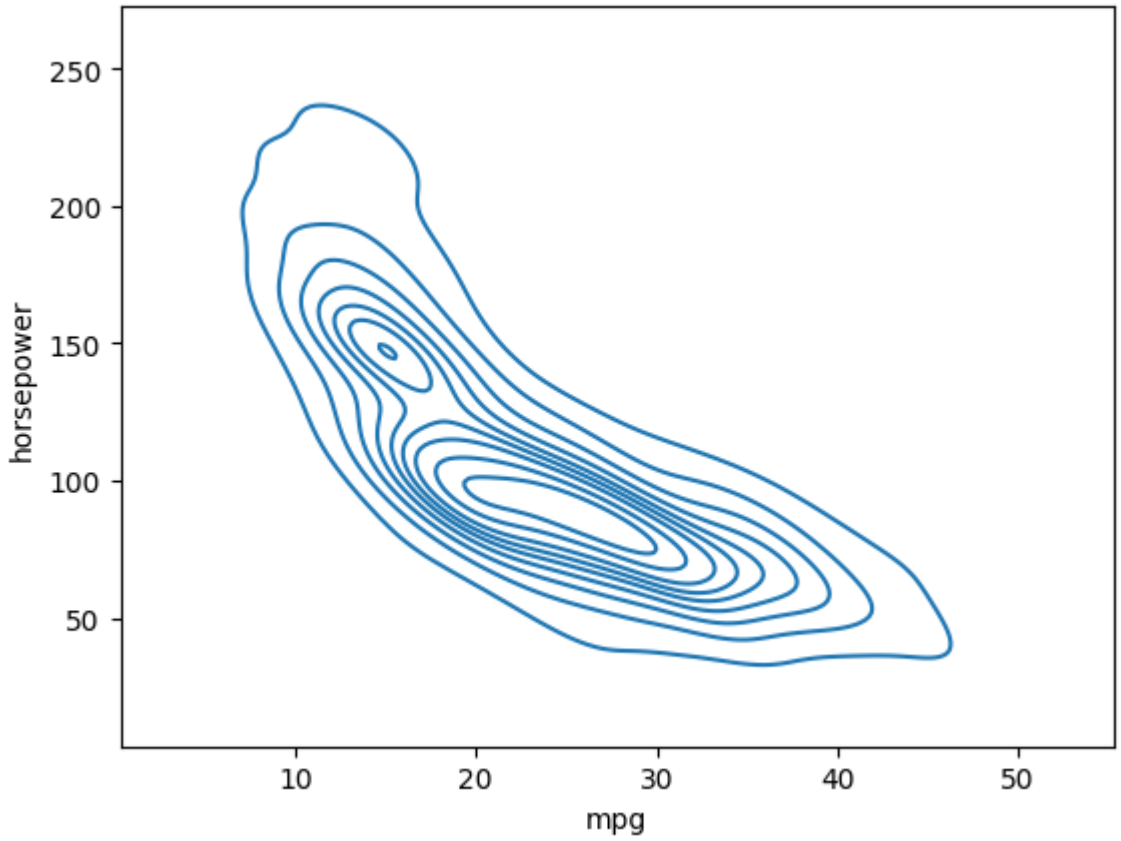


Looks like a heartbeat!

Also, try a histogram. (Are you just reading this or coding along).

Till now we have been using just one column (Univariate KDE), let's two now.

```
sns.kdeplot(data=car, x=car.mpg, y=car.horsepower)
```

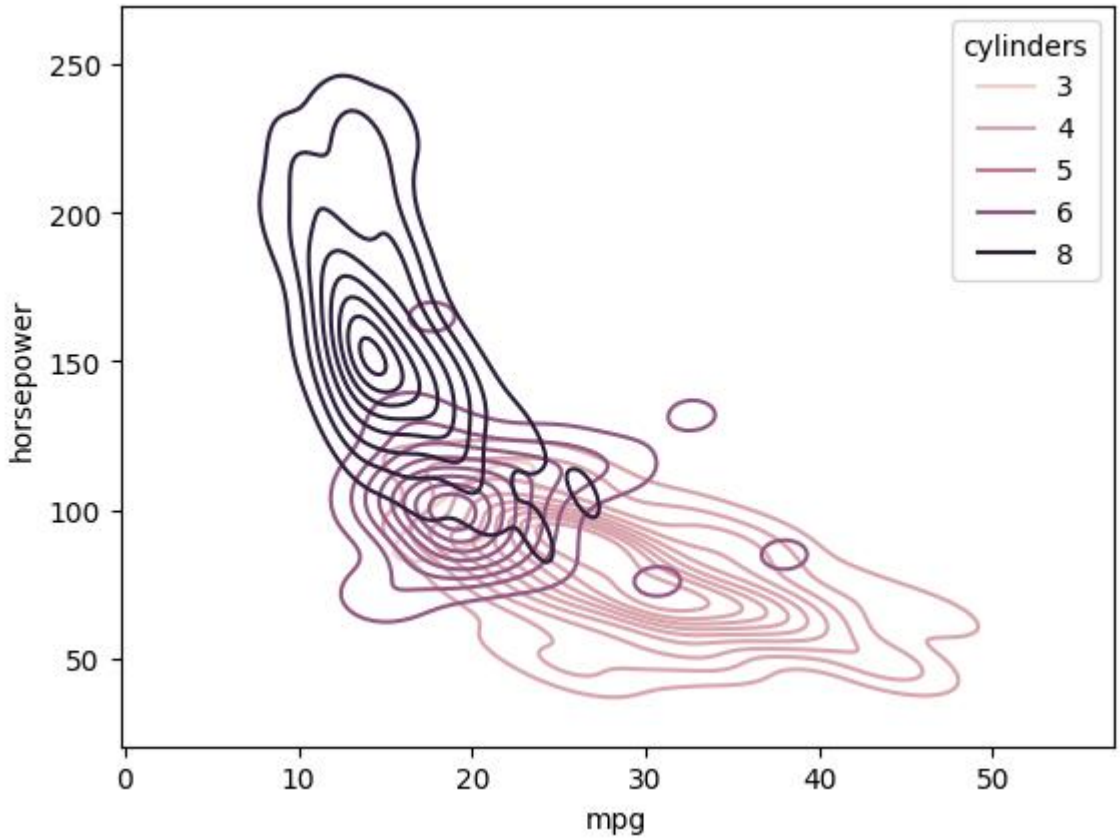


Imagine this like a mountain.

The above graph tells us the density of data present in both columns. You should try to visualize the above graph in 3D.

We can also do something to know the density of cars giving more horsepower and mileage based on the number of cylinders present.

```
sns.kdeplot(data=car, x=car.mpg, y=car.horsepower, hue=car.cylinders)
```

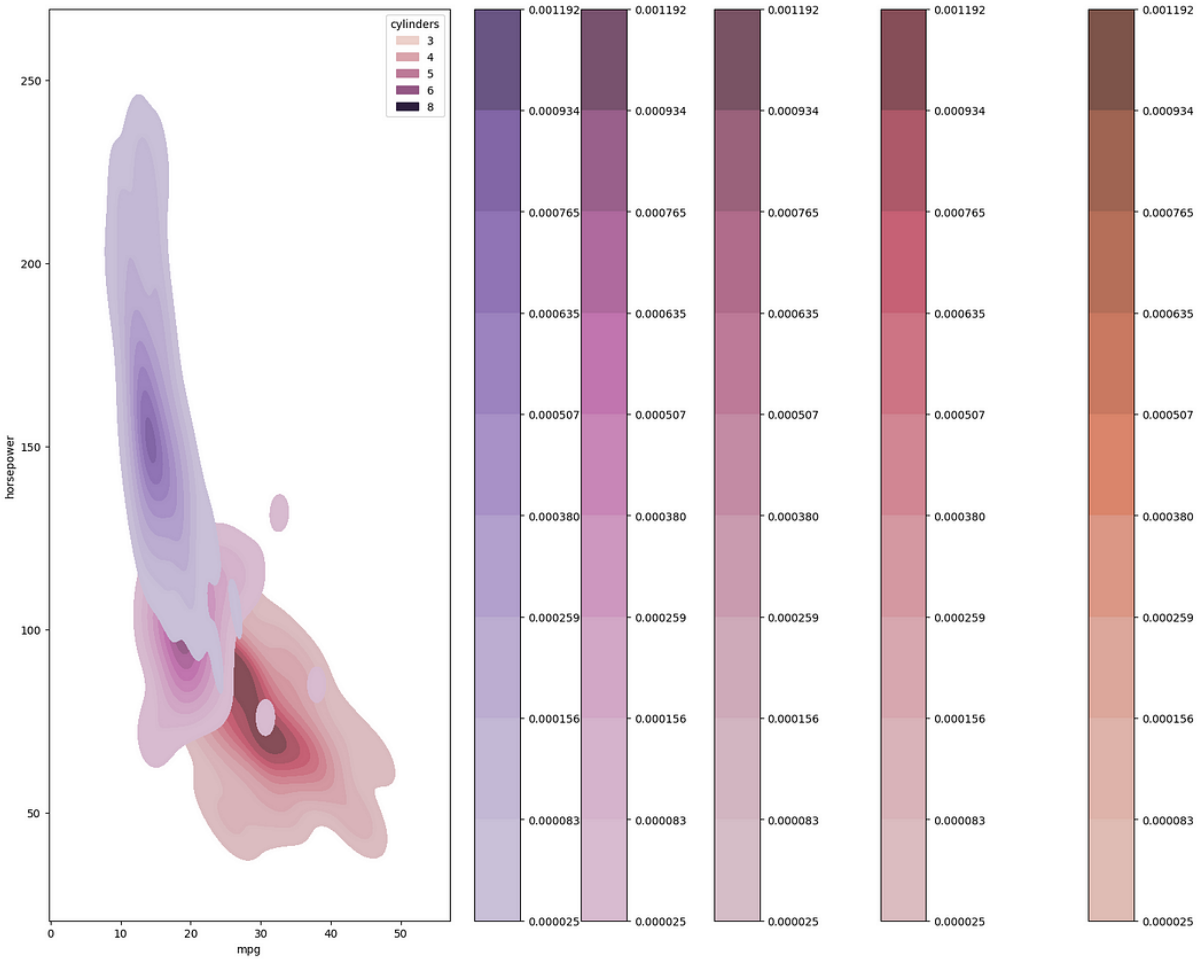


spaghetti

So, cars with 8 cylinders are giving less mileage but more horsepower (I know that is very obvious!). The black color spaghetti indicates that.

Let's do something crazy.

```
plt.figure(figsize=(20, 15))
sns.kdeplot(data=car, x=car.mpg, y=car.horsepower, hue=car.cylinders, shade=True, cbar=True)
```



Ever seen something like this?

If we set the **shade** and the **cbar** parameters to true we will have some good colors indicating our density in 2D, the dark color places mean high density, and the light color places indicate low density.

That's it.

Day — 24☑

Day-25: Customer Segmentation using Hierarchical Clustering 🏪🛍️

Today, I found an amazing website called EnjoyAlgorithms.

From that website, I learned about Hierarchical Clustering.

Hierarchical Clustering is another clustering algorithm similar to KMeans and DBSCAN.

In my previous projects, I have used the KMeans algorithm and the DBSCAN algorithm and performed some clustering on some sample datasets, but I never touched Hierarchical Clustering.

This is the first time I am learning about the Hierarchical Clustering algorithm and it turns out to be easy.

I am going to use the Mall Customer Dataset, so get the dataset using the below link before following along.

Click this 👉 Mall Customers Dataset

Before getting started with the code, let me tell you a few things about customer segmentation, why we need to segment customers, and what is the purpose of using clustering algorithms to segment customers.

Let's say you are running an online store. For the past few years, you have been collecting some data about your customers like their gender, annual income, age, and spending score(A score you give to each customer based on how much money they spend on products on your online store if someone buys so many products from your site then they will have more spending score than someone who rarely buys).

Now you have decided to run an ad campaign by sending emails and targeting ads on Facebook, YouTube, and other social platforms. It is not efficient if you run ads on all your customers, because customers who buy rarely from you are just going to ignore your emails and ads on social media.

If you want to be efficient in running your ad campaign you need to target the right groups of customers who have good spending scores and annual income.

This is where clustering comes into play. With the help of clustering, you can group your customers into different groups based on their annual income and spending scores.

To do clustering we need to use clustering algorithms this is where the Hierarchical Clustering algorithm comes into play(You can also use other algorithms).

Now, let's learn more about Hierarchical Clustering and how it works.

Hierarchical Clustering works differently when compared to other clustering algorithms.

Let's there are 10 data points like this,

[100, 50, 60, 140, 300, 70, 90, 110, 120, 80].

The hierarchical clustering first groups,

[50, 60], [70, 80], [90, 100], [110, 120], [140]

Then it will group the above 5 clusters further,

[50, 60, 70, 80], [90, 100, 110, 120], [140]

Now let's cluster the above 3 clusters

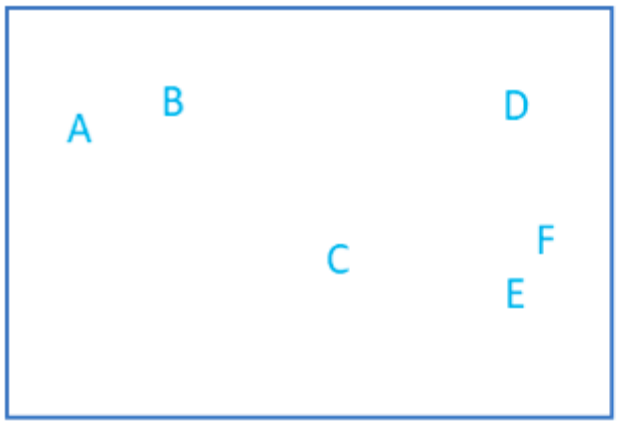
[50, 60, 70, 80], [90, 100, 110, 120, 140]

Finally,

[50, 60, 70, 80, 90, 100, 110, 120, 140]

So, based on the similarities of the data points it will just keep on clustering until there is only one cluster left.

Let's understand hierarchical clustering in another way.



Note the alphabets!

See the alphabet in the above box, as you see A and B are close to each so are E and F now see the Dendrogram on the right A & B and E & F are clustered first. After the first clustering, now check what is close to A & B there seems to be nothing, for E & F, D seems to be close to them now cluster [E, F, D]. Then, [E, F, D, C]. Finally, [A, B, E, F, D, C].

If you didn't understand anything till now, no problem you will understand everything after seeing the code and the graphs.

Here is the code:

```
import pandas as pd
df = pd.read_csv('Mall_Customers.csv')
df.head()
'''''
OUTPUT:

CustomerID Gender Age Annual Income (k$) Spending Score (1-100)
```

```
0 1 Male 19 15 39
1 2 Male 21 15 81
2 3 Female 20 16 6
3 4 Female 23 16 77
4 5 Female 31 17 40
"""
```

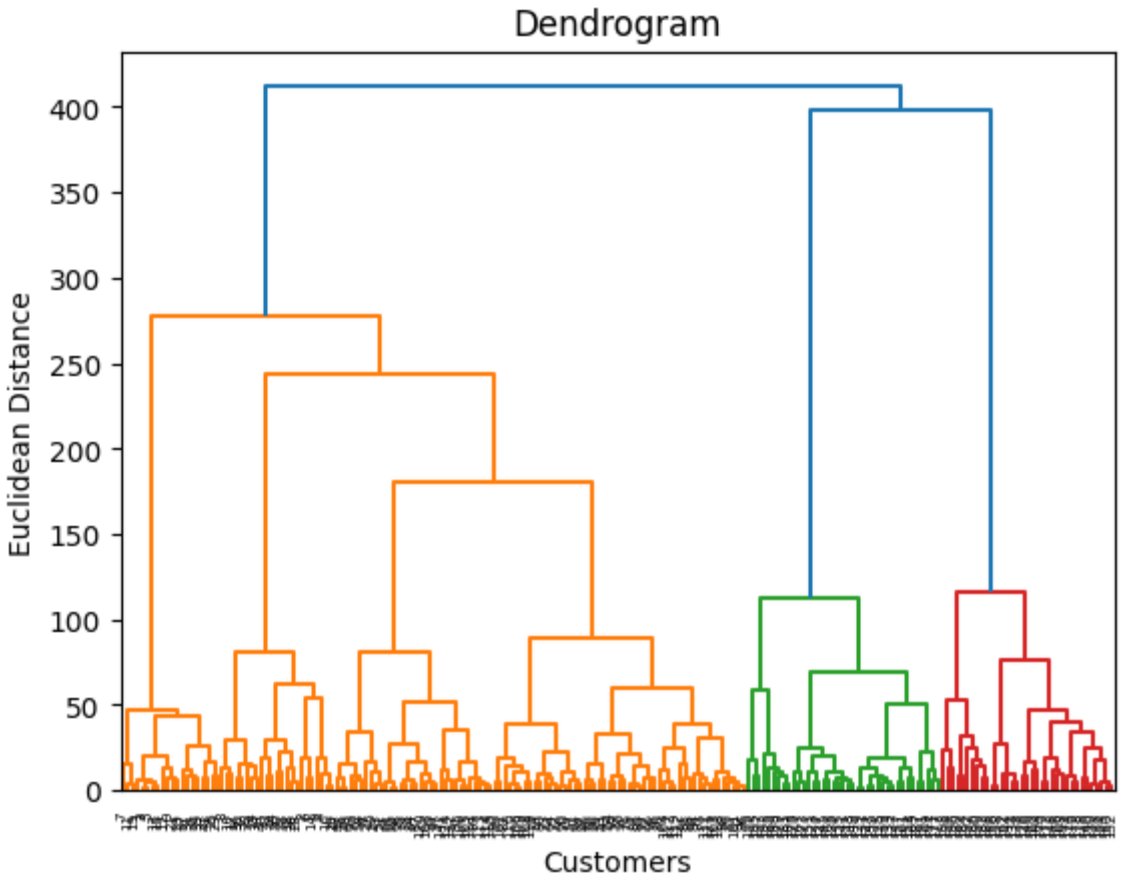
We are not going to use all the columns.

```
features = df.iloc[:, 2:5]
```

From SciPy we need to import something called 👇,

```
import scipy.cluster.hierarchy as sch
dendrogram = sch.dendrogram(sch.linkage(features, method='ward'))
plt.title("Dendrogram")
plt.xlabel("Customers")
plt.ylabel("Euclidean Distance")
plt.show()
```

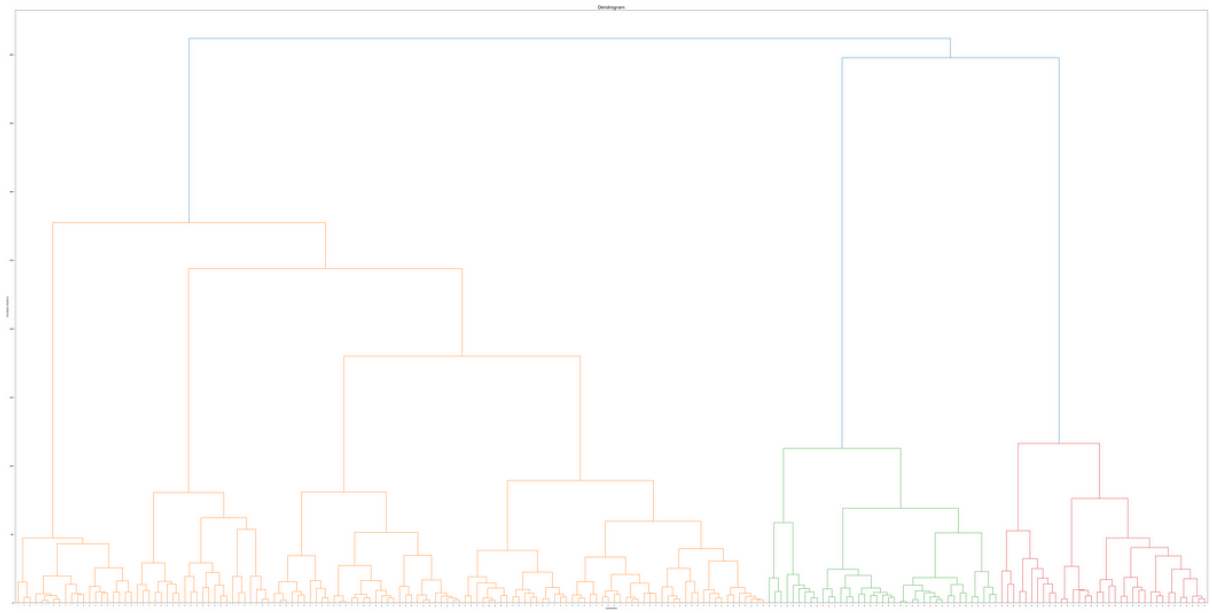
Using a dendrogram we can visually see how our dataset is being clustered. Below is the graph.



It's so messy!

If you want to see the values in the X-axis clearly, you can just increase the size of the figure.

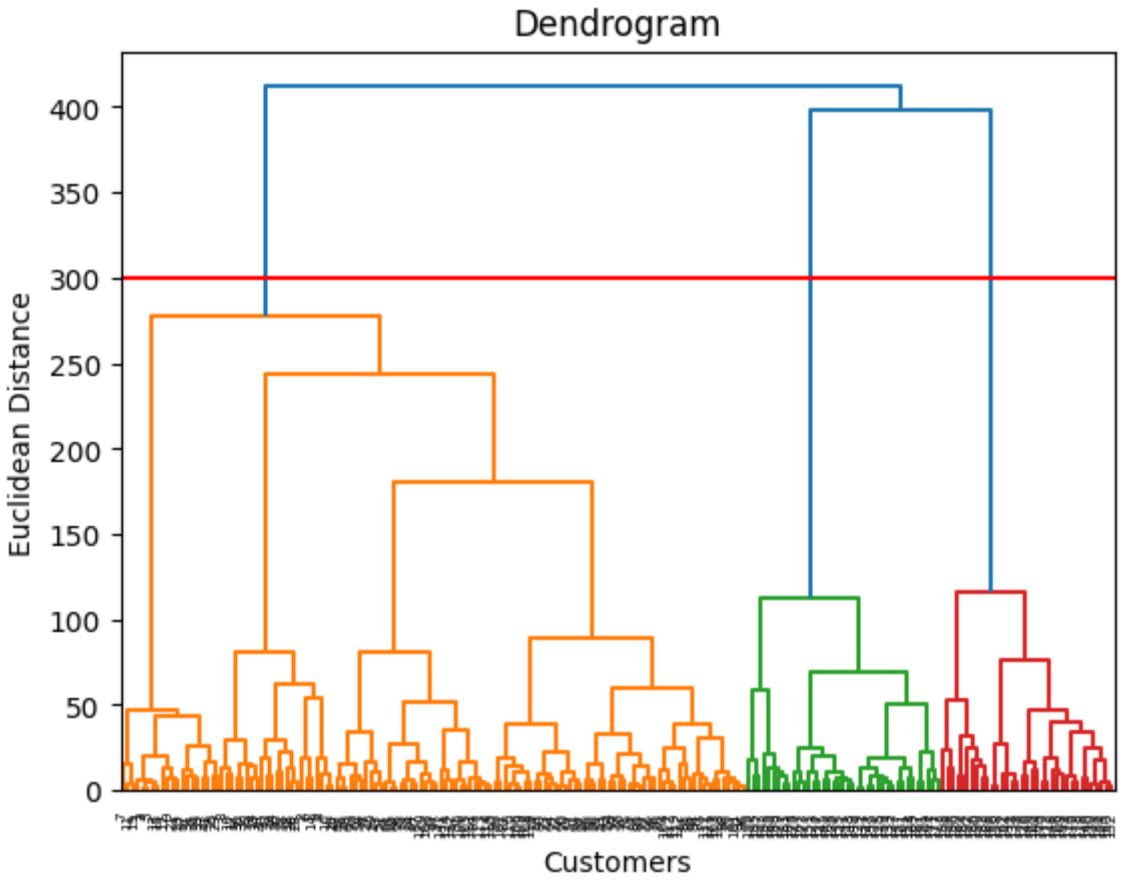
```
plt.figure(figsize=(100, 50))
dendrogram = sch.dendrogram(sch.linkage(features, method='ward'))
plt.title("Dendrogram", fontsize=20)
plt.xlabel("Customers")
plt.ylabel("Euclidean Distance")
plt.show()
```



Zoom it!

Now it's time to decide how many clusters we want. If I want 3 clusters I will draw a horizontal line from 300 in the Y-axis. If I want 2 clusters I will draw a horizontal line from 400 in the Y-axis. See the below code to understand clearly.

```
dendrogram = sch.dendrogram(sch.linkage(features, method='ward'))
plt.axhline(y=300, color='r')
plt.title("Dendrogram")
plt.xlabel("Customers")
plt.ylabel("Euclidean Distance")
plt.show()
```



Got it?

The ***plt.axhline()*** is used to draw a horizontal line from the Y-axis. Now we have 3 clusters see the above graph, Orange, Green, and Red are the 3 Clusters.

Here is the main step. Hierarchical Clustering is available in the ***sklearn.cluster*** module and there are two variants of it, one named **Agglomerative Hierarchical Clustering** and the other named **Divisive Hierarchical Clustering**.

If we move from the bottom of the dendrogram from many clusters to one cluster it is known as **Agglomerative Hierarchical Clustering**. Inversely, If you move from the top of the dendrogram

from one cluster to many clusters we call it **Divisive Hierarchical Clustering**.

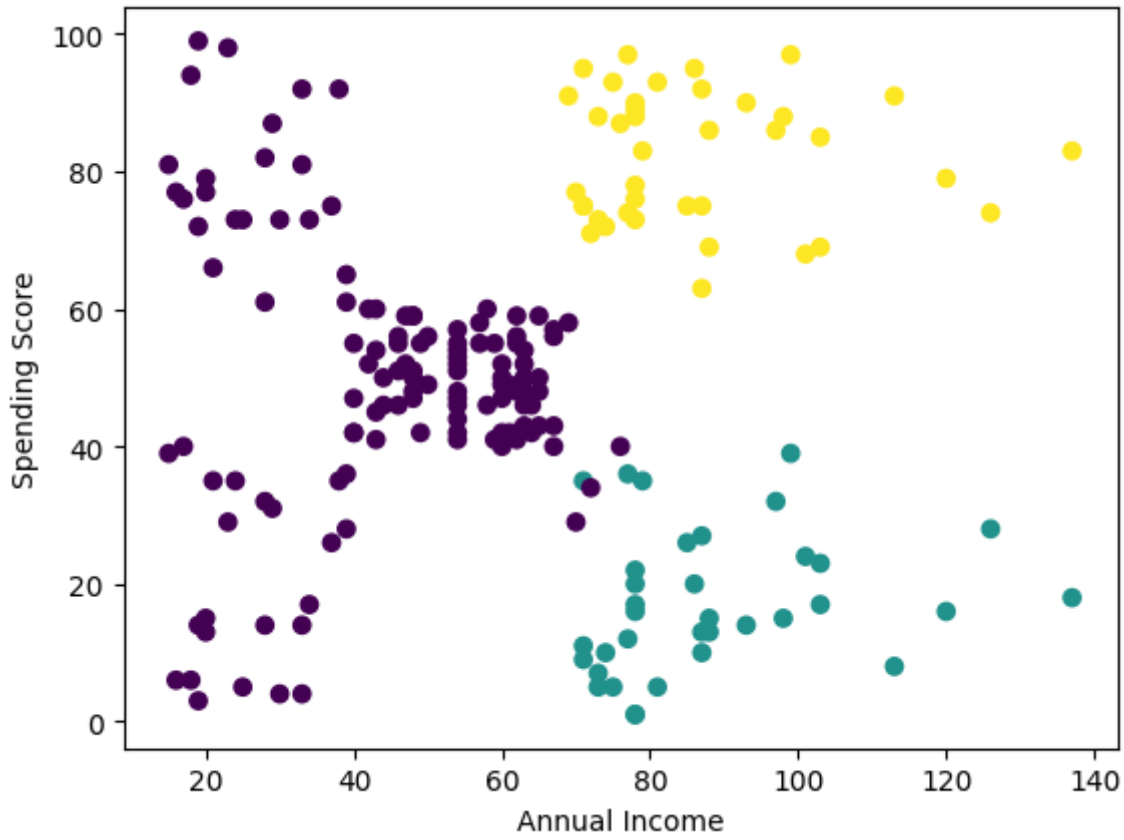
Mostly Divisive Hierarchical Clustering will not be used.

```
from sklearn.cluster import AgglomerativeClustering
cluster = AgglomerativeClustering(n_clusters=3, affinity='euclidean',
linkage='ward')
cluster.fit_predict(features)

"""
OUTPUT:
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 2, 1, 2, 1, 2, 1, 2,
       0, 2, 1, 2, 1, 2, 1, 2, 1, 2, 0, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2,
       1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2,
       1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2,
       1, 2])
"""
```

So, I am going to do 3 Clusters. I don't know much about the other parameters(affinity and linkage).

Let's plot and see our clusters

```
plt.scatter(features['Annual Income (k$)'], features['Spending Score (1-100)'], c=cluster.labels_)
plt.xlabel('Annual Income')
plt.ylabel('Spending Score')
plt.show()
```

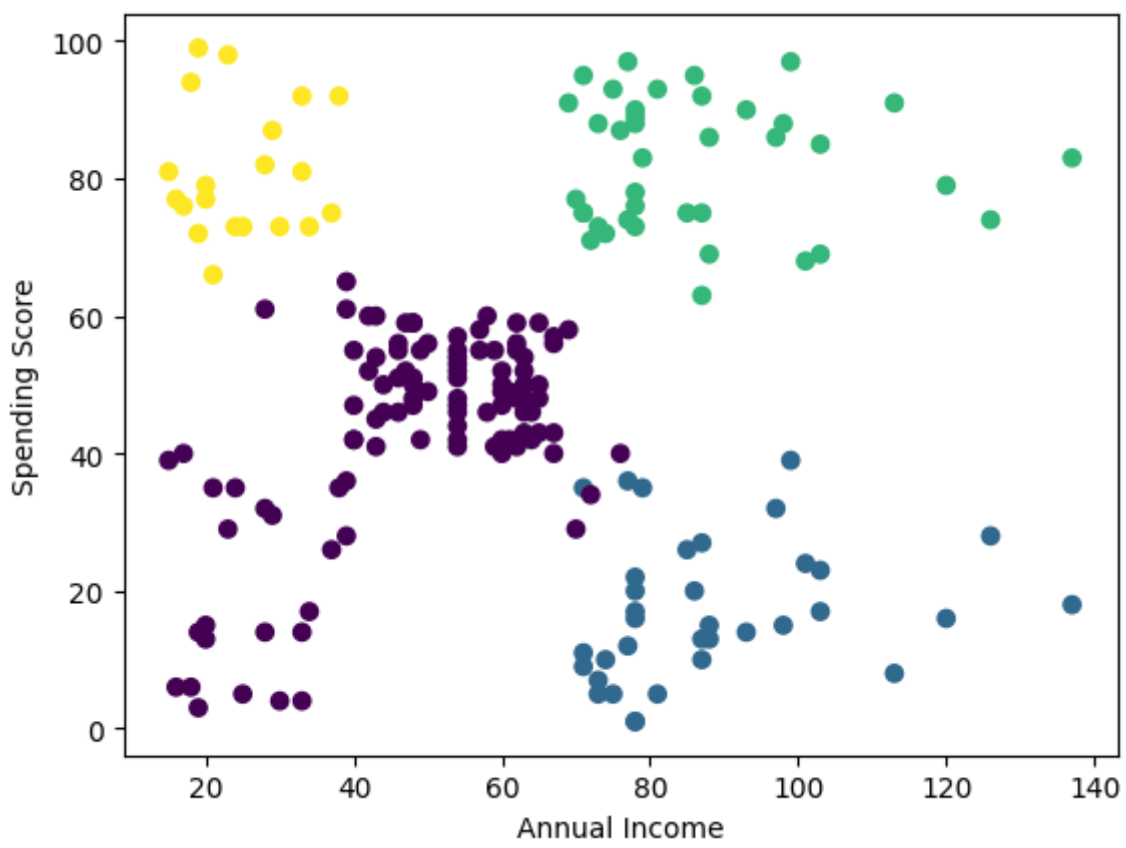


Yellow People are Rich!

From the above cluster, the yellow people have high annual income and they also have good spending scores. (Target the yellow people)

Let's try 4 clusters.

```
from sklearn.cluster import AgglomerativeClustering
cluster = AgglomerativeClustering(n_clusters=4, affinity='euclidean',
linkage='ward')
cluster.fit_predict(features)

plt.scatter(features['Annual Income (k$)'], features['Spending Score
(1-100)'], c=cluster.labels_)
plt.xlabel('Annual Income')
plt.ylabel('Spending Score')
plt.show()
```



Now it's green!

See the above graph, the green people have high incomes they also have good spending scores.

Interestingly, the yellow people above have low annual income but their spending score is high(Maybe they are kids spending their parent's money or housewives spending their husband's money).

Now after clustering our customers into 4 groups, you know whom to target.

Our target is the Yellow and Green clusters.

That's it.

Day — 25☑

Day-26: Anomaly Detection Using OneClassSVM😇😇😈😇😇

Today, I was not feeling so motivated to do anything, I wasted a lot of time, and I did nothing productive today. But I don't want to break my streak, just 5 days more to complete my challenge including this day.

So, to keep my streak going and tackle my bad mood I decide to do something really simple.

I used a different technique to do outlier detection.

In one of my previous projects, I used the **Isolation Forest** and the **Local Outlier Factor(LOF)** to detect anomalies, so this time I decided to use a different technique.

The technique I used today is OneClassSVM.

So, here is the code.

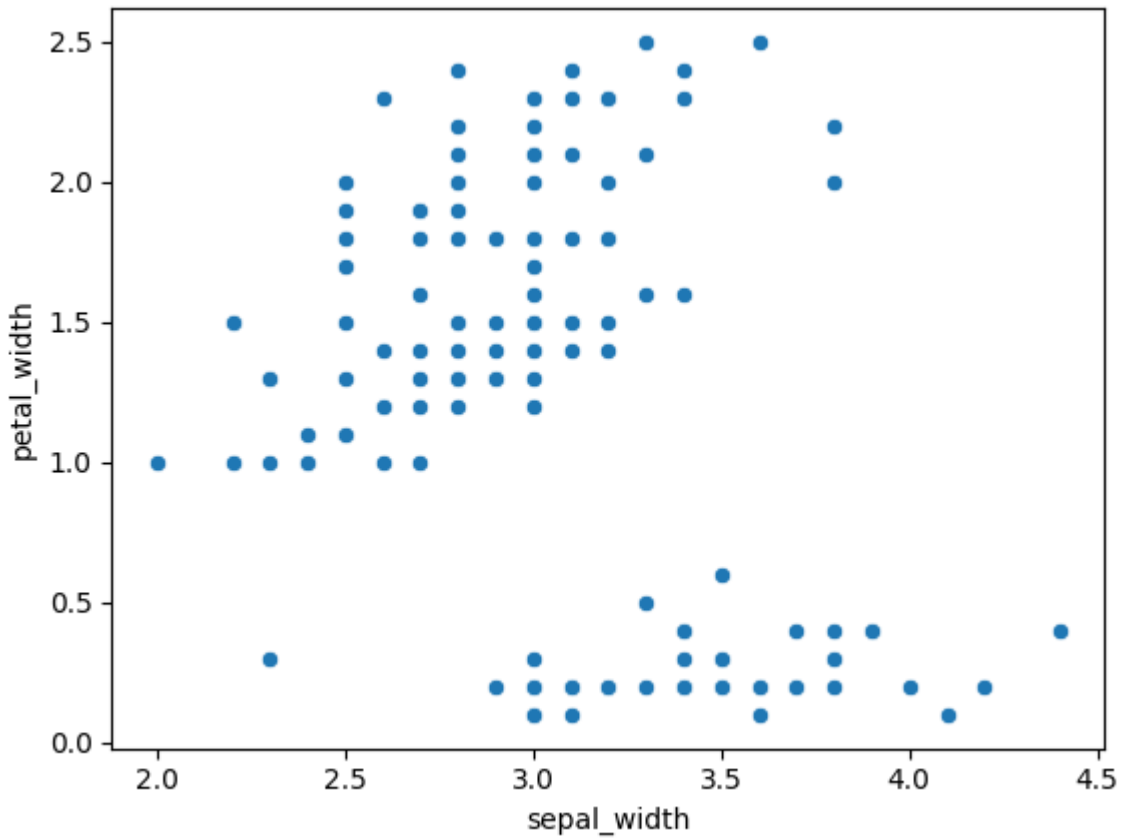
I imported Seaborn because in Seaborn there are plenty of toy datasets that we can use to learn and apply different techniques. I used the "Iris" dataset. Also, when it comes to Anomaly detection or unsupervised learning in general there will be a lot of visualizations.

```
import seaborn as sns
```

```
df = sns.load_dataset('iris')
df.head()
"""
OUTPUT:
 sepal_length sepal_width petal_length petal_width species
0 5.1 3.5 1.4 0.2 setosa
1 4.9 3.0 1.4 0.2 setosa
2 4.7 3.2 1.3 0.2 setosa
3 4.6 3.1 1.5 0.2 setosa
4 5.0 3.6 1.4 0.2 setosa
"""
```

We don't want to use all the columns in the "Iris" dataset, we just need to choose 2 columns. Let's try the ***sepal_width*** and ***petal_width*** columns, and find the outliers using the OneClassSVM.

```
sns.scatterplot(df, x=df.sepal_width, y=df.petal_width)
```



There are few outliers!

Let's import the OneClassSVM.

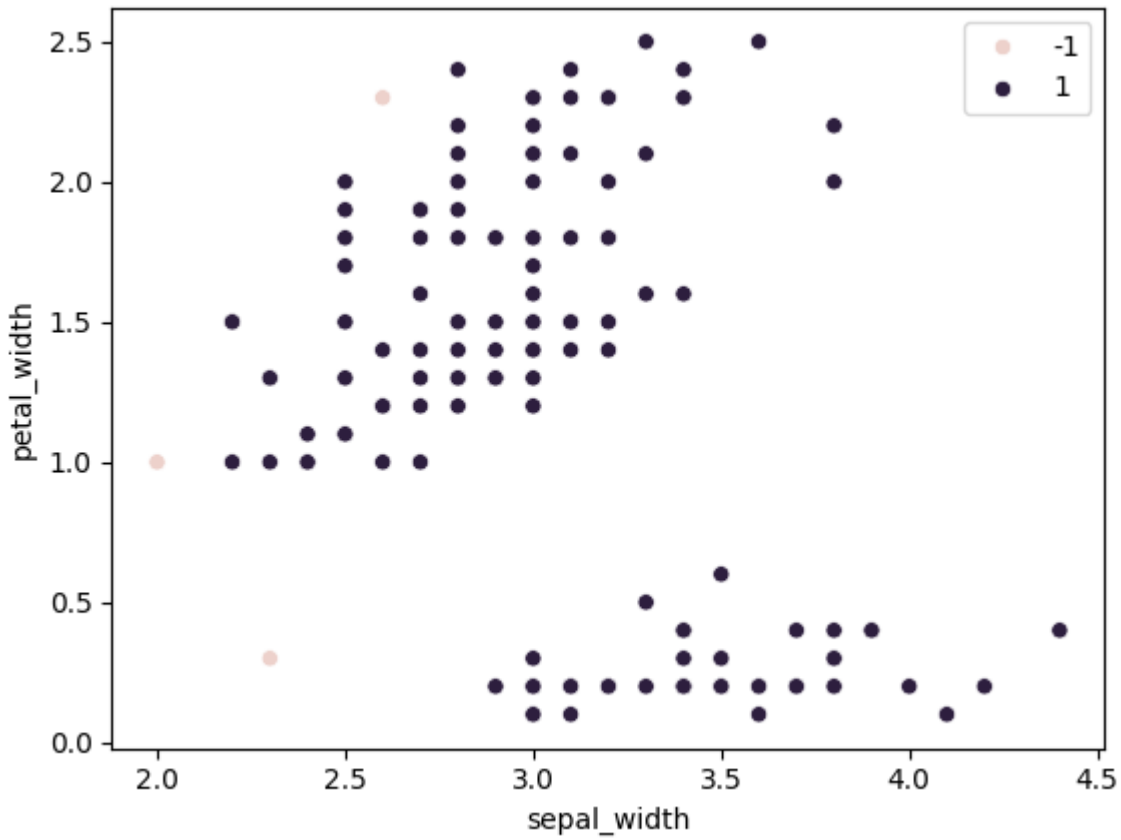
```
from sklearn.svm import OneClassSVM
ocsvm = OneClassSVM(nu=0.001)

X = df[['sepal_width', 'petal_width']]

ocsvm.fit(X)
ocsvm.predict(X)
"""
OUTPUT:
array([ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1, -1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1, -1,  1,  1,  1,  1,  1,  1,  1,
```

```
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1])
"""
```

So, after importing the OneClassSVM, I created an estimator named **ocsvm**. You can also see a parameter called **nu,** you can think of the **nu** parameter as a radio nob. If you increase the value in the nu parameter you will find more outliers if you decrease the value you will find fewer outliers.

I just want to find the outliers that are far from most of the data points. Let's visualize and see how our OneClassSVM performing.

```
sns.scatterplot(df, x=df.sepal_width, y=df.petal_width,
hue=ocsvm.predict(X))
```



Not bad!

As you can see from the above scatter plot, our estimator has found 3 outliers in our data. Play with the **nu** parameter and see how it is working.

That's it.

Day — 26☑

Day-27: Clustering Using GMM, KMeans, and DBSCAN🧜‍♂️🧜‍♂️🧜‍♀️🧜‍♀️🧜‍♂️🧜‍♂️

My understanding about Clustering is getting better day by day.

Today, after reading this article you will have a good understanding of KMeans and DBSCAN and when to use them.

Before moving on to KMeans and DBSCAN, let's know a little about Gaussian Mixture Model.

If you have been following my 30-day challenge you might have already know that I have already written an article about KMeans and DBSCAN(but this one will be different).

So, today along with those 2 topics, I learned another topic which is GMM(new for me). My goal in this challenge is to learn one new topic everyday and do a project related to that topic.

Let me first share with you the code of Gaussian Mixture. The purpose of the code is to cluster the classic "Iris" dataset. Let's see how GMM performs.

Gaussian Mixture

Import seaborn and load the "Iris" dataset.

```
import seaborn as sns
df = sns.load_dataset('iris')
df.head()
"""
OUTPUT:

sepal_length sepal_width petal_length petal_width species
0 5.1 3.5 1.4 0.2 setosa
1 4.9 3.0 1.4 0.2 setosa
2 4.7 3.2 1.3 0.2 setosa
3 4.6 3.1 1.5 0.2 setosa
4 5.0 3.6 1.4 0.2 setosa
"""
```

We got 4 features and 1 label(species columns) but our task is clustering so let's remove the label and also reduce the dimension of the dataset using PCA(Check out my previous projects you find PCA).

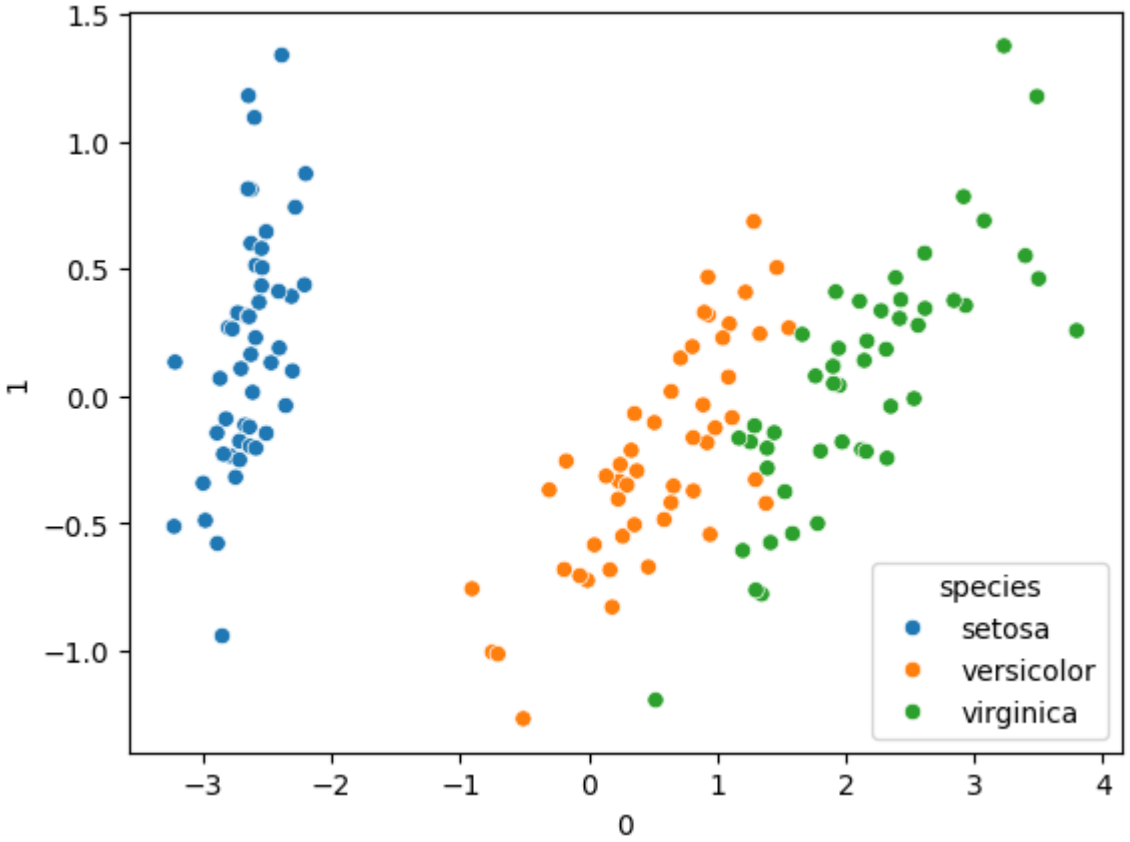
```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X = df.drop(columns='species')
features = pca.fit_transform(X)
```

Now we have dimensionality reduced "Iris" dataset named **features** with just 2 dimensions(features).

After performing **PCA** the data inside the variable **features** will be a NumPy array but in order to plot our data using Seaborn we need to

convert the NumPy array into a Pandas Data Frame, let's do that and then plot the data points in the variable **features**.

```
import pandas as pd
features_df = pd.DataFrame(features, columns=['0', '1'])
sns.scatterplot(features_df, x=features_df['0'], y=features_df['1'],
hue=df.species)
```



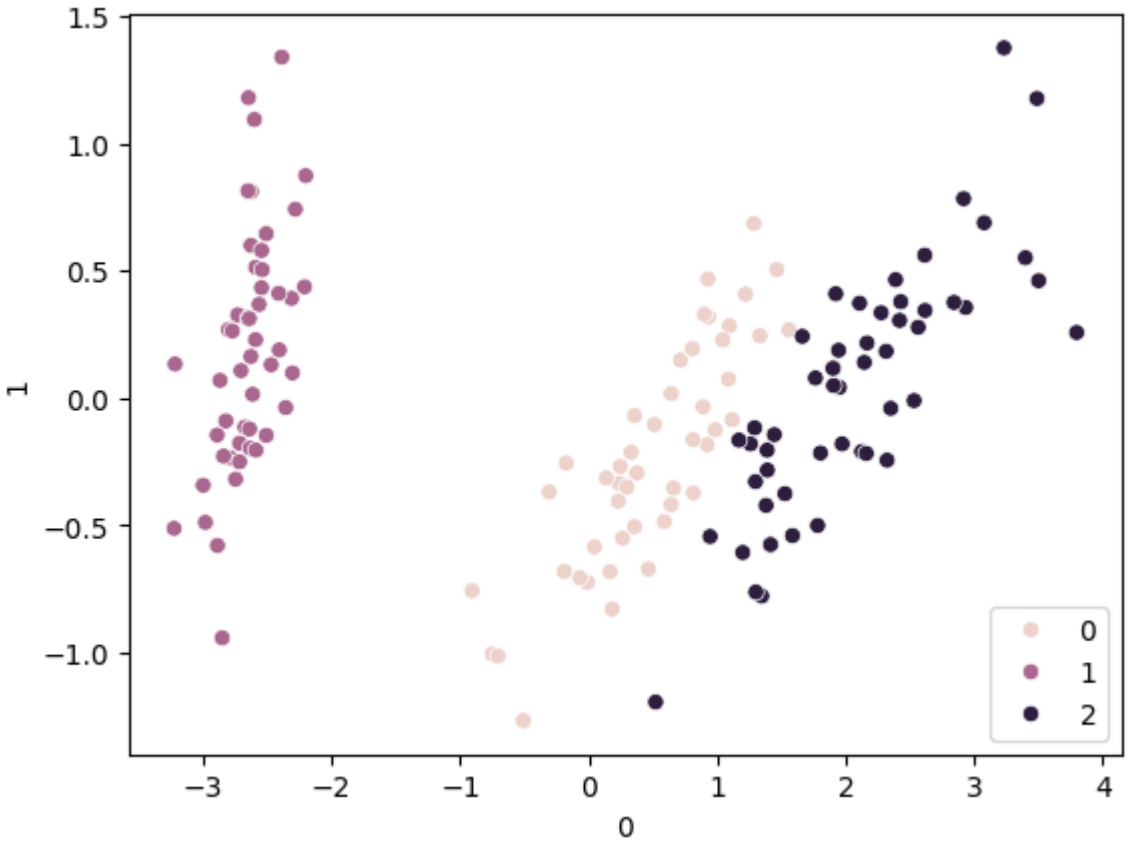
3 Clusters!

The above scatter plot shows us the original clusters, let's create a Gaussian Mixture estimator and see how well it can cluster. If the clusters made by our Gaussian Mixture estimator is similar to the one above then our estimator is performing well if not it is not performing well.

```
from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=3)
gmm.fit(features_df)
gmm_cluster = gmm.predict(features_df)
```

The clusters made by the GMM is stored in the ***gmm_cluster*** variable, now let's pass that in the **hue** parameter.

```
sns.scatterplot(features_df, x=features_df['0'], y=features_df['1'], hue=gmm_cluster)
```

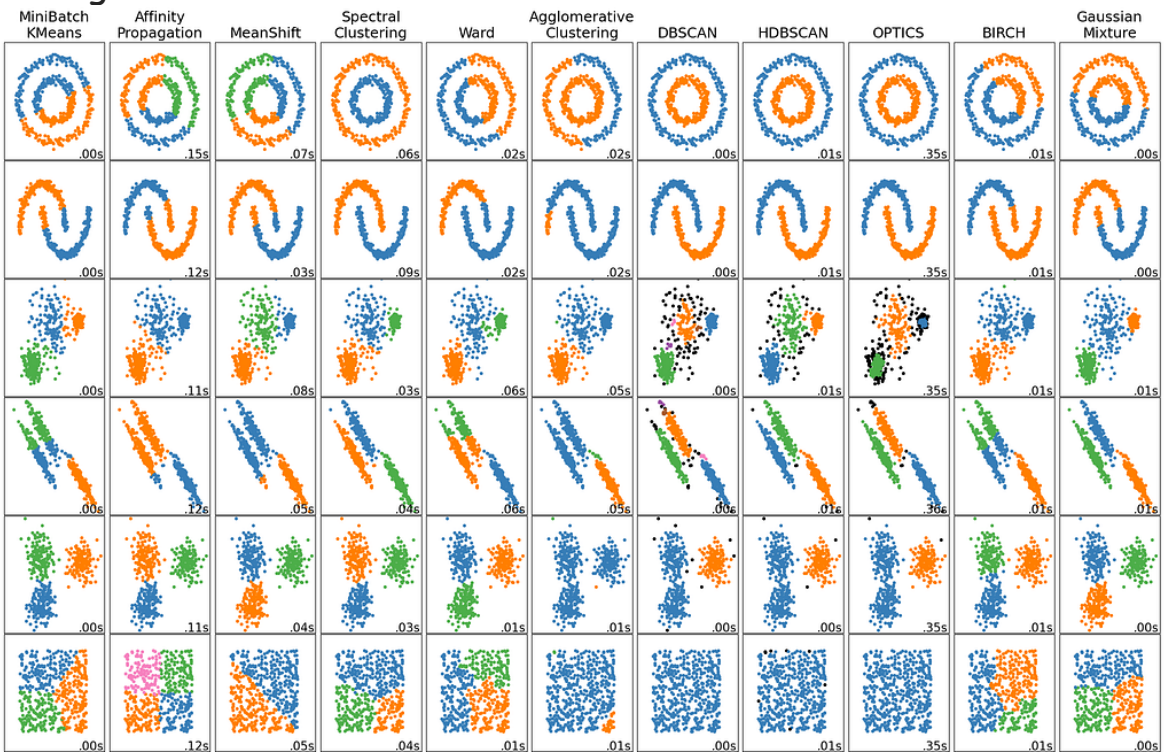


Great!

Compare the above scatter plot with the original scatter plot there is not that much of a difference. Our Gaussian Mixture Model is performing well.

Note: Gaussian Mixture is not only used for clustering, we can also use it for anomaly detection, density estimation, feature extraction, and data generation.

Now, here is the interesting part. See the below picture before reading ahead.



Source: Scikit-Learn Documentation

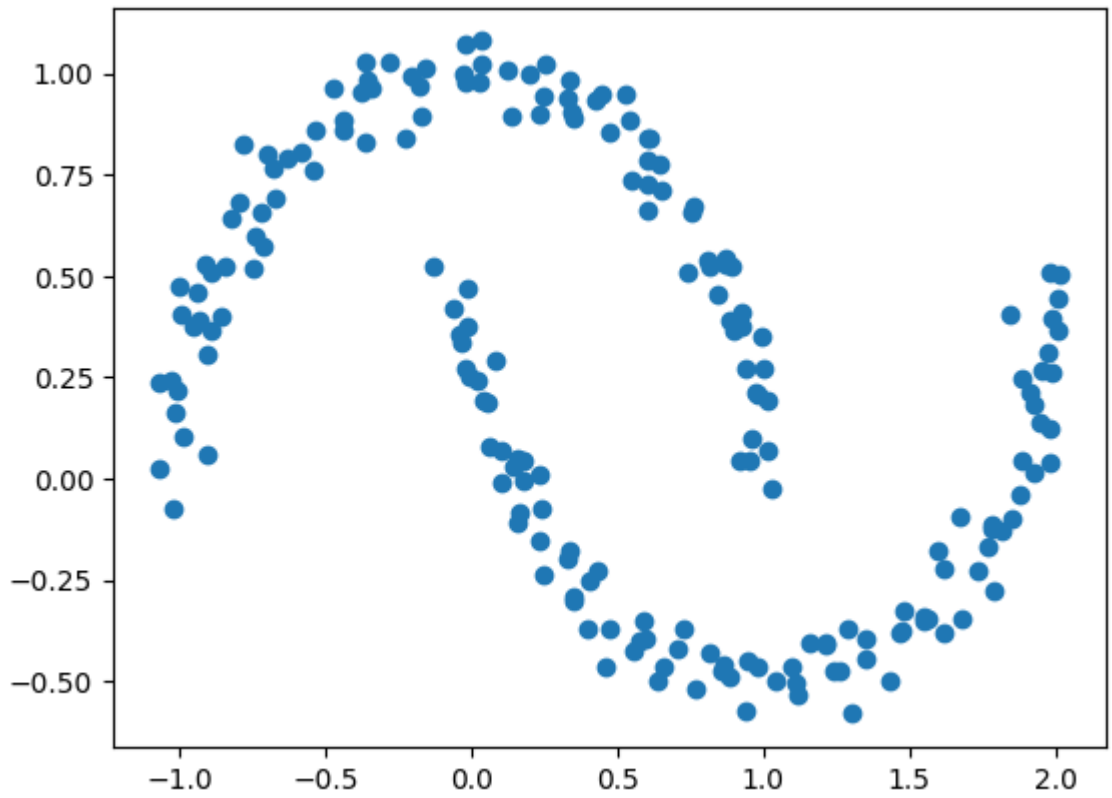
The above picture tell us when to use which clustering technique. In this article however we are only going to focus on the Horse-Shoe🧲 shaped graph in the second row.

The right technique to cluster data points which are distributed like horse shoe is DBSCAN. We can not use KMeans for this type of distribution.

Let's understand the above statement practically.

Before that we need to create some data points which looks like horse shoe, to do that use the below code. (The ***make_moons*** dataset with noise=0.05 gives us a horse shoe shaped data point distribution).

```
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt
Xmoon, ymoon = make_moons(200, noise=0.05, random_state=0)
plt.scatter(Xmoon[:, 0], Xmoon[:, 1]);
```

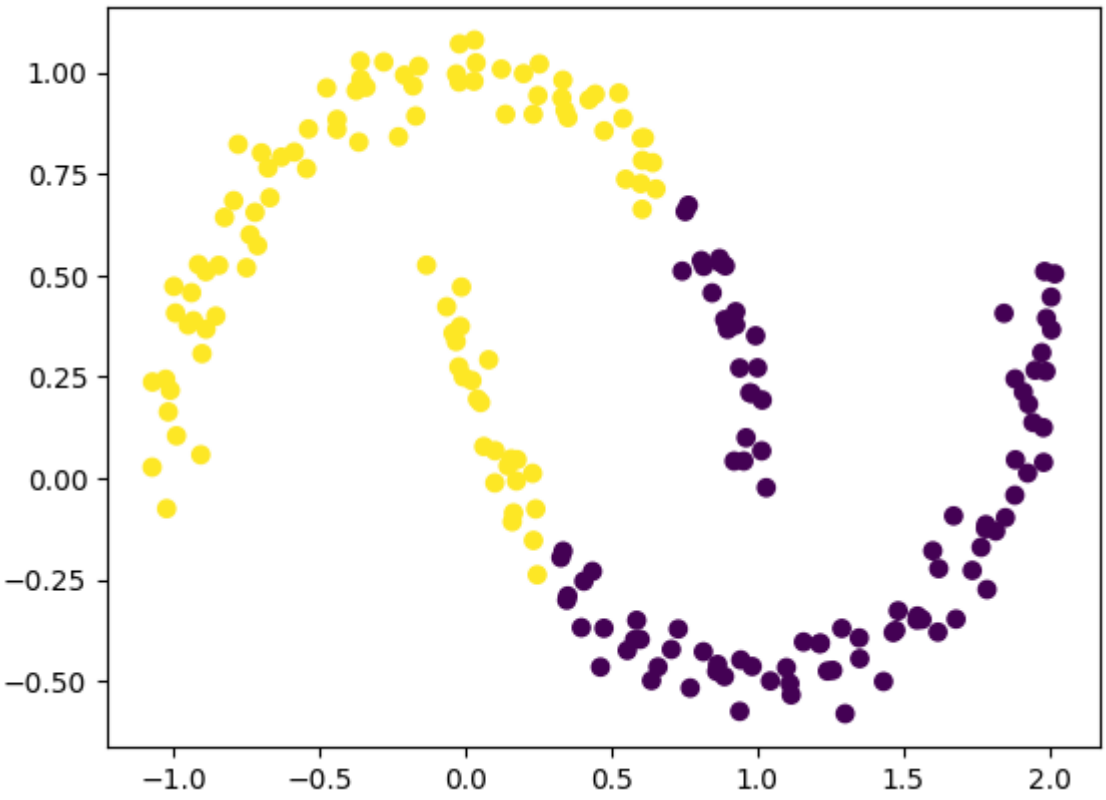


Nice!

Now, let's try KMeans to cluster the above data points.

KMeans

```
from sklearn.cluster import KMeans, DBSCAN
kmeans = KMeans(n_clusters=2, random_state=786)
kmeans.fit(Xmoon)
kmeans_clusters = kmeans.predict(Xmoon)
plt.scatter(Xmoon[:, 0], Xmoon[:, 1], c=kmeans_clusters, cmap='viridis')
```

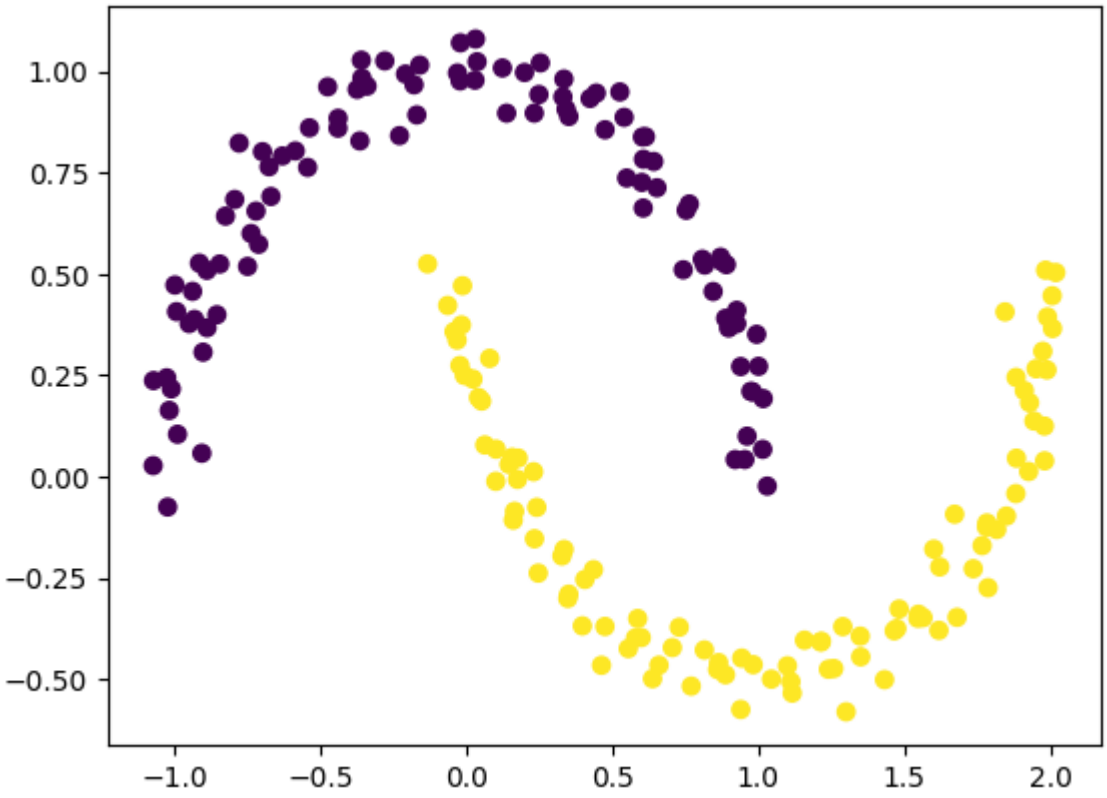


Nah!

Let's try DBSCAN now.

DBSCAN

```
dbscan = DBSCAN(eps=0.2, min_samples=5)
dbscan.fit(Xmoon)
dbscan_clusters = dbscan.labels_
plt.scatter(Xmoon[:, 0], Xmoon[:, 1], c=dbscan_clusters, cmap='viridis')
```



Great!

So, now you know when to use KMeans and when to use DBSCAN.

That's it.

Day — 27☑

Day-28: Association Rule Learning Using FP-Growth 🍞 🧈 🥛

Today's topic is **FP-Growth** Algorithm (Frequent Patterns Growth Algorithm).

In unsupervised learning, there are two important topics, one is **clustering** and another one is **association**.

Most people know about clustering but very few people know about association. Let me first give you a brief explanation of what it is.

Imagine you own a supermarket and in your database, you have some data about people's buying patterns. For example,

Customer-1: [Milk, Bread, Butter, Curd]

Customer-2: [Bread, Butter, Banana, Onion]

Customer-3: [Ice Cream, Chocolate, Biscuit]

...

...

and so on.

Now what is the best thing you can do with this data?

If you know people who are buying "Bread" tend to buy "Butter" most of the time, then you can place "Butter" and "Bread" together in your supermarket, which can potentially increase your sales.

(For an experiment, go to a nearby supermarket and see how products are placed. See what is beside the "Soap" counter, is there "Shampoo"? See what is beside the "Tooth Brush" counter, is there "Tooth Paste"?)

These products are placed together purposefully.

To do these placements you need to understand the buying patterns of your customers and then you need to know which two products should be kept together.

This process is known as **Market Basket Analysis**.

This is what we can achieve using association rule mining algorithms like Apriori and FP-Growth.

In this article, we are going to look at the FP-Growth algorithm.

Before moving on you need to know what is an "Itemset"

Itemset — Itemset means a group of items like ["Milk", "Butter", "Banana"] or sometimes it can be a single item like just "Milk".

As the name suggests, ***Frequent Pattern Growth.***

This algorithm will find the frequently occurring patterns in the dataset(containing customer's purchasing patterns).

You will understand this more clearly once we get into the code.

We also need to set a threshold(min_support) which will tell the algorithm to only find frequent patterns that are above the threshold. For example, if there are 10 instances in the dataset and our threshold is 0.6(60%) if pattern ["Milk", "Butter"] occurs in 6 instances in the dataset then it will be considered as frequent otherwise not.

Let's get into the code.

First, create a sample dataset like the one below.

```
dataset = [['Milk', 'Curd', 'Biscuit', 'Bread', 'Eggs', 'Banana'],
        ['Coffee', 'Curd', 'Biscuit', 'Bread', 'Eggs', 'Banana'],
        ['Milk', 'Salt', 'Bread', 'Eggs'],
        ['Milk', 'Unicorn', 'Salt', 'Bread', 'Banana'],
        ['Salt', 'Curd', 'Sugar', 'Bread', 'Ice cream', 'Eggs']]
```

Before applying the FP-Growth algorithm, we need to do some preprocessing on the above data. This can be done using something called **TransactionEncoder** available in the **mlxtend library**.

The TransactionEncoder works similarly to a OneHotEncoder. See code below to understand clearly.

```
import pandas as pd
from mlxtend.preprocessing import TransactionEncoder

te = TransactionEncoder()
te_array = te.fit_transform(dataset)

te_array
"""
OUTPUT:
array([[ True,  True,  True, False,  True,  True, False,  True, False,
        False, False],
       [ True,  True,  True,  True,  True,  True, False, False, False,
        False, False],
       [False, False,  True, False, False,  True, False,  True,  True,
        False, False],
       [ True, False,  True, False, False, False, False,  True,  True,
        False,  True],
       [False, False,  True, False,  True,  True,  True, False,  True,
         True, False]])
"""
```

After applying TransactionEncoder we will get an array of Boolean values. If it does not make sense to you now, no problem go ahead you will get it.

```
te.columns_
"""
OUTPUT:
['Banana',
 'Biscuit',
 'Bread',
 'Coffee',
 'Curd',
 'Eggs',
 'Ice cream',
 'Milk',
 'Salt',
 'Sugar',
 'Unicorn']
"""

df = pd.DataFrame(te_array, columns=te.columns_)
df
"""
OUTPUT:
 Banana Biscuit Bread Coffee Curd Eggs Ice cream Milk Salt Sugar Unicorn
0 True True True False True True False True False False False
1 True True True True True True False False False False False
2 False False True False False True False True True False False
3 True False True False False False False True True False True
4 False False True False True True True False True True False
"""
```

After converting the array into a data frame we can clearly see the purpose of the Boolean values.

Let's take "Banana" the first column. If "Banana" is present in the 0th index of the original dataset then the value will be "True" else "False".

Now, let's apply the FP-Growth Algorithm.

```
from mlxtend.frequent_patterns import fpgrowth

fpgrowth(df, min_support=0.6) #Setting the threshold to 60%

"""
OUTPUT:
 support itemsets
0 1.0 (2)
1 0.8 (5)
2 0.6 (7)
3 0.6 (4)
4 0.6 (0)
5 0.6 (8)
6 0.8 (2, 5)
7 0.6 (2, 7)
8 0.6 (4, 5)
9 0.6 (2, 4)
10 0.6 (2, 4, 5)
11 0.6 (0, 2)
12 0.6 (8, 2)
"""
```

The above output shows the index value in the “itemsets” column, let’s replace that with the original value.

```
fpgrowth(df, min_support=0.6, use_colnames=True)
"""
OUTPUT:
 support itemsets
0 1.0 (Bread)
1 0.8 (Eggs)
2 0.6 (Milk)
3 0.6 (Curd)
4 0.6 (Banana)
5 0.6 (Salt)
6 0.8 (Bread, Eggs)
7 0.6 (Bread, Milk)
8 0.6 (Eggs, Curd)
9 0.6 (Bread, Curd)
10 0.6 (Bread, Eggs, Curd)
11 0.6 (Bread, Banana)
12 0.6 (Bread, Salt)
"""
```

Now, from the above output we can see, that the item “Bread” is being purchased by all the customers.

Let’s focus on itemsets with two values.

See index 6 which says, [“Bread”, “Egg”] is being purchased by customers together 80% of the time(Maybe for bread omelet). This means we should place those two products together.

Similar to the above example, see the index 7, 8, 9, 10, 11, and 12. Products on those indexes should also be kept together for better sales.

That's it.

Day — 28☑

Day-29: Clustering Using Bayesian Gaussian Mixture 🌻 💐 🌷

Today was tough. I was not in the mood at all. But I convinced myself by saying, "Just 2 days".

As I was not in a good mood. I decided to take on a simple topic. But as I already told you I should not code on the topic that I already did. So, today I decided to code on a topic called Bayesian Gaussian Mixture.

Bayesian Gaussian Mixture is just another technique used for clustering.

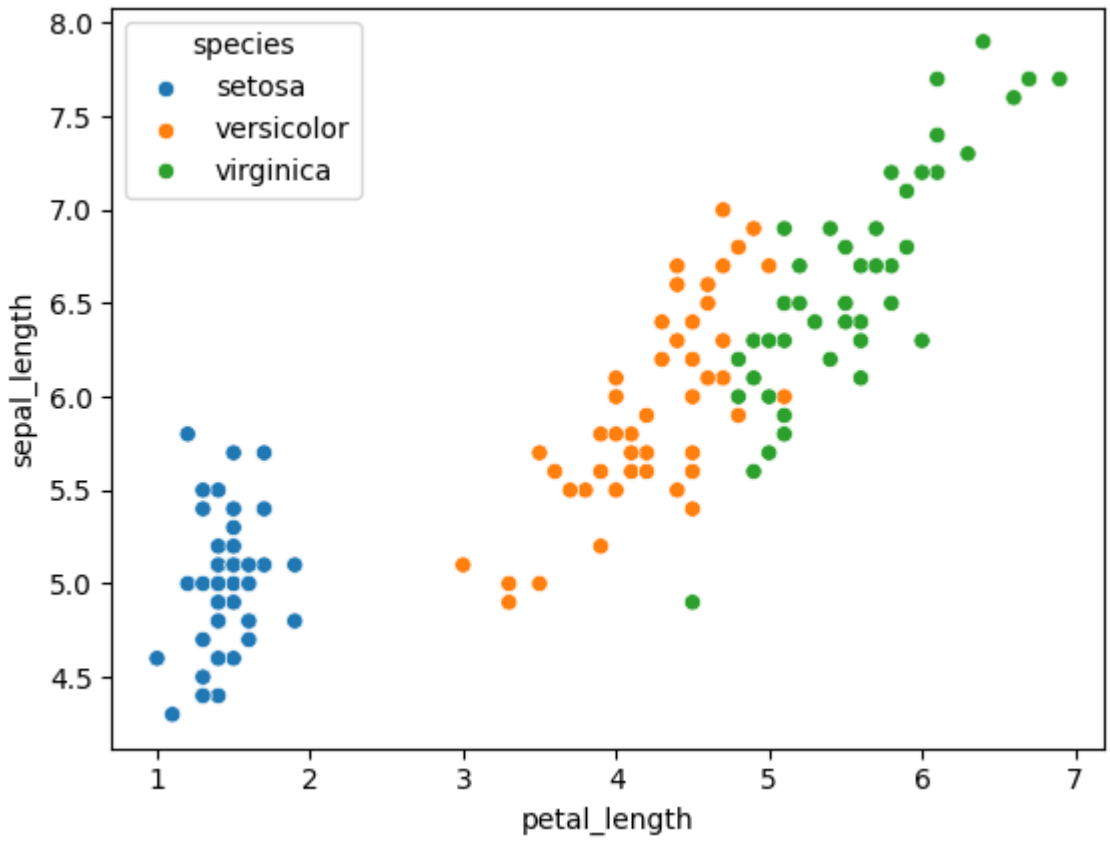
Let me get into the code straight away.

As usual, I used the "Iris" dataset again. Then I separated the dataset into features(X) and label(y).

```
import seaborn as sns

iris = sns.load_dataset('iris')
X = iris.drop(columns='species')
y = iris['species']
```

Then I took the ***petal_lenght*** and ***sepal_length*** columns and plotted the data points.

```
sns.scatterplot(iris, x='petal_length', y='sepal_length', hue='species')
```



Without wasting any time, I created the **BayesianGaussianMixture** estimator and fitted the features(X).

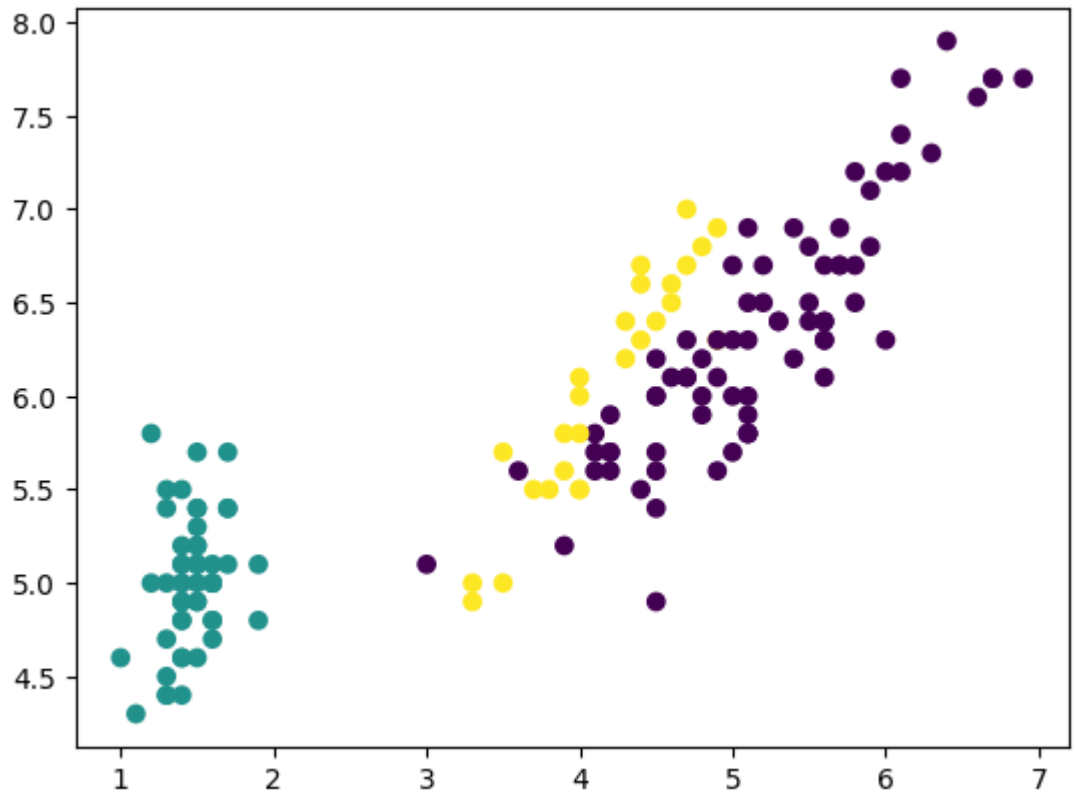
```
bgmm = BayesianGaussianMixture(n_components=3, random_state=42)
bgmm.fit(X)
y_pred = bgmm.predict(X)
```

The clusters predicted by the **BayesianGaussianMixture** are stored in the y_pred variable.

```
y_pred
"""
OUTPUT:
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 0, 0, 2, 2, 0, 2, 0, 2, 0, 0, 2,
       0, 0, 0, 2, 0, 2, 2, 0, 2, 2, 2, 0, 0, 2, 2, 2, 2, 0, 0, 0, 2, 2,
       0, 2, 0, 0, 2, 2, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
"""
```

Now, let's plot the clusters using matplotlib.

```
import matplotlib.pyplot as plt

plt.scatter(X['petal_length'], X['sepal_length'], c=y_pred,
cmap='viridis')
```



Bad Cluster!

As you can see a lot of points are clustered incorrectly. Maybe I need to learn more about this topic and the hyperparameters available to tune the estimator.

But anyway. Day 29 is done.

That's it.

Day — 29☑

Day-30: Dimensionality Reduction Using LDA and SVD ✊

Today is the last day of my Challenge.

In this article, I am gonna share with you the code to do dimensionality reduction using Linear Discriminant Analysis(LDA) and Truncated **Singular Value Decomposition**(SVD).

Let's get into the code straight away.

First, import the necessary libraries.

```
import numpy as np
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris
from sklearn.decomposition import TruncatedSVD
```

Load the "Iris" dataset.

```
# Load iris dataset as an example
iris = load_iris()
X = iris.data
y = iris.target
```

Now let's split the features and label into train and test sets.

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

After splitting, we need to Standardize our data, so let's do that using StandardScaler.

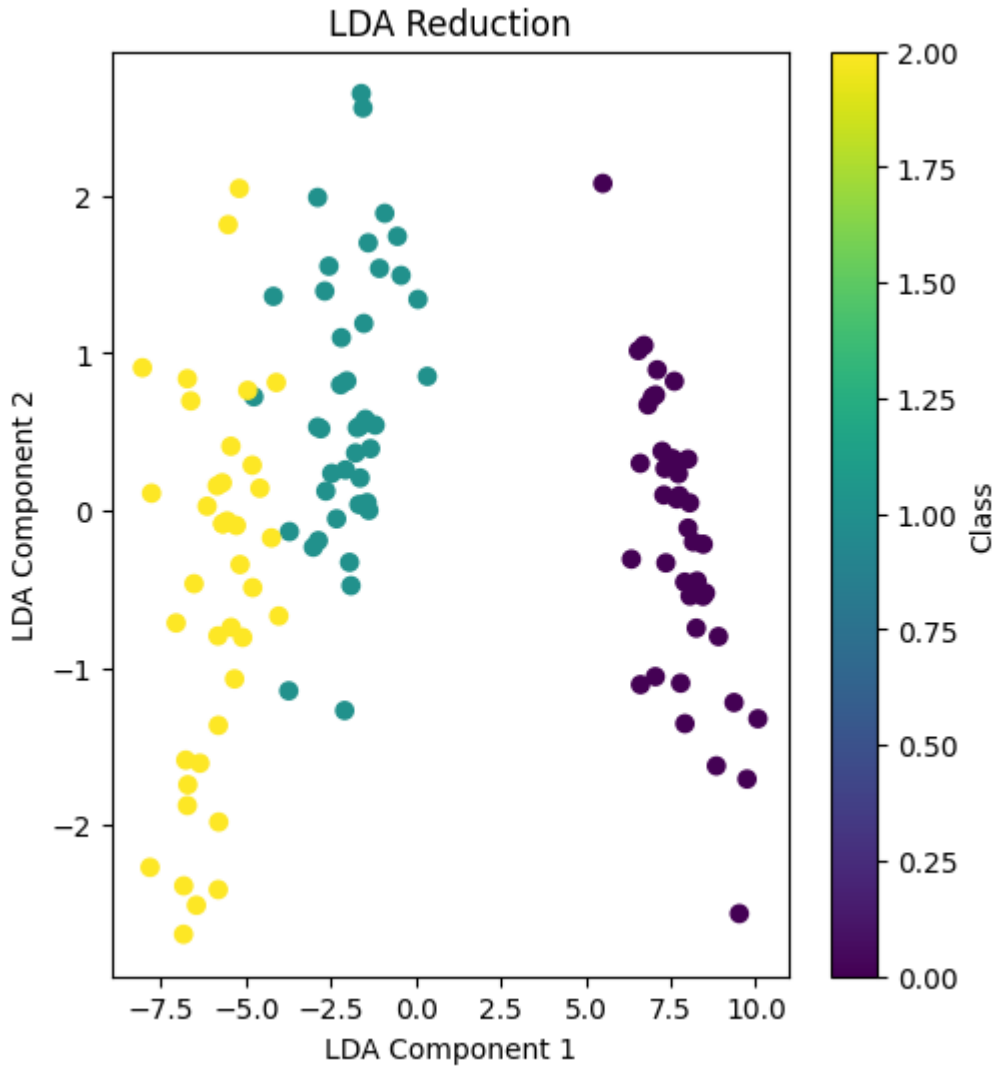
```
# Standardize the data (important for PCA)
scaler = StandardScaler()
X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.transform(X_test)
```

Now, we are good to go. Let's first try LDA.

LDA

```
# Apply LDA
lda = LinearDiscriminantAnalysis()
X_train_lda = lda.fit_transform(X_train, y_train)
import matplotlib.pyplot as plt

# Visualize LDA
plt.figure(figsize=(12, 6))

# Plot LDA
plt.subplot(1, 2, 1)
plt.scatter(X_train_lda[:, 0], X_train_lda[:, 1], c=y_train,
```

```
cmap='viridis')
plt.title('LDA Reduction')
plt.xlabel('LDA Component 1')
plt.ylabel('LDA Component 2')
plt.colorbar(label='Class')
```

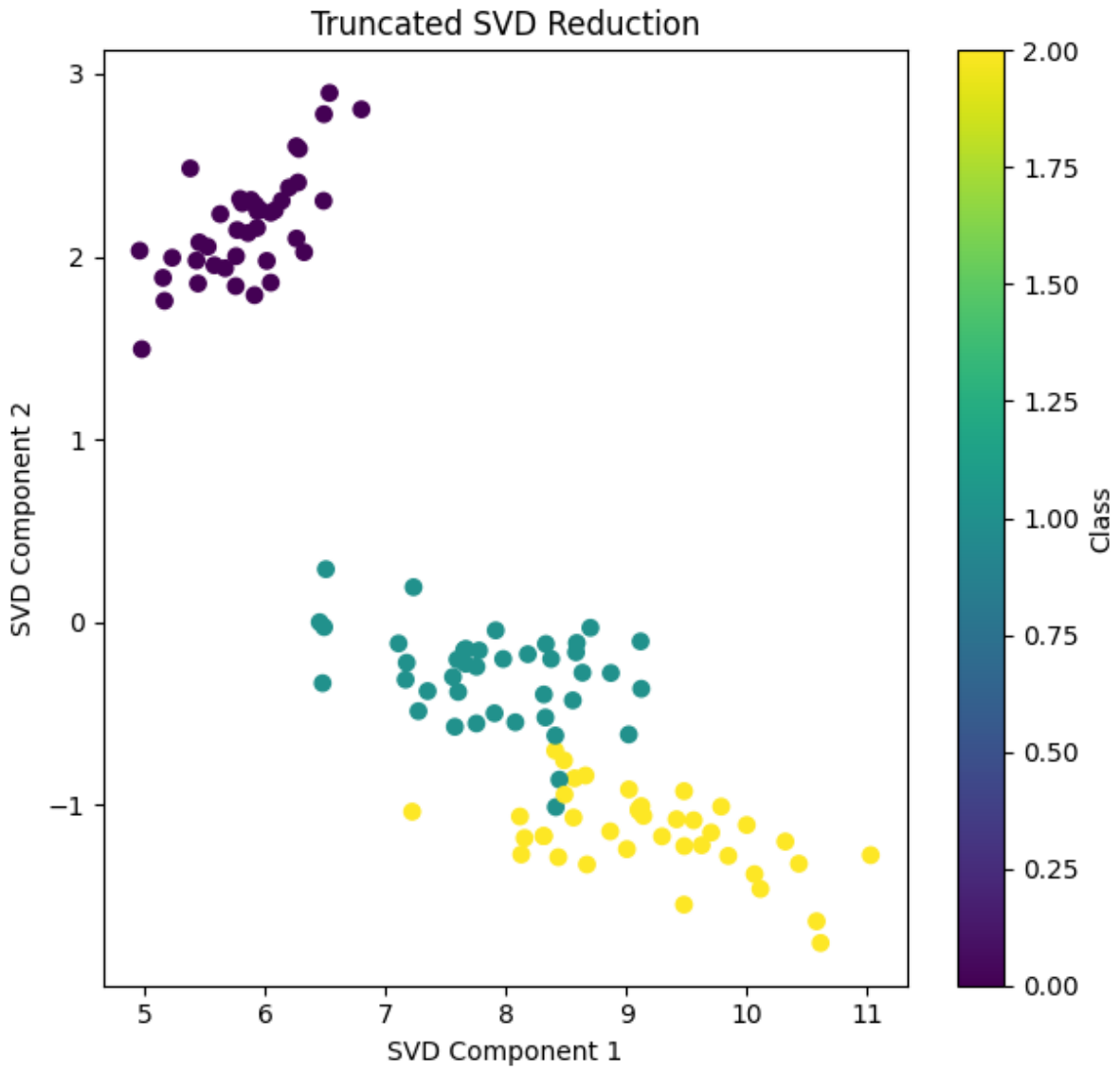


LDA

SVD

```
# Apply SVD
svd = TruncatedSVD(n_components=3)
X_train_svd = svd.fit_transform(X_train)
```

```
import matplotlib.pyplot as plt

# Visualize SVD
plt.figure(figsize=(12, 6))


# Visualize Truncated SVD
plt.subplot(1, 2, 2)
plt.scatter(X_train_svd[:, 0], X_train_svd[:, 1], c=y_train,
cmap='viridis')
plt.title('Truncated SVD Reduction')
plt.xlabel('SVD Component 1')
plt.ylabel('SVD Component 2')
plt.colorbar(label='Class')

plt.tight_layout()
plt.show()
```



SVD

That's it.

Day — 30☑

Congratulations! You did it.

Being consistent is not easy. But if you show up consistently regardless of how you feel and do the work you are supposed to do you will get what you deserve.

I hope you learned something from this book. Keep learning.

Thank you!

With love,

Abbas Ali.