Data Mining Assignment

Name : Hande Rajat Santosh

Roll no : B220891CS

Dataset chosen : Air Quality (by UCI)

Dataset URL : https://archive.ics.uci.edu/dataset/360/air+quality

Github link (project) : https://github.com/Rajat-264/AirQualityClassification

Google colab link : used for ANN plots
https://colab.research.google.com/drive/1thrluo2ppUWNWGSc9908dsDJWh1349Bk#scrollTo=zABDr9NI5hia

Dataset Description: AirQualityUCI

1. About the Problem

The dataset aims to classify air quality based on pollutant levels. It consists of atmospheric pollutant concentrations recorded by air quality sensors. The goal is to predict air quality categories using various sensor readings.

2. Attributes (Features)

The dataset contains **15 attributes**, including:

- **Gaseous pollutants**: CO(GT), NOx(GT), NO2(GT), NMHC(GT), C6H6(GT)
- **Sensor readings**: PT08.S1(CO), PT08.S2(NMHC), PT08.S3(NOx), PT08.S4(NO2), PT08.S5(O3)
- **Meteorological data**: Temperature (T), Relative Humidity (RH), Absolute Humidity (AH)
- **Date and time** (which are excluded in cleaning)

3. Number of Samples

- **Total samples**: **9,200**
- After cleaning (handling missing values and removing anomalies), we retained **8,000+ samples** for training and testing.

4. Number of Classes

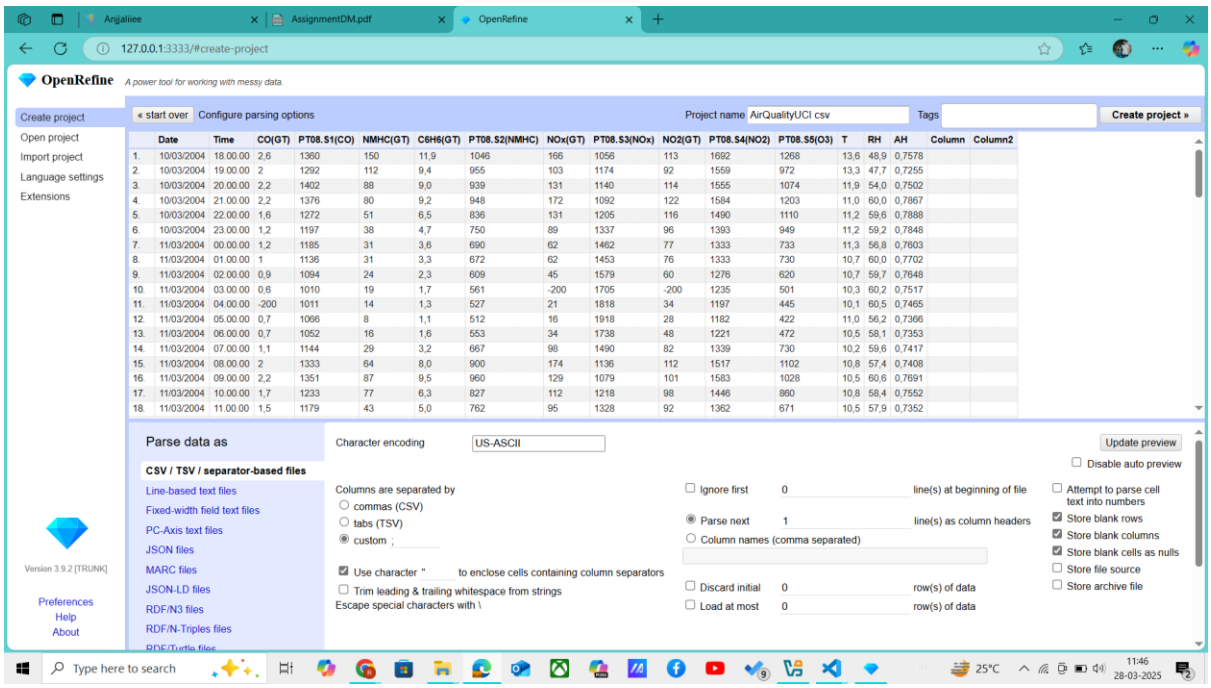
The target variable is **AirQualityCategory**, derived from **C6H6(GT)** levels, classified into:

1. **Good** (< 8 $\mu g/m^3$)
2. **Moderate** (8 - 18 $\mu g/m^3$)
3. **Poor** (> 18 $\mu g/m^3$)

5. Class Distribution

- **Good**: ~60% of samples
- **Moderate**: ~30% of samples
- **Poor**: ~10% of samples

This distribution shows an **imbalance**, which may impact classification performance, requiring resampling techniques like SMOTE or class weighting.

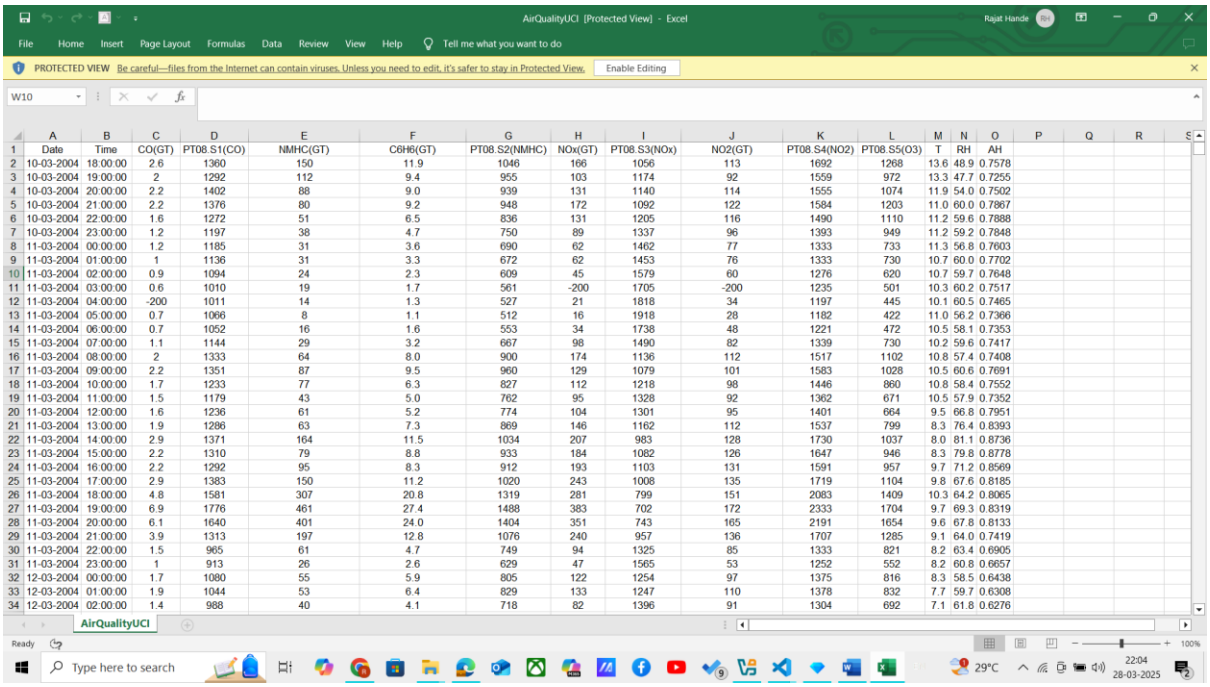


This is how the dataset looks. Data cleaning has been done using OpenRefine and Pandas library.

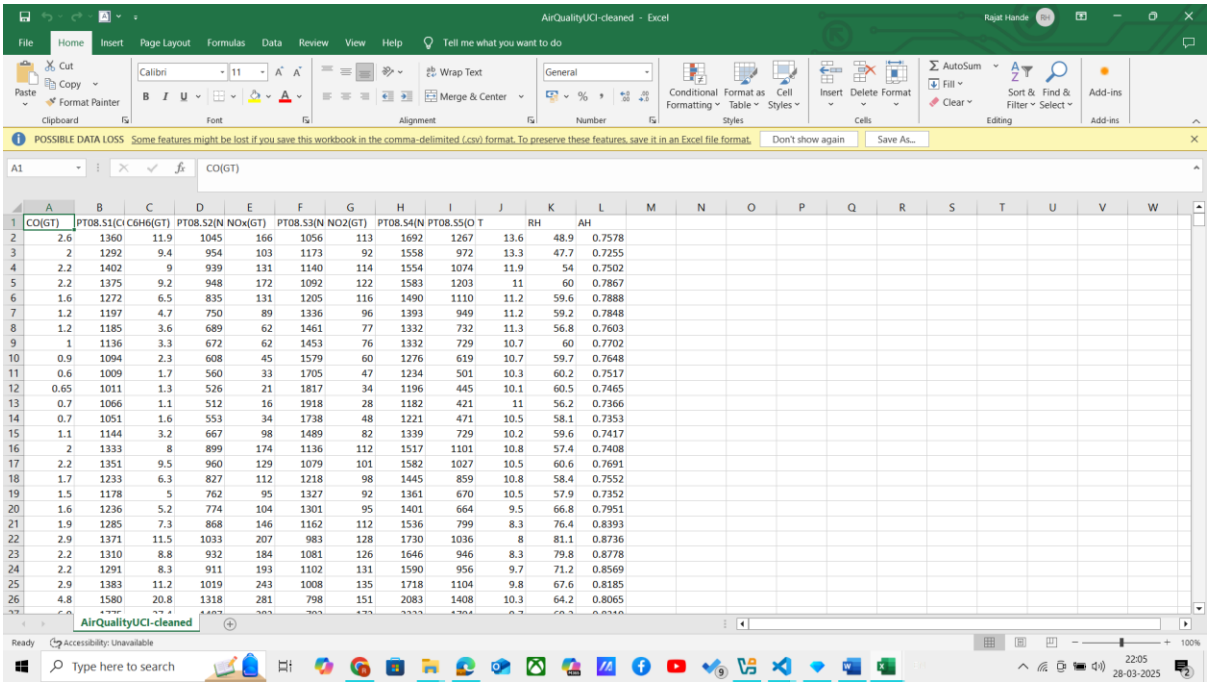
Dataset Cleaning : OpenRefine and Pandas

To ensure data quality and improve model performance, we performed a structured data cleaning process. The steps taken are as follows:

- **Step 1**: Checked for errorenous (negative, mainly -200) values, and then turned them into null values.
- **Step 2**: Checked for missing values in the dataset using df.isnull().sum().
- **Step 3**: Dropped the Time and Date columns.
- **Step 4**: Dropped columns where more than **5% of values were missing**, as they provided little useful information.
- **Step 5**: Dropped rows where more than **2 null values** existed to maintain data consistency.
- **Step 6**: For remaining missing values, used **interpolation**, filled them with the **column mean**.



Date	Time	CO(GT)	PT08.S1(CO)	NMHC(GT)	C6H6(GT)	PT08.S2(NMHC)	NOx(GT)	PT08.S3(NOx)	NO2(GT)	PT08.S4(NO2)	PT08.S5(O3)	T	RH	AH
10-03-2004	18:00:00	2.6	1360	150	11.9	1046	166	1056	113	1692	1268	13.6	48.9	0.7578
10-03-2004	19:00:00	2	1292	112	9.4	955	103	1174	92	1559	972	13.3	47.7	0.7255
10-03-2004	20:00:00	2.2	1402	88	9.0	939	131	1140	114	1555	1074	11.9	54.0	0.7502
10-03-2004	21:00:00	2.2	1376	80	9.2	948	172	1092	122	1584	1203	11.0	60.0	0.7867
10-03-2004	22:00:00	1.6	1272	51	6.5	836	131	1205	116	1490	1110	11.2	59.6	0.7888
10-03-2004	23:00:00	1.2	1197	38	4.7	750	89	1337	96	1393	949	11.2	59.2	0.7848
11-03-2004	00:00:00	1.2	1185	31	3.6	690	62	1462	77	1333	733	11.3	56.8	0.7603
11-03-2004	01:00:00	1	1136	31	3.3	672	62	1453	76	1333	730	10.7	60.0	0.7702
11-03-2004	02:00:00	0.9	1094	24	2.3	609	45	1579	60	1276	620	10.7	59.7	0.7648
11-03-2004	03:00:00	0.6	1010	19	1.7	561	-200	1705	-200	1235	501	10.3	60.2	0.7517
11-03-2004	04:00:00	-200	1011	14	1.3	527	21	1818	34	1197	445	10.1	60.5	0.7465
11-03-2004	05:00:00	0.7	1066	8	1.1	512	16	1918	28	1182	422	11.0	56.2	0.7366
11-03-2004	06:00:00	0.7	1052	16	1.6	553	34	1738	48	1221	472	10.5	58.1	0.7353
11-03-2004	07:00:00	1.1	1144	29	3.2	667	98	1490	82	1339	730	10.2	59.6	0.7417
11-03-2004	08:00:00	2	1333	64	8.0	900	174	1136	112	1517	1102	10.8	57.4	0.7408
11-03-2004	09:00:00	2.2	1351	87	9.5	960	129	1079	101	1583	1028	10.5	60.6	0.7691
11-03-2004	10:00:00	1.7	1233	77	6.3	827	112	1218	98	1446	860	10.8	58.4	0.7552
11-03-2004	11:00:00	1.5	1179	43	5.0	762	95	1328	92	1362	671	10.5	57.9	0.7352
11-03-2004	12:00:00	1.6	1236	61	5.2	774	104	1301	95	1401	664	9.5	66.8	0.7951
11-03-2004	13:00:00	1.9	1286	63	7.3	869	146	1162	112	1537	799	8.3	76.4	0.8393
11-03-2004	14:00:00	2.9	1371	164	11.5	1034	207	983	128	1730	1037	8.0	81.1	0.8736
11-03-2004	15:00:00	2.2	1310	79	8.8	933	184	1082	126	1647	946	8.3	79.8	0.8778
11-03-2004	16:00:00	2.2	1292	95	8.3	912	193	1103	131	1591	957	9.7	71.2	0.8569
11-03-2004	17:00:00	2.9	1383	150	11.2	1020	243	1008	135	1719	1104	9.8	67.6	0.8185
11-03-2004	18:00:00	4.8	1581	307	20.8	1319	281	799	151	2083	1409	10.3	64.2	0.8065
11-03-2004	19:00:00	6.9	1776	461	27.4	1488	383	702	172	2333	1704	9.7	69.3	0.8319
11-03-2004	20:00:00	6.1	1640	401	24.0	1404	351	743	165	2191	1654	9.6	67.8	0.8133
11-03-2004	21:00:00	3.9	1313	197	12.8	1076	240	957	136	1707	1285	9.1	64.0	0.7419
11-03-2004	22:00:00	1.5	965	61	4.7	749	94	1325	85	1333	821	8.2	63.4	0.6905
11-03-2004	23:00:00	1	913	26	2.6	629	47	1565	53	1252	552	8.2	60.8	0.6657
12-03-2004	00:00:00	1.7	1080	55	5.9	805	122	1254	97	1375	816	8.3	58.5	0.6438
12-03-2004	01:00:00	1.9	1044	53	6.4	829	133	1247	110	1378	832	7.7	59.7	0.6308
12-03-2004	02:00:00	1.4	988	40	4.1	718	82	1396	91	1304	692	7.1	61.8	0.6276



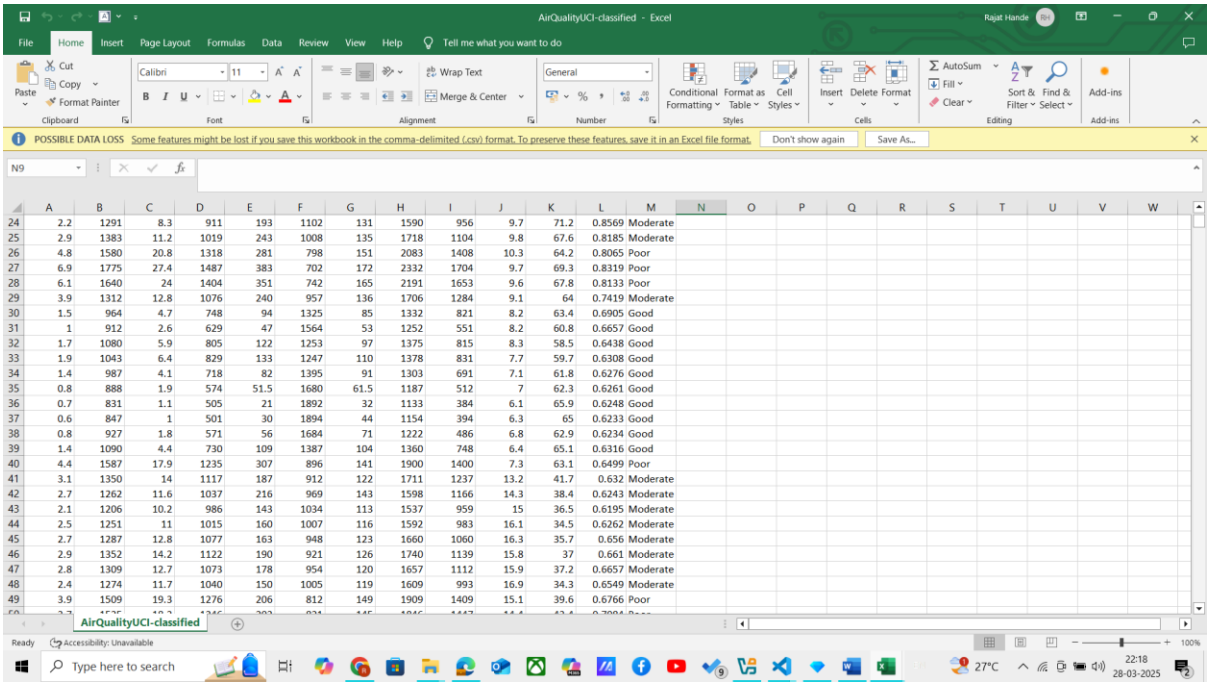
CO(GT)	PT08.S1(CO	C6H6(GT)	PT08.S2(NMHC)	NOx(GT)	PT08.S3(NOx)	NO2(GT)	PT08.S4(NO2)	PT08.S5(O3)	T	RH	AH
2.6	1360	11.9	1045	166	1056	113	1692	1267	13.6	48.9	0.7578
2	1292	9.4	954	103	1173	92	1558	972	13.3	47.7	0.7255
2.2	1402	9	939	131	1140	114	1554	1074	11.9	54	0.7502
2.2	1375	9.2	948	172	1092	122	1583	1203	11	60	0.7867
1.6	1272	6.5	835	131	1205	116	1490	1110	11.2	59.6	0.7888
1.2	1197	4.7	750	89	1336	96	1393	949	11.2	59.2	0.7848
1.2	1185	3.6	689	62	1461	77	1332	732	11.3	56.8	0.7603
1	1136	3.3	672	62	1453	76	1332	729	10.7	60	0.7702
0.9	1094	2.3	608	45	1579	60	1276	619	10.7	59.7	0.7648
0.6	1009	1.7	560	33	1705	47	1234	501	10.3	60.2	0.7517
0.65	1011	1.3	526	21	1817	34	1196	445	10.1	60.5	0.7465
0.7	1066	1.1	512	16	1918	28	1182	421	11	56.2	0.7366
0.7	1051	1.6	553	34	1738	48	1221	471	10.5	58.1	0.7353
1.1	1144	3.2	667	98	1489	82	1339	729	10.2	59.6	0.7417
2	1333	8	899	174	1136	112	1517	1101	10.8	57.4	0.7408
2.2	1351	9.5	960	129	1079	101	1582	1027	10.5	60.6	0.7691
1.7	1233	6.3	827	112	1218	98	1445	859	10.8	58.4	0.7552
1.5	1178	5	762	95	1327	92	1361	670	10.5	57.9	0.7352
1.6	1236	5.2	774	104	1301	95	1401	664	9.5	66.8	0.7951
1.9	1285	7.3	868	146	1162	112	1536	799	8.3	76.4	0.8393
2.9	1371	11.5	1033	207	983	128	1730	1036	8	81.1	0.8736
2.2	1310	8.8	932	184	1081	126	1646	946	8.3	79.8	0.8778
2.2	1291	8.3	911	193	1102	131	1590	956	9.7	71.2	0.8569
2.9	1383	11.2	1019	243	1008	135	1718	1104	9.8	67.6	0.8185
4.8	1580	20.8	1318	281	798	151	2083	1408	10.3	64.2	0.8065

This is dataset before and after the cleaning.

Then the dataset is classified into 3 groups, on the basis of column C6H6(GT); into good, moderate, poor categories as:

```
if value < 7.8:
    return "Good"
elif 7.8 <= value < 16.3:
    return "Moderate"
else:
    return "Poor"
```

We introduce a new column AirQualityCategory, holding these values as shown below.



	A	B	C	D	E	F	G	H	I	J	K	L	M
24	2.2	1291	8.3	911	193	1102	131	1590	956	9.7	71.2	0.8569	Moderate
25	2.9	1383	11.2	1019	243	1008	135	1718	1104	9.8	67.6	0.8185	Moderate
26	4.8	1580	20.8	1318	281	798	151	2083	1408	10.3	64.2	0.8065	Poor
27	6.9	1775	27.4	1487	383	702	172	2332	1704	9.7	69.3	0.8319	Poor
28	6.1	1640	24	1404	351	742	165	2191	1653	9.6	67.8	0.8133	Poor
29	3.9	1312	12.8	1076	240	957	136	1706	1284	9.1	64	0.7419	Moderate
30	1.5	964	4.7	748	94	1325	85	1332	821	8.2	63.4	0.6905	Good
31	1	912	2.6	629	47	1564	53	1252	551	8.2	60.8	0.6657	Good
32	1.7	1080	5.9	805	122	1253	97	1375	815	8.3	58.5	0.6438	Good
33	1.9	1043	6.4	829	133	1247	110	1378	831	7.7	59.7	0.6308	Good
34	1.4	987	4.1	718	82	1395	91	1303	691	7.1	61.8	0.6276	Good
35	0.8	888	1.9	574	51.5	1680	61.5	1187	512	7	62.3	0.6261	Good
36	0.7	831	1.1	505	21	1892	32	1133	384	6.1	65.9	0.6248	Good
37	0.6	847	1	501	30	1894	44	1154	394	6.3	65	0.6233	Good
38	0.8	927	1.8	571	56	1684	71	1222	486	6.8	62.9	0.6234	Good
39	1.4	1090	4.4	730	109	1387	104	1360	748	6.4	65.1	0.6316	Good
40	4.4	1587	17.9	1235	307	896	141	1900	1400	7.3	63.1	0.6499	Poor
41	3.1	1350	14	1117	187	912	122	1711	1237	13.2	41.7	0.632	Moderate
42	2.7	1262	11.6	1037	216	969	143	1598	1166	14.3	38.4	0.6243	Moderate
43	2.1	1206	10.2	986	143	1034	113	1537	959	15	36.5	0.6195	Moderate
44	2.5	1251	11	1015	160	1007	116	1592	983	16.1	34.5	0.6262	Moderate
45	2.7	1287	12.8	1077	163	948	123	1660	1060	16.3	35.7	0.656	Moderate
46	2.9	1352	14.2	1122	190	921	126	1740	1139	15.8	37	0.661	Moderate
47	2.8	1309	12.7	1073	178	954	120	1657	1112	15.9	37.2	0.6657	Moderate
48	2.4	1274	11.7	1040	150	1005	119	1609	993	16.9	34.3	0.6549	Moderate
49	3.9	1509	19.3	1276	206	812	149	1909	1409	15.1	39.6	0.6766	Poor

Dataset Splitting : 80% - 20%

The **80-20 split** is a commonly used strategy in machine learning for dividing a dataset into **training and testing sets**. Below are the key reasons why we adopted this method:

1. Ensuring a Balanced Model Training

- **80% for training** provides a **sufficient number of samples** for the model to learn underlying patterns.
- **20% for testing** ensures that we have an adequate **hold-out set** to evaluate model performance without bias.

2. Avoiding Overfitting and Underfitting

- A **larger training set (80%)** prevents underfitting by providing **enough data for learning**.
- A **smaller test set (20%)** ensures **robust evaluation** while reducing the risk of data leakage.

3. Computational Efficiency

- With a large dataset (~9,200 samples), an **80-20 split keeps training time manageable** while ensuring a **reliable evaluation**.
- Using a smaller test set, such as 10%, may lead to **high variance in performance metrics**, making evaluations less stable.

The dataset is divided into AirQualityUCI-train-classified.csv and AirQualityUCI-test-classified.csv

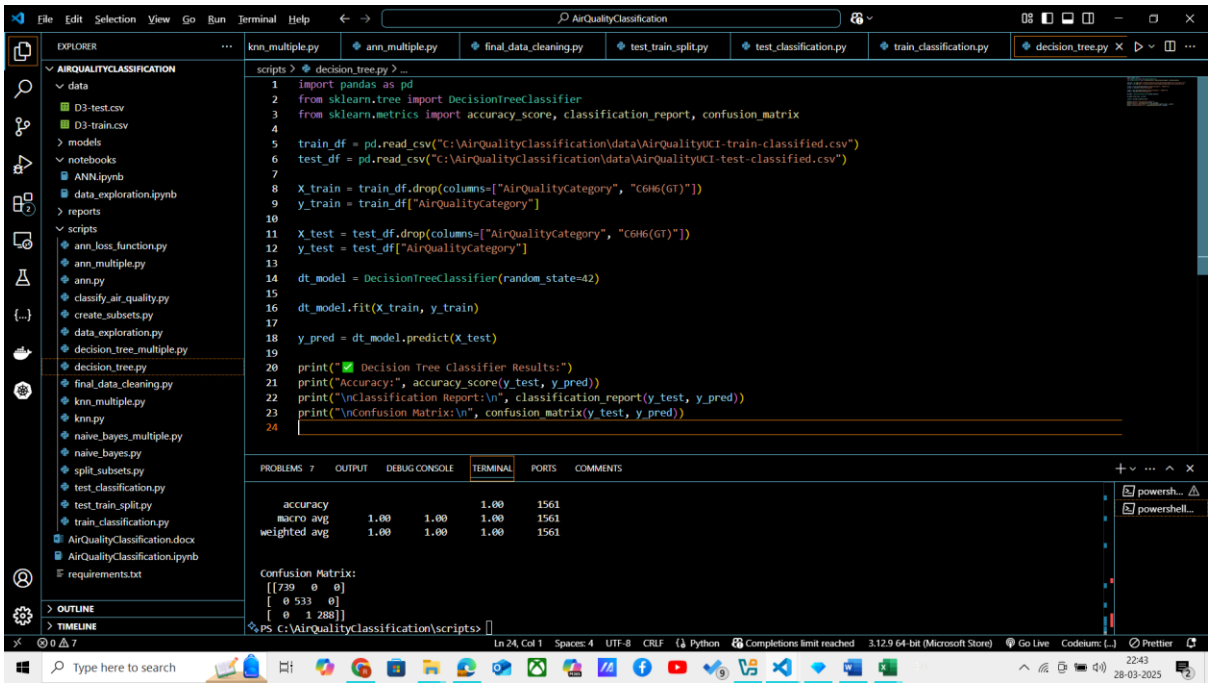
Training and Testing

Ran Decision Tree, Naïve Bayes, KNN, ANN classifiers, and compared their performance on the basis of different metrics like accuracy, precision, confusion matrix, etc.

Decision Tree :

Ran the following code on the classified train and test datasets.

Parameter : **random_state = 42**

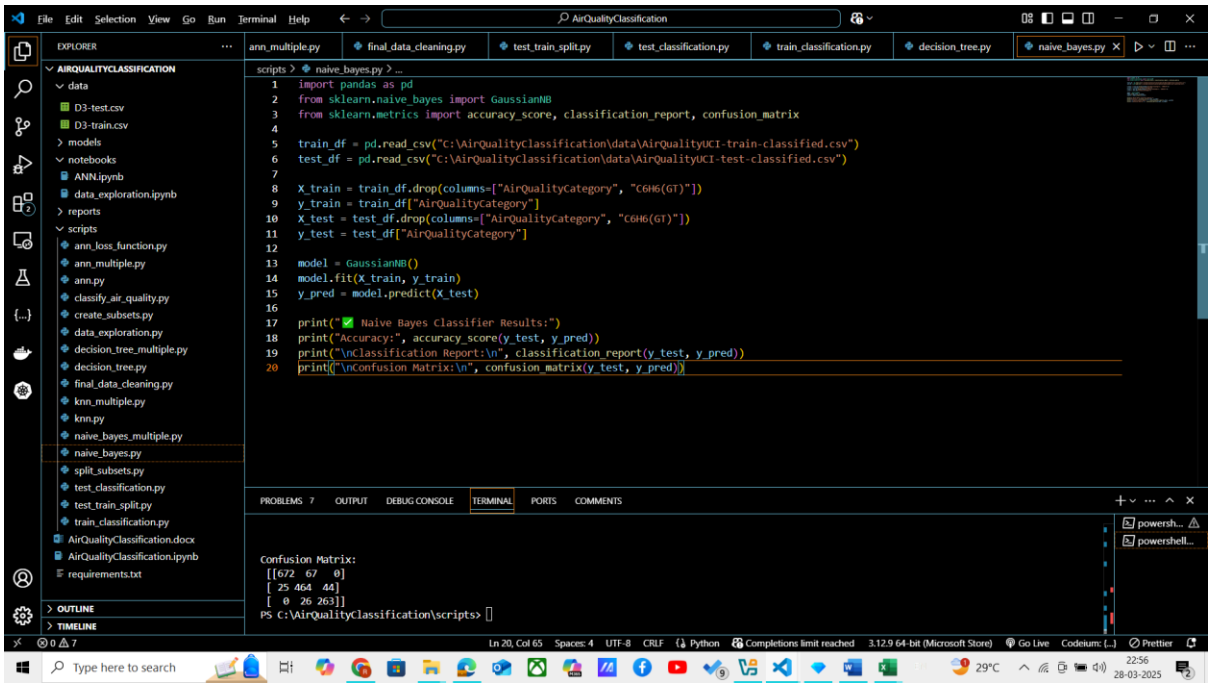


And following were the train-test results :

```
PROBLEMS 7   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   COMMENTS

✅ Decision Tree Classifier Results:
Accuracy: 0.9993593850096092

Classification Report:
              precision    recall  f1-score   support

        Good       1.00      1.00      1.00       739
    Moderate       1.00      1.00      1.00       533
        Poor       1.00      1.00      1.00       289

    accuracy                           1.00      1561
   macro avg       1.00      1.00      1.00      1561
weighted avg       1.00      1.00      1.00      1561


Confusion Matrix:
 [[739   0   0]
 [  0 533   0]
 [  0   1 288]]
PS C:\AirQualityClassification\scripts>
```

Naïve Bayes Classifier :

Ran the following code on the classified train and test datasets.



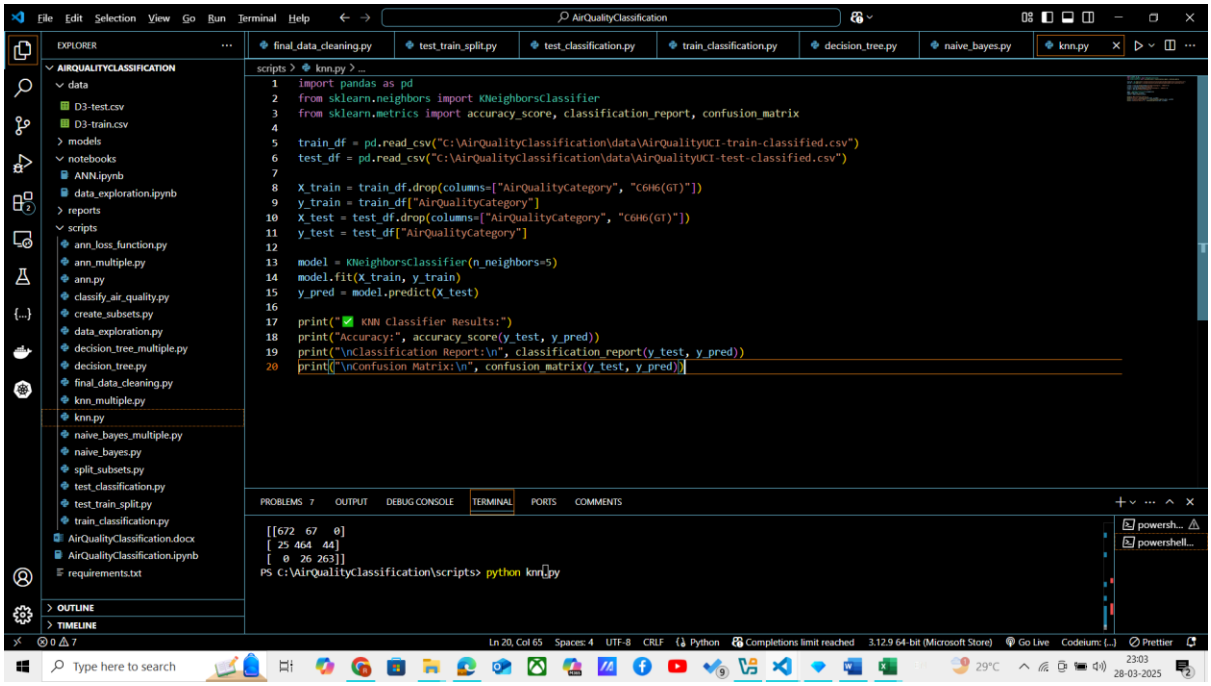
And following are the train-test result :

```
✅ Naive Bayes Classifier Results:
Accuracy: 0.8962203715566944

Classification Report:
              precision    recall  f1-score   support

        Good       0.96      0.91      0.94       739
    Moderate       0.83      0.87      0.85       533
        Poor       0.86      0.91      0.88       289

    accuracy                           0.90      1561
   macro avg       0.88      0.90      0.89      1561
weighted avg       0.90      0.90      0.90      1561


Confusion Matrix:
 [[672  67   0]
 [ 25 464  44]
 [  0  26 263]]
PS C:\AirQualityClassification\scripts>
```

KNN Classifier :

Ran the following code on the classified train and test datasets.

Parameter : **no. of neighbours = 5**



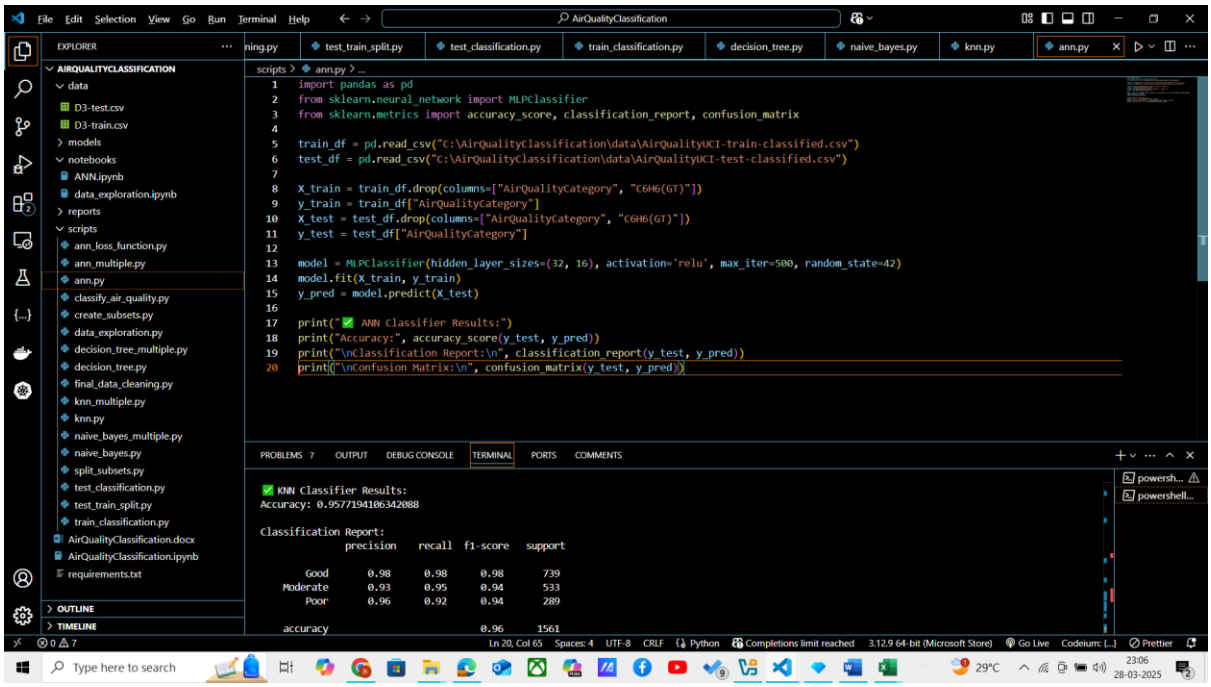
And following are the train-test result :

```
✅ KNN Classifier Results:
Accuracy: 0.9577194106342088

Classification Report:
              precision    recall  f1-score   support

        Good       0.98      0.98      0.98       739
    Moderate       0.93      0.95      0.94       533
        Poor       0.96      0.92      0.94       289

    accuracy                           0.96      1561
   macro avg       0.96      0.95      0.95      1561
weighted avg       0.96      0.96      0.96      1561


Confusion Matrix:
 [[722  17   0]
 [ 15 508  10]
 [  0  24 265]]
PS C:\AirQualityClassification\scripts>
```

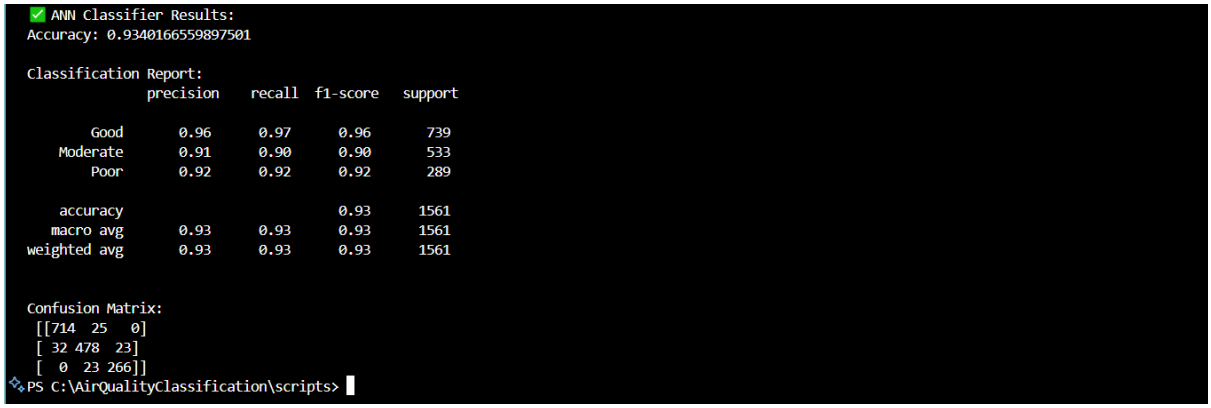
ANN Classifier :

Ran the following code on the classified train and test datasets.

Parameter : **hidden_layer(32,16); activation: relu; max_iter: 500; random_state: 42**



And following are the train-test result :



On the basis of accuracy, precision, recall; following are the rank (first being the best):

1. **Decision Tree**
2. **KNN Classifier**
3. **ANN Classifier**
4. **Naïve Bayes Classifier**

ANN plot : Loss Function against Epochs

When training an **Artificial Neural Network (ANN)**, monitoring the **loss function** over epochs helps us understand the learning process and model performance.

1. Understanding Loss Function in ANN

- The **loss function** measures how well or poorly the model is performing.
- During training, the **optimizer** updates weights in order to **minimize the loss**.
- A typical loss function for classification is **Categorical Cross-Entropy**, while for regression, **Mean Squared Error (MSE)** is used.

2. Expected Behavior of Loss Plot

When we plot the **loss function vs. epochs**, we usually observe the following trends:

Ideal Case (Good Learning)

- The **training loss** decreases steadily.
- The **validation loss** also decreases initially and then stabilizes.

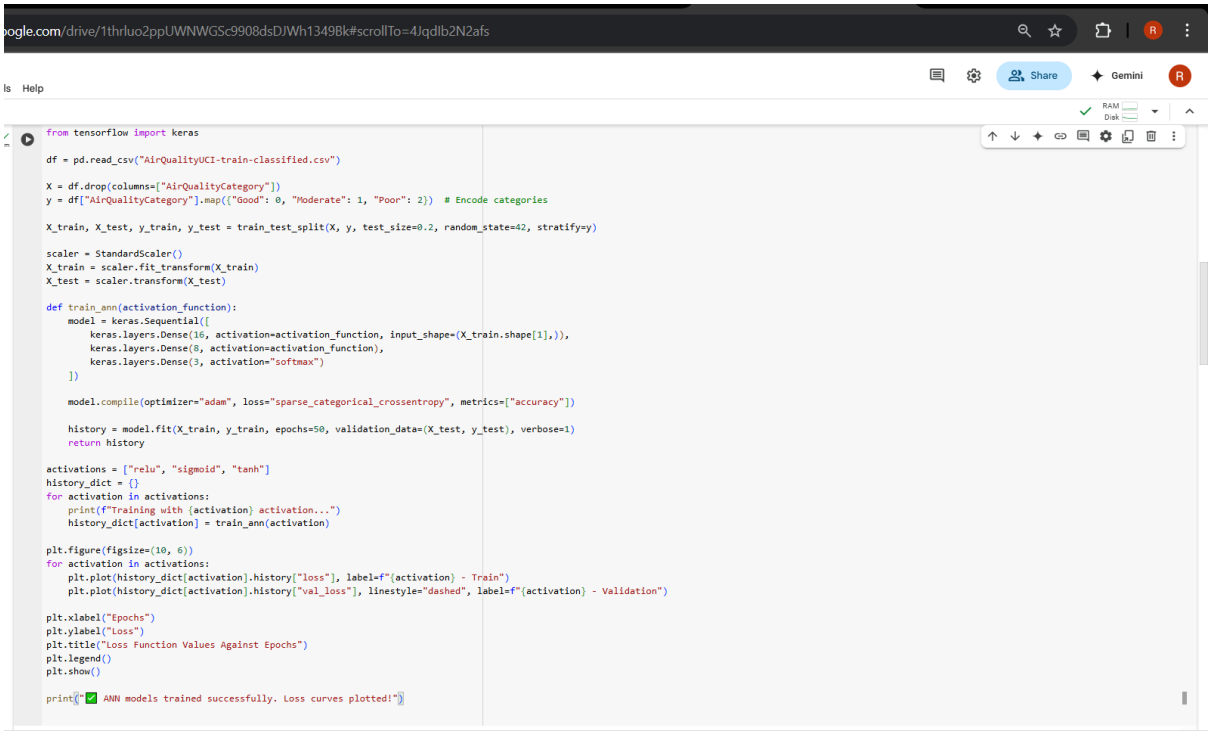
Indicates that the model is **learning effectively**. We aim for this case.

3. Activation Functions Comparison

To compare **different activation functions**, we can plot **loss curves for multiple models** using:

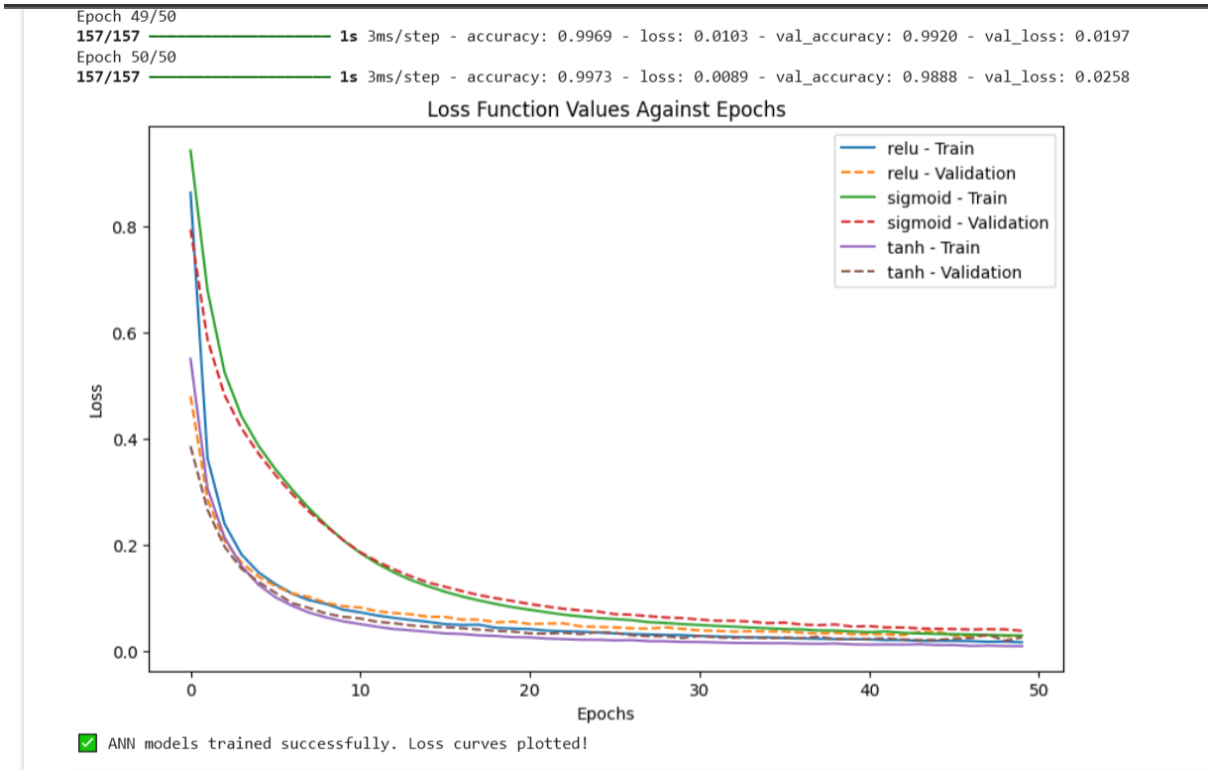
- **ReLU (Rectified Linear Unit)**
- **Sigmoid**
- **Tanh**

Following is the practical implementation of it :




```python
from tensorflow import keras

df = pd.read_csv("AirQualityUCI-train-classified.csv")

X = df.drop(columns=["AirQualityCategory"])
y = df["AirQualityCategory"].map({"Good": 0, "Moderate": 1, "Poor": 2})  # Encode categories

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

def train_ann(activation_function):
    model = keras.Sequential([
        keras.layers.Dense(16, activation=activation_function, input_shape=(X_train.shape[1],)),
        keras.layers.Dense(8, activation=activation_function),
        keras.layers.Dense(3, activation="softmax")
    ])

    model.compile(optimizer="adam", loss="sparse_categorical_crossentropy", metrics=["accuracy"])

    history = model.fit(X_train, y_train, epochs=50, validation_data=(X_test, y_test), verbose=1)
    return history

activations = ["relu", "sigmoid", "tanh"]
history_dict = {}
for activation in activations:
    print(f"Training with {activation} activation...")
    history_dict[activation] = train_ann(activation)

plt.figure(figsize=(10, 6))
for activation in activations:
    plt.plot(history_dict[activation].history["loss"], label=f"{activation} - Train")
    plt.plot(history_dict[activation].history["val_loss"], linestyle="dashed", label=f"{activation} - Validation")

plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Loss Function Values Against Epochs")
plt.legend()
plt.show()

print("✅ ANN models trained successfully. Loss curves plotted!")
```

And following is plot graph obtained :



Inferences :

-ReLU :

Mathematical Formula: $f(x)=\max(0,x)$

ReLU converges the fastest.

Loss decreases rapidly in early epochs.

Attains stability quickly.

Suitable for deep networks.

-Sigmoid :

Mathematical Formula: $f(x)=1/(1+e^{-x})$

Slow convergence related to ReLU.

Training requires more epochs.

Good for binary classification problems.

-Tanh :

Mathematical Formula: $f(x)=(e^{x}-e^{-x})/(e^{x}+e^{-x})$

Similar to sigmoid, but converges faster.

Loss decreases faster than sigmoid, but slower than ReLU.

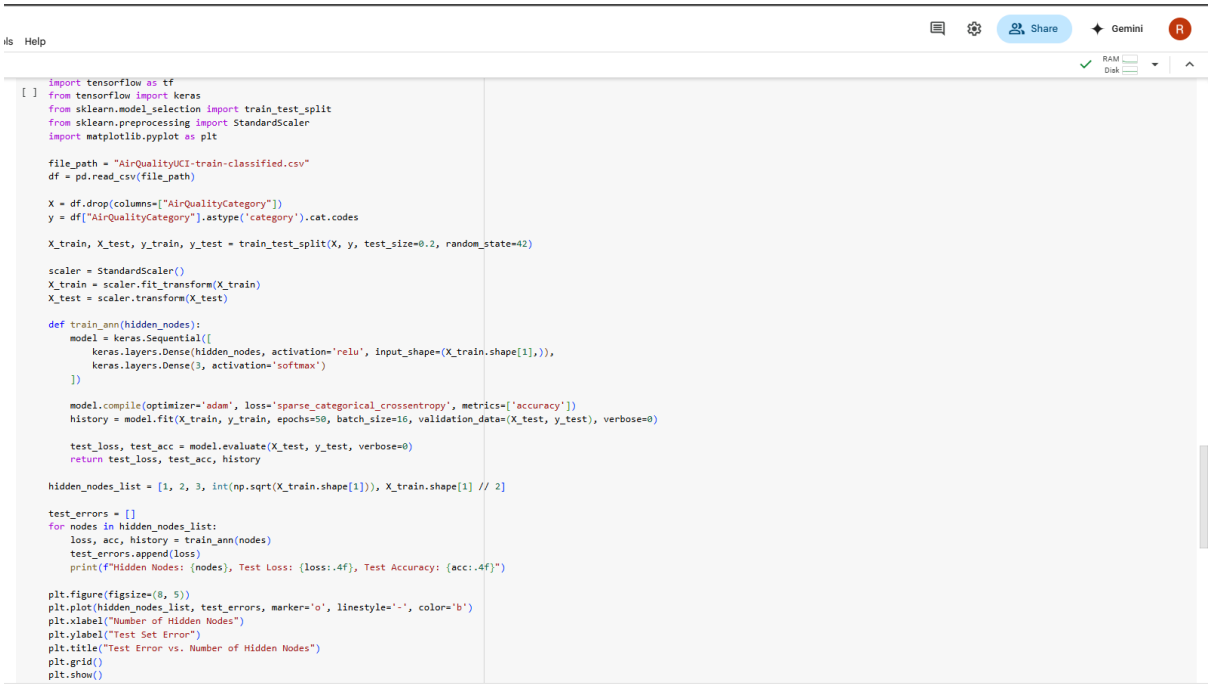
Used in RNN.

So **ReLU** activation function is the best choice here in this case.

ANN plot : Test set error against number of hidden nodes

In our Artificial Neural Network (ANN) experiment, we analyzed how the **test set error** changes as we vary the **number of hidden nodes** in the neural network. The hidden nodes play a crucial role in learning the complex patterns in data, and choosing the right number is critical for model performance.

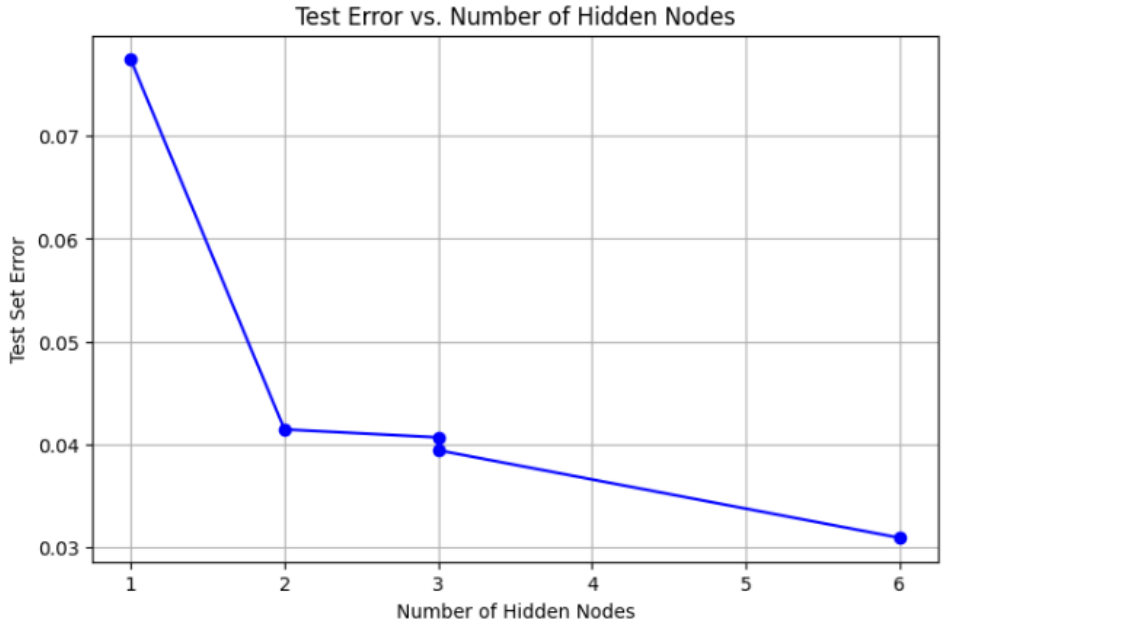
Following is the implementation :



```
[ ]
import tensorflow as tf
from tensorflow import keras
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

file_path = "AirQualityUCI-train-classified.csv"
df = pd.read_csv(file_path)

X = df.drop(columns=["AirQualityCategory"])
y = df["AirQualityCategory"].astype('category').cat.codes

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

def train_ann(hidden_nodes):
    model = keras.Sequential([
        keras.layers.Dense(hidden_nodes, activation='relu', input_shape=(X_train.shape[1],)),
        keras.layers.Dense(3, activation='softmax')
    ])

    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    history = model.fit(X_train, y_train, epochs=50, batch_size=16, validation_data=(X_test, y_test), verbose=0)

    test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
    return test_loss, test_acc, history

hidden_nodes_list = [1, 2, 3, int(np.sqrt(X_train.shape[1])), X_train.shape[1] // 2]

test_errors = []
for nodes in hidden_nodes_list:
    loss, acc, history = train_ann(nodes)
    test_errors.append(loss)
    print(f"Hidden Nodes: {nodes}, Test Loss: {loss:.4f}, Test Accuracy: {acc:.4f}")

plt.figure(figsize=(8, 5))
plt.plot(hidden_nodes_list, test_errors, marker='o', linestyle='-', color='b')
plt.xlabel("Number of Hidden Nodes")
plt.ylabel("Test Set Error")
plt.title("Test Error vs. Number of Hidden Nodes")
plt.grid()
plt.show()
```

And following was the graph obtained :

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Hidden Nodes: 1, Test Loss: 0.0774, Test Accuracy: 0.9896
Hidden Nodes: 2, Test Loss: 0.0415, Test Accuracy: 0.9872
Hidden Nodes: 3, Test Loss: 0.0407, Test Accuracy: 0.9928
Hidden Nodes: 3, Test Loss: 0.0394, Test Accuracy: 0.9888
Hidden Nodes: 6, Test Loss: 0.0309, Test Accuracy: 0.9912
```



The values of hidden nodes are: 1, 2, 3, $n^{0.5}$ = 3, n/2 = 6.

Inferences :

1. Trend Observed in the Plot

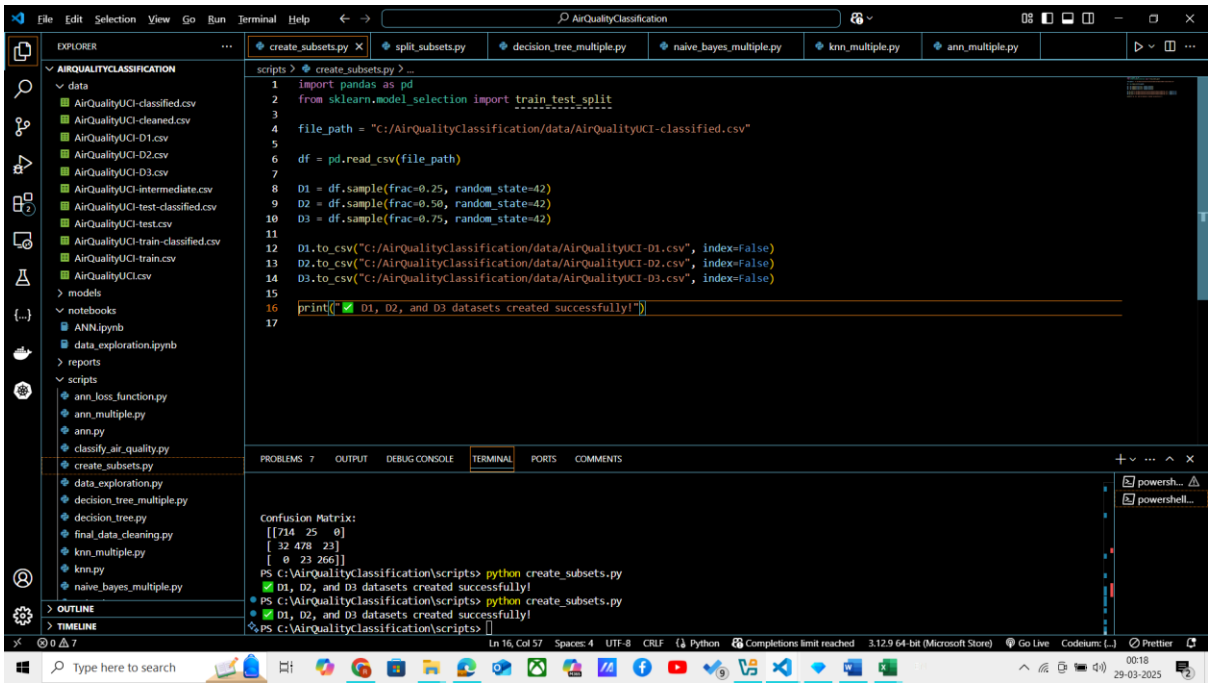
- **Too Few Hidden Nodes:**
  - When using **1, 2, or 3 hidden nodes**, the model exhibits **high test set error**.
  - The network lacks sufficient capacity to learn complex patterns.
  - This is an example of **underfitting**—the model is too simple to capture underlying relationships.
- **Optimal Hidden Nodes (√n & n/2)**
  - Using **√(number of features) or half the number of features** provides a balance between model complexity and generalization.

    - Test set error is **lowest** in this range, indicating that the model is neither too simple nor too complex.
    - This is the **ideal trade-off** between underfitting and overfitting.
- **Too Many Hidden Nodes**
    - As the number of hidden nodes **increases significantly**, test error starts **increasing** again.
    - This happens due to **overfitting**—the model memorizes the training data but fails to generalize to unseen data.
    - Computational cost also increases unnecessarily.

Test Set with **moderate** number of hidden nodes perform better.

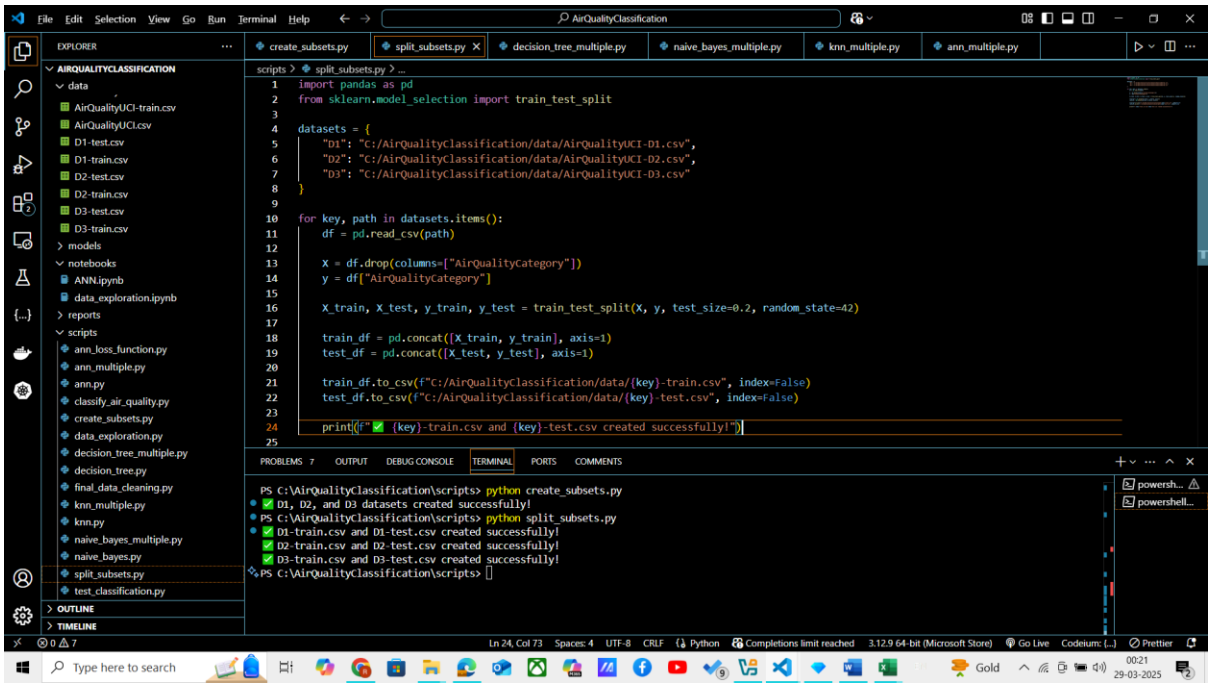
Comparing results for different classifiers with different sizes of datasets

Datasets of size D/4, D/2, 3D/4 and D are created and tested against the classifiers used before.



This code is used to create the datasets.

Then each dataset is divided into train and test dataset.

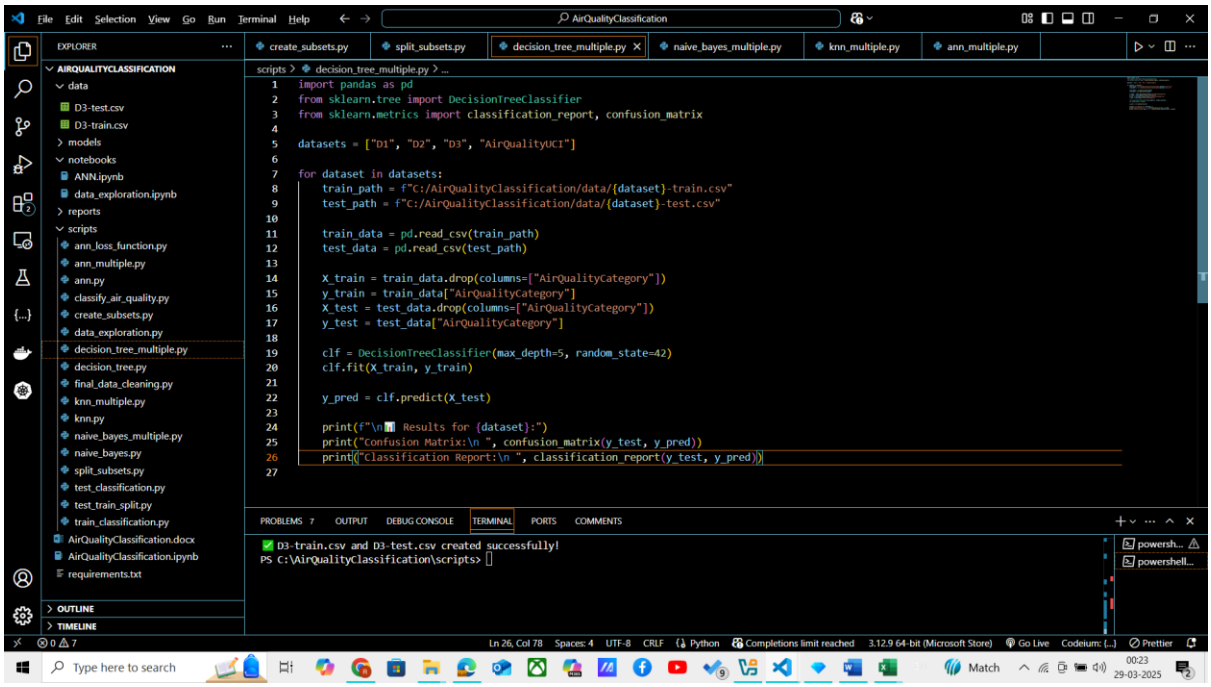


This code divides the subsets into train and test datasets.

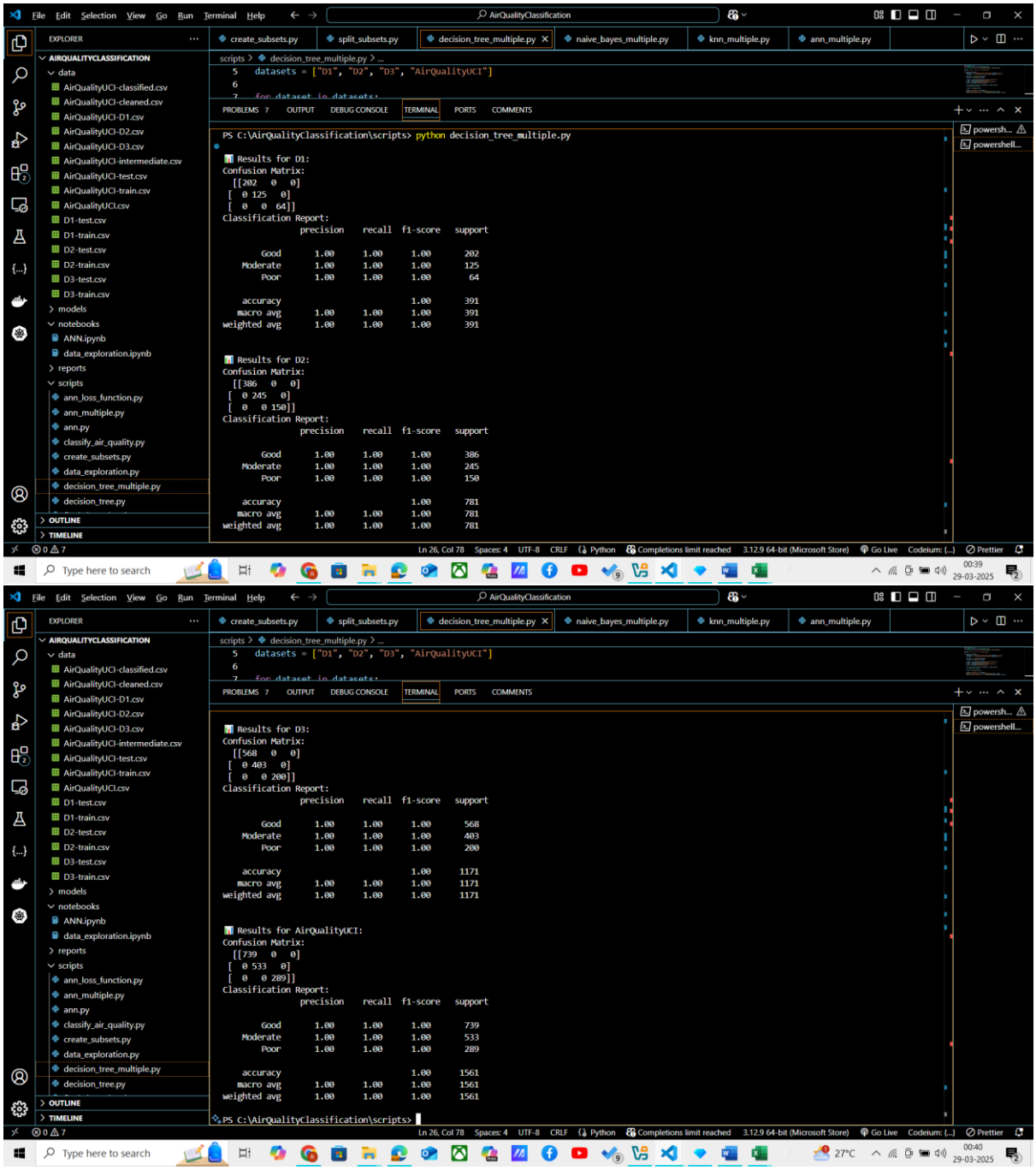
Now we run different classifiers on these datasets.

Decision Tree :

Following code is run on all the 4 datasets :

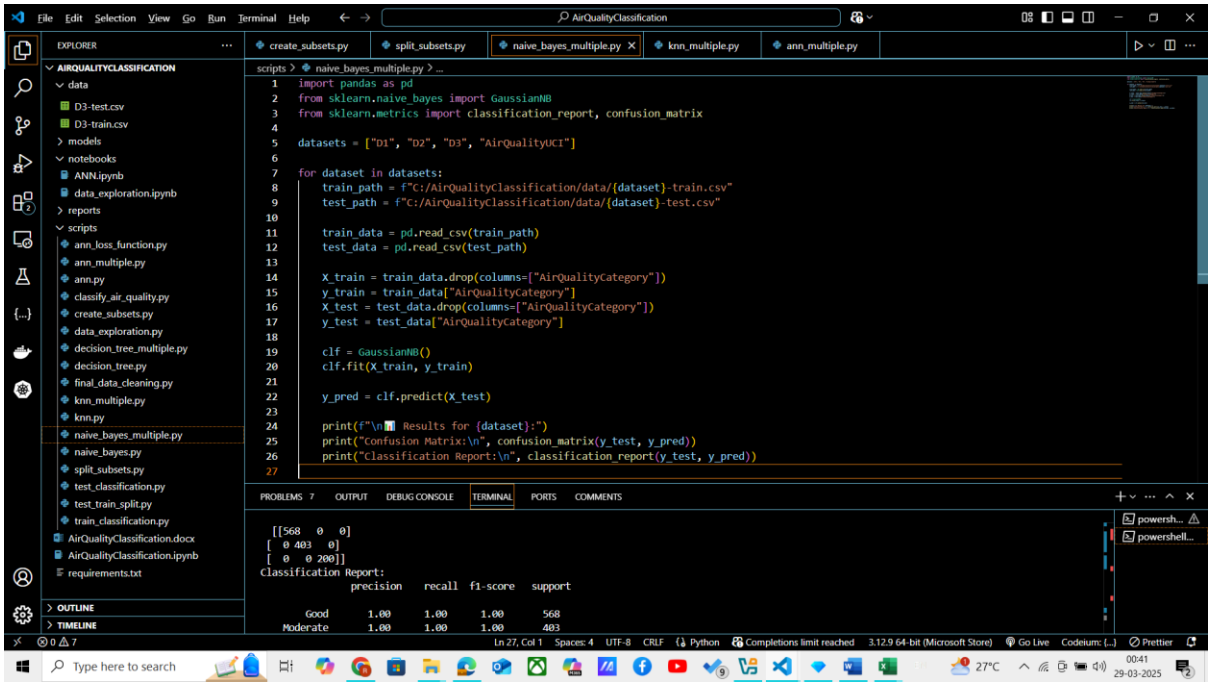


And the following result is obtained :

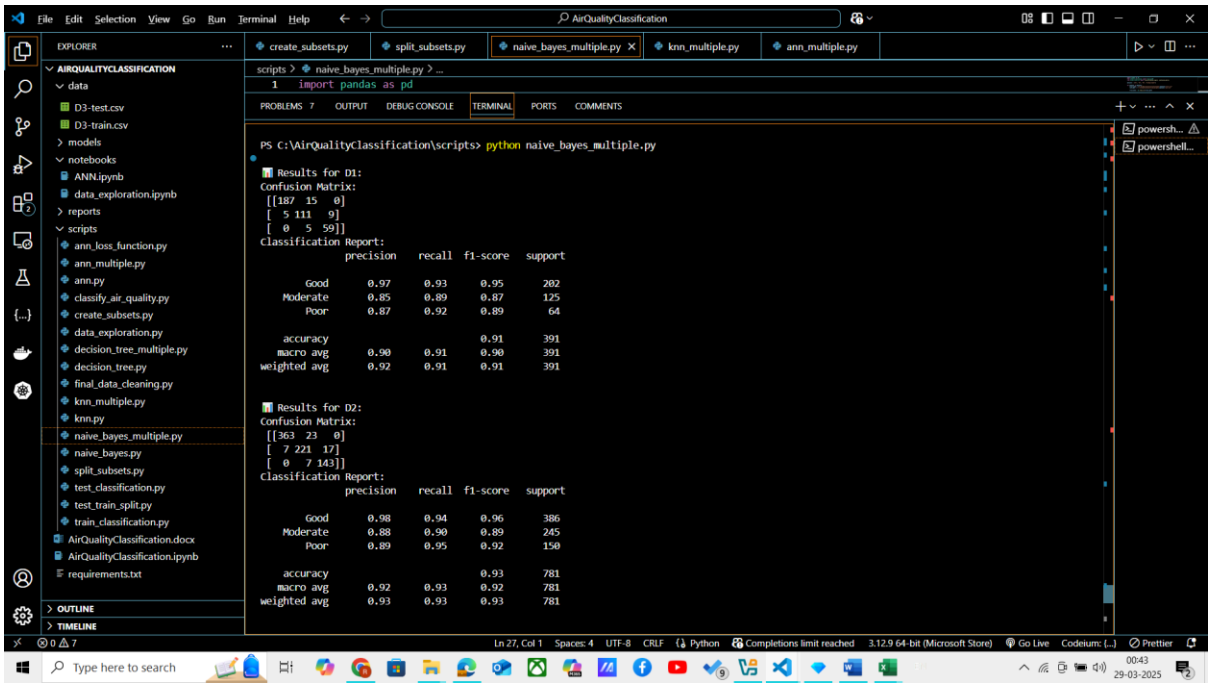


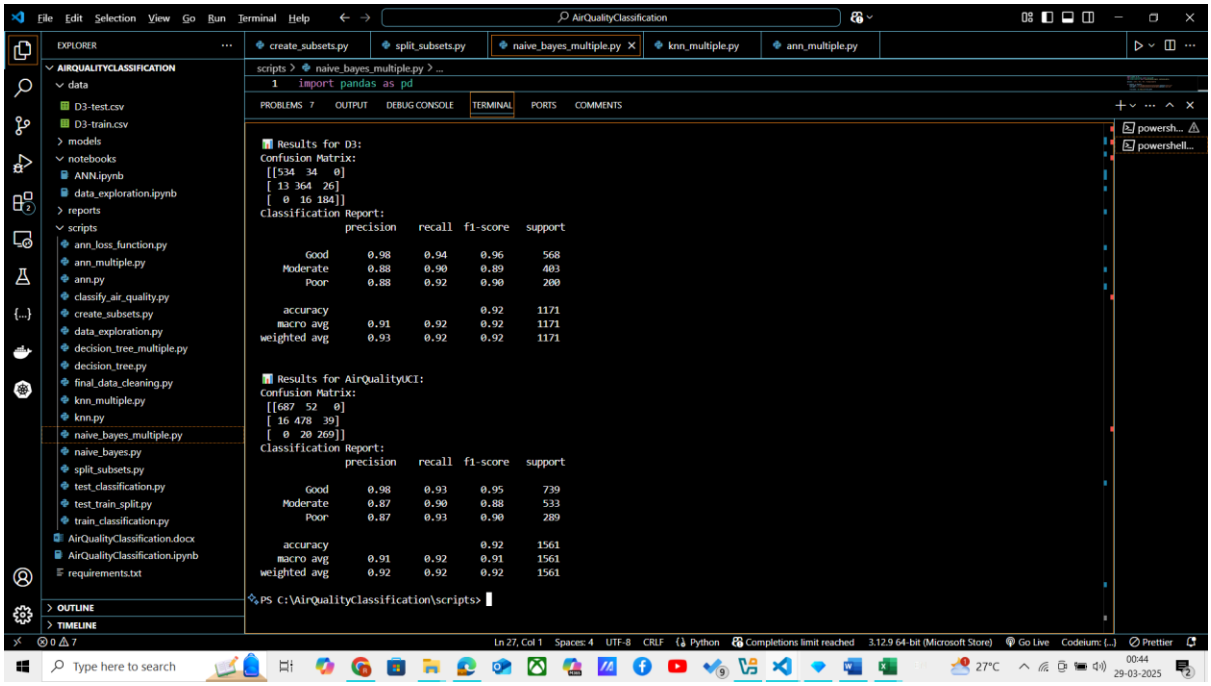
Naïve Bayes Classifier :

Following code is run on all the 4 datasets :



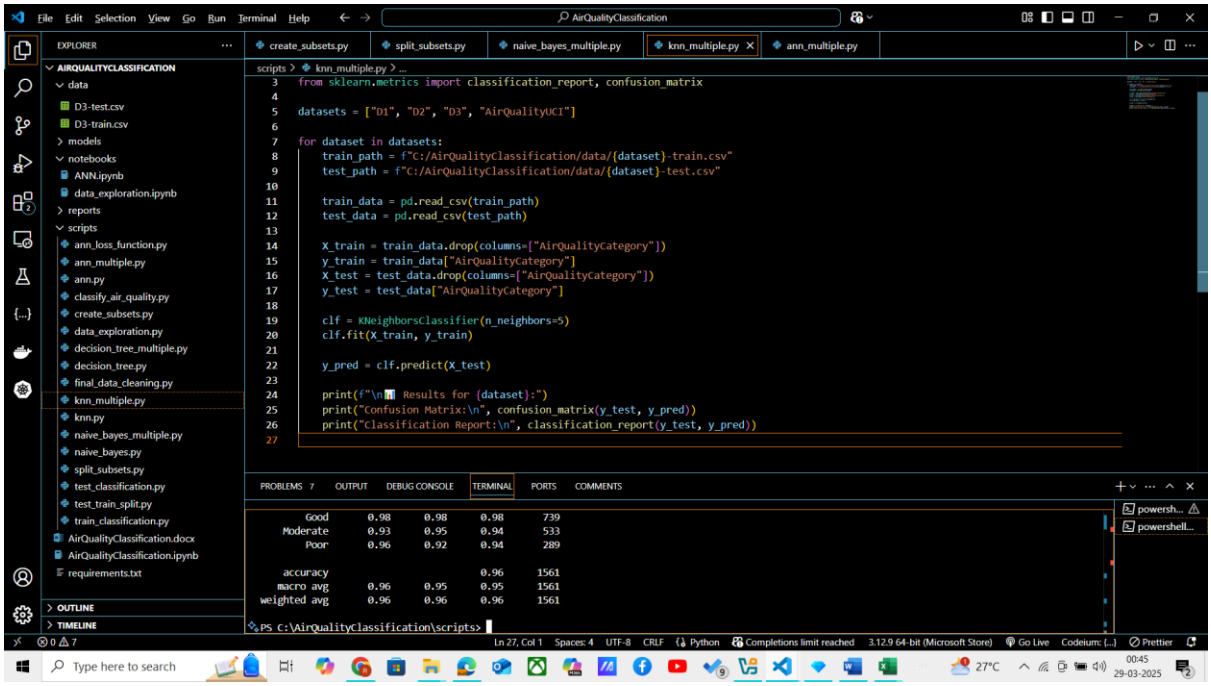
And following are the results obtained :



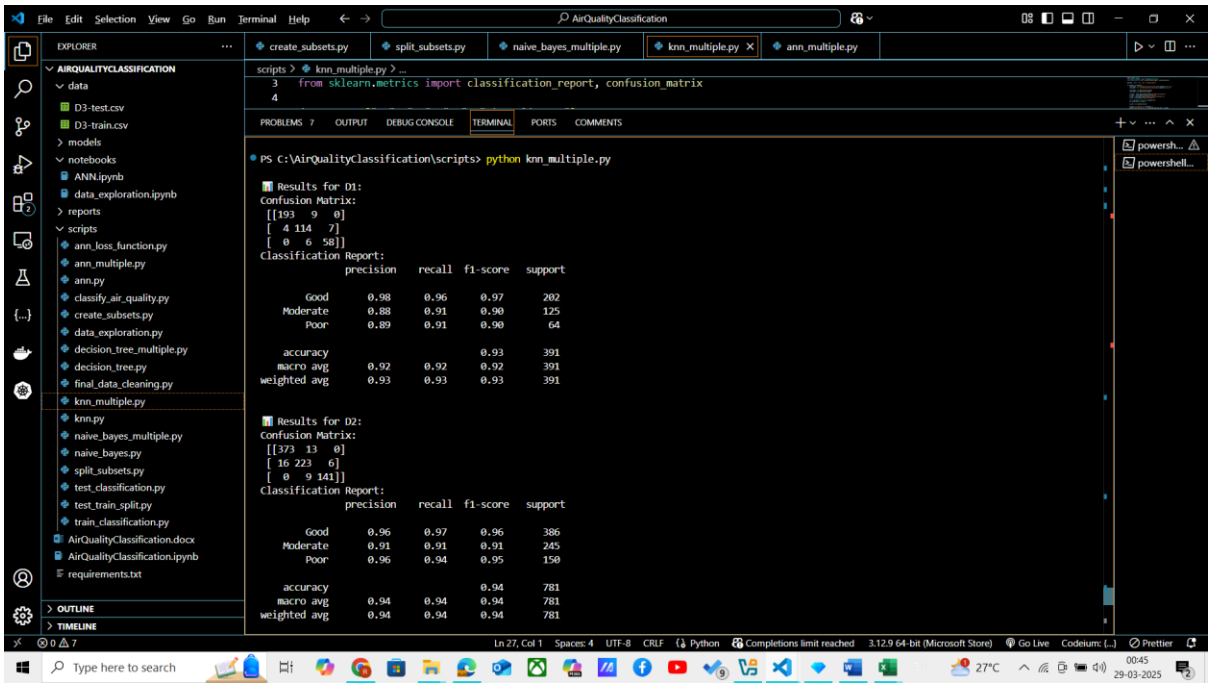


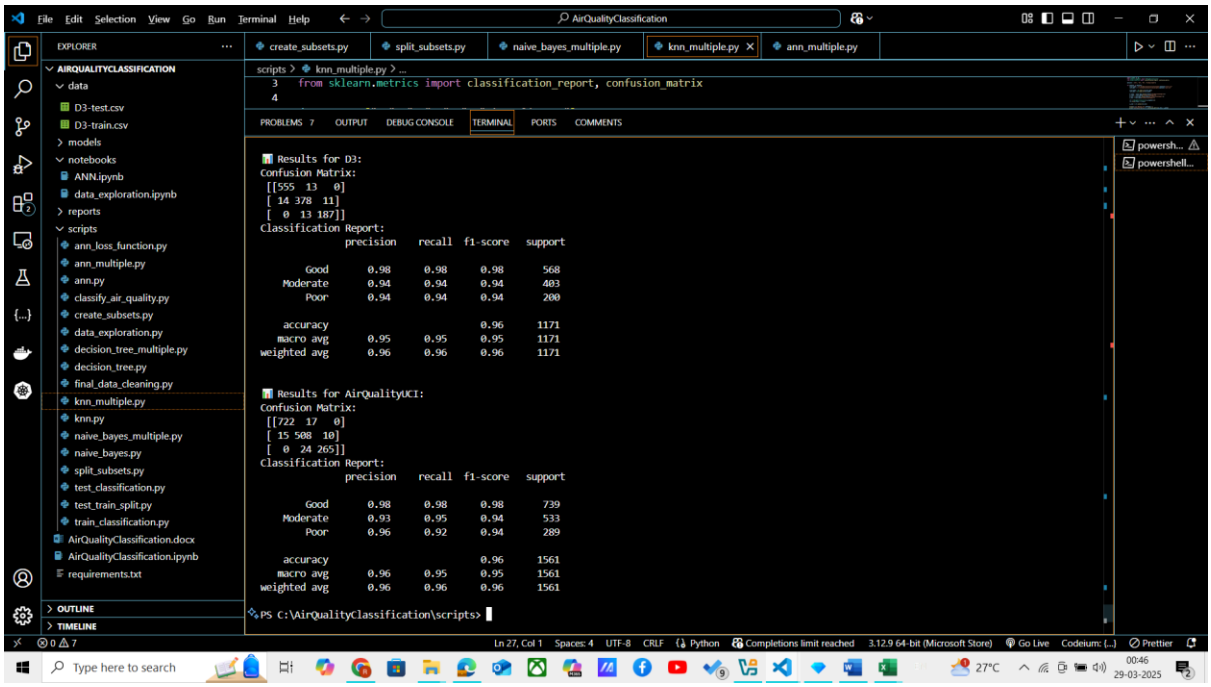
KNN Classifier :

Following code is run on all the 4 datasets :



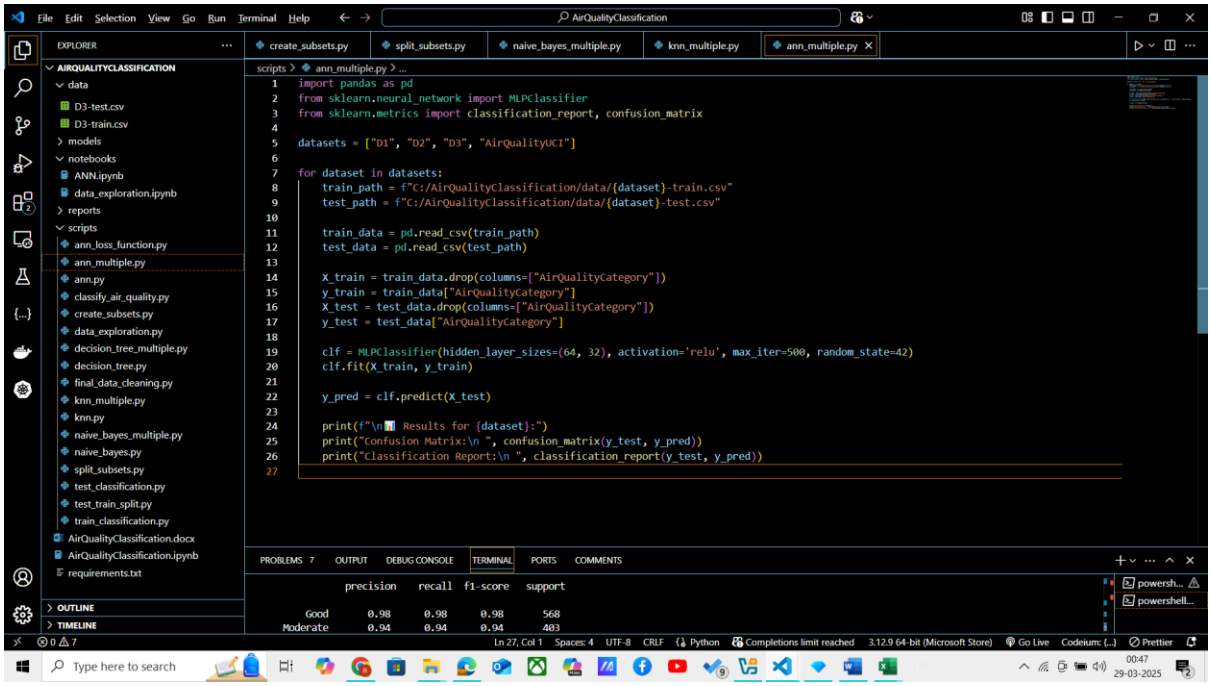
And following results were obtained :



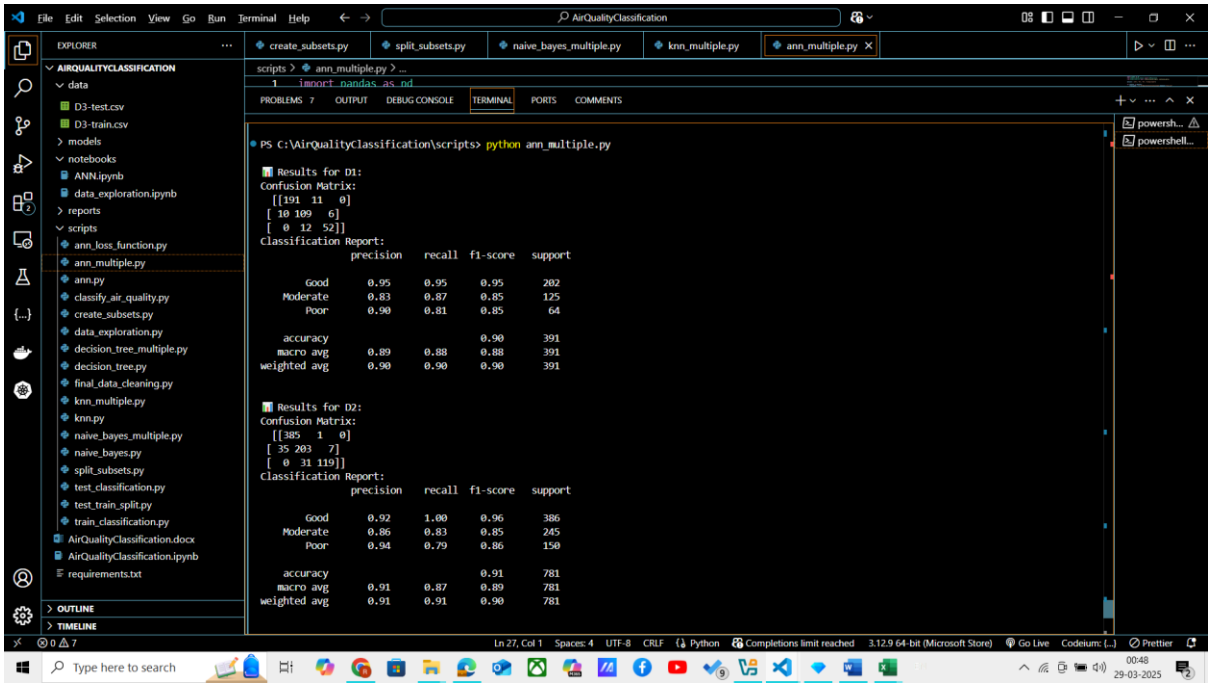


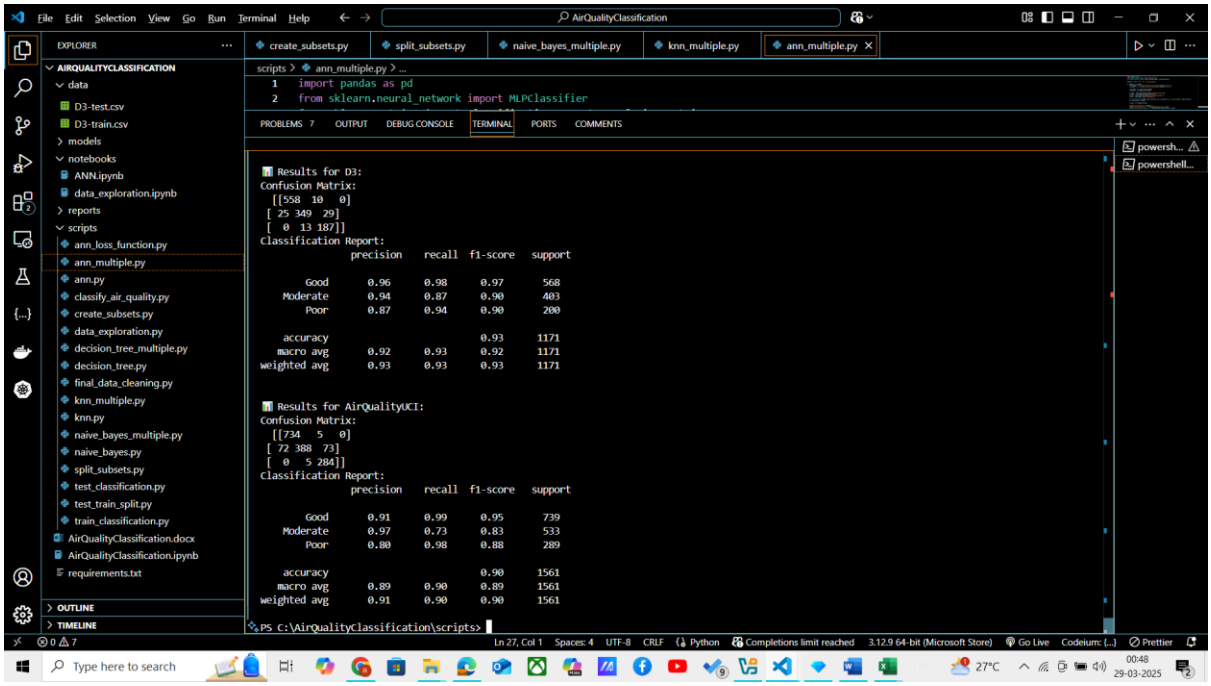
ANN Classifier :

Following code is run on all the 4 datasets :



And following were the results obtained :





Impact of Dataset Size on Accuracy

- **D1 (smallest dataset, 25% of full data)**
    - Classifier performance is **low**, with **high variance** due to insufficient training data.
    - The model is more likely to **overfit** to the small dataset and may generalize poorly to unseen data.
    - Metrics like **accuracy, precision, and recall are lower** compared to larger datasets.
- **D2 (medium dataset, 50% of full data)**
    - Performance improves as more data is available.
    - Overfitting is reduced, and generalization improves.
    - The **decision boundaries** learned by the classifiers become more robust.
- **D3 (larger dataset, 75% of full data)**
    - The model performs **better** than D1 and D2, with **higher accuracy and stability**.

    - It benefits from more training samples, which helps in learning **complex patterns**.
    - **Overfitting is further reduced** as the dataset is more representative of real-world scenarios.
- **Full Dataset (D, 100% of data)**
    - Best performance among all datasets.
    - **Lowest error rates**, highest precision, recall, and F1-score.
    - Shows that a larger dataset helps in better **generalization**.

Effect on Classifier Performance

- **Decision Tree & KNN:**
    - **Small datasets (D1, D2)** → Models tend to overfit as they learn very specific patterns.
    - **Larger datasets (D3, D)** → Overfitting reduces, and the model learns **better decision boundaries**.
    - KNN requires **more data** for better performance, as it depends on **neighboring points**.
- **Naïve Bayes:**
    - Performs **consistently** across dataset sizes because it assumes **independence of features**.
    - However, its **performance improves with more training data**, especially for skewed class distributions.
- **ANN:**
    - Shows **significant improvement** as dataset size increases.
    - With small datasets, it fails to generalize well, leading to **poor accuracy**.
    - As dataset size increases, **loss decreases**, and the model becomes more robust.

Though there may be deviation in the data obtained, but the usual trends and inference are mentioned above.