VIET NAM NATIONAL UNIVERSITY HCM CITY

UNIVERSITY OF INFORMATION TECHNOLOGY

<u>FACULTY OF INFORMATION SYSTEMS</u>

FINAL REPORT

<u>TOPIC</u>:

PREDICT US TRAFFIC ACCIDENT SEVERITY

<u>SUBJECT:</u>

DATA MINING

Practical instructors : M.S Vu Minh Sang

Class: IS403.N21.HTCL

Group implementation: FOUR BEST

Nguyen Thi My Tran	-	20520322
Ton Nu Tu Quyen	-	20520296
Thái Tăng Đức	-	20521203
Trần Anh Huy	-	20520551

Ho Chi Minh City, Jun 2023

THANK YOU

First of all, we sincerely express our gratitude to M.S. Vu Minh Sang for providing us with the expertise necessary to complete this project, as well as for your enthusiastic and sincere guidance and assistance. I believe the group's report would be extremely difficult to finish without your passionate supervision.

This is also an opportunity for each team member to collaborate, improve their cooperation abilities, learn from one another, and, most importantly, implement products during the course.

During the implementation of the project, the team applied the knowledge they had been taught and used new things, with the desire to be able to complete the work most perfectly. However, with limited time, knowledge, and experience, shortcomings cannot be avoided, so the group is looking forward to receiving valuable suggestions from you to help the group supplement and improve their knowledge to better serve future projects and actual work.

Finally, my team wishes you a lot of health to continue to carry out your noble mission of imparting knowledge to future generations

The authors would like to express their sincere thanks.

Ho Chi Minh City, Jun 14, 2023

Group

FOUR BEST

INSTRUCTOR COMMENTS

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

TABLE OF CONTENTS

1. IDENTIFICATION OF DATA MINING PROBLEMS

1.1. Reason for choosing the topic

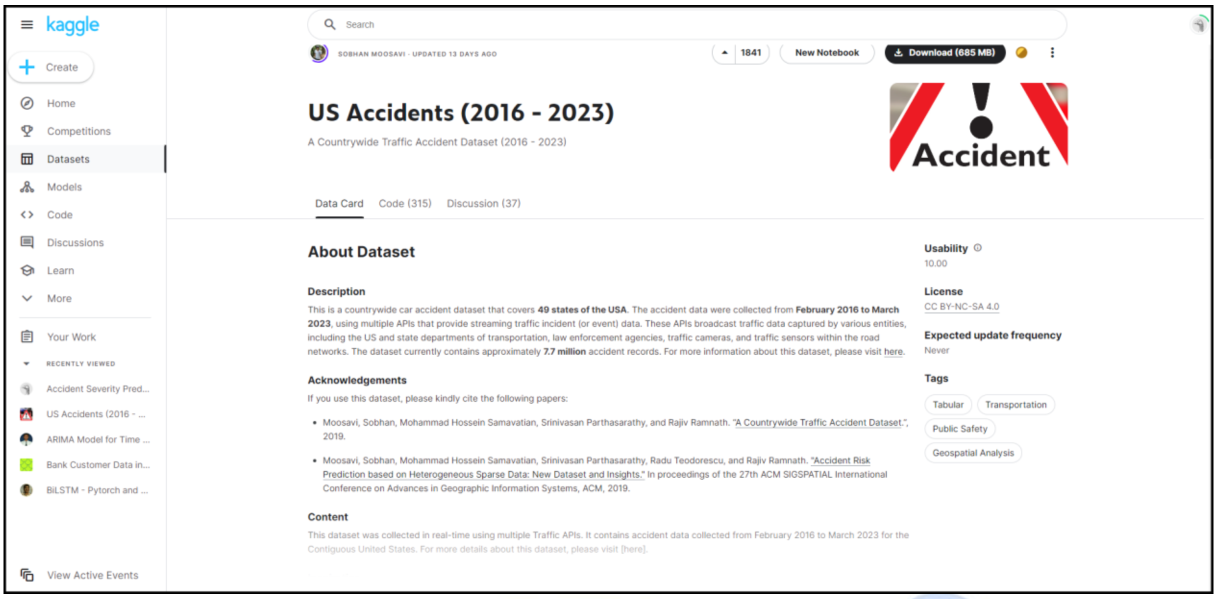
Traffic accidents are one of the most pressing issues today, considered to be one of the biggest disasters that threaten human life and health. The consequences of traffic accidents are very serious, not only affecting people's mental health but also leading to poverty, backwardness, and disease. Up to 70% of the incidents and fatalities are young people, who are pillars of their families. The consequences of traffic accidents are immeasurable, as they impact and cause harm to society and the families of those affected. However, traffic accidents are still increasing and becoming more serious in recent years. To reduce the impact of traffic accidents, many efforts have been made, from improving transportation infrastructure to raising awareness among road users. However, predicting the severity of traffic accidents remains a major challenge, particularly in the context of increasing numbers of vehicles on the road.

In this study, we will use a dataset on traffic accidents in the United States to propose a model for predicting accident severity. We will use data mining techniques to analyze the factors that influence accident severity and build an accurate prediction model. The results of this study may help government agencies and traffic management authorities to develop preventive measures and reduce traffic accidents, while also contributing to the research and application of data mining techniques in the field of transportation.

1.2. About dataset

1.2.1. Dataset information

- Dataset name: US Accidents (2016 - 2023)
- This is a countrywide car accident dataset that covers 49 states of the USA. This dataset was collected in real-time using multiple Traffic APIs. It contains accident data collected from February 2016 to March 2023 for the Contiguous United States.
- Last updated date: 28/05/2023



US Accidents (2016 - 2023)

A Countrywide Traffic Accident Dataset (2016 - 2023)

About Dataset

Description

This is a countrywide car accident dataset that covers **49 states of the USA**. The accident data were collected from **February 2016 to March 2023**, using multiple APIs that provide streaming traffic incident (or event) data. These APIs broadcast traffic data captured by various entities, including the US and state departments of transportation, law enforcement agencies, traffic cameras, and traffic sensors within the road networks. The dataset currently contains approximately **7.7 million** accident records. For more information about this dataset, please visit here.

Acknowledgements

If you use this dataset, please kindly cite the following papers:

- Moosavi, Sobhan, Mohammad Hossein Samavatian, Srinivasan Parthasarathy, and Rajiv Ramnath. "A Countrywide Traffic Accident Dataset.", 2019.
- Moosavi, Sobhan, Mohammad Hossein Samavatian, Srinivasan Parthasarathy, Radu Teodorescu, and Rajiv Ramnath. "Accident Risk Prediction based on Heterogeneous Sparse Data: New Dataset and Insights." In proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, ACM, 2019.

Content

This dataset was collected in real-time using multiple Traffic APIs. It contains accident data collected from February 2016 to March 2023 for the Contiguous United States. For more details about this dataset, please visit [here].

Usability
10.00

License
CC BY-NC-SA 4.0

Expected update frequency
Never

Tags
Tabular, Transportation, Public Safety, Geospatial Analysis

1.2.2. Author

- Author name: Sobhan Moosavi
- The author is a scientist in the United States.

1.2.3. Number of lines, columns, earnings period

- The data includes: 46 columns, approximately 7.7 million accident records
- Due to the large size of the original dataset, in order to facilitate research on our device, we filtered and extracted the data rows from 2022 to 2023. The resulting dataset includes: 46 columns, 1.048.575 accident records.
- After preprocessing: 32 columns, 87436 accident records
- After encoding: 61 columns, 87436 accident records

1.2.3. Dataset source

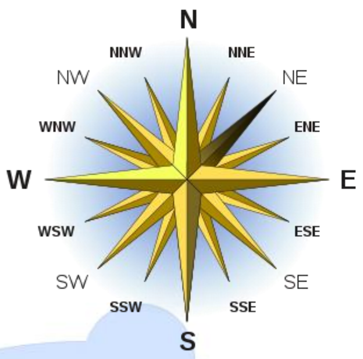
Data Link: https://www.kaggle.com/datasets/sobhanmoosavi/us-accidents

1.2.4. Detailed description of the data store's attribute columns

No	Attribute	Description	Data Type	Significant
1	ID		String	This is a unique identifier of the accident record.
2	Source		String	Source of raw accident data

3	Severity		Int	Shows the severity of the accident, a number between 1 and 4, where 1 indicates the least impact on traffic (i.e., short delay as a result of the accident) and 4 indicates a significant impact on traffic (i.e., long delay).
4	Start_Time	Start Time	DateTime	Shows start time of the accident in local time zone.
5	End_Time	End Time	DateTime	Shows end time of the accident in local time zone. End time here refers to when the impact of accident on traffic flow was dismissed.
6	Start_Lat	Start Latitude	Int	Shows latitude in GPS coordinate of the start point.
7	Start_Lng	Start Longitude	Int	Shows longitude in GPS coordinate of the start point.
8	End_Lat	End Latitude	Int	Shows latitude in GPS coordinate of the end point.
9	End_Lng	End Longitude	Int	Shows longitude in GPS coordinate of the end point.
10	Distance(mi)		Float	The length of the road extent affected by the accident in miles (1 mile = 1,609344 km = 1.609,344 m).
11	Description		String	Shows a human provided description of the accident.
12	Street		String	Shows the street name in address field.

14	City		String	Shows the city in address field.
15	County		String	Shows the county in address field.
16	State		String	Shows the state in address field.
17	Zipcode		Int	Shows the zipcode in address field.
18	Country		String	Shows the country in address field.
19	Timezone		String	Shows timezone based on the location of the accident (eastern, central, etc.).
20	Airport_Code	Airport Code	String	Denotes an airport-based weather station which is the closest one to location of the accident.
21	Weather_Timestamp	Weather Timestamp	Datetime	Shows the time-stamp of weather observation record (in local time).
22	Temperature(F)		Float	Shows the temperature (in Fahrenheit).
23	Wind_Chill(F)	Wind Chill	Float	Shows the wind chill (in Fahrenheit).
24	Humidity(%)		Int	Shows the humidity (in percentage).
25	Pressure(in)		Float	Shows the air pressure (in inches).
26	Visibility(mi)		Float	Shows visibility (in miles).
27	Wind_Direction	Wind Direction	String	Shows wind direction. The possible values are:

				- Calm: no wind - E, East: wind from the east - ENE, NNE, NNW, NW, ...  N NNW NNE NW NE WNW ENE W E WSW ESE SW SE SSW SSE S
28	Wind_Speed(mph)	Wind Speed	Float	Shows wind speed (in miles per hour).
29	Precipitation(in)	Precipitation	Float	Shows precipitation amount in inches, if there is any.
30	Weather_Condition	Weather Condition	String	Shows the weather condition (rain, snow, thunderstorm, fog, etc.)
31	Amenity		Boolean	A POI annotation which indicates presence of amenity in a nearby location.
32	Bump		Boolean	A POI annotation which indicates presence of speed bump or hump in a nearby location.
33	Crossing		Boolean	A POI annotation which indicates presence of crossing in a nearby location.
33	Give_Way	Give Way	Boolean	A POI annotation which indicates presence of give_way in a nearby location.

34	Junction		Boolean	A POI annotation which indicates presence of junction in a nearby location.
35	No_Exit	No Exit	Boolean	A POI annotation which indicates presence of no_exit in a nearby location.
36	Railway		Boolean	A POI annotation which indicates presence of railway in a nearby location.
37	Roundabout		Boolean	A POI annotation which indicates presence of roundabout in a nearby location.
38	Station		Boolean	A POI annotation which indicates presence of station in a nearby location.
39	Stop		Boolean	A POI annotation which indicates presence of stop in a nearby location.
40	Traffic_Calming	Traffic Calming	Boolean	A POI annotation which indicates presence of traffic_calming in a nearby location.
41	Traffic_Signal	Traffic Signal	Boolean	A POI annotation which indicates presence of traffic_signal in a nearby location.
42	Turning_Loop	Turning Loop	Boolean	A POI annotation which indicates presence of turning_loop in a nearby location.

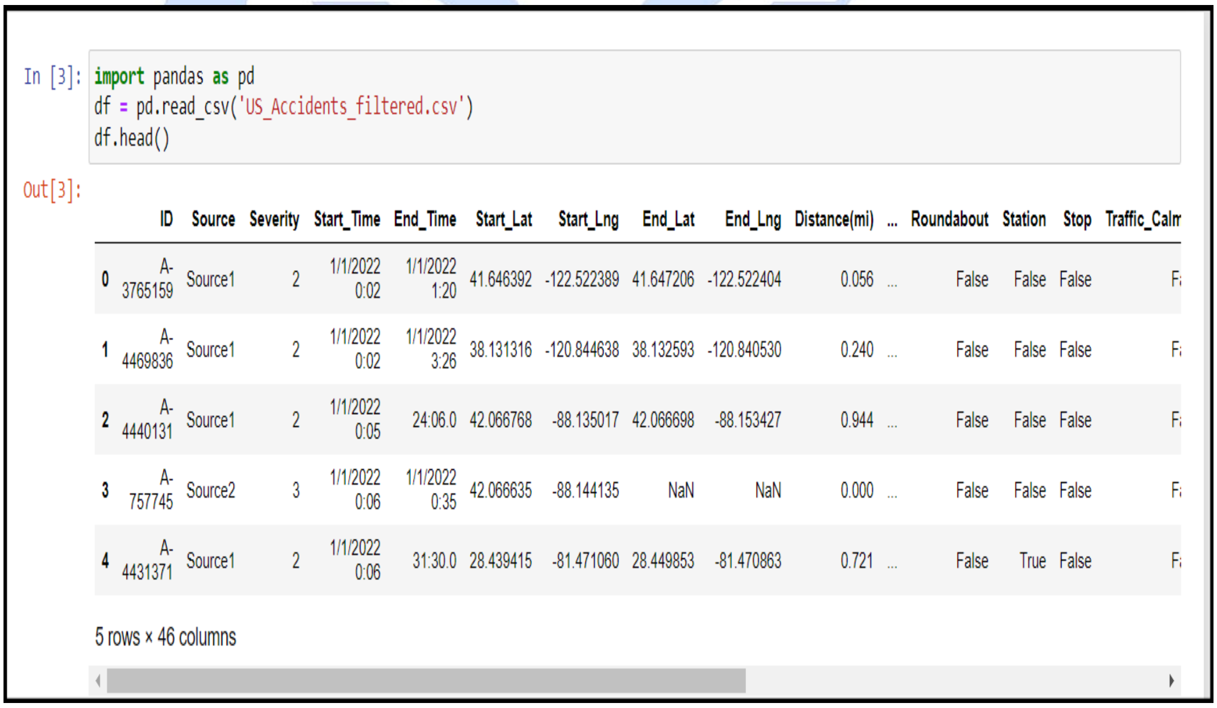
43	Sunrise_Sunset	Sunrise Sunset	String	Shows the period of day (i.e. day or night) based on sunrise/sunset.
44	Civil_Twilight	Civil Twilight	String	Shows the period of day (i.e. day or night) based on civil twilight.
45	Nautical_Twilight	Nautical Twilight	String	Shows the period of day (i.e. day or night) based on nautical twilight.
46	Astronomical_Twilight	Astronomical Twilight	String	Shows the period of day (i.e. day or night) based on astronomical twilight.

1.3. Descriptive Statistics and Exploratory Data Analysis (EDA)

1.3.1. Descriptive Statistics

Calculate the values: Count, Min, Max, Mean, Median, Quantile, Range, Mode and Variance on dataset.

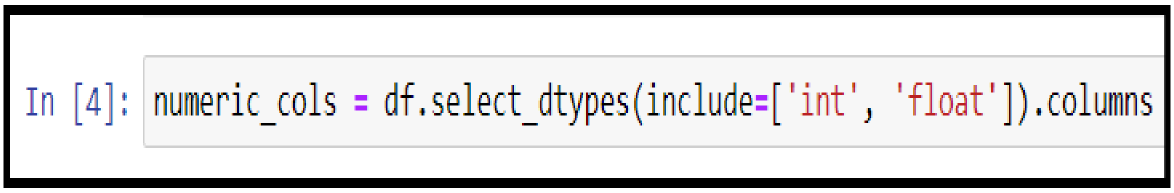
Step 1: Import necessary libraries and read original data from CSV file.



```
In [3]: import pandas as pd
        df = pd.read_csv('US_Accidents_filtered.csv')
        df.head()
```

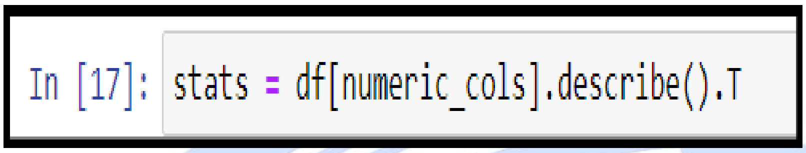
```
Out[3]:
          ID   Source  Severity  Start_Time  End_Time  Start_Lat   Start_Lng    End_Lat     End_Lng  Distance(mi)  ...  Roundabout  Station  Stop  Traffic_Calm
0         A-  Source1         2    1/1/2022  1/1/2022  41.646392 -122.522389  41.647206 -122.522404         0.056  ...       False    False  False            Fa
     3765159                           0:02      1:20
1         A-  Source1         2    1/1/2022  1/1/2022  38.131316 -120.844638  38.132593 -120.840530         0.240  ...       False    False  False            Fa
     4469836                           0:02      3:26
2         A-  Source1         2    1/1/2022   24:06.0  42.066768  -88.135017  42.066698  -88.153427         0.944  ...       False    False  False            Fa
     4440131                           0:05
3         A-  Source2         3    1/1/2022  1/1/2022  42.066635  -88.144135        NaN         NaN         0.000  ...       False    False  False            Fa
      757745                           0:06      0:35
4         A-  Source1         2    1/1/2022   31:30.0  28.439415  -81.471060  28.449853  -81.470863         0.721  ...       False     True  False            Fa
     4431371                           0:06

5 rows × 46 columns
```

Step 2: We notice that the dataset contains two numeric data types, which are Int and Float. We proceed to filter out columns with data type Int or Float to calculate.



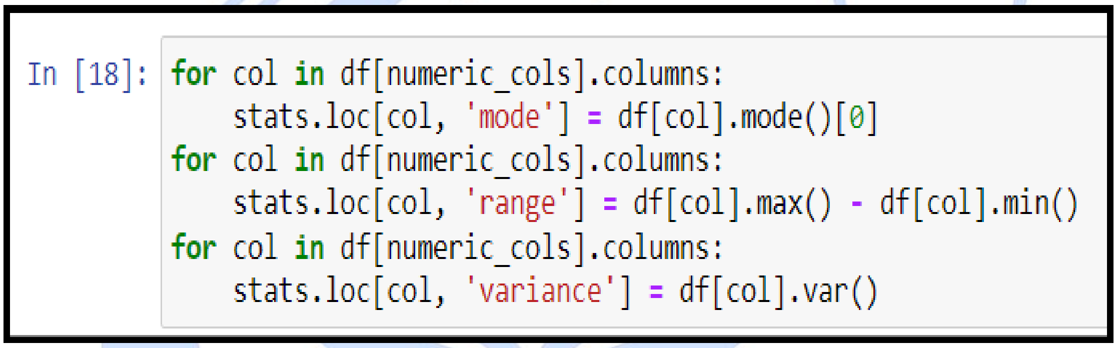
```
In [4]: numeric_cols = df.select_dtypes(include=['int', 'float']).columns
```

Step 3: Use the describe() method to compute descriptive statistics for all columns with data types of Int or Float.



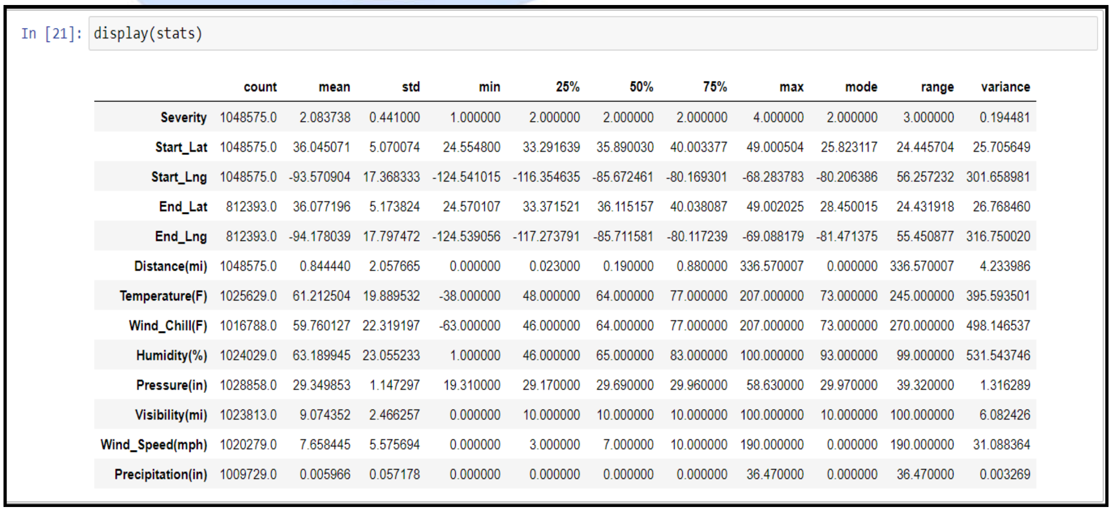
```
In [17]: stats = df[numeric_cols].describe().T
```

Step 4: Calculate mode, range, and variance for all columns with data types of Int or Float.



```
In [18]: for col in df[numeric_cols].columns:
             stats.loc[col, 'mode'] = df[col].mode()[0]
         for col in df[numeric_cols].columns:
             stats.loc[col, 'range'] = df[col].max() - df[col].min()
         for col in df[numeric_cols].columns:
             stats.loc[col, 'variance'] = df[col].var()
```

Step 5: Display the result.



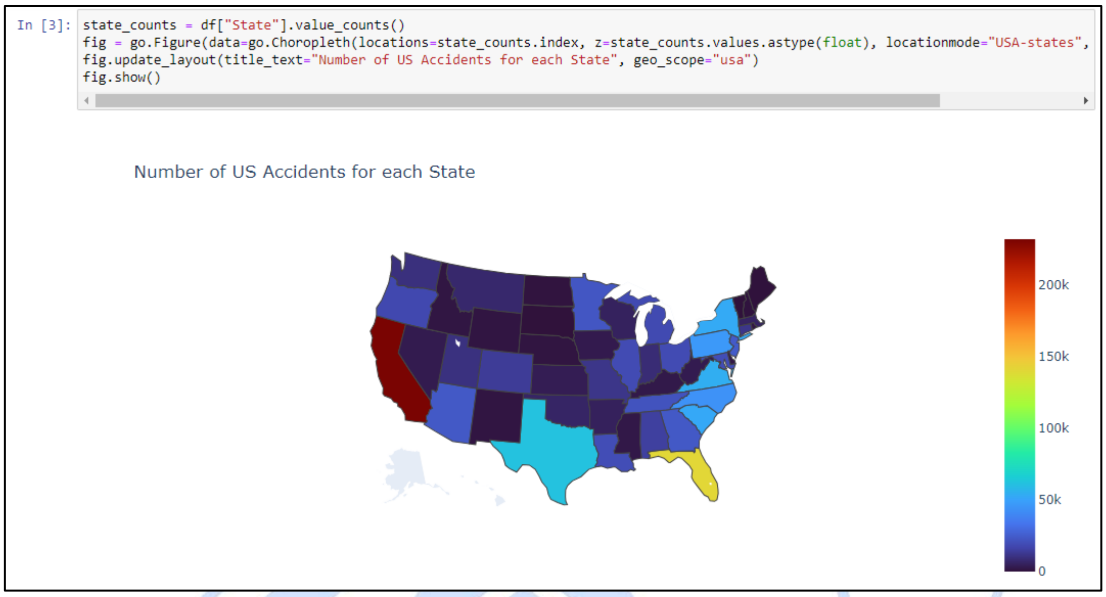
```
In [21]: display(stats)
```

	count	mean	std	min	25%	50%	75%	max	mode	range	variance
Severity	1048575.0	2.083738	0.441000	1.000000	2.000000	2.000000	2.000000	4.000000	2.000000	3.000000	0.194481
Start_Lat	1048575.0	36.045071	5.070074	24.554800	33.291639	35.890030	40.003377	49.000504	25.823117	24.445704	25.705649
Start_Lng	1048575.0	-93.570904	17.368333	-124.541015	-116.354635	-85.672461	-80.169301	-68.283783	-80.206386	56.257232	301.658981
End_Lat	812393.0	36.077196	5.173824	24.570107	33.371521	36.115157	40.038087	49.002025	28.450015	24.431918	26.768460
End_Lng	812393.0	-94.178039	17.797472	-124.539056	-117.273791	-85.711581	-80.117239	-69.088179	-81.471375	55.450877	316.750020
Distance(mi)	1048575.0	0.844440	2.057665	0.000000	0.023000	0.190000	0.880000	336.570007	0.000000	336.570007	4.233986
Temperature(F)	1025629.0	61.212504	19.889532	-38.000000	48.000000	64.000000	77.000000	207.000000	73.000000	245.000000	395.593501
Wind_Chill(F)	1016788.0	59.760127	22.319197	-63.000000	46.000000	64.000000	77.000000	207.000000	73.000000	270.000000	498.146537
Humidity(%)	1024029.0	63.189945	23.055233	1.000000	46.000000	65.000000	83.000000	100.000000	93.000000	99.000000	531.543746
Pressure(in)	1028858.0	29.349853	1.147297	19.310000	29.170000	29.690000	29.960000	58.630000	29.970000	39.320000	1.316289
Visibility(mi)	1023813.0	9.074352	2.466257	0.000000	10.000000	10.000000	10.000000	100.000000	10.000000	100.000000	6.082426
Wind_Speed(mph)	1020279.0	7.658445	5.575694	0.000000	3.000000	7.000000	10.000000	190.000000	0.000000	190.000000	31.088364
Precipitation(in)	1009729.0	0.005966	0.057178	0.000000	0.000000	0.000000	0.000000	36.470000	0.000000	36.470000	0.003269

1.3.2. Exploratory Data Analysis (EDA)

- **Visualizing the number of accidents that occurred in each state**

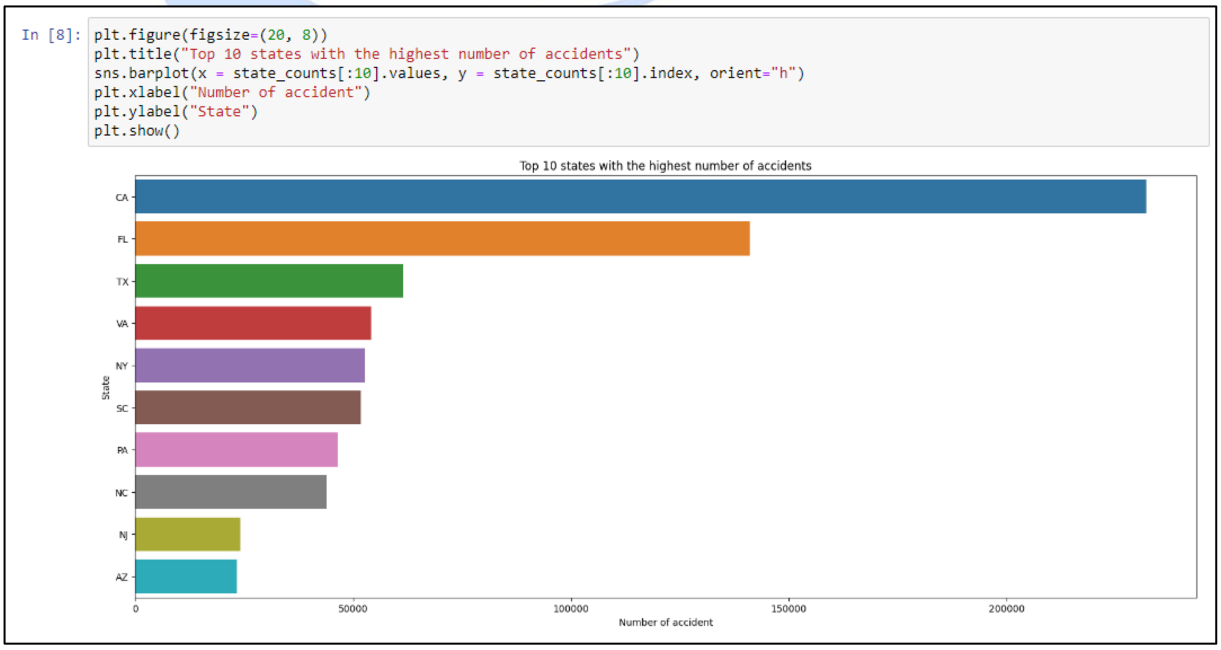
  Firstly, we visualize the density of accidents in each state across the United State on the map.



```
In [3]: state_counts = df["State"].value_counts()
        fig = go.Figure(data=go.Choropleth(locations=state_counts.index, z=state_counts.values.astype(float), locationmode="USA-states",
        fig.update_layout(title_text="Number of US Accidents for each State", geo_scope="usa")
        fig.show()
```

Looking at the map, it can be seen that most states are shaded in dark to light blue (indicating a low frequency of accidents), with only a few states colored in warm hues (indicating a high frequency of accidents).

We will list the top 10 states with the most accidents to identify which states they are.

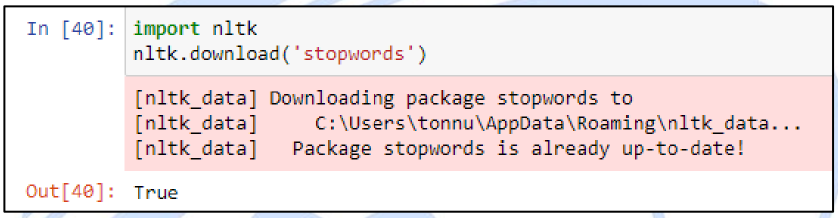


```
In [8]: plt.figure(figsize=(20, 8))
        plt.title("Top 10 states with the highest number of accidents")
        sns.barplot(x = state_counts[:10].values, y = state_counts[:10].index, orient="h")
        plt.xlabel("Number of accident")
        plt.ylabel("State")
        plt.show()
```

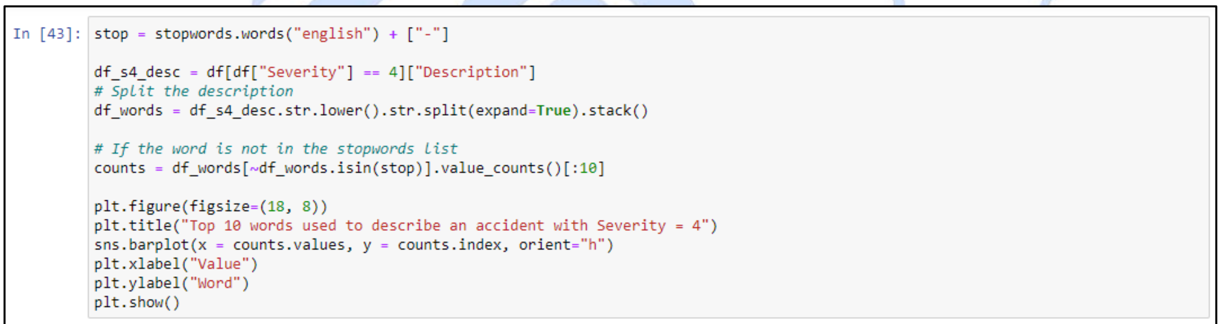
So CA (California) is the state that with the highest number of accidents, above 200000 accidents. And FL (Florida) is the second one with the number of accident close to 150000. Most of the others are approximately around 50000.

- **Find out most frequent words in the description of an accident with "Severity = 4"**

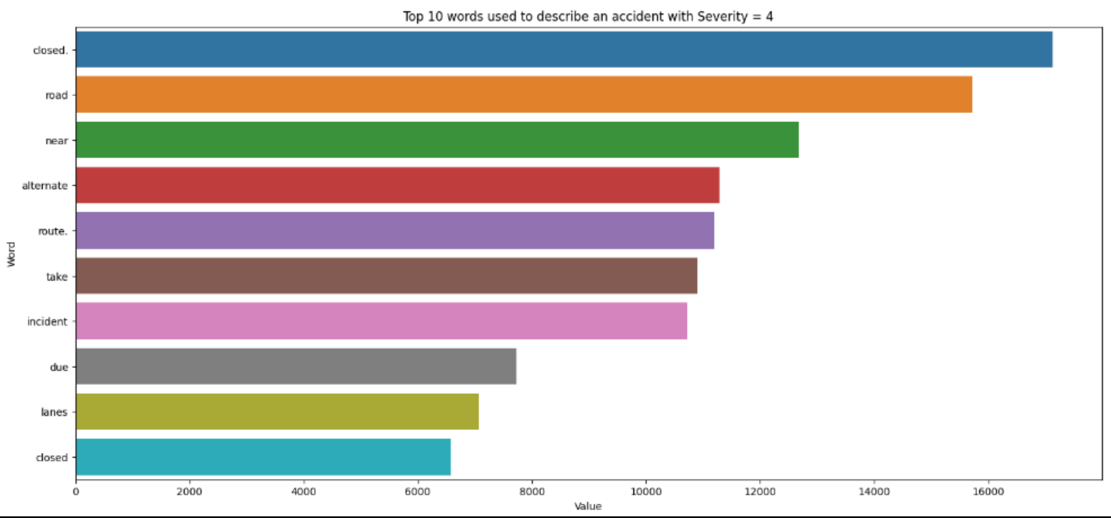
We are going to compute the most frequent words in the description column of the accidents with a value of severity equal to 4, using some stopwords from the english language. We will need to download stopwords package for this part.



```
In [40]: import nltk
         nltk.download('stopwords')

         [nltk_data] Downloading package stopwords to
         [nltk_data]     C:\Users\tonnu\AppData\Roaming\nltk_data...
         [nltk_data]   Package stopwords is already up-to-date!

Out[40]: True
```



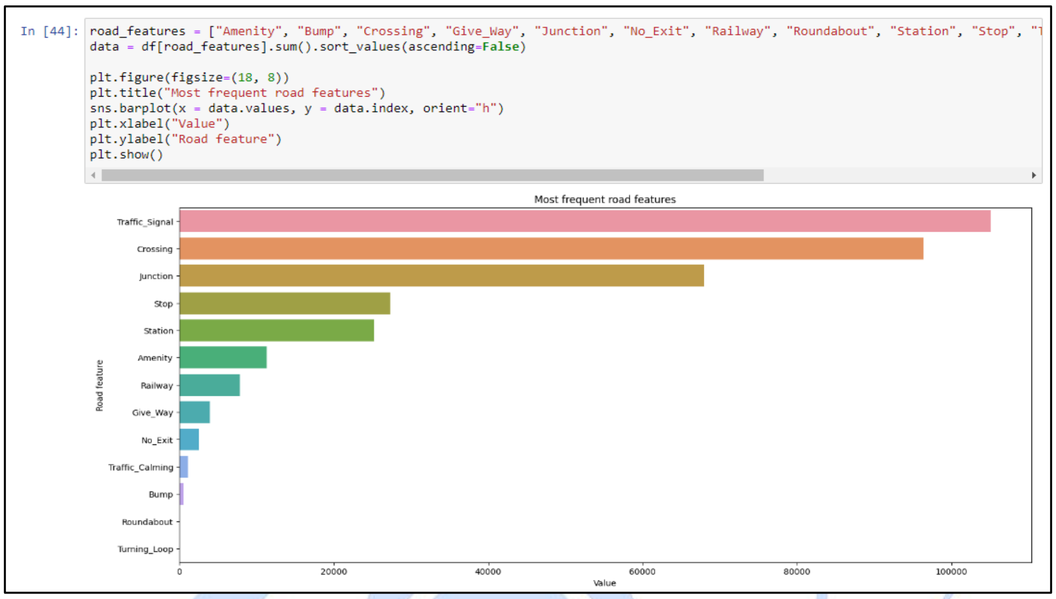
```
In [43]: stop = stopwords.words("english") + ["-"]

         df_s4_desc = df[df["Severity"] == 4]["Description"]
         # Split the description
         df_words = df_s4_desc.str.lower().str.split(expand=True).stack()

         # If the word is not in the stopwords list
         counts = df_words[~df_words.isin(stop)].value_counts()[:10]

         plt.figure(figsize=(18, 8))
         plt.title("Top 10 words used to describe an accident with Severity = 4")
         sns.barplot(x = counts.values, y = counts.index, orient="h")
         plt.xlabel("Value")
         plt.ylabel("Word")
         plt.show()
```



We can see that the most used word in the description is "closed". Next in line is the word "road".

- **Analyze which are the most frequent road features in accidents**

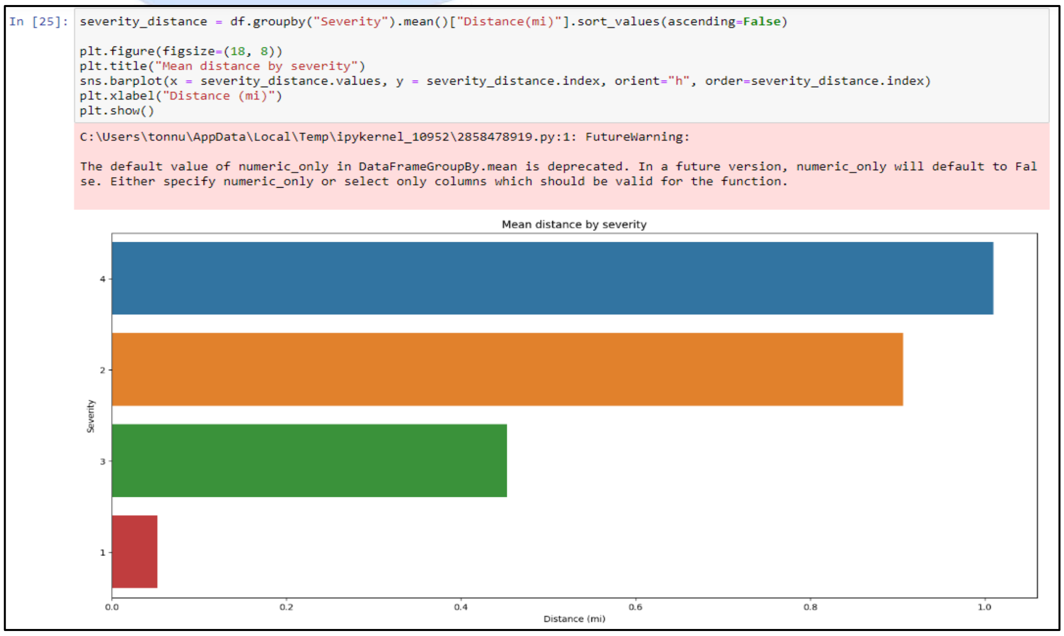
  After finding out that the word “road” is used the most in the description with “Serverity = 4”, we will extract columns related to road features in the dataset and see which road feature is the most frequent location of accidents.



```
In [44]: road_features = ["Amenity", "Bump", "Crossing", "Give_Way", "Junction", "No_Exit", "Railway", "Roundabout", "Station", "Stop", "T
         data = df[road_features].sum().sort_values(ascending=False)

         plt.figure(figsize=(18, 8))
         plt.title("Most frequent road features")
         sns.barplot(x = data.values, y = data.index, orient="h")
         plt.xlabel("Value")
         plt.ylabel("Road feature")
         plt.show()
```

  As we can see, most of the accidents occured near traffic signal and crossing. This is understandable because at places with traffic signals, especially at locations with many crossing, there are often a lot of vehicles passing by, making it easier to occure traffic accidents.

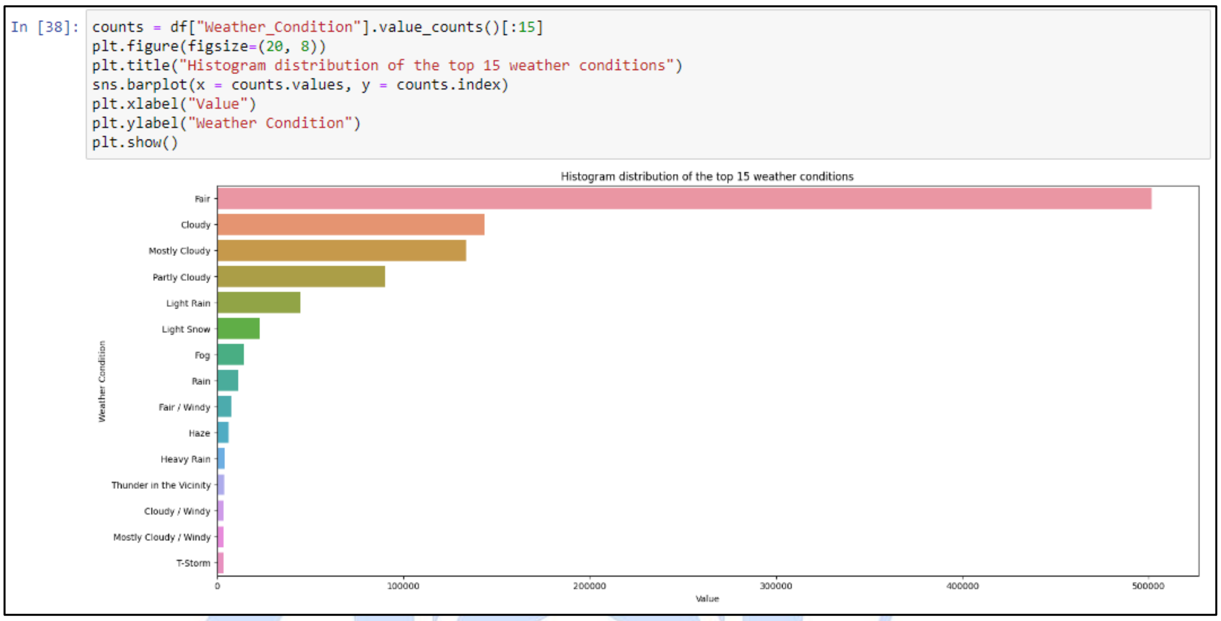
- **Calculate mean distance group by each Serverity level**



```
In [25]: severity_distance = df.groupby("Severity").mean()["Distance(mi)"].sort_values(ascending=False)

         plt.figure(figsize=(18, 8))
         plt.title("Mean distance by severity")
         sns.barplot(x = severity_distance.values, y = severity_distance.index, orient="h", order=severity_distance.index)
         plt.xlabel("Distance (mi)")
         plt.show()

         C:\Users\tonnu\AppData\Local\Temp\ipykernel_10952\2858478919.py:1: FutureWarning:

         The default value of numeric_only in DataFrameGroupBy.mean is deprecated. In a future version, numeric_only will default to Fal
         se. Either specify numeric_only or select only columns which should be valid for the function.
```

It is clear that the serverity level 4 – the most significant impact would lead the length of the road extent affected by the accident to be the highest number. However, it is quite surprising to see that severity level 2 has a relatively high mean distance of impact, second only to severity level 4.

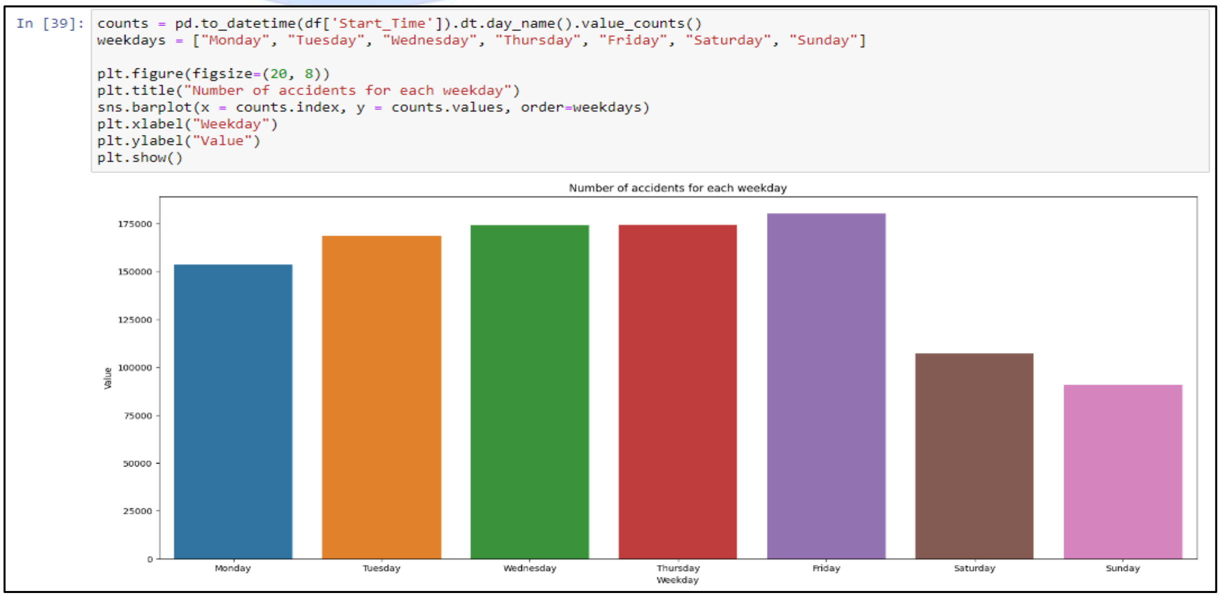
- **Analyze weather condition**



```
In [38]: counts = df["Weather_Condition"].value_counts()[:15]
         plt.figure(figsize=(20, 8))
         plt.title("Histogram distribution of the top 15 weather conditions")
         sns.barplot(x = counts.values, y = counts.index)
         plt.xlabel("Value")
         plt.ylabel("Weather Condition")
         plt.show()
```

The highest number of accidents occurs when the weather is fair. This is normal because when the weather is stable, more people decide to go out, leading to more crowded traffic and a higher risk of accidents.

- **Analyze number of accidents occured in each weekday**



```
In [39]: counts = pd.to_datetime(df['Start_Time']).dt.day_name().value_counts()
         weekdays = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]

         plt.figure(figsize=(20, 8))
         plt.title("Number of accidents for each weekday")
         sns.barplot(x = counts.index, y = counts.values, order=weekdays)
         plt.xlabel("Weekday")
         plt.ylabel("Value")
         plt.show()
```

Accidents tend to occur more frequently on weekdays when people are going to work, while the number of accidents on weekends is only about half of that on other days. This can be explained by the fact that on working days, people tend to be in more of a hurry when driving, increasing the risk of accidents compared to weekends.

1.4. Problem description

Our project aims to predict the severity of accidents in the United States by analyzing a range of features that have been recorded in this dataset. The goal is to provide insights into what factors contribute to the severity of accidents, and to identify potential high-risk areas or situations.

1.5. Data mining tools

- Python: version 3.11.2

- Jupyter Notebook

- Visual Studio Code

- Python's libraries: numpy, matplotlib, seaborn, pandas, category_encoders, sklearn, plotly, nltk

1.6. Source code and result

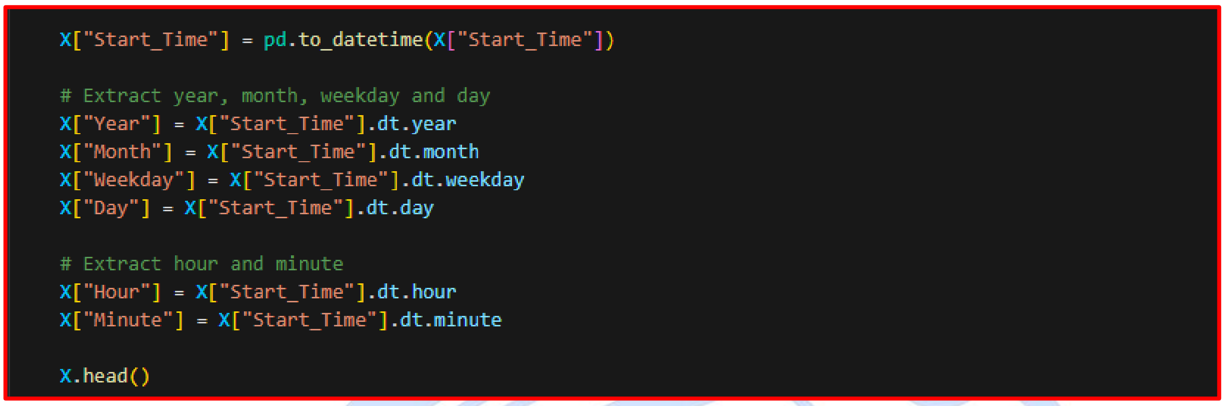
Link One Drive:

https://uithcm-my.sharepoint.com/:f:/g/personal/20520322_ms_uit_edu_vn/EtJTxv1VKU5MgUTjiiw8edkBBtoruPFlx7KqkyPFSw_GOQ?e=WqY5Ml

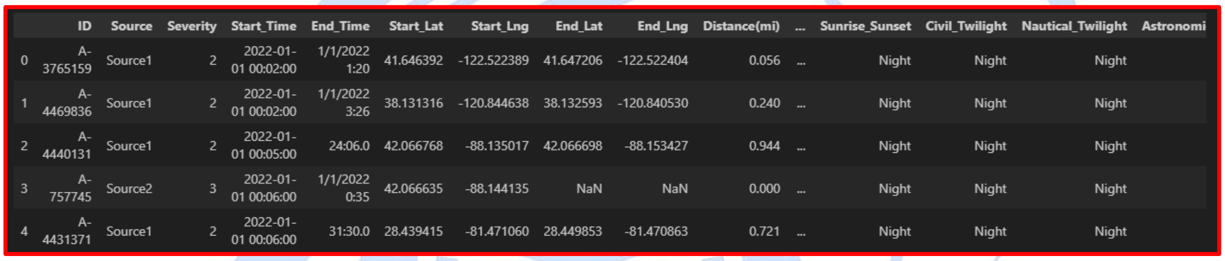
2. DATA PREPROCESSING PROCESS

2.1. Feature addition

We decided to decompose the Start_Time feature in year, month, day, weekday, hour and minute, in order to feed them to the models.



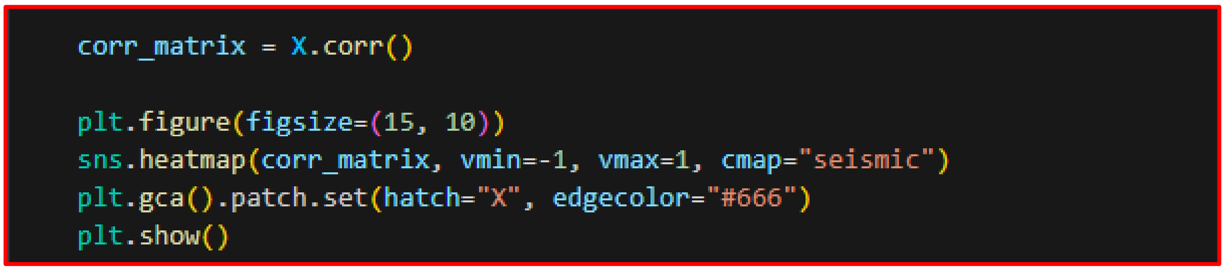
```python
X["Start_Time"] = pd.to_datetime(X["Start_Time"])

# Extract year, month, weekday and day
X["Year"] = X["Start_Time"].dt.year
X["Month"] = X["Start_Time"].dt.month
X["Weekday"] = X["Start_Time"].dt.weekday
X["Day"] = X["Start_Time"].dt.day

# Extract hour and minute
X["Hour"] = X["Start_Time"].dt.hour
X["Minute"] = X["Start_Time"].dt.minute

X.head()
```



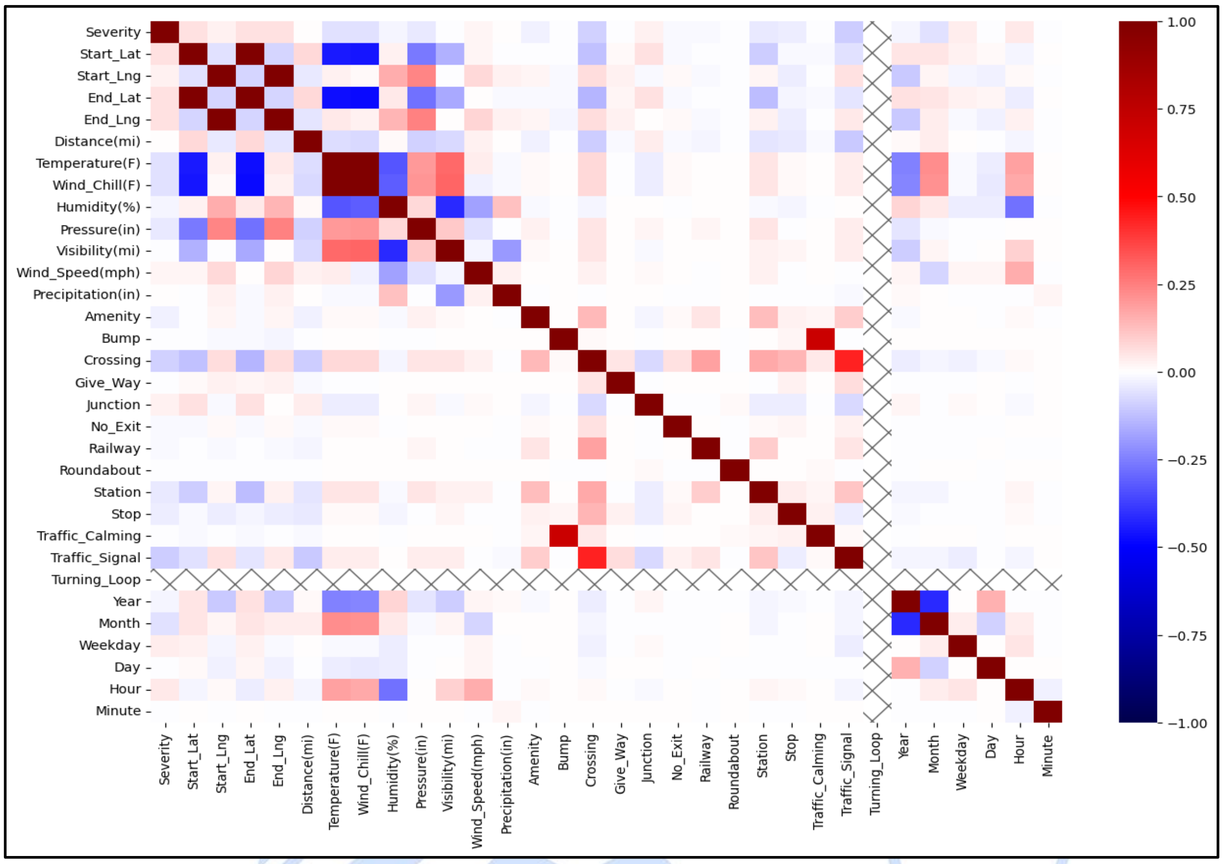
	ID	Source	Severity	Start_Time	End_Time	Start_Lat	Start_Lng	End_Lat	End_Lng	Distance(mi)	...	Sunrise_Sunset	Civil_Twilight	Nautical_Twilight	Astronomi
0	A-3765159	Source1	2	2022-01-01 00:02:00	1/1/2022 1:20	41.646392	-122.522389	41.647206	-122.522404	0.056	...	Night	Night	Night	
1	A-4469836	Source1	2	2022-01-01 00:02:00	1/1/2022 3:26	38.131316	-120.844638	38.132593	-120.840530	0.240	...	Night	Night	Night	
2	A-4440131	Source1	2	2022-01-01 00:05:00	24:06.0	42.066768	-88.135017	42.066698	-88.153427	0.944	...	Night	Night	Night	
3	A-757745	Source2	3	2022-01-01 00:06:00	1/1/2022 0:35	42.066635	-88.144135	NaN	NaN	0.000	...	Night	Night	Night	
4	A-4431371	Source1	2	2022-01-01 00:06:00	31:30.0	28.439415	-81.471060	28.449853	-81.470863	0.721	...	Night	Night	Night	

2.2. Check correlation between features

In the next block is presented the correlation matrix between all the possible features, in the form of an heatmap. Cùng với đó, chúng ta có thể quan sát mối tương quan giữa các tính năng khác nhau của tập dữ liệu, để kiểm tra xem một số tính năng có tương quan cao hay không và loại bỏ một trong số chúng.



```python
corr_matrix = X.corr()

plt.figure(figsize=(15, 10))
sns.heatmap(corr_matrix, vmin=-1, vmax=1, cmap="seismic")
plt.gca().patch.set(hatch="X", edgecolor="#666")
plt.show()
```

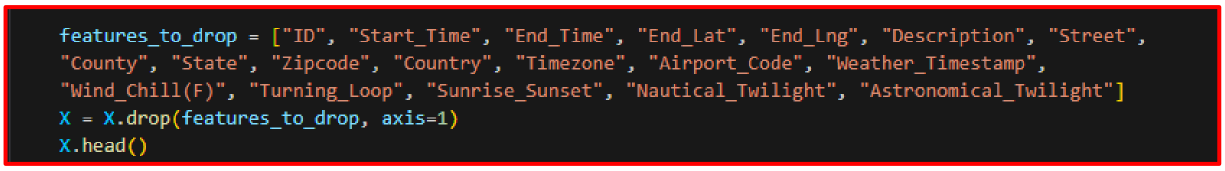


From the matrix we can see that the start and end GPS coordinates of the accidents are highly correlated. Moreover, the wind chill (temperature) is directly proportional to the temperature, so we can also drop one of them.

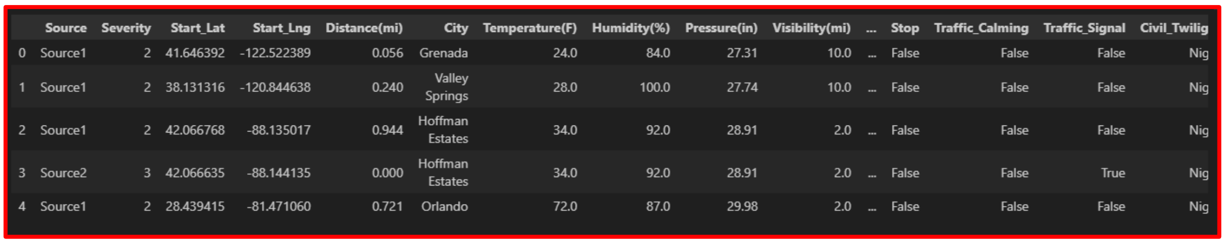
We can also see that the presence of a traffic signal is slightly correlated to the severity of an accident meaning that maybe traffic lights can help the traffic flow when an accident occurs.

2.3. Feature selection

Here is the process of feature selection, in order to select the best features from which our models can learn.



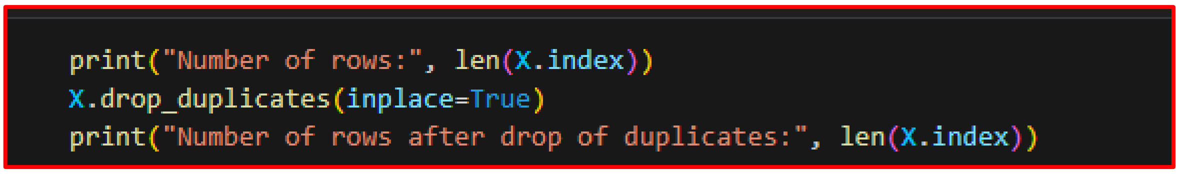
```
features_to_drop = ["ID", "Start_Time", "End_Time", "End_Lat", "End_Lng", "Description", "Street",
"County", "State", "Zipcode", "Country", "Timezone", "Airport_Code", "Weather_Timestamp",
"Wind_Chill(F)", "Turning_Loop", "Sunrise_Sunset", "Nautical_Twilight", "Astronomical_Twilight"]
X = X.drop(features_to_drop, axis=1)
X.head()
```



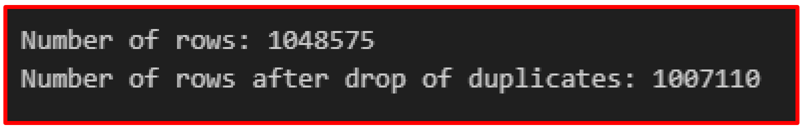
```
   Source  Severity  Start_Lat   Start_Lng  Distance(mi)      City  Temperature(F)  Humidity(%)  Pressure(in)  Visibility(mi)  ...  Stop  Traffic_Calming  Traffic_Signal  Civil_Twilig
0  Source1        2  41.646392  -122.522389        0.056   Grenada            24.0         84.0         27.31            10.0  ...  False            False           False           Nig
1  Source1        2  38.131316  -120.844638        0.240    Valley            28.0        100.0         27.74            10.0  ...  False            False           False           Nig
                                                           Springs
2  Source1        2  42.066768   -88.135017        0.944   Hoffman            34.0         92.0         28.91             2.0  ...  False            False           False           Nig
                                                           Estates
3  Source2        3  42.066635   -88.144135        0.000   Hoffman            34.0         92.0         28.91             2.0  ...  False            False            True           Nig
                                                           Estates
4  Source1        2  28.439415   -81.471060        0.721   Orlando            72.0         87.0         29.98             2.0  ...  False            False           False           Nig
```

2.4. Drop duplicates

In this section we are going to check if there are some duplicates in the dataset.



```
print("Number of rows:", len(X.index))
X.drop_duplicates(inplace=True)
print("Number of rows after drop of duplicates:", len(X.index))
```

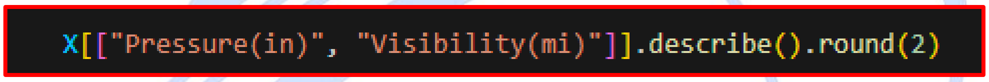


```
Number of rows: 1048575
Number of rows after drop of duplicates: 1007110
```

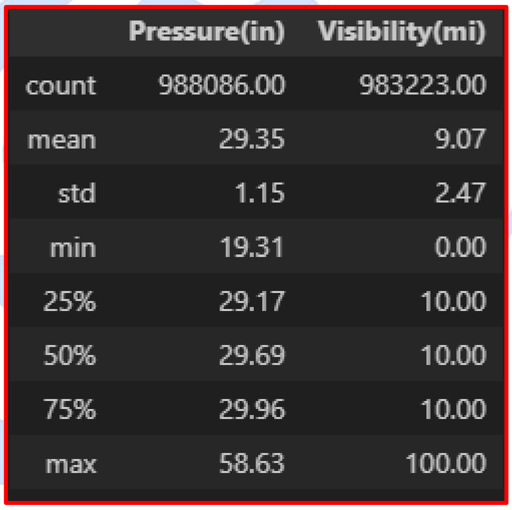
2.5. Handle erroneous and missing values

Here we are going to clean the dataset from erroneous or missing values.

Let's instead analyze Pressure and Visibility:

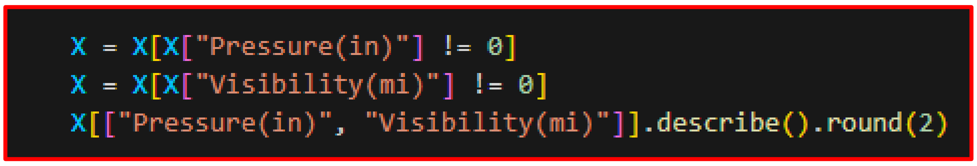


```
X[["Pressure(in)", "Visibility(mi)"]].describe().round(2)
```

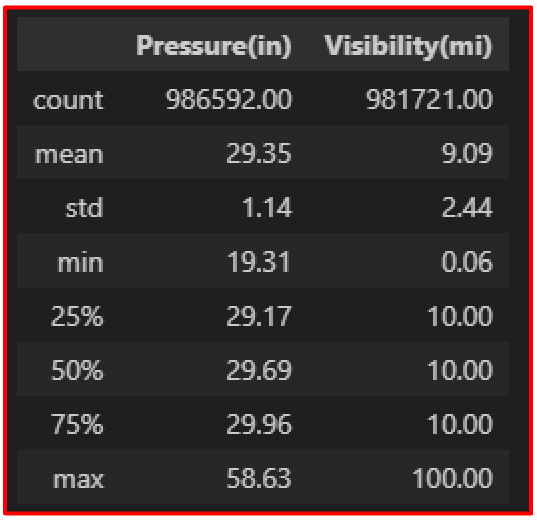


	Pressure(in)	Visibility(mi)
count	988086.00	983223.00
mean	29.35	9.07
std	1.15	2.47
min	19.31	0.00
25%	29.17	10.00
50%	29.69	10.00
75%	29.96	10.00
max	58.63	100.00

We can see that the minimum value is 0, meaning that some records are missing them and replaced them by putting zeros. For this reason, we are going to drop the records with missing values for these two columns.

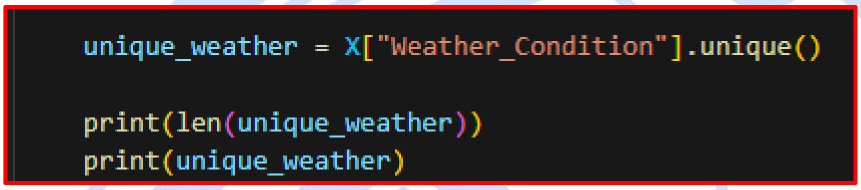


```
X = X[X["Pressure(in)"] != 0]
X = X[X["Visibility(mi)"] != 0]
X[["Pressure(in)", "Visibility(mi)"]].describe().round(2)
```



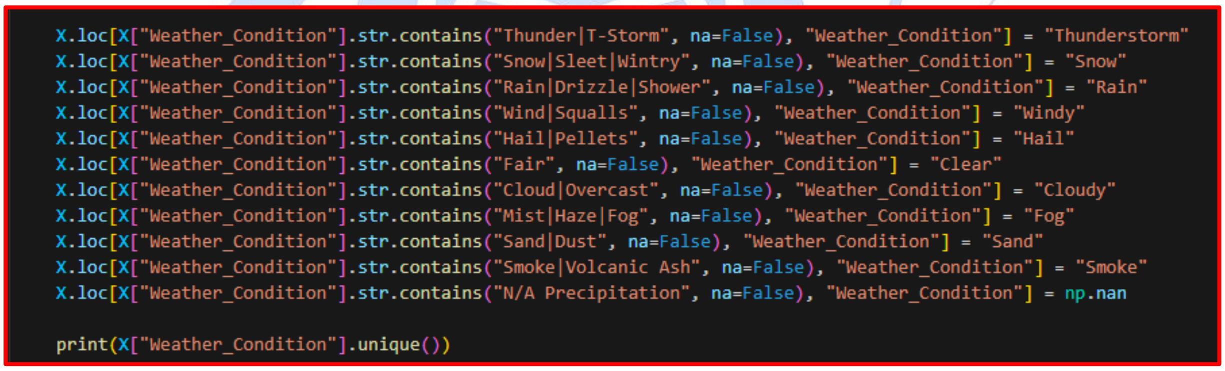
	Pressure(in)	Visibility(mi)
count	986592.00	981721.00
mean	29.35	9.09
std	1.14	2.44
min	19.31	0.06
25%	29.17	10.00
50%	29.69	10.00
75%	29.96	10.00
max	58.63	100.00

If we analyze the weather conditions, we can see that there are lots of them, so it's better to reduce the number of unique conditions.

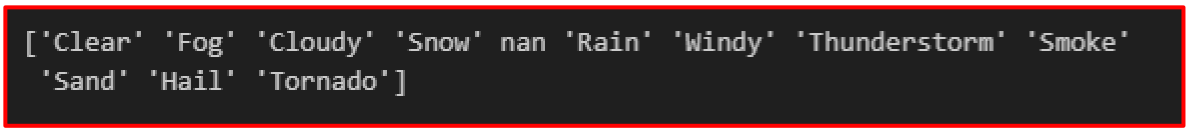


```
unique_weather = X["Weather_Condition"].unique()

print(len(unique_weather))
print(unique_weather)
```

To do so, we are going to replace them with a more generic description:

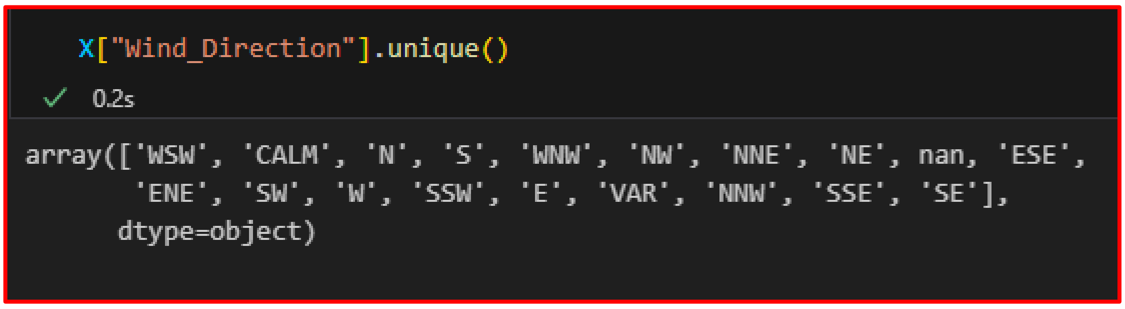


```
X.loc[X["Weather_Condition"].str.contains("Thunder|T-Storm", na=False), "Weather_Condition"] = "Thunderstorm"
X.loc[X["Weather_Condition"].str.contains("Snow|Sleet|Wintry", na=False), "Weather_Condition"] = "Snow"
X.loc[X["Weather_Condition"].str.contains("Rain|Drizzle|Shower", na=False), "Weather_Condition"] = "Rain"
X.loc[X["Weather_Condition"].str.contains("Wind|Squalls", na=False), "Weather_Condition"] = "Windy"
X.loc[X["Weather_Condition"].str.contains("Hail|Pellets", na=False), "Weather_Condition"] = "Hail"
X.loc[X["Weather_Condition"].str.contains("Fair", na=False), "Weather_Condition"] = "Clear"
X.loc[X["Weather_Condition"].str.contains("Cloud|Overcast", na=False), "Weather_Condition"] = "Cloudy"
X.loc[X["Weather_Condition"].str.contains("Mist|Haze|Fog", na=False), "Weather_Condition"] = "Fog"
X.loc[X["Weather_Condition"].str.contains("Sand|Dust", na=False), "Weather_Condition"] = "Sand"
X.loc[X["Weather_Condition"].str.contains("Smoke|Volcanic Ash", na=False), "Weather_Condition"] = "Smoke"
X.loc[X["Weather_Condition"].str.contains("N/A Precipitation", na=False), "Weather_Condition"] = np.nan

print(X["Weather_Condition"].unique())
```



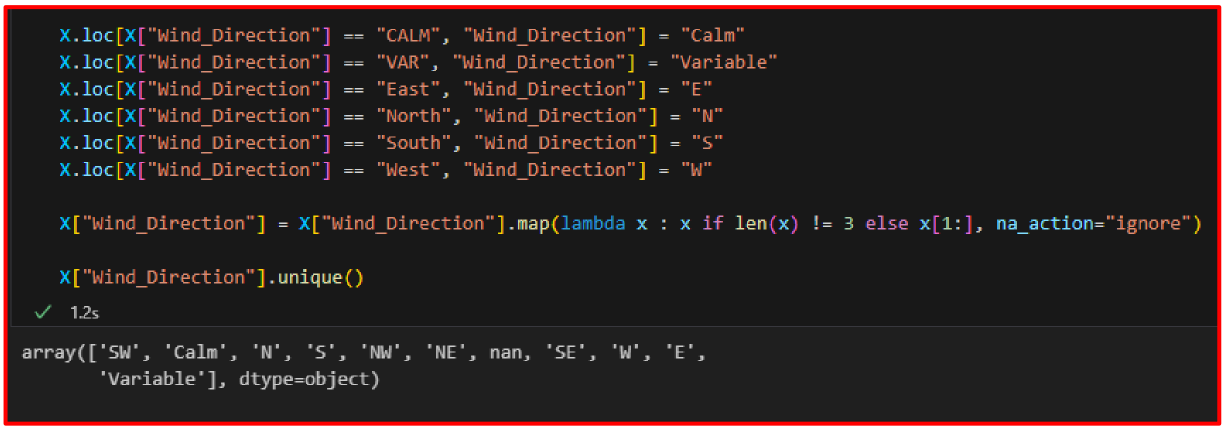
```
['Clear' 'Fog' 'Cloudy' 'Snow' nan 'Rain' 'Windy' 'Thunderstorm' 'Smoke'
 'Sand' 'Hail' 'Tornado']
```

Let's check also the Wind_Direction field:



```
X["Wind_Direction"].unique()
✓ 0.2s

array(['WSW', 'CALM', 'N', 'S', 'WNW', 'NW', 'NNE', 'NE', nan, 'ESE',
       'ENE', 'SW', 'W', 'SSW', 'E', 'VAR', 'NNW', 'SSE', 'SE'],
      dtype=object)
```

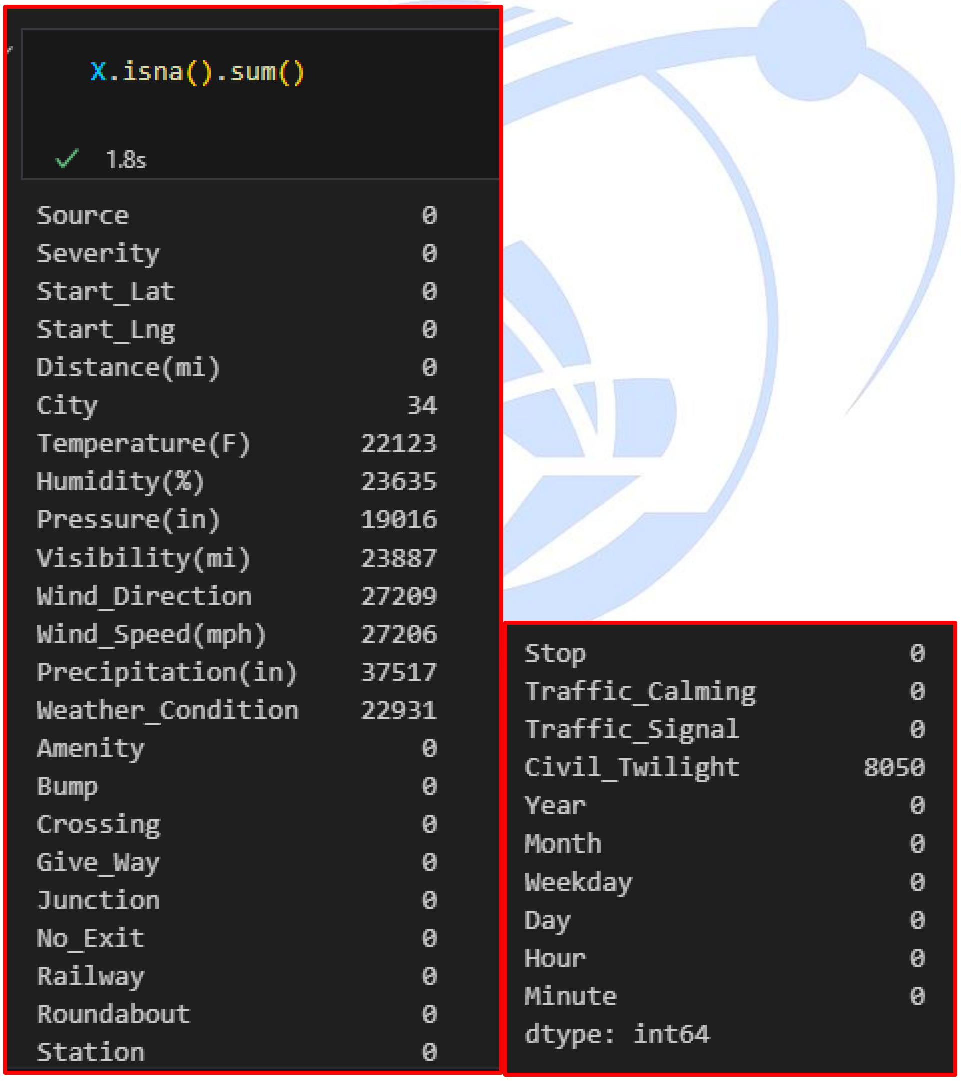
As we can see, we can group the values like we did with Weather_Condition:



```
X.loc[X["Wind_Direction"] == "CALM", "Wind_Direction"] = "Calm"
X.loc[X["Wind_Direction"] == "VAR", "Wind_Direction"] = "Variable"
X.loc[X["Wind_Direction"] == "East", "Wind_Direction"] = "E"
X.loc[X["Wind_Direction"] == "North", "Wind_Direction"] = "N"
X.loc[X["Wind_Direction"] == "South", "Wind_Direction"] = "S"
X.loc[X["Wind_Direction"] == "West", "Wind_Direction"] = "W"

X["Wind_Direction"] = X["Wind_Direction"].map(lambda x : x if len(x) != 3 else x[1:], na_action="ignore")

X["Wind_Direction"].unique()
```

```
array(['SW', 'Calm', 'N', 'S', 'NW', 'NE', nan, 'SE', 'W', 'E',
       'Variable'], dtype=object)
```

Next, let's analyze the missing values:

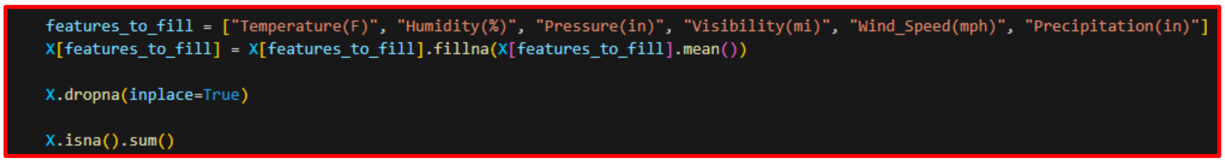


```
X.isna().sum()
```

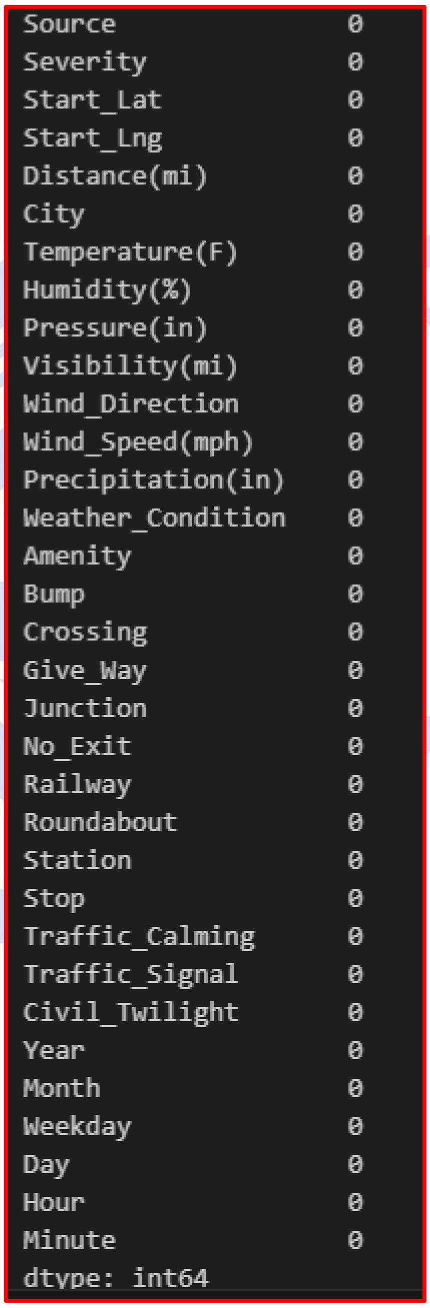
```
Source                   0
Severity                 0
Start_Lat                0
Start_Lng                0
Distance(mi)             0
City                    34
Temperature(F)       22123
Humidity(%)          23635
Pressure(in)         19016
Visibility(mi)       23887
Wind_Direction       27209
Wind_Speed(mph)      27206
Precipitation(in)    37517
Weather_Condition    22931
Amenity                  0
Bump                     0
Crossing                 0
Give_Way                 0
Junction                 0
No_Exit                  0
Railway                  0
Roundabout               0
Station                  0
Stop                     0
Traffic_Calming          0
Traffic_Signal           0
Civil_Twilight        8050
Year                     0
Month                    0
Weekday                  0
Day                      0
Hour                     0
Minute                   0
dtype: int64
```

Since a lot of records do not have informations about Precipitation, we are going to drop the feature.

For numerical features we are going to fill the missing features with the mean, while for categorical features like City, Wind_Direction, Weather_Condition and Civil_Twilight, we are going to delete the records with missing informations.



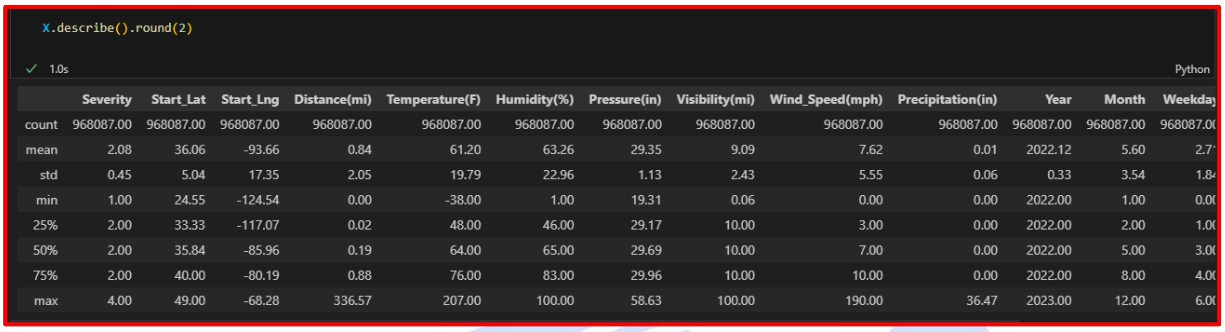
```
features_to_fill = ["Temperature(F)", "Humidity(%)", "Pressure(in)", "Visibility(mi)", "Wind_Speed(mph)", "Precipitation(in)"]
X[features_to_fill] = X[features_to_fill].fillna(X[features_to_fill].mean())

X.dropna(inplace=True)

X.isna().sum()
```



```
Source               0
Severity             0
Start_Lat            0
Start_Lng            0
Distance(mi)         0
City                 0
Temperature(F)       0
Humidity(%)          0
Pressure(in)         0
Visibility(mi)       0
Wind_Direction       0
Wind_Speed(mph)      0
Precipitation(in)    0
Weather_Condition    0
Amenity              0
Bump                 0
Crossing             0
Give_Way             0
Junction             0
No_Exit              0
Railway              0
Roundabout           0
Station              0
Stop                 0
Traffic_Calming      0
Traffic_Signal       0
Civil_Twilight       0
Year                 0
Month                0
Weekday              0
Day                  0
Hour                 0
Minute               0
dtype: int64
```

2.6. Check features variance

In this section we are going to check the variance for each feature in order to remove features with a very low variance beacuse they can't help to discriminate instances.

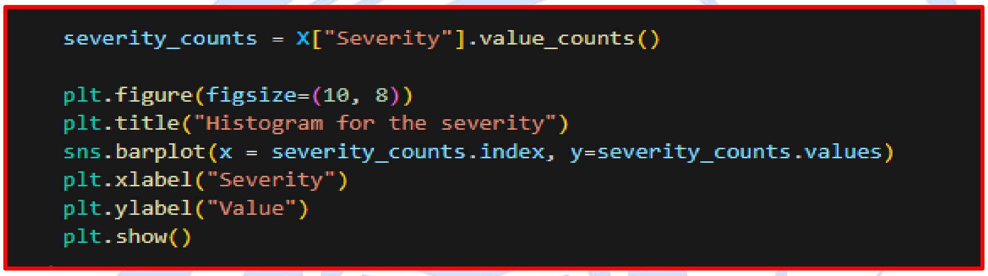


```python
X.describe().round(2)
```

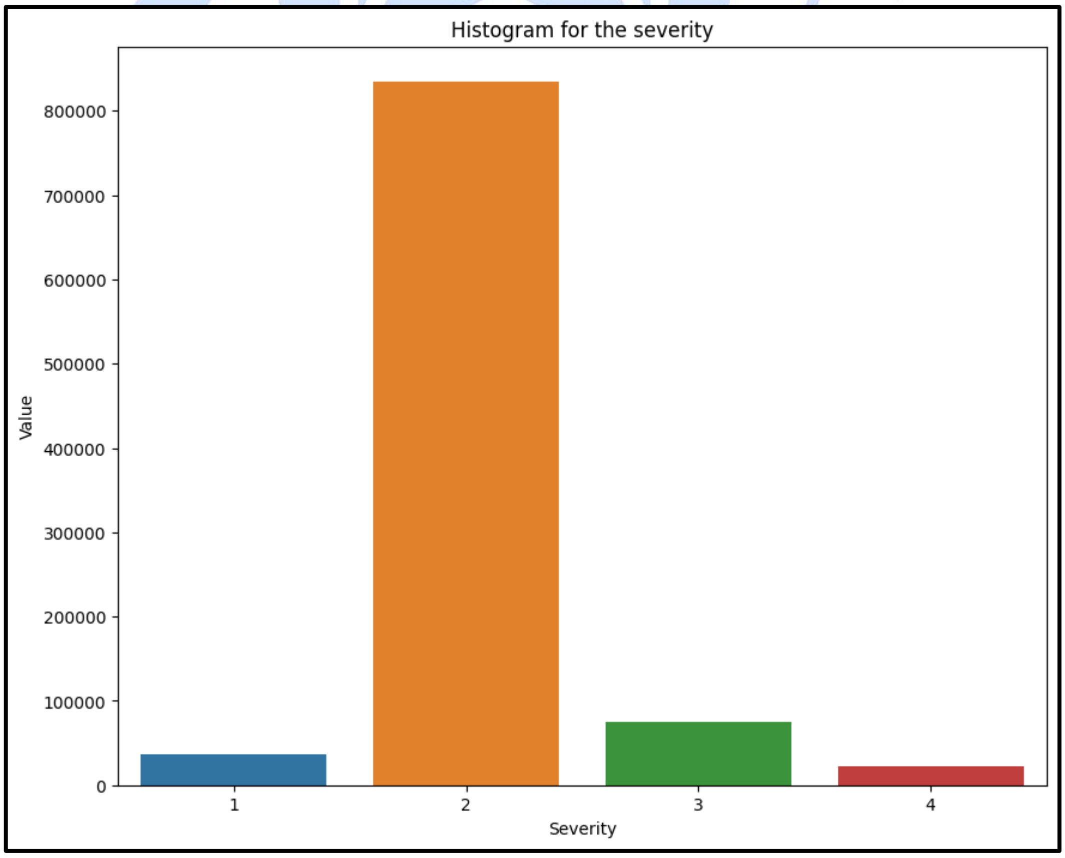
✓ 1.0s Python

	Severity	Start_Lat	Start_Lng	Distance(mi)	Temperature(F)	Humidity(%)	Pressure(in)	Visibility(mi)	Wind_Speed(mph)	Precipitation(in)	Year	Month	Weekda
count	968087.00	968087.00	968087.00	968087.00	968087.00	968087.00	968087.00	968087.00	968087.00	968087.00	968087.00	968087.00	968087.0
mean	2.08	36.06	-93.66	0.84	61.20	63.26	29.35	9.09	7.62	0.01	2022.12	5.60	2.7
std	0.45	5.04	17.35	2.05	19.79	22.96	1.13	2.43	5.55	0.06	0.33	3.54	1.8
min	1.00	24.55	-124.54	0.00	-38.00	1.00	19.31	0.06	0.00	0.00	2022.00	1.00	0.0
25%	2.00	33.33	-117.07	0.02	48.00	46.00	29.17	10.00	3.00	0.00	2022.00	2.00	1.0
50%	2.00	35.84	-85.96	0.19	64.00	65.00	29.69	10.00	7.00	0.00	2022.00	5.00	3.0
75%	2.00	40.00	-80.19	0.88	76.00	83.00	29.96	10.00	10.00	0.00	2022.00	8.00	4.0
max	4.00	49.00	-68.28	336.57	207.00	100.00	58.63	100.00	190.00	36.47	2023.00	12.00	6.0

2.7. Handle unbalanced data

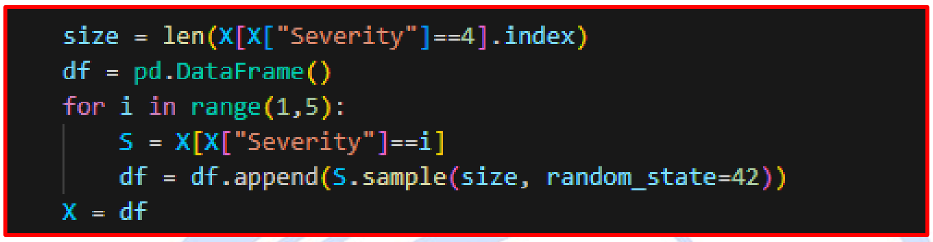


```python
severity_counts = X["Severity"].value_counts()

plt.figure(figsize=(10, 8))
plt.title("Histogram for the severity")
sns.barplot(x = severity_counts.index, y=severity_counts.values)
plt.xlabel("Severity")
plt.ylabel("Value")
plt.show()
```

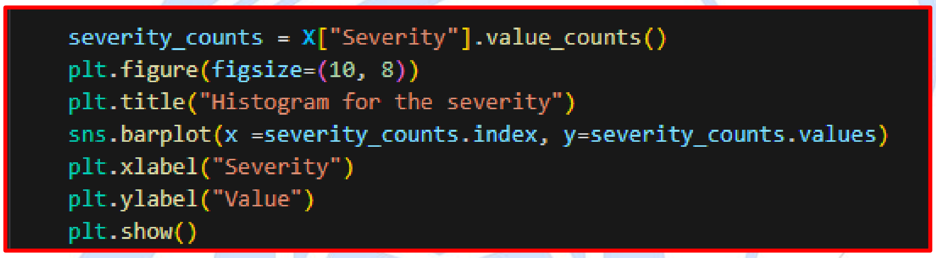


The severity attribute as we can see from the previous plot is highly unbalanced, the number of accident with the severity 1 is very small instead the number of accident with severity 2 is much higher

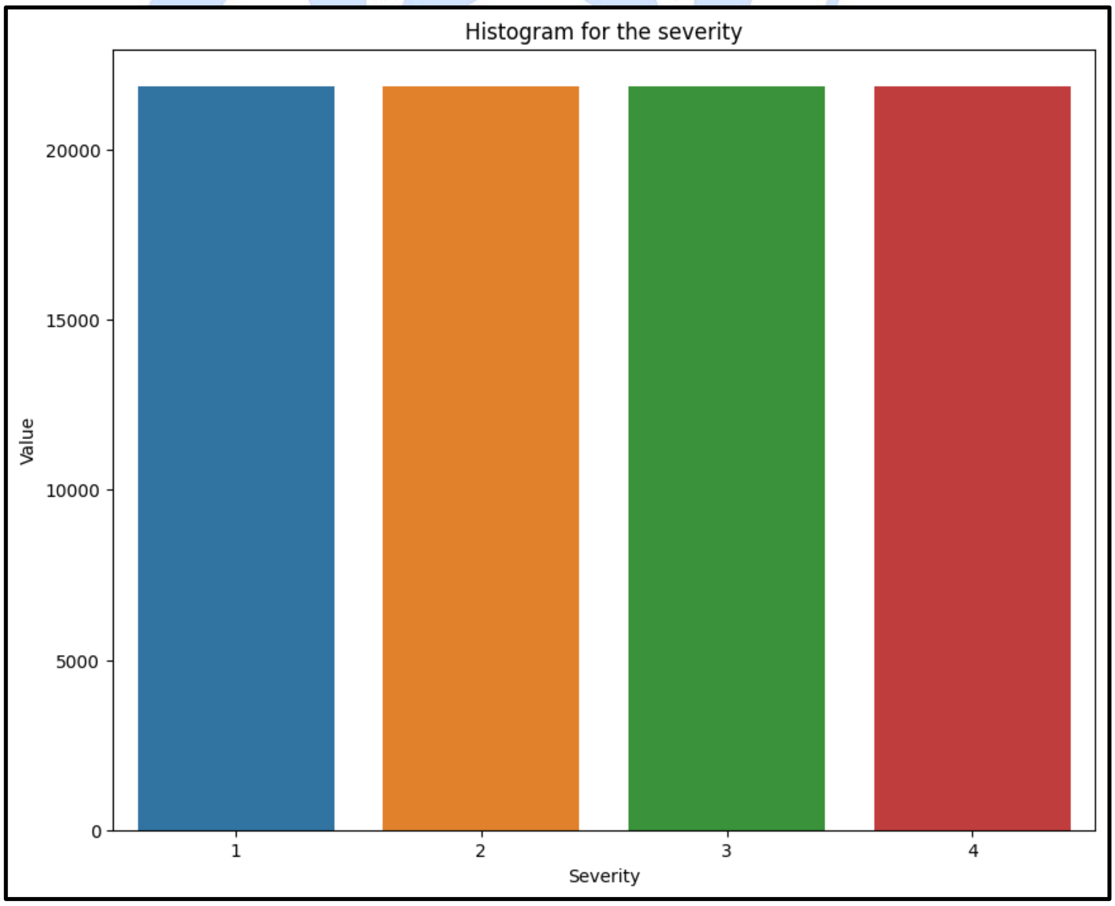
So, in order to balance the data we are going to undersample all the categories to the number of records of the minority category, in this case the severity 4. We thought this was a good choice since this leaves us with a good amount of records for each category.



```python
size = len(X[X["Severity"]==4].index)
df = pd.DataFrame()
for i in range(1,5):
    S = X[X["Severity"]==i]
    df = df.append(S.sample(size, random_state=42))
X = df
```

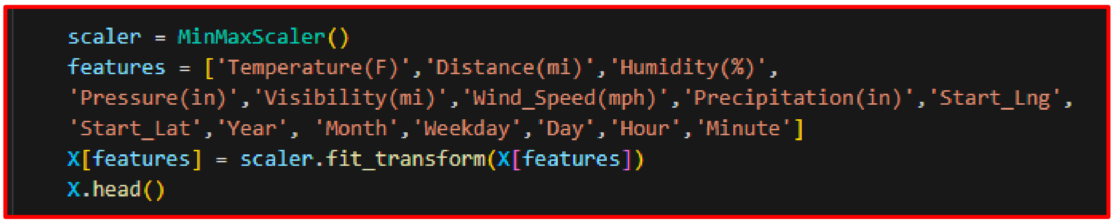


```python
severity_counts = X["Severity"].value_counts()
plt.figure(figsize=(10, 8))
plt.title("Histogram for the severity")
sns.barplot(x =severity_counts.index, y=severity_counts.values)
plt.xlabel("Severity")
plt.ylabel("Value")
plt.show()
```

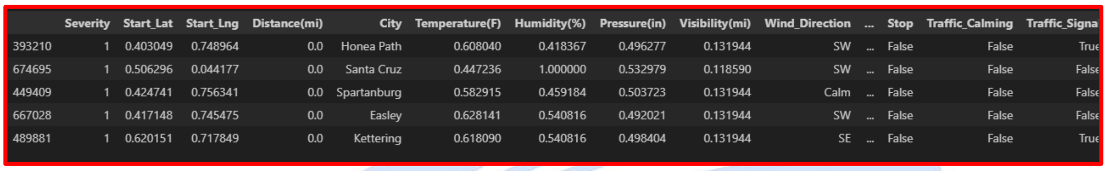


2.8. Feature scaling

In this section we are going to scale and normalize the features.To improve the performance of our models, we normalized the values of the continuous features.



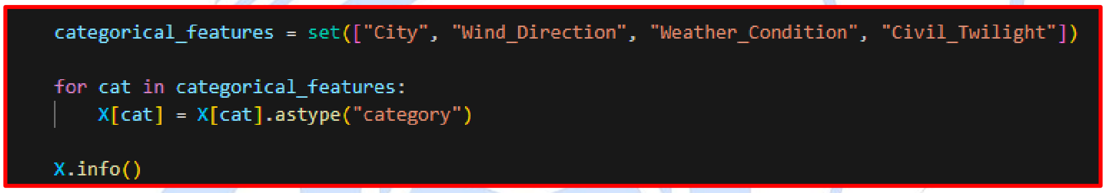
```python
scaler = MinMaxScaler()
features = ['Temperature(F)','Distance(mi)','Humidity(%)',
'Pressure(in)','Visibility(mi)','Wind_Speed(mph)','Precipitation(in)','Start_Lng',
'Start_Lat','Year', 'Month','Weekday','Day','Hour','Minute']
X[features] = scaler.fit_transform(X[features])
X.head()
```



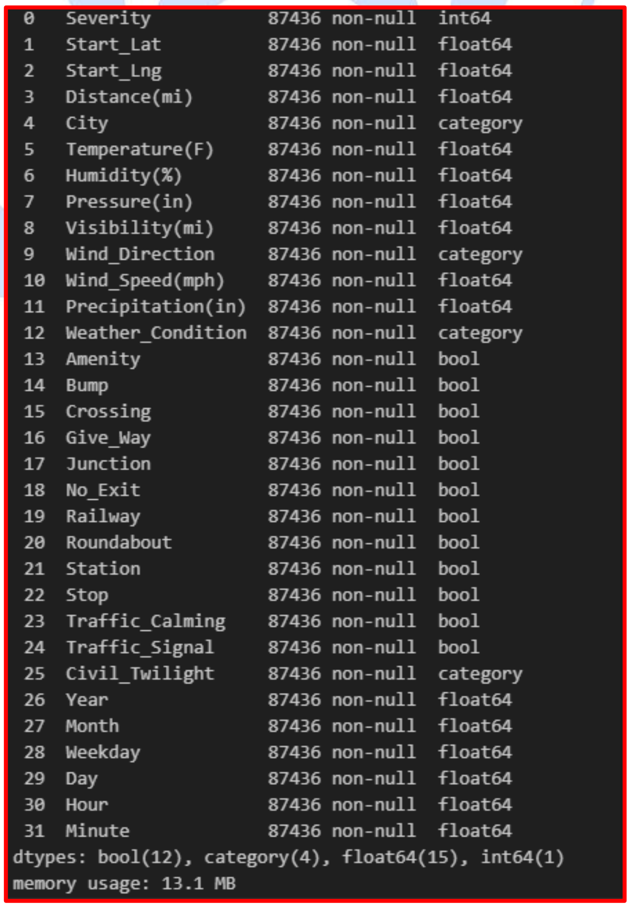
	Severity	Start_Lat	Start_Lng	Distance(mi)	City	Temperature(F)	Humidity(%)	Pressure(in)	Visibility(mi)	Wind_Direction	...	Stop	Traffic_Calming	Traffic_Signa
393210	1	0.403049	0.748964	0.0	Honea Path	0.608040	0.418367	0.496277	0.131944	SW	...	False	False	True
674695	1	0.506296	0.044177	0.0	Santa Cruz	0.447236	1.000000	0.532979	0.118590	SW	...	False	False	False
449409	1	0.424741	0.756341	0.0	Spartanburg	0.582915	0.459184	0.503723	0.131944	Calm	...	False	False	False
667028	1	0.417148	0.745475	0.0	Easley	0.628141	0.540816	0.492021	0.131944	SW	...	False	False	False
489881	1	0.620151	0.717849	0.0	Kettering	0.618090	0.540816	0.498404	0.131944	SE	...	False	False	True

2.9. Feature encoding

Finally, in this section we are going to encode the categorical features.

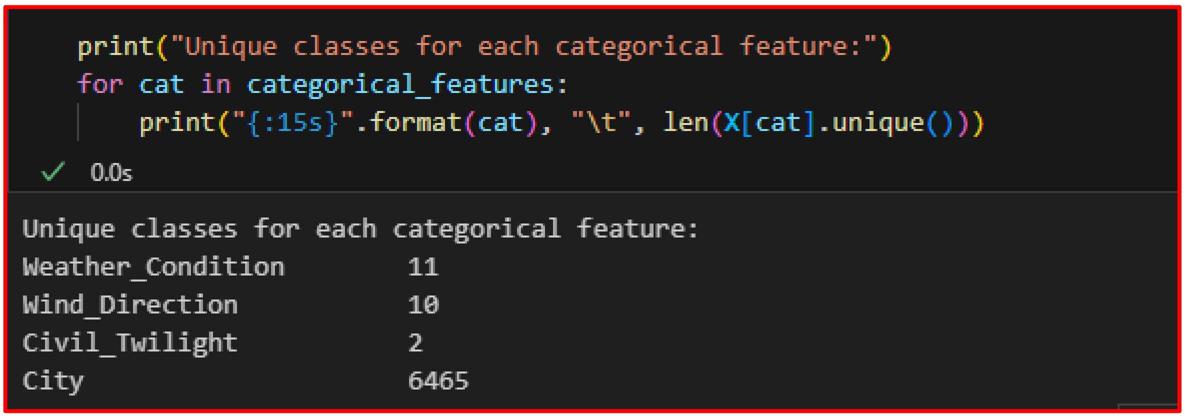


```python
categorical_features = set(["City", "Wind_Direction", "Weather_Condition", "Civil_Twilight"])

for cat in categorical_features:
    X[cat] = X[cat].astype("category")

X.info()
```



```
 0   Severity           87436 non-null  int64
 1   Start_Lat          87436 non-null  float64
 2   Start_Lng          87436 non-null  float64
 3   Distance(mi)       87436 non-null  float64
 4   City               87436 non-null  category
 5   Temperature(F)     87436 non-null  float64
 6   Humidity(%)        87436 non-null  float64
 7   Pressure(in)       87436 non-null  float64
 8   Visibility(mi)     87436 non-null  float64
 9   Wind_Direction     87436 non-null  category
 10  Wind_Speed(mph)    87436 non-null  float64
 11  Precipitation(in)  87436 non-null  float64
 12  Weather_Condition  87436 non-null  category
 13  Amenity            87436 non-null  bool
 14  Bump               87436 non-null  bool
 15  Crossing           87436 non-null  bool
 16  Give_Way           87436 non-null  bool
 17  Junction           87436 non-null  bool
 18  No_Exit            87436 non-null  bool
 19  Railway            87436 non-null  bool
 20  Roundabout         87436 non-null  bool
 21  Station            87436 non-null  bool
 22  Stop               87436 non-null  bool
 23  Traffic_Calming    87436 non-null  bool
 24  Traffic_Signal     87436 non-null  bool
 25  Civil_Twilight     87436 non-null  category
 26  Year               87436 non-null  float64
 27  Month              87436 non-null  float64
 28  Weekday            87436 non-null  float64
 29  Day                87436 non-null  float64
 30  Hour               87436 non-null  float64
 31  Minute             87436 non-null  float64
dtypes: bool(12), category(4), float64(15), int64(1)
memory usage: 13.1 MB
```

First of all, we show the number of unique classes for each categorical feature.

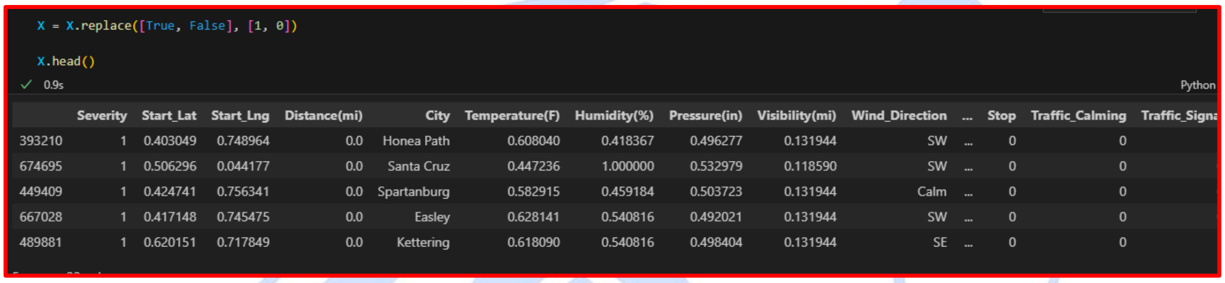


```
print("Unique classes for each categorical feature:")
for cat in categorical_features:
    print("{:15s}".format(cat), "\t", len(X[cat].unique()))
```

✓ 0.0s

```
Unique classes for each categorical feature:
Weather_Condition        11
Wind_Direction           10
Civil_Twilight           2
City                     6465
```

Let's first encode the boolean values in a numerical form.

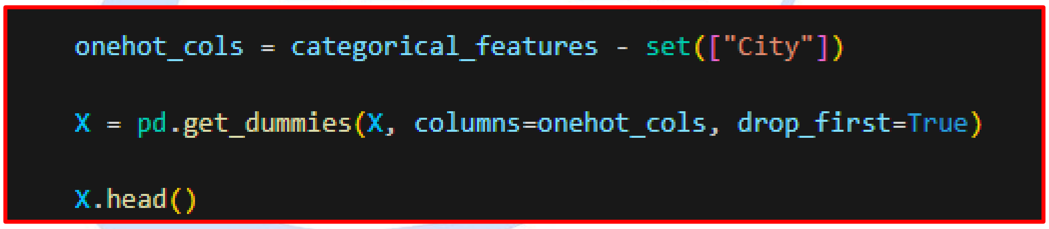


```
X = X.replace([True, False], [1, 0])

X.head()
```

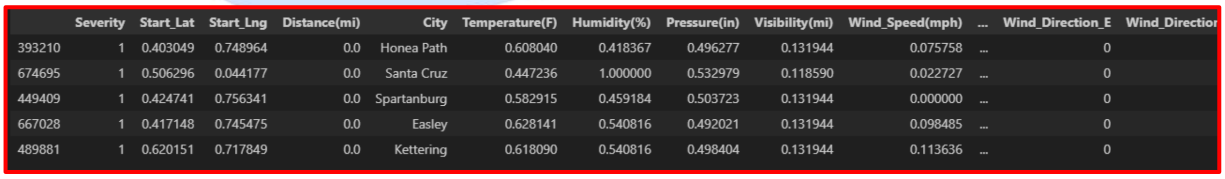
✓ 0.9s Python

	Severity	Start_Lat	Start_Lng	Distance(mi)	City	Temperature(F)	Humidity(%)	Pressure(in)	Visibility(mi)	Wind_Direction	...	Stop	Traffic_Calming	Traffic_Signa
393210	1	0.403049	0.748964	0.0	Honea Path	0.608040	0.418367	0.496277	0.131944	SW	...	0	0	
674695	1	0.506296	0.044177	0.0	Santa Cruz	0.447236	1.000000	0.532979	0.118590	SW	...	0	0	
449409	1	0.424741	0.756341	0.0	Spartanburg	0.582915	0.459184	0.503723	0.131944	Calm	...	0	0	
667028	1	0.417148	0.745475	0.0	Easley	0.628141	0.540816	0.492021	0.131944	SW	...	0	0	
489881	1	0.620151	0.717849	0.0	Kettering	0.618090	0.540816	0.498404	0.131944	SE	...	0	0	

Now we can encode the categorical features using the method get_dummies() which converts the features with the one-hot encoding.

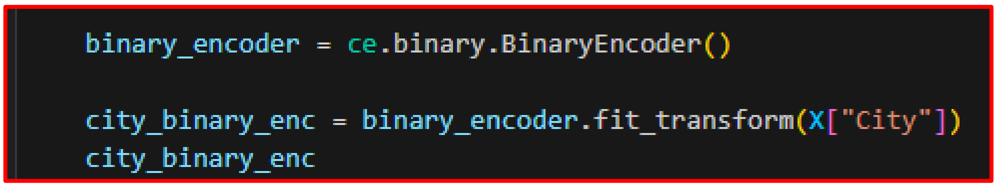


```
onehot_cols = categorical_features - set(["City"])

X = pd.get_dummies(X, columns=onehot_cols, drop_first=True)

X.head()
```

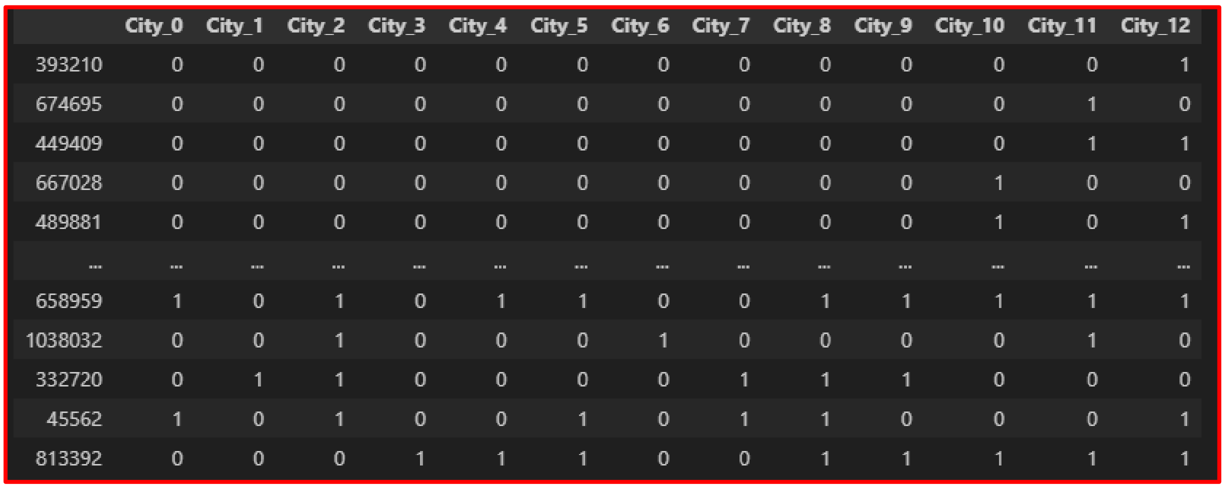


	Severity	Start_Lat	Start_Lng	Distance(mi)	City	Temperature(F)	Humidity(%)	Pressure(in)	Visibility(mi)	Wind_Speed(mph)	...	Wind_Direction_E	Wind_Directior
393210	1	0.403049	0.748964	0.0	Honea Path	0.608040	0.418367	0.496277	0.131944	0.075758	...	0	
674695	1	0.506296	0.044177	0.0	Santa Cruz	0.447236	1.000000	0.532979	0.118590	0.022727	...	0	
449409	1	0.424741	0.756341	0.0	Spartanburg	0.582915	0.459184	0.503723	0.131944	0.000000	...	0	
667028	1	0.417148	0.745475	0.0	Easley	0.628141	0.540816	0.492021	0.131944	0.098485	...	0	
489881	1	0.620151	0.717849	0.0	Kettering	0.618090	0.540816	0.498404	0.131944	0.113636	...	0	

Now, remains only to encode the City feature. In order to, reduce the usage of memory and the number of features we used the BinaryEncoder included in the library category_encoders.



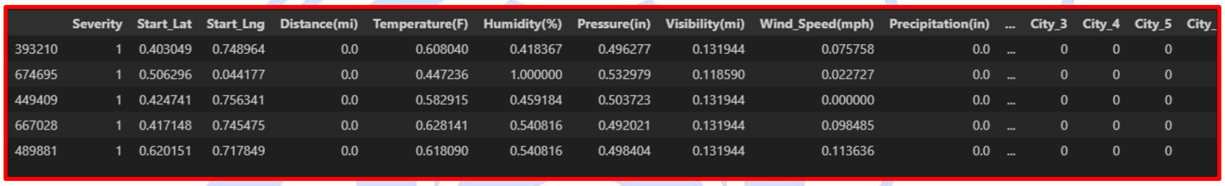
```
binary_encoder = ce.binary.BinaryEncoder()

city_binary_enc = binary_encoder.fit_transform(X["City"])
city_binary_enc
```



	City_0	City_1	City_2	City_3	City_4	City_5	City_6	City_7	City_8	City_9	City_10	City_11	City_12
393210	0	0	0	0	0	0	0	0	0	0	0	0	1
674695	0	0	0	0	0	0	0	0	0	0	0	1	0
449409	0	0	0	0	0	0	0	0	0	0	0	1	1
667028	0	0	0	0	0	0	0	0	0	0	1	0	0
489881	0	0	0	0	0	0	0	0	0	0	1	0	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...
658959	1	0	1	0	1	1	0	0	1	1	1	1	1
1038032	0	0	1	0	0	0	1	0	0	0	0	1	0
332720	0	1	1	0	0	0	0	1	1	1	0	0	0
45562	1	0	1	0	0	1	0	1	1	0	0	0	1
813392	0	0	0	1	1	1	0	0	1	1	1	1	1

Finally, we can merge the two dataframes and obtain the final dataframe X with the categorical features encoded.

```
X = pd.concat([X, city_binary_enc], axis=1).drop("City", axis=1)

X.head()
```



	Severity	Start_Lat	Start_Lng	Distance(mi)	Temperature(F)	Humidity(%)	Pressure(in)	Visibility(mi)	Wind_Speed(mph)	Precipitation(in)	...	City_3	City_4	City_5	City_
393210	1	0.403049	0.748964	0.0	0.608040	0.418367	0.496277	0.131944	0.075758	0.0	...	0	0	0	
674695	1	0.506296	0.044177	0.0	0.447236	1.000000	0.532979	0.118590	0.022727	0.0	...	0	0	0	
449409	1	0.424741	0.756341	0.0	0.582915	0.459184	0.503723	0.131944	0.000000	0.0	...	0	0	0	
667028	1	0.417148	0.745475	0.0	0.628141	0.540816	0.492021	0.131944	0.098485	0.0	...	0	0	0	
489881	1	0.620151	0.717849	0.0	0.618090	0.540816	0.498404	0.131944	0.113636	0.0	...	0	0	0	

3. APPLICATION OF DATA MINING ALGORITHMS FOR DATASET

3.1. Split data before building the model

After preprocessing the data, we obtained the final DataFrame X with the categorical features encoded.

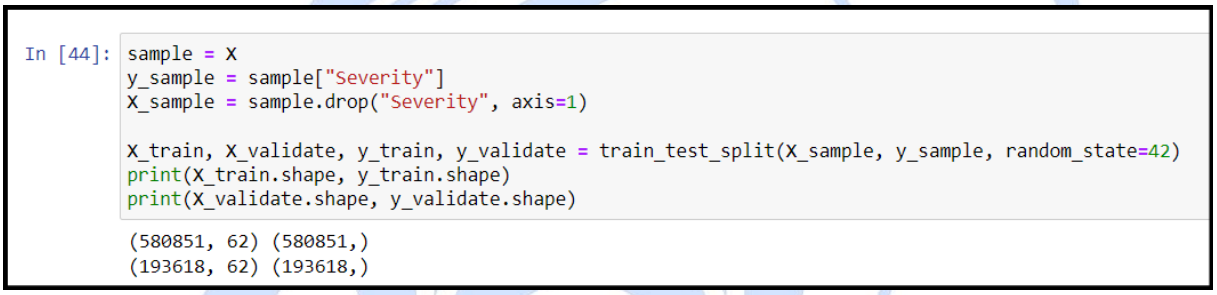
Step 1: Split the DataFrame X into two parts: a training/validation set (X) and a test set (X_test).

```
In [43]: # Train/Validation - Test split
         X, X_test = train_test_split(X, test_size=.2, random_state=42)
         print(X.shape, X_test.shape)

         (774469, 63) (193618, 63)
```

- The train_test_split() function from scikit-learn is used to perform the split. The test_size parameter is set to 0.2, which means that 20% of the data will be allocated to the test set and the remaining 80% will be used for training and validation.

- The random_state parameter is set to 42 to ensure that the split is reproducible.
- The resulting shapes of the two sets are printed using the shape attribute of each DataFrame, which returns a tuple containing the number of rows and columns (in that order).

Step 2: Assign a reference to the DataFrame X to a new variable sample. Split the DataFrame sample into two parts: a training set (X_train and y_train) and a validation set (X_validate and y_validate), using the train_test_split() function from scikit-learn.



```
In [44]: sample = X
         y_sample = sample["Severity"]
         X_sample = sample.drop("Severity", axis=1)

         X_train, X_validate, y_train, y_validate = train_test_split(X_sample, y_sample, random_state=42)
         print(X_train.shape, y_train.shape)
         print(X_validate.shape, y_validate.shape)

         (580851, 62) (580851,)
         (193618, 62) (193618,)
```

- First, the DataFrame sample is assigned to a new variable sample. Then, the target variable Severity is extracted from sample and assigned to y_sample, while the remaining features are assigned to X_sample.
- Next, train_test_split() is used to split X_sample and y_sample into a training set and a validation set. The random_state parameter is set to 42 to ensure that the split is reproducible.
- Finally, the shapes of the resulting sets are printed using the shape attribute of each DataFrame or Series.

3.2. Using Logistic Regression algorithm

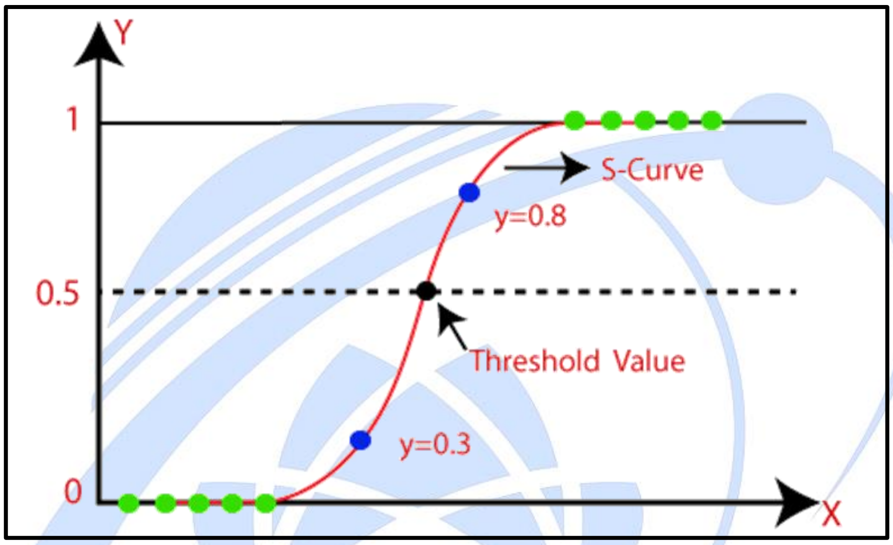
3.2.1. Logistic Regression algorithm introduction

Logistic regression is one of the most popular Machine Learning algorithms, which comes under the Supervised Learning technique. It is used for predicting the categorical dependent variable using a given set of independent variables.

The outcome must be a categorical or discrete value. It can be either Yes or No, 0 or 1, true or False, etc. but instead of giving the exact value as 0 and 1, it gives the probabilistic values which lie between 0 and 1.

For example, if we are modeling people's sex as male or female from their height, then the first class could be male and the logistic regression model could be written as the probability of male given a person's height, or more formally

The below image is showing the logistic function.



3.2.2. Selected reasons

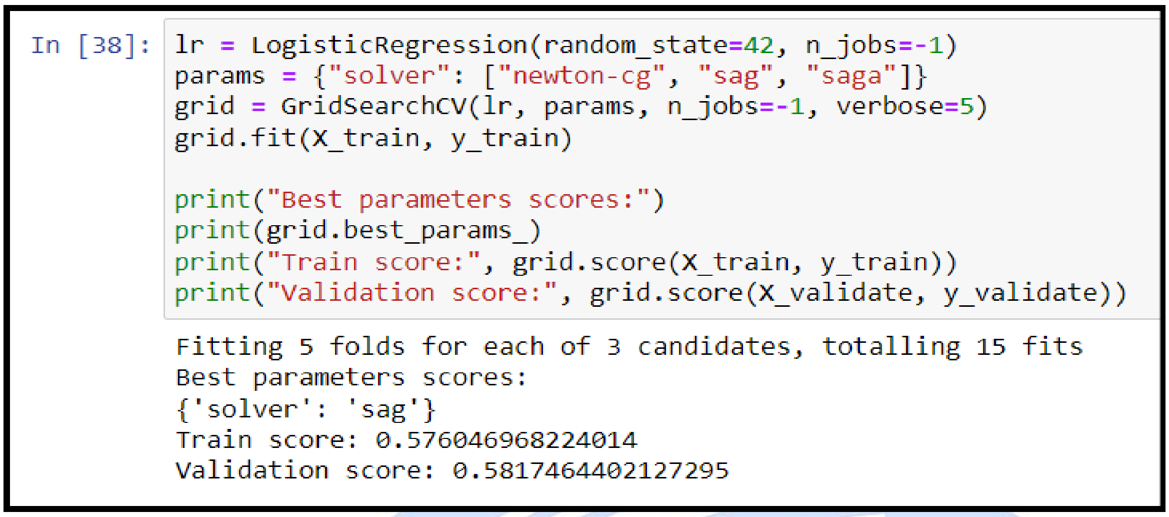
- Because this is a data classification algorithm, so it is suitable for the problem posed.
- The algorithm gives high accuracy and fast running time.

3.2.3. Build and predict with Logistic Regression model

Step 1:

- Implementing a logistic regression model with hyperparameter tuning using GridSearchCV.
- The LogisticRegression object is instantiated with random_state and n_jobs set.
- The hyperparameters to be tuned are specified in a dictionary called params.
- A GridSearchCV object is created and fit to the training data

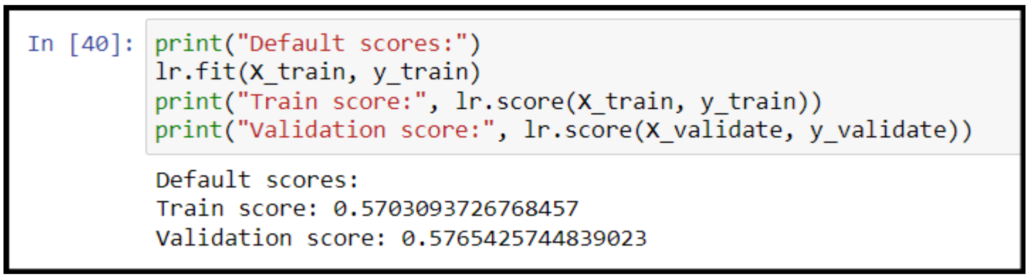
- The best hyperparameters are outputted along with the model's accuracy scores on the training and validation data.



```
In [38]: lr = LogisticRegression(random_state=42, n_jobs=-1)
         params = {"solver": ["newton-cg", "sag", "saga"]}
         grid = GridSearchCV(lr, params, n_jobs=-1, verbose=5)
         grid.fit(X_train, y_train)

         print("Best parameters scores:")
         print(grid.best_params_)
         print("Train score:", grid.score(X_train, y_train))
         print("Validation score:", grid.score(X_validate, y_validate))

         Fitting 5 folds for each of 3 candidates, totalling 15 fits
         Best parameters scores:
         {'solver': 'sag'}
         Train score: 0.576046968224014
         Validation score: 0.5817464402127295
```

- The output means that the logistic regression model was trained and evaluated using GridSearchCV with 5-fold cross-validation on 3 hyperparameter combinations (newton-cg, sag, saga). The best hyperparameter combination was found to be {'solver': 'sag'}.
- The accuracy score on the training data was 0.576, indicating that the model predicted the class labels correctly for 57.6% of the training examples.
- The accuracy score on the validation data was 0.582, indicating that the model predicted the class labels correctly for 58.2% of the validation examples.

Step 2: Fitting a logistic regression model (lr) to the training dataset (X_train and y_train) and printing the training and validation scores for the default hyperparameters.

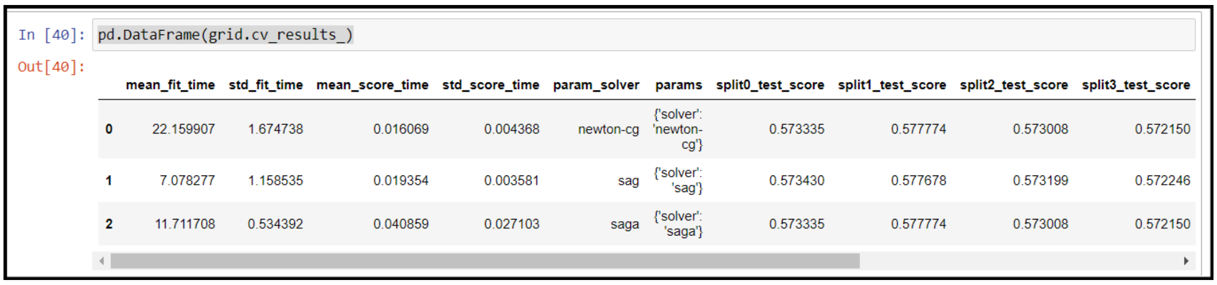


```
In [40]: print("Default scores:")
         lr.fit(X_train, y_train)
         print("Train score:", lr.score(X_train, y_train))
         print("Validation score:", lr.score(X_validate, y_validate))

         Default scores:
         Train score: 0.5703093726768457
         Validation score: 0.5765425744839023
```

- The training score of 0.570 indicates that the model correctly predicted 57% of the training samples.

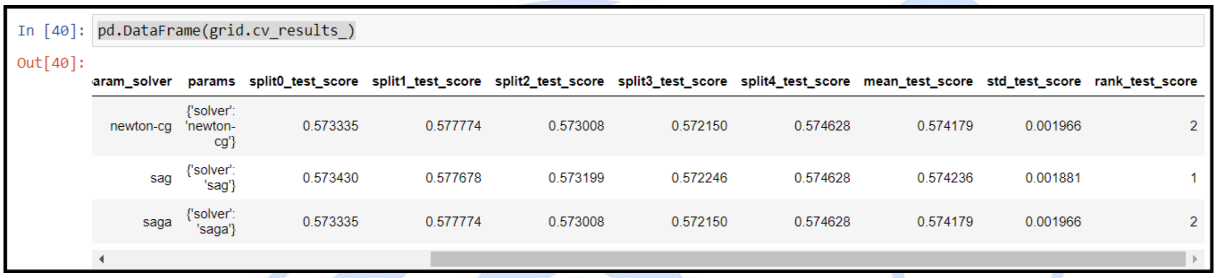
- The validation score of 0.577 indicates that the model correctly predicted 58% of the validation samples.

Step 3: Creates a Pandas DataFrame that contains the results of a grid search cross-validation.



```
In [40]: pd.DataFrame(grid.cv_results_)
Out[40]:
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_solver	params	split0_test_score	split1_test_score	split2_test_score	split3_test_score
0	22.159907	1.674738	0.016069	0.004368	newton-cg	{'solver': 'newton-cg'}	0.573335	0.577774	0.573008	0.572150
1	7.078277	1.158535	0.019354	0.003581	sag	{'solver': 'sag'}	0.573430	0.577678	0.573199	0.572246
2	11.711708	0.534392	0.040859	0.027103	saga	{'solver': 'saga'}	0.573335	0.577774	0.573008	0.572150



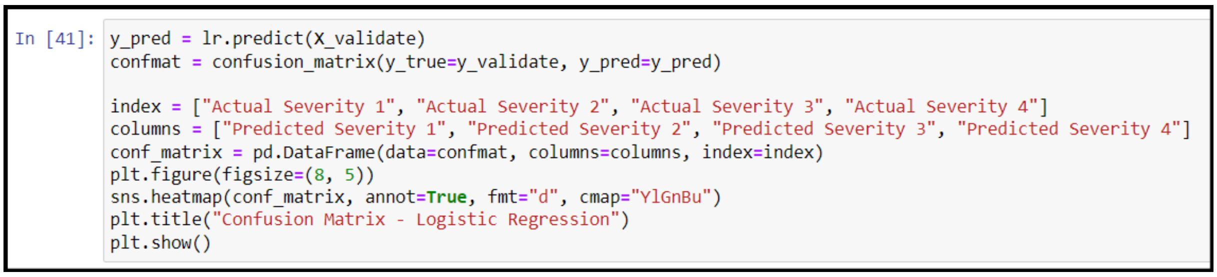
```
In [40]: pd.DataFrame(grid.cv_results_)
Out[40]:
```

aram_solver	params	split0_test_score	split1_test_score	split2_test_score	split3_test_score	split4_test_score	mean_test_score	std_test_score	rank_test_score
newton-cg	{'solver': 'newton-cg'}	0.573335	0.577774	0.573008	0.572150	0.574628	0.574179	0.001966	2
sag	{'solver': 'sag'}	0.573430	0.577678	0.573199	0.572246	0.574628	0.574236	0.001881	1
saga	{'solver': 'saga'}	0.573335	0.577774	0.573008	0.572150	0.574628	0.574179	0.001966	2

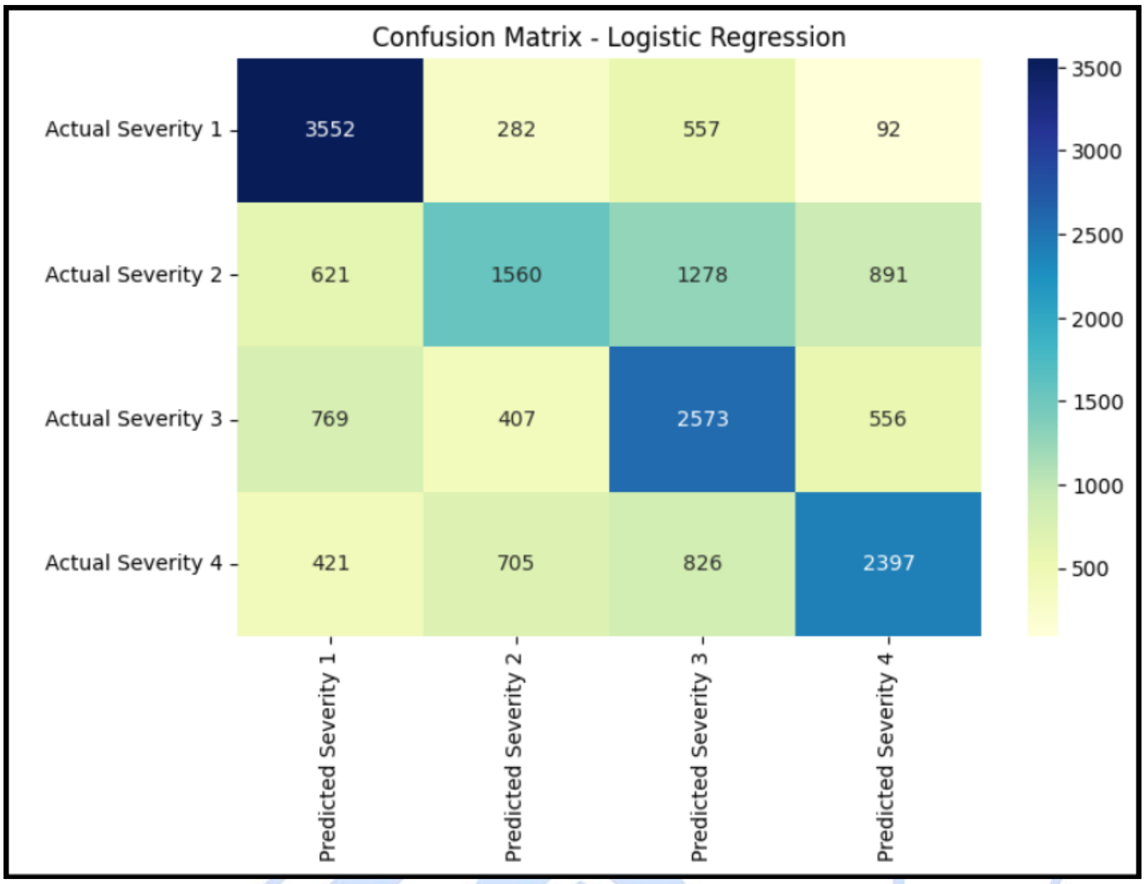
- The resulting DataFrame contains a row for each combination of hyperparameters and columns for various metrics, including the mean test score, standard deviation of the test score, and the values of the hyperparameter.

Step 4:

- Using a logistic regression model (lr) to predict the labels of the validation dataset (X_validate).
- Creating a confusion matrix to visualize the performance of the model. The confusion matrix is stored in a Pandas DataFrame and displayed using a heatmap from the Seaborn library.



```
In [41]: y_pred = lr.predict(X_validate)
         confmat = confusion_matrix(y_true=y_validate, y_pred=y_pred)

         index = ["Actual Severity 1", "Actual Severity 2", "Actual Severity 3", "Actual Severity 4"]
         columns = ["Predicted Severity 1", "Predicted Severity 2", "Predicted Severity 3", "Predicted Severity 4"]
         conf_matrix = pd.DataFrame(data=confmat, columns=columns, index=index)
         plt.figure(figsize=(8, 5))
         sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="YlGnBu")
         plt.title("Confusion Matrix - Logistic Regression")
         plt.show()
```



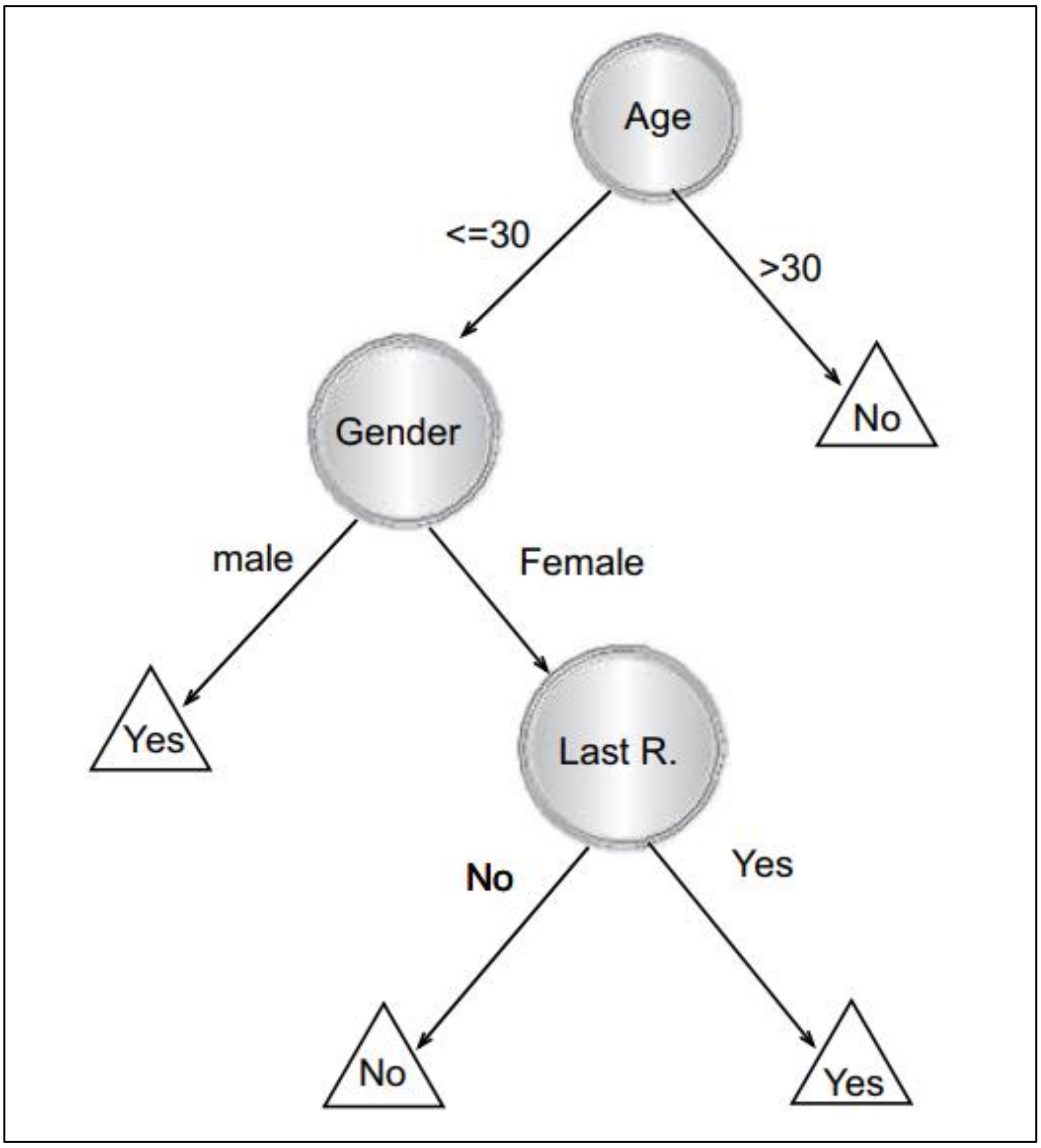
3.3. Using Decision Tree algorithm

3.3.1. Decision Tree Algorithm introduction

A decision tree is a classifier expressed as a recursive partition of the instance space. The decision tree consists of nodes that form a rooted tree, meaning it is a directed tree with a node called “root” that has no incoming edges. All other nodes have exactly one incoming edge. A node with outgoing edges is called an internal or test node. All other nodes are called leaves (also known as terminal or decision nodes). In a decision tree, each internal node splits the instance space into two or more sub-spaces according to a certain discrete function of the input attributes values. In the simplest and most frequent case, each test considers a single attribute, such that the instance space is partitioned according to the attribute’s value. In the case of numeric attributes, the condition refers to a range.

Each leaf is assigned to one class representing the most appropriate target value. Alternatively, the leaf may hold a probability vector indicating the probability of the target attribute having a certain value. Instances are classified by navigating them from the root of the tree down to a leaf, according to the outcome of the tests along the path.

For example, this figure describes a decision tree that reasons whether or not a potential customer will respond to a direct mailing. Internal nodes are represented as circles, whereas leaves are denoted as triangles.



[1]

3.3.2. Selected reasons

Some advantages of decision trees are:

- Simple to understand and to interpret. Trees can be visualized.

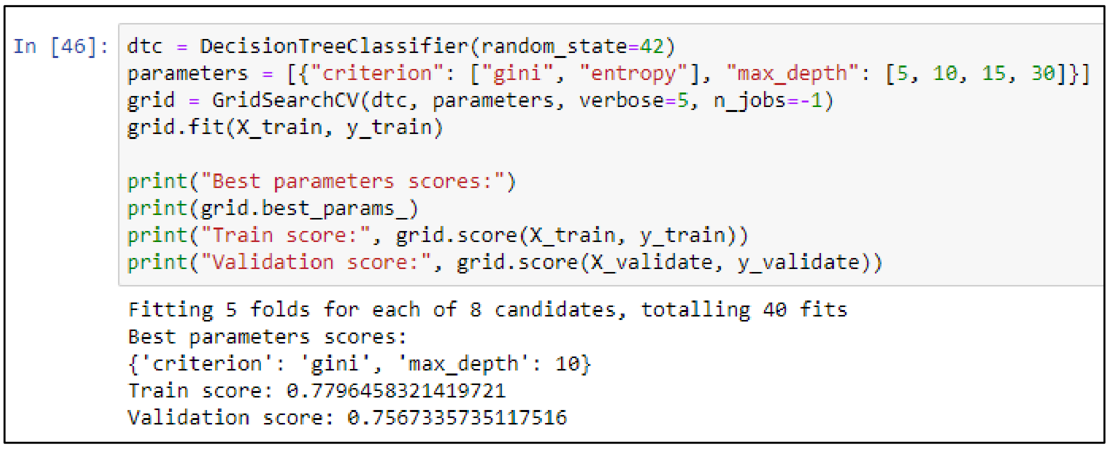
- Requires little data preparation. Other techniques often require data normalization, dummy variables need to be created and blank values to be removed. Note however that this module does not support missing values.
- The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.
- Able to handle both numerical and categorical data.
- Able to handle multi-output problems.
- Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.
- Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.
- Performs well even if its assumptions are somewhat violated by the true model from which the data were generated. [2]

3.3.3. Build and predict with Decision Tree model

Step 1:

- Create a decision tree model (dtc) using DecisionTreeClassifier from sklearn.tree, with the random_state parameter set to 42.
- Create a list of parameters to try in a grid search. This list includes two parameters: criterion (the method for measuring node impurity) and max_depth (the maximum depth of the tree).
- Create a grid search object with the decision tree model ‘dtc’, the parameters to try ‘parameters’, ‘verbose=5’ to print the grid search results, and n_jobs=-1 to use all available CPU cores.
- Train the model on the training set ‘X_train’ and ‘y_train’, and performs a grid search to find the best parameters for the model. Then print out the best parameters, the score on the training set and the score on the validation set

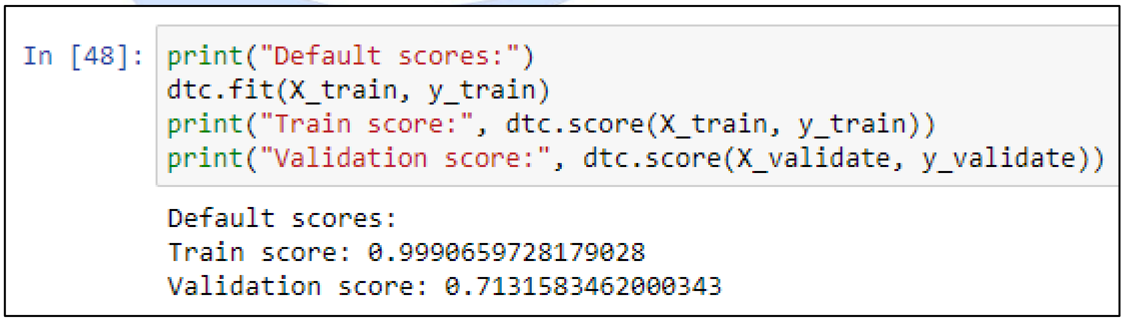
of the best model found by the grid search.



```
In [46]: dtc = DecisionTreeClassifier(random_state=42)
         parameters = [{"criterion": ["gini", "entropy"], "max_depth": [5, 10, 15, 30]}]
         grid = GridSearchCV(dtc, parameters, verbose=5, n_jobs=-1)
         grid.fit(X_train, y_train)

         print("Best parameters scores:")
         print(grid.best_params_)
         print("Train score:", grid.score(X_train, y_train))
         print("Validation score:", grid.score(X_validate, y_validate))

         Fitting 5 folds for each of 8 candidates, totalling 40 fits
         Best parameters scores:
         {'criterion': 'gini', 'max_depth': 10}
         Train score: 0.7796458321419721
         Validation score: 0.7567335735117516
```

- The output means that the Decision Tree model was trained and the best parameters scores are “criterion: gini” and “max_depth: 10”.
- The accuracy score on the training data was approximate 0.78, indicating that the model predicted the class labels correctly for 7.8% of the training examples.
- The accuracy score on the validation data was approximate 0.757, indicating that the model predicted the class labels correctly for 75.7% of the validation examples.

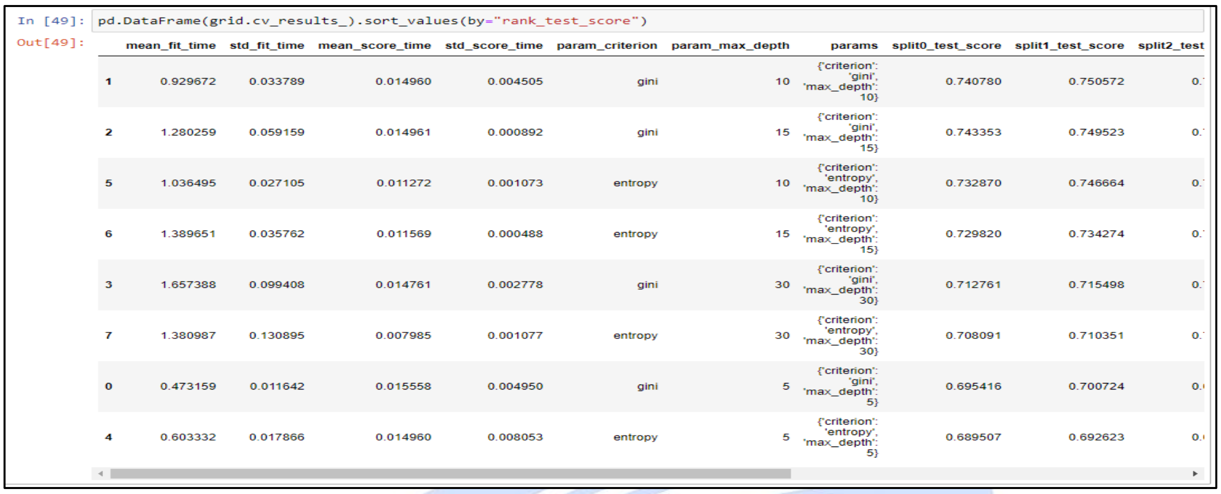
Step 2: Fitting the Decision Tree model (dtc) to the training dataset (X_train and y_train) and printing the training and validation scores for the default hyperparameters.



```
In [48]: print("Default scores:")
         dtc.fit(X_train, y_train)
         print("Train score:", dtc.score(X_train, y_train))
         print("Validation score:", dtc.score(X_validate, y_validate))

         Default scores:
         Train score: 0.9990659728179028
         Validation score: 0.7131583462000343
```

- The training score approximate of 0.999 indicates that the model correctly predicted 99.9% of the training samples.
- The validation score approximate of 0.713 indicates that the model correctly predicted 71.3% of the validation samples.

Step 3: Creates a DataFrame from the grid search results and sorts the results in ascending order of the validation score.

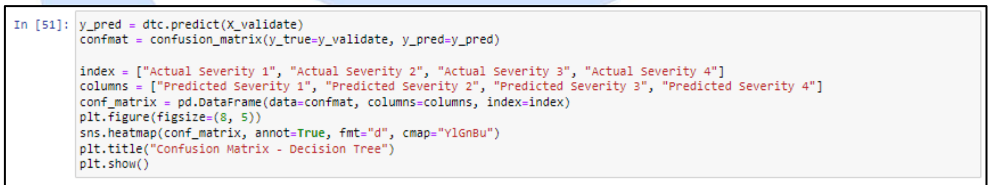


```
In [49]: pd.DataFrame(grid.cv_results_).sort_values(by="rank_test_score")
Out[49]:
```

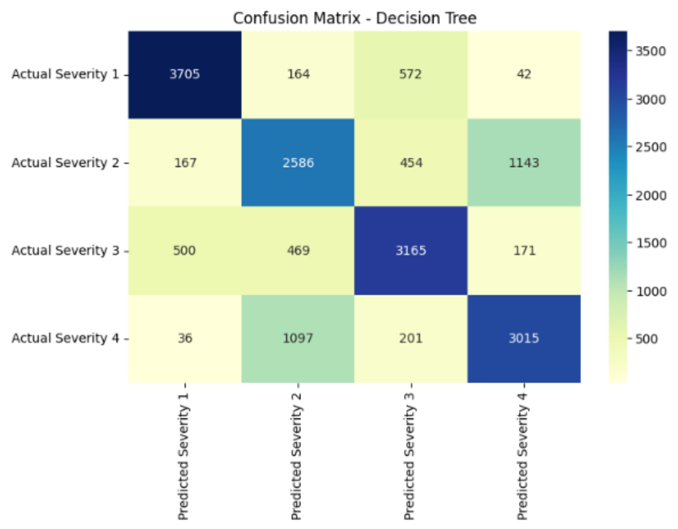
	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_criterion	param_max_depth	params	split0_test_score	split1_test_score	split2_test
1	0.929672	0.033789	0.014960	0.004505	gini	10	{'criterion': 'gini', 'max_depth': 10}	0.740780	0.750572	0.
2	1.280259	0.059159	0.014961	0.000892	gini	15	{'criterion': 'gini', 'max_depth': 15}	0.743353	0.749523	0.
5	1.036495	0.027105	0.011272	0.001073	entropy	10	{'criterion': 'entropy', 'max_depth': 10}	0.732870	0.746664	0.
6	1.389651	0.035762	0.011569	0.000488	entropy	15	{'criterion': 'entropy', 'max_depth': 15}	0.729820	0.734274	0.
3	1.657388	0.099408	0.014761	0.002778	gini	30	{'criterion': 'gini', 'max_depth': 30}	0.712761	0.715498	0.
7	1.380987	0.130895	0.007985	0.001077	entropy	30	{'criterion': 'entropy', 'max_depth': 30}	0.708091	0.710351	0.
0	0.473159	0.011642	0.015558	0.004950	gini	5	{'criterion': 'gini', 'max_depth': 5}	0.695416	0.700724	0.
4	0.603332	0.017866	0.014960	0.008053	entropy	5	{'criterion': 'entropy', 'max_depth': 5}	0.689507	0.692623	0.

Step 4:

- Predicts the labels of the validation set (X_validate) using the decision tree model (dtc) trained.
- Calculates the confusion matrix by comparing the true labels (y_validate) with the predicted labels (y_pred).
- Creating a confusion matrix to visualize the performance of the model. The confusion matrix is stored in a Pandas DataFrame and displayed using a heatmap from the Seaborn library.

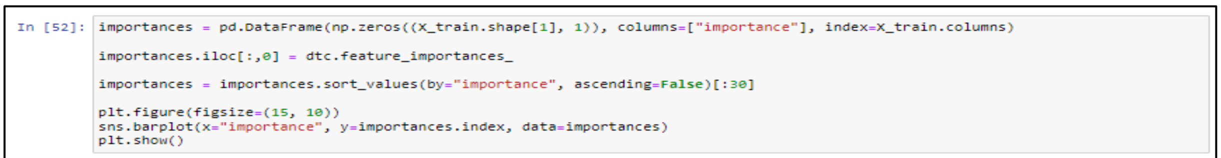


```
In [51]: y_pred = dtc.predict(X_validate)
         confmat = confusion_matrix(y_true=y_validate, y_pred=y_pred)

         index = ["Actual Severity 1", "Actual Severity 2", "Actual Severity 3", "Actual Severity 4"]
         columns = ["Predicted Severity 1", "Predicted Severity 2", "Predicted Severity 3", "Predicted Severity 4"]
         conf_matrix = pd.DataFrame(data=confmat, columns=columns, index=index)
         plt.figure(figsize=(8, 5))
         sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="YlGnBu")
         plt.title("Confusion Matrix - Decision Tree")
         plt.show()
```

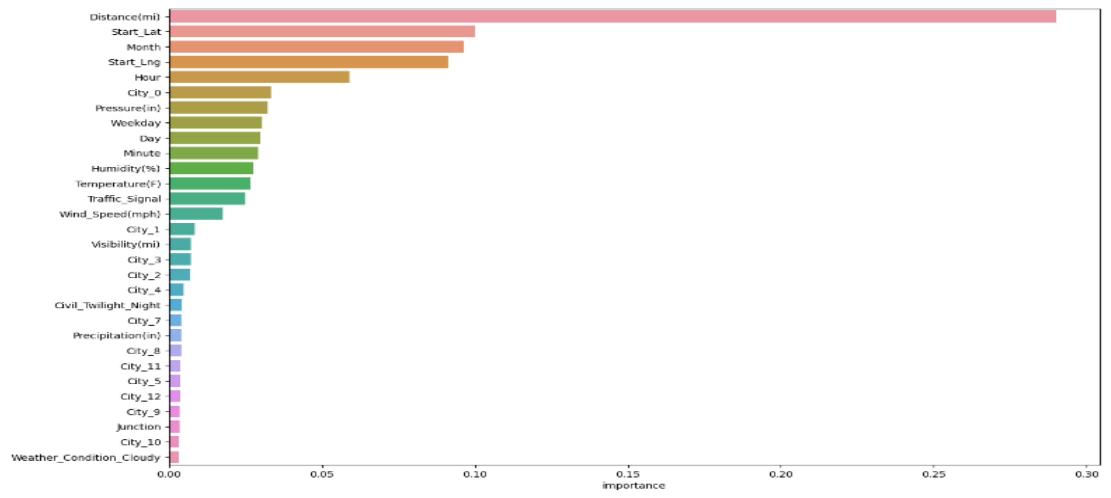


Step 5:

- Find out top 30 important features by creating a DataFrame with the number of rows equal to the number of columns in X_train and one column named “importance”, using the feature names in X_train as the index.

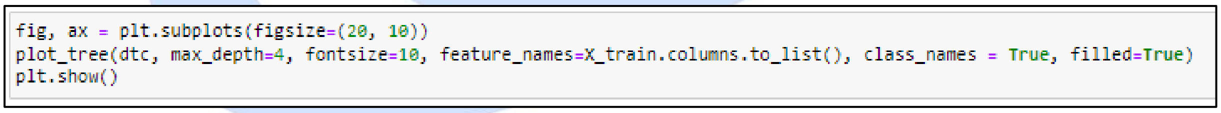


```
In [52]: importances = pd.DataFrame(np.zeros((X_train.shape[1], 1)), columns=["importance"], index=X_train.columns)

         importances.iloc[:,0] = dtc.feature_importances_

         importances = importances.sort_values(by="importance", ascending=False)[:30]

         plt.figure(figsize=(15, 10))
         sns.barplot(x="importance", y=importances.index, data=importances)
         plt.show()
```

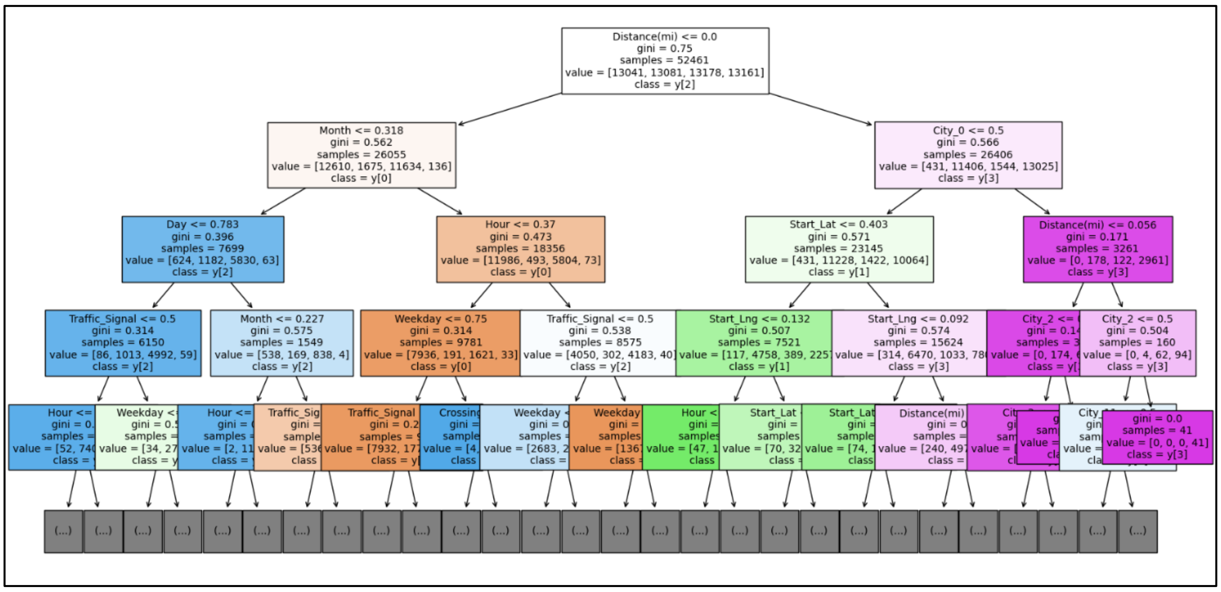


Step 6:

- Creates a visualization of the Decision Tree model ‘dtc’ using the plot_tree() function from sklearn.tree, with max_depth = 4.



```
fig, ax = plt.subplots(figsize=(20, 10))
plot_tree(dtc, max_depth=4, fontsize=10, feature_names=X_train.columns.to_list(), class_names = True, filled=True)
plt.show()
```



3.4. Using Random Forest algorithm

3.4.1. Random Forest Algorithm introduction

The Random Forest algorithm is a popular machine learning technique that is used for both classification and regression tasks. It is an ensemble learning method that combines the predictions of multiple decision trees to generate a final prediction. Random Forests are known for their robustness and high accuracy, making them widely used in various domains, including finance, healthcare, and natural language processing.

One of the advantages of Random Forests is their ability to handle large datasets with high dimensionality. They are also less prone to overfitting compared to individual decision trees. Furthermore, Random Forests can provide measures of feature importance, allowing us to identify the most influential features in the dataset.

However, it's worth noting that Random Forests can be computationally expensive, especially when dealing with a large number of trees or complex datasets. Additionally, they may not perform well on datasets with strong linear relationships, as decision trees are not naturally suited for capturing such patterns. Overall, the Random Forest algorithm is a powerful and versatile technique for machine learning tasks, offering a good balance between accuracy, interpretability, and robustness.

3.4.2. Selected reasons

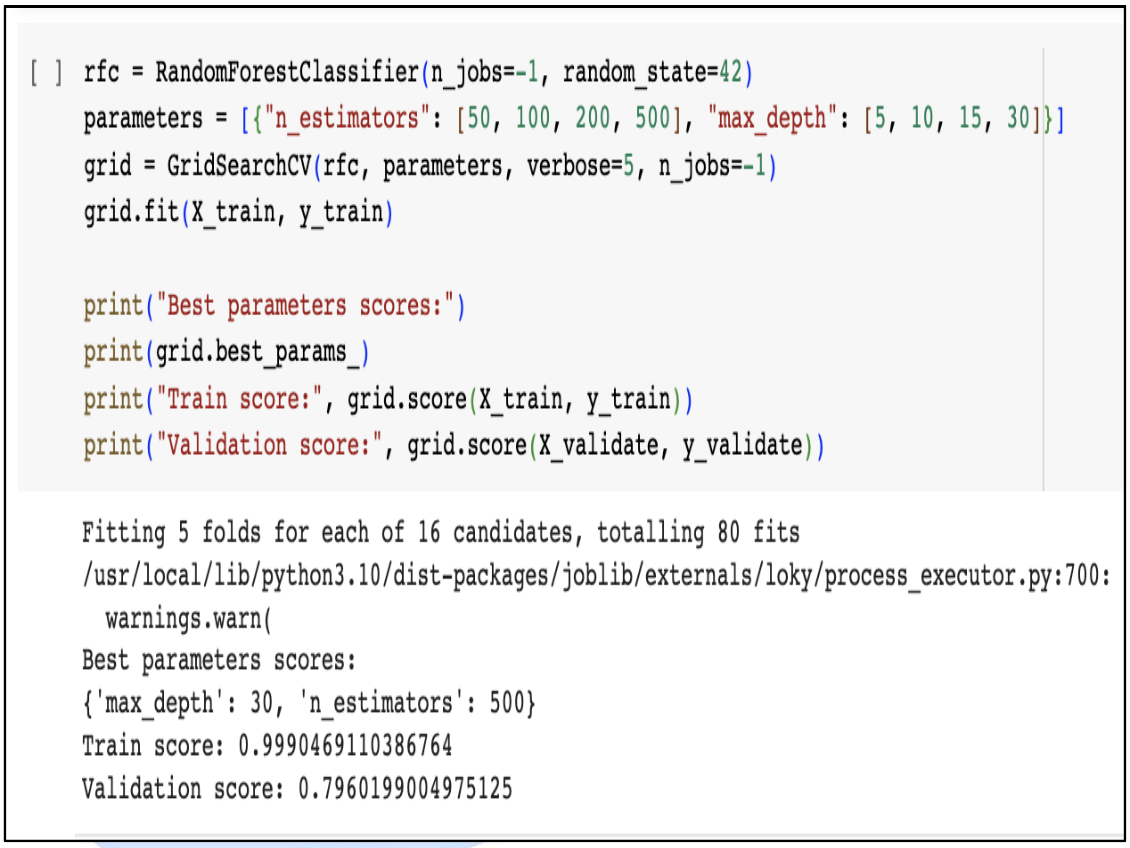
- Because this is a data classification algorithm, so it is suitable for the problem posed.
- The algorithm gives high accuracy and fast running time.

3.4.3. Build and predict with Random Forest model

Step 1:

- Implementing the Random Forest classifier: The code initializes a Random Forest classifier (rfc) using the Random Forest Classifier class.

- The Random Forest classifier object is instantiated with n_jobs and random_state.
- The hyperparameters to be tuned are specified in a dictionary called parameters.
- A GridSearchCV object is created and fit to the training data
- The best hyperparameters are outputted along with the model's accuracy scores on the training and validation data.

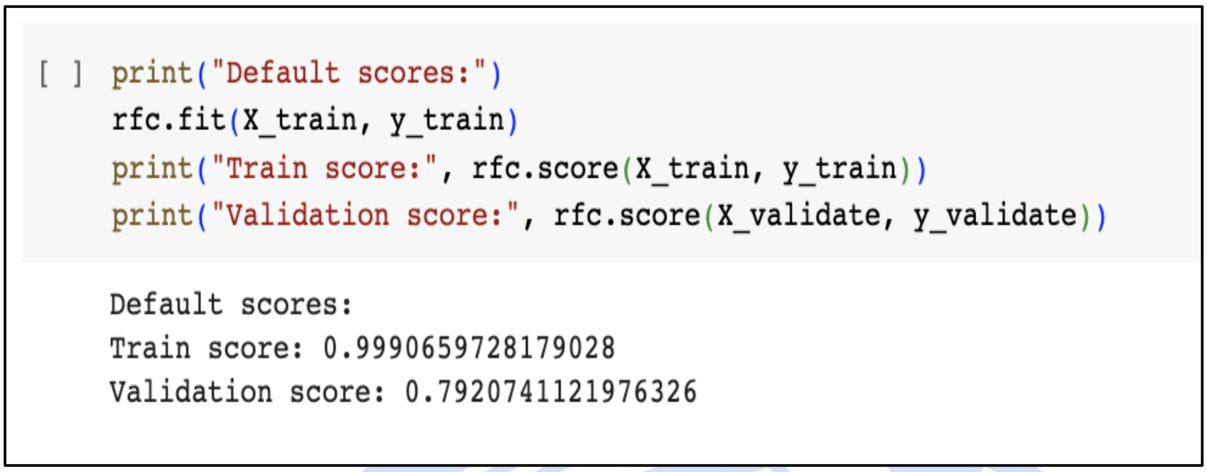


```
[ ] rfc = RandomForestClassifier(n_jobs=-1, random_state=42)
    parameters = [{"n_estimators": [50, 100, 200, 500], "max_depth": [5, 10, 15, 30]}]
    grid = GridSearchCV(rfc, parameters, verbose=5, n_jobs=-1)
    grid.fit(X_train, y_train)

    print("Best parameters scores:")
    print(grid.best_params_)
    print("Train score:", grid.score(X_train, y_train))
    print("Validation score:", grid.score(X_validate, y_validate))
```

```
Fitting 5 folds for each of 16 candidates, totalling 80 fits
/usr/local/lib/python3.10/dist-packages/joblib/externals/loky/process_executor.py:700:
  warnings.warn(
Best parameters scores:
{'max_depth': 30, 'n_estimators': 500}
Train score: 0.9990469110386764
Validation score: 0.7960199004975125
```

- The output means that the Random Forest classifier model was trained and evaluated using GridSearchCV with 5-fold cross-validation on 2 hyperparameter combinations (n_estimators, max_depth).
- The accuracy score on the training data was 0.999, indicating that the model predicted the class labels correctly for 99,9% of the training examples.
  The accuracy score on the validation data was 0.796, indicating that the model predicted the class labels correctly for 79,6% of the validation examples

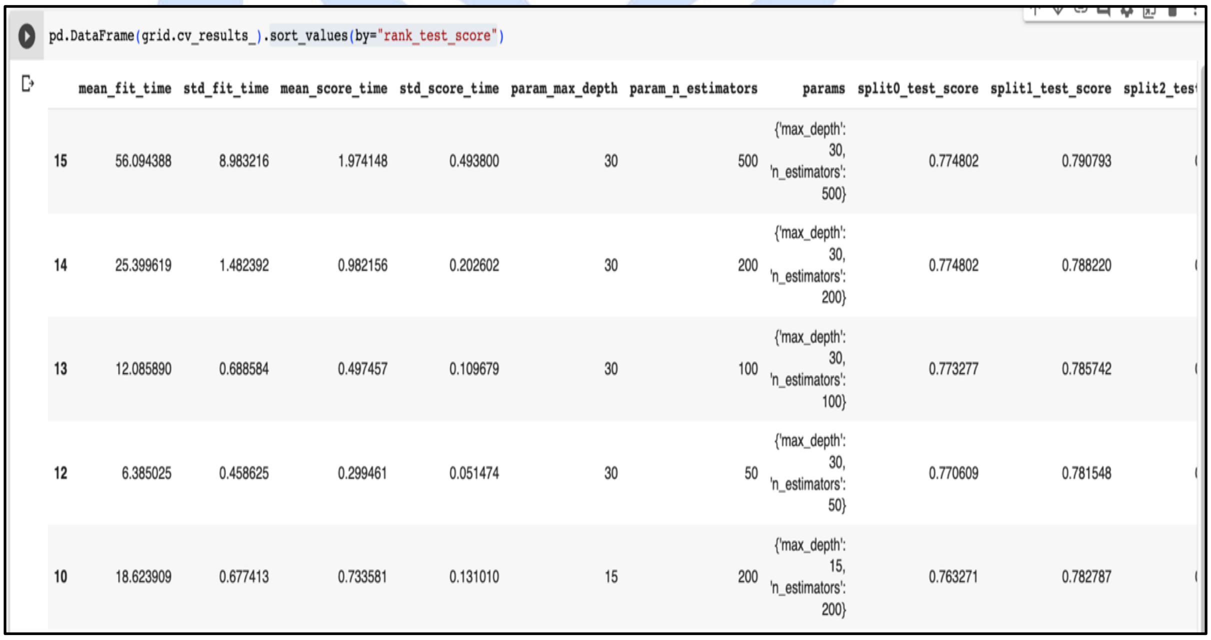
Step 2: Fitting a Random Forest classifier model (rfc) to the training dataset (X_train and y_train) and printing the training and validation scores for the default hyperparameters.



```
[ ] print("Default scores:")
    rfc.fit(X_train, y_train)
    print("Train score:", rfc.score(X_train, y_train))
    print("Validation score:", rfc.score(X_validate, y_validate))

    Default scores:
    Train score: 0.9990659728179028
    Validation score: 0.7920741121976326
```

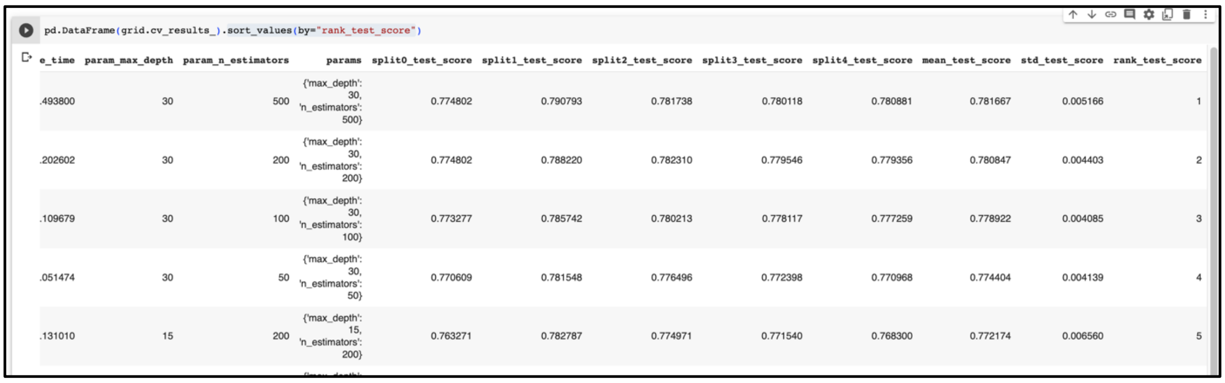
- The training score of 0.999 indicates that the model correctly predicted 99% of the training samples.
- The validation score of 0.79 indicates that the model correctly predicted 79% of the validation samples.

Step 3: Creates a Pandas DataFrame that contains the results of a grid search cross-validation and sort_values by rank_test_score.



```
pd.DataFrame(grid.cv_results_).sort_values(by="rank_test_score")
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	param_n_estimators	params	split0_test_score	split1_test_score	split2_tes
15	56.094388	8.983216	1.974148	0.493800	30	500	{'max_depth': 30, 'n_estimators': 500}	0.774802	0.790793	(
14	25.399619	1.482392	0.982156	0.202602	30	200	{'max_depth': 30, 'n_estimators': 200}	0.774802	0.788220	(
13	12.085890	0.688584	0.497457	0.109679	30	100	{'max_depth': 30, 'n_estimators': 100}	0.773277	0.785742	(
12	6.385025	0.458625	0.299461	0.051474	30	50	{'max_depth': 30, 'n_estimators': 50}	0.770609	0.781548	(
10	18.623909	0.677413	0.733581	0.131010	15	200	{'max_depth': 15, 'n_estimators': 200}	0.763271	0.782787	(

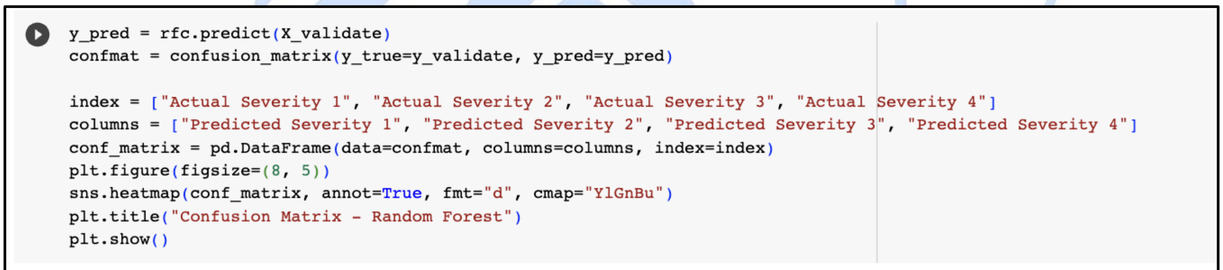


```
pd.DataFrame(grid.cv_results_).sort_values(by="rank_test_score")
```

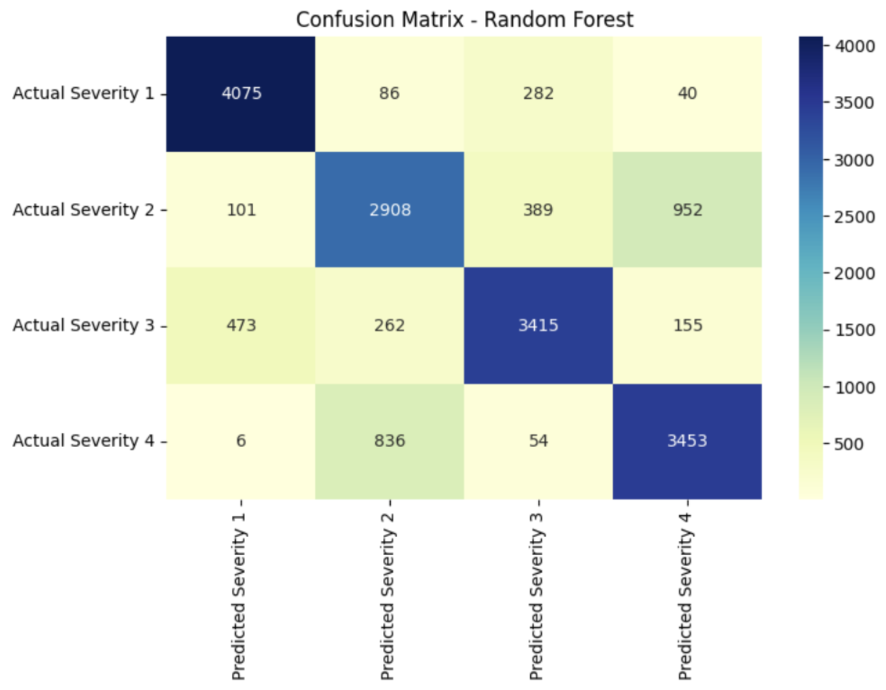
e_time	param_max_depth	param_n_estimators	params	split0_test_score	split1_test_score	split2_test_score	split3_test_score	split4_test_score	mean_test_score	std_test_score	rank_test_score
.493800	30	500	{'max_depth': 30, 'n_estimators': 500}	0.774802	0.790793	0.781738	0.780118	0.780881	0.781667	0.005166	1
.202602	30	200	{'max_depth': 30, 'n_estimators': 200}	0.774802	0.788220	0.782310	0.779546	0.779356	0.780847	0.004403	2
.109679	30	100	{'max_depth': 30, 'n_estimators': 100}	0.773277	0.785742	0.780213	0.778117	0.777259	0.778922	0.004085	3
.051474	30	50	{'max_depth': 30, 'n_estimators': 50}	0.770609	0.781548	0.776496	0.772398	0.770968	0.774404	0.004139	4
.131010	15	200	{'max_depth': 15, 'n_estimators': 200}	0.763271	0.782787	0.774971	0.771540	0.768300	0.772174	0.006560	5

- The resulting DataFrame contains a row for each combination of hyperparameters and columns for various metrics, including the mean test score, standard deviation of the test score, and the values of the hyperparameter.

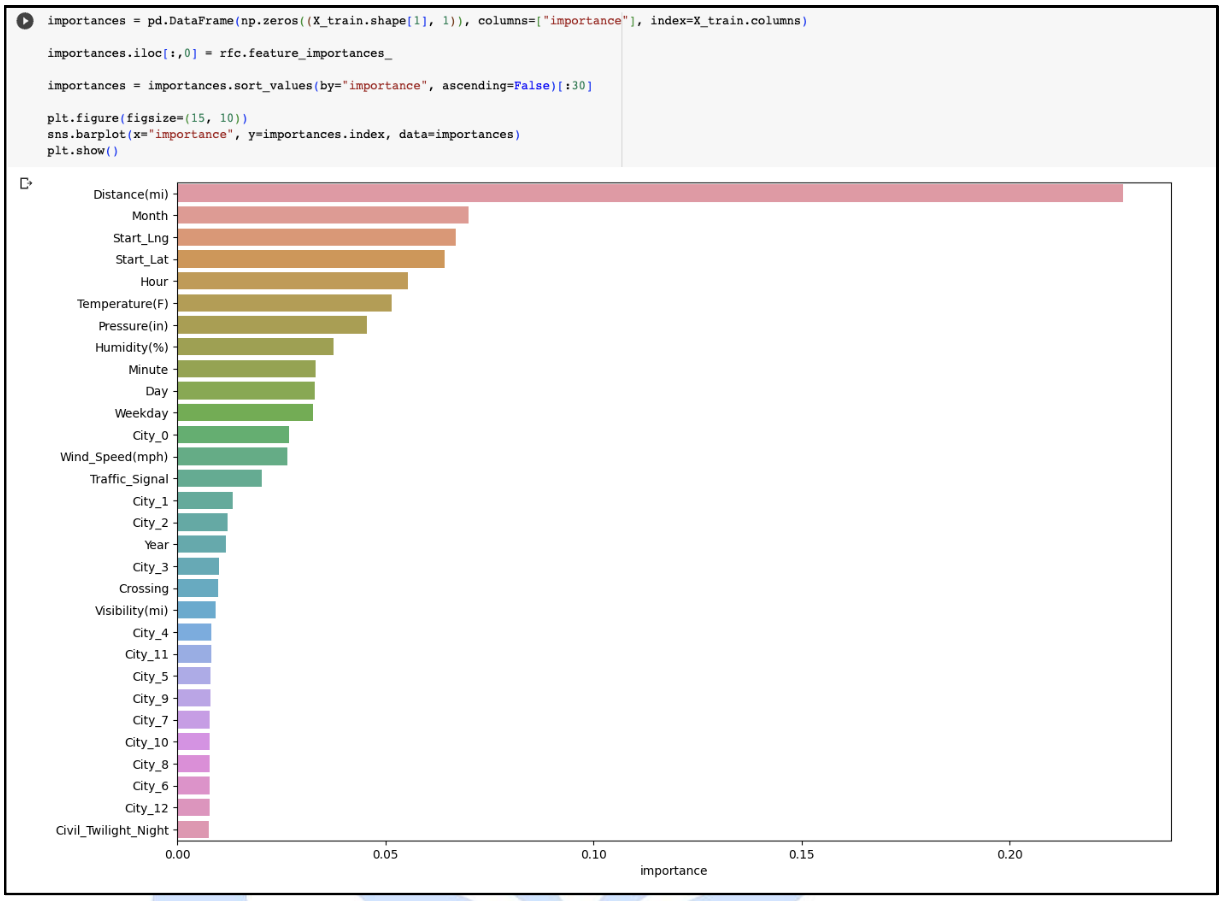
Step 4: Creating a confusion matrix to visualize the performance of the model. The confusion matrix is stored in a Pandas DataFrame and displayed using a heatmap from the Seaborn library.



```
y_pred = rfc.predict(X_validate)
confmat = confusion_matrix(y_true=y_validate, y_pred=y_pred)

index = ["Actual Severity 1", "Actual Severity 2", "Actual Severity 3", "Actual Severity 4"]
columns = ["Predicted Severity 1", "Predicted Severity 2", "Predicted Severity 3", "Predicted Severity 4"]
conf_matrix = pd.DataFrame(data=confmat, columns=columns, index=index)
plt.figure(figsize=(8, 5))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="YlGnBu")
plt.title("Confusion Matrix - Random Forest")
plt.show()
```



Step 5: Generates a bar plot that visually represents the feature importances of the top 30 features determined by the random forest classifier. The higher the importance score of a feature, the more influential it is in making predictions with the random forest model.



```
importances = pd.DataFrame(np.zeros((X_train.shape[1], 1)), columns=["importance"], index=X_train.columns)

importances.iloc[:,0] = rfc.feature_importances_

importances = importances.sort_values(by="importance", ascending=False)[:30]

plt.figure(figsize=(15, 10))
sns.barplot(x="importance", y=importances.index, data=importances)
plt.show()
```

3.5. Using Naive Bayes algorithm

3.5.1. Naive Bayes Algorithm introduction

Naïve Bayes algorithms is a classification technique based on applying Bayes' theorem with a strong assumption that all the predictors are independent to each other. In simple words, the assumption is that the presence of a feature in a class is independent to the presence of any other feature in the same class.

For example, a phone may be considered as smart if it is having touch screen, internet facility, good camera etc. Though all these features are dependent on each other, they contribute independently to the probability of that the phone is a smart phone.

In Bayesian classification, the main interest is to find the posterior probabilities i.e. the probability of a label given some observed features, $P(L \mid features)$. With the help of Bayes theorem, we can express this in quantitative form as follows [3]:

$$P(L|features)\frac{P(L)P(features|L)}{P(features)}$$

- Here, $P(L \mid features)$ is the posterior probability of class.
- $P(L)$ is the prior probability of class.
- $P(features \mid L)$ is the likelihood which is the probability of predictor given class.
- $P(features)$ is the prior probability of predictor.

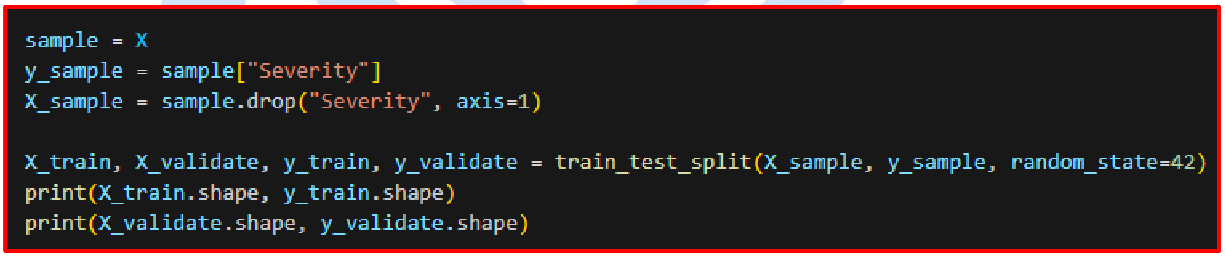
3.5.2. Selected reasons

- Because this is a data classification algorithm, so it is suitable for the problem posed and with text description fields

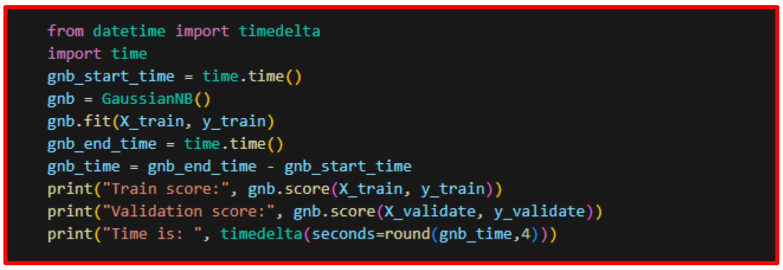
- The algorithm gives high accuracy and fast running time

3.5.3. Build and predict with Naive Bayes model

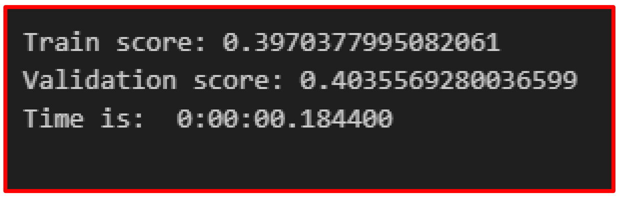
Step 1: Use GaussianNB function to train and predict Severity.



```
sample = X
y_sample = sample["Severity"]
X_sample = sample.drop("Severity", axis=1)

X_train, X_validate, y_train, y_validate = train_test_split(X_sample, y_sample, random_state=42)
print(X_train.shape, y_train.shape)
print(X_validate.shape, y_validate.shape)
```



```
from datetime import timedelta
import time
gnb_start_time = time.time()
gnb = GaussianNB()
gnb.fit(X_train, y_train)
gnb_end_time = time.time()
gnb_time = gnb_end_time - gnb_start_time
print("Train score:", gnb.score(X_train, y_train))
print("Validation score:", gnb.score(X_validate, y_validate))
print("Time is: ", timedelta(seconds=round(gnb_time,4)))
```



```
Train score: 0.3970377995082061
Validation score: 0.4035569280036599
Time is:  0:00:00.184400
```

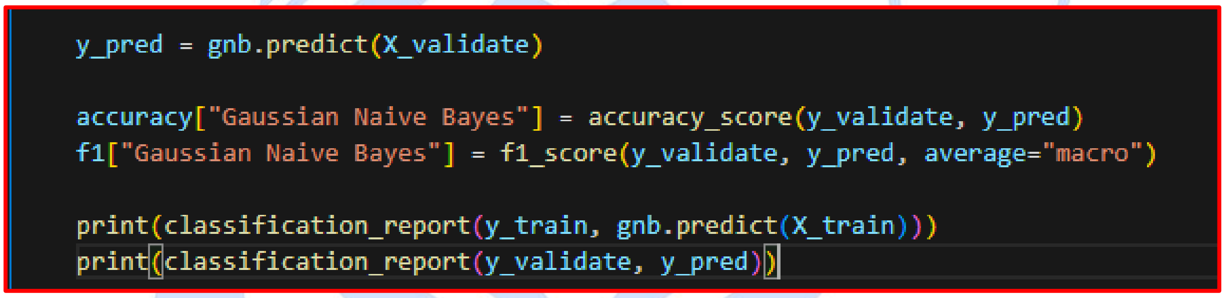
- The accuracy score on the training data was 0.397, indicating that the model predicted the class labels correctly for 39.7% of the training examples.
- The accuracy score on the validation data was 0.404, indicating that the model predicted the class labels correctly for 40.4% of the validation examples.

Step 2:

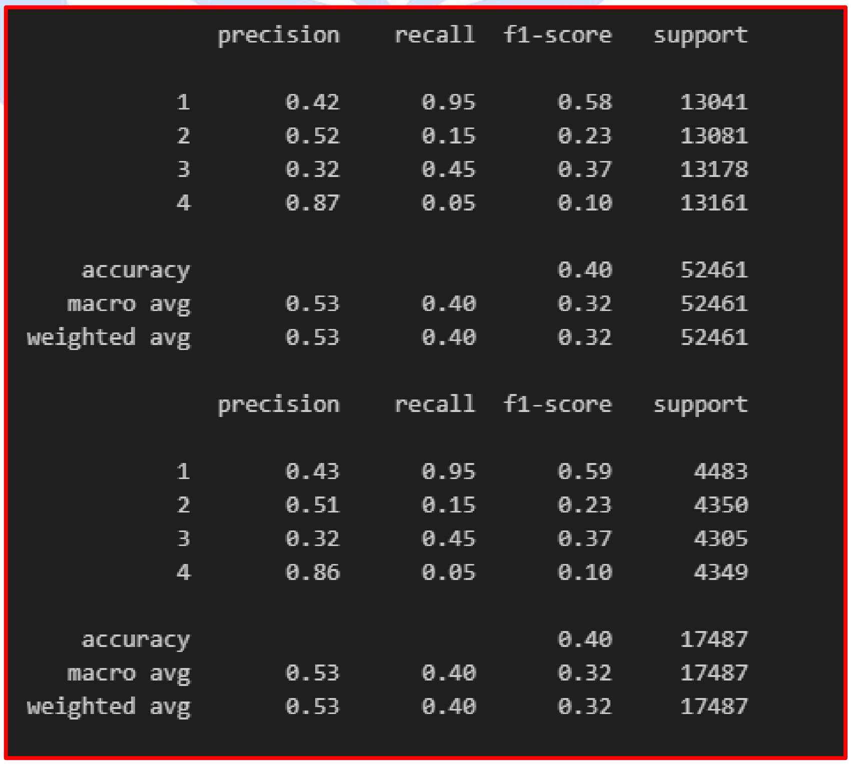
-Using a Naïve Bayes model to predict the labels of the validation dataset (X_validate) and calculating the accuracy and F1 score of the predictions.

-The accuracy and F1 score are then stored in two dictionaries (accuracy and f1) for later use.

-The classification_report function is then used to print the precision, recall, and F1 score for each class in the training and validation datasets.

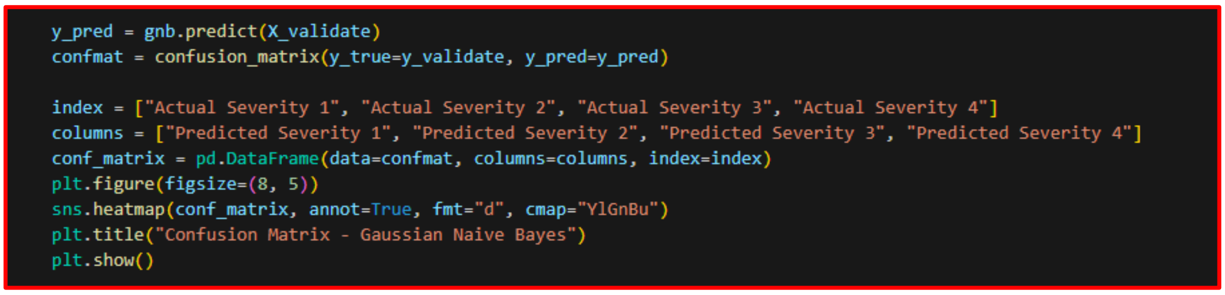


```python
y_pred = gnb.predict(X_validate)

accuracy["Gaussian Naive Bayes"] = accuracy_score(y_validate, y_pred)
f1["Gaussian Naive Bayes"] = f1_score(y_validate, y_pred, average="macro")

print(classification_report(y_train, gnb.predict(X_train)))
print(classification_report(y_validate, y_pred))
```

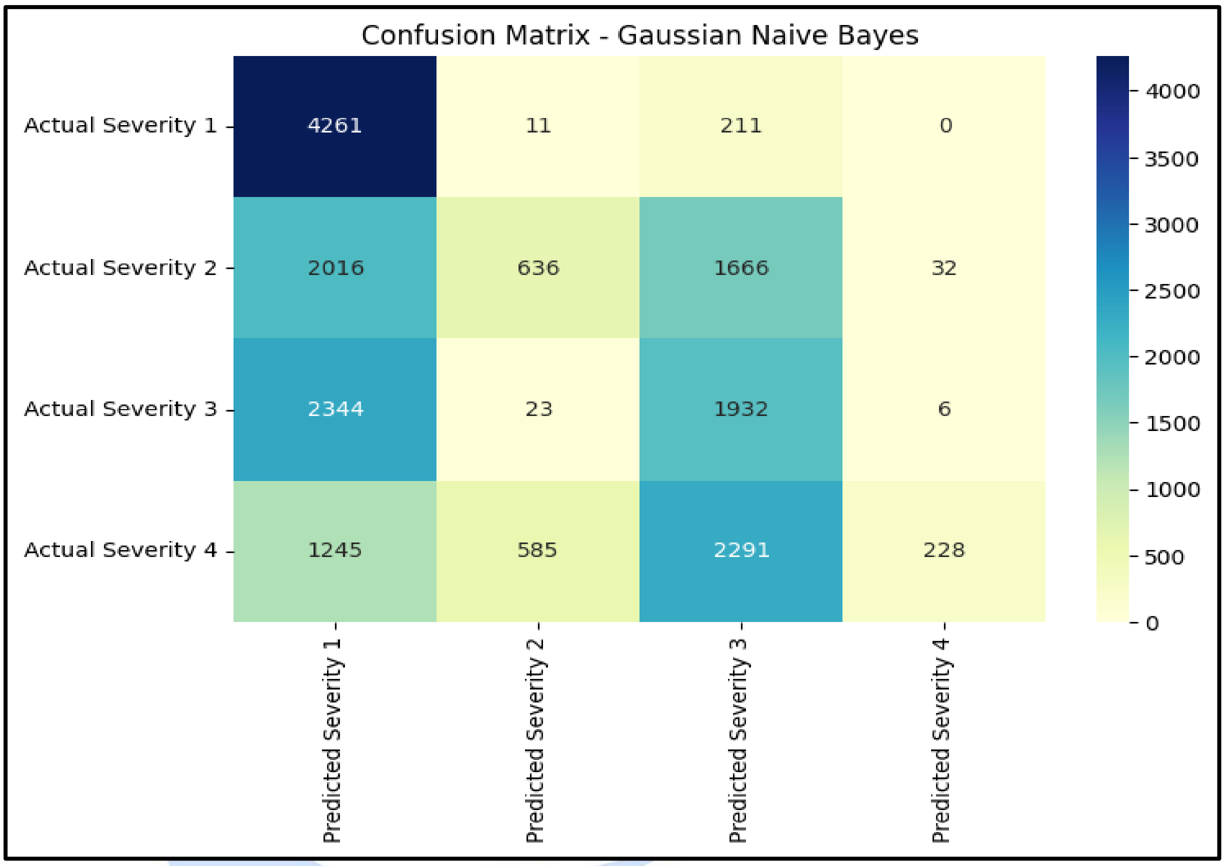


```
              precision    recall  f1-score   support

           1       0.42      0.95      0.58     13041
           2       0.52      0.15      0.23     13081
           3       0.32      0.45      0.37     13178
           4       0.87      0.05      0.10     13161

    accuracy                           0.40     52461
   macro avg       0.53      0.40      0.32     52461
weighted avg       0.53      0.40      0.32     52461

              precision    recall  f1-score   support

           1       0.43      0.95      0.59      4483
           2       0.51      0.15      0.23      4350
           3       0.32      0.45      0.37      4305
           4       0.86      0.05      0.10      4349

    accuracy                           0.40     17487
   macro avg       0.53      0.40      0.32     17487
weighted avg       0.53      0.40      0.32     17487
```

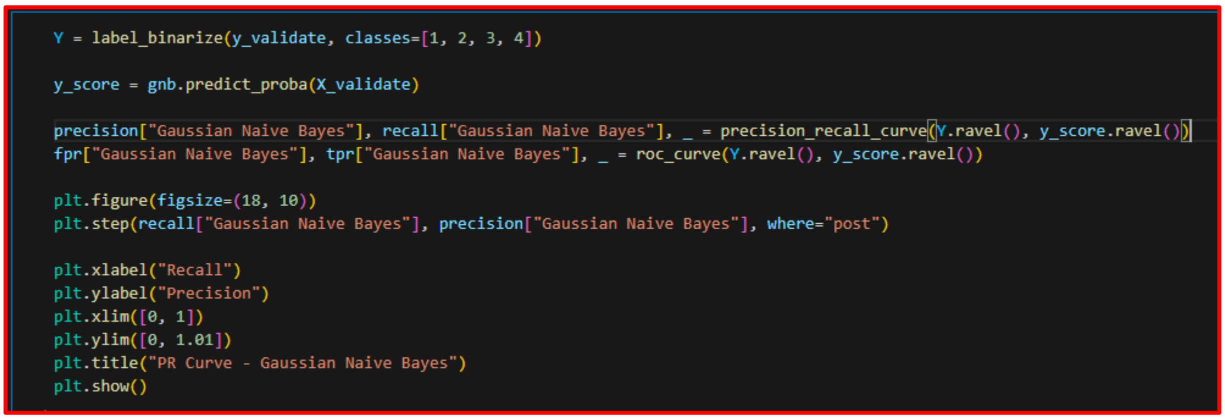
Step 3: Calculate the confusion matrix and plot it on the heatmap



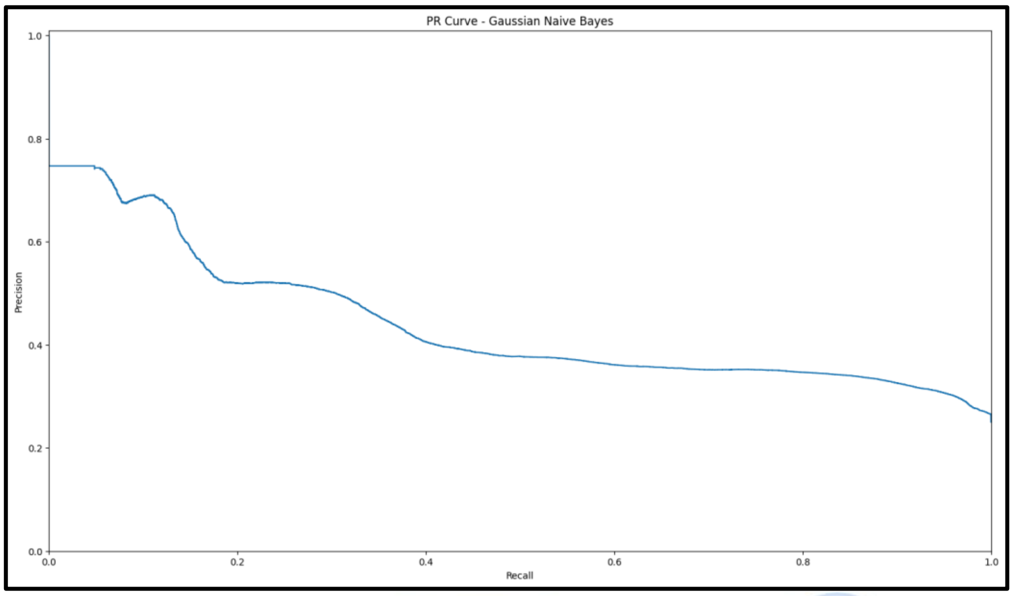
```python
y_pred = gnb.predict(X_validate)
confmat = confusion_matrix(y_true=y_validate, y_pred=y_pred)

index = ["Actual Severity 1", "Actual Severity 2", "Actual Severity 3", "Actual Severity 4"]
columns = ["Predicted Severity 1", "Predicted Severity 2", "Predicted Severity 3", "Predicted Severity 4"]
conf_matrix = pd.DataFrame(data=confmat, columns=columns, index=index)
plt.figure(figsize=(8, 5))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="YlGnBu")
plt.title("Confusion Matrix - Gaussian Naive Bayes")
plt.show()
```



Step 4: Draw a Precision-Recall (PR) Curve for the Logistic Regression model on the validation dataset, which is a way to evaluate the performance of the classification model.

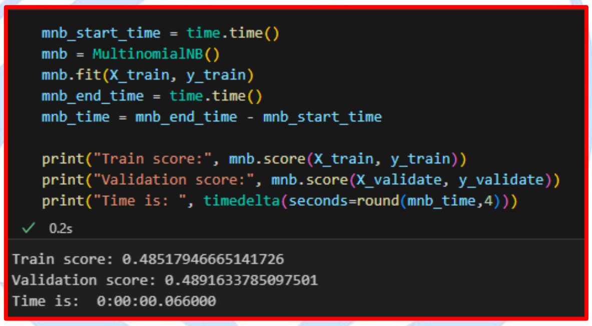


```python
Y = label_binarize(y_validate, classes=[1, 2, 3, 4])

y_score = gnb.predict_proba(X_validate)

precision["Gaussian Naive Bayes"], recall["Gaussian Naive Bayes"], _ = precision_recall_curve(Y.ravel(), y_score.ravel())
fpr["Gaussian Naive Bayes"], tpr["Gaussian Naive Bayes"], _ = roc_curve(Y.ravel(), y_score.ravel())

plt.figure(figsize=(18, 10))
plt.step(recall["Gaussian Naive Bayes"], precision["Gaussian Naive Bayes"], where="post")

plt.xlabel("Recall")
plt.ylabel("Precision")
plt.xlim([0, 1])
plt.ylim([0, 1.01])
plt.title("PR Curve - Gaussian Naive Bayes")
plt.show()
```



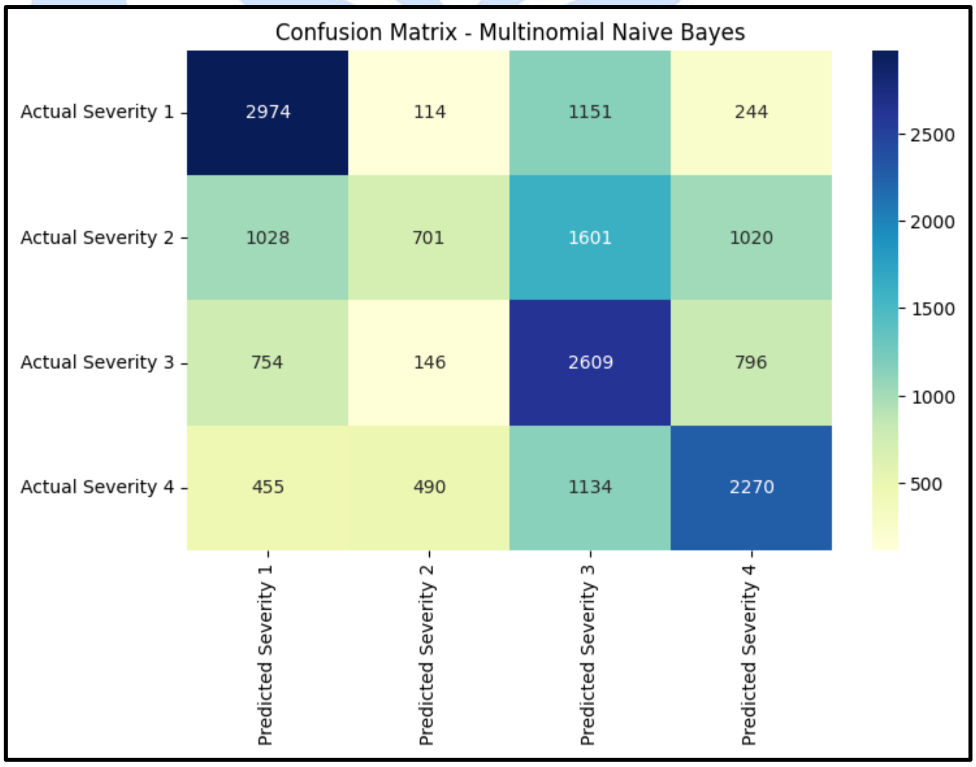
Step 5: Similar to the GaussianNB function, we test MultinomialNB and BernoulliNB to check

- MultinomialNB:



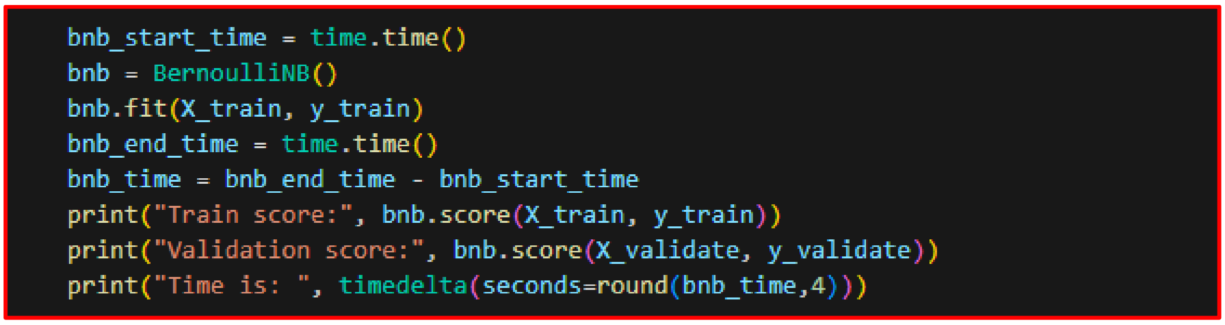
```
mnb_start_time = time.time()
mnb = MultinomialNB()
mnb.fit(X_train, y_train)
mnb_end_time = time.time()
mnb_time = mnb_end_time - mnb_start_time

print("Train score:", mnb.score(X_train, y_train))
print("Validation score:", mnb.score(X_validate, y_validate))
print("Time is: ", timedelta(seconds=round(mnb_time,4)))
```

```
✓ 0.2s
Train score: 0.48517946665141726
Validation score: 0.4891633785097501
Time is:  0:00:00.066000
```

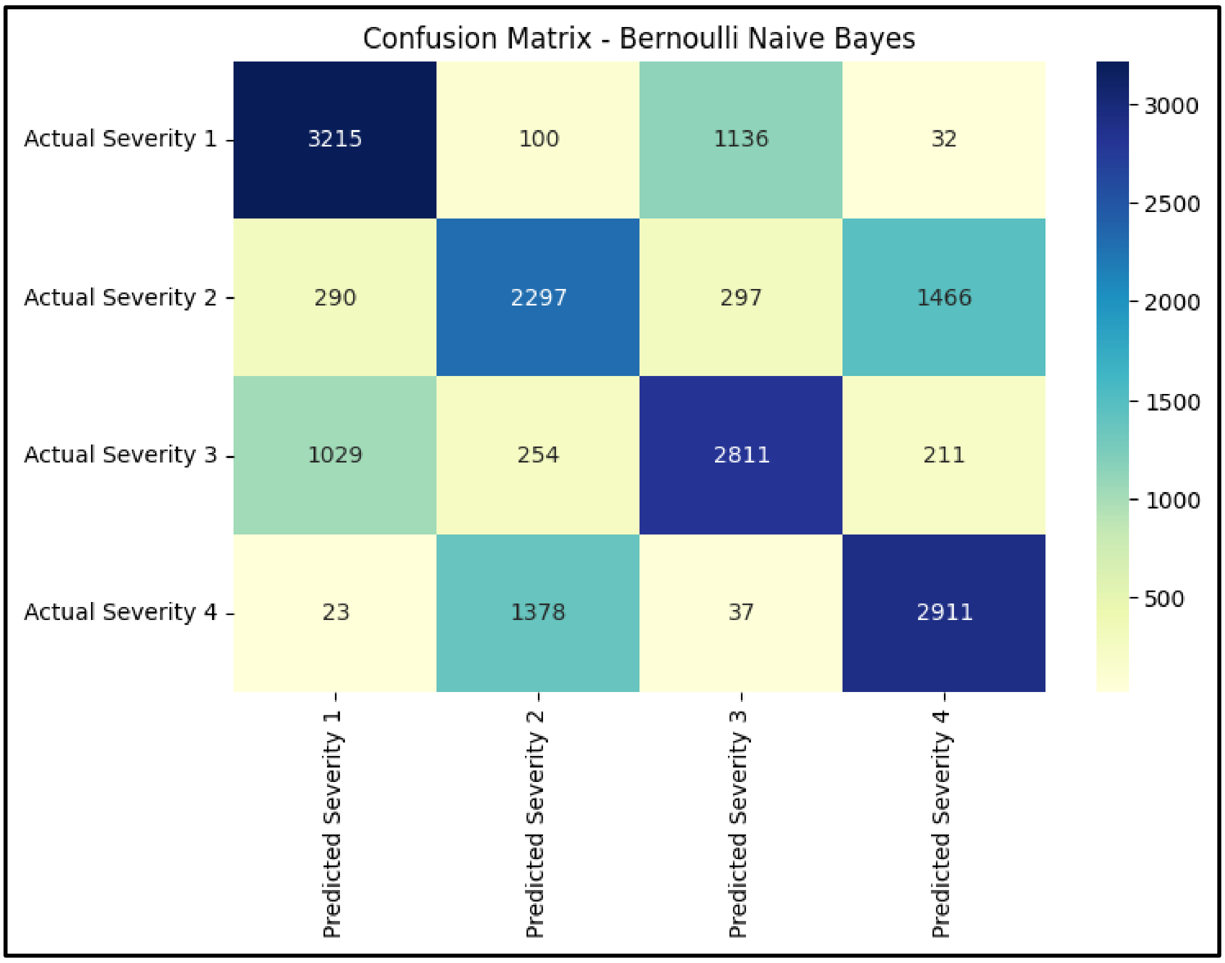


	Predicted Severity 1	Predicted Severity 2	Predicted Severity 3	Predicted Severity 4
Actual Severity 1	2974	114	1151	244
Actual Severity 2	1028	701	1601	1020
Actual Severity 3	754	146	2609	796
Actual Severity 4	455	490	1134	2270

- BernoulliNB:



```
bnb_start_time = time.time()
bnb = BernoulliNB()
bnb.fit(X_train, y_train)
bnb_end_time = time.time()
bnb_time = bnb_end_time - bnb_start_time
print("Train score:", bnb.score(X_train, y_train))
print("Validation score:", bnb.score(X_validate, y_validate))
print("Time is: ", timedelta(seconds=round(bnb_time,4)))
```



As we see MultinomialNB function gives better and faster results than the GaussianNB function and the BernoulliNB function gives better results than the MultinomialNB function but is slower.

4. ALGORITHMS EVALUATION AND PREDICTING

4.1. Accuracy Algorithm Evaluation

4.1.1. Accuracy Algorithm Evaluation Introduction

When building a Machine Learning model, we need an assessment to see if the model is working and to compare the capabilities of the models. In this article, I will introduce methods of evaluating classification models.

The performance of a model is often evaluated against the test data set. Specifically, suppose that the output of the model when input is the test set is described by the vector y_pred - which is the output prediction vector with each element being the predicted class of a data point in the test set. We need to compare this y_pred prediction vector with the real class vector of the data, described by the y_true vector.

There are many ways to evaluate a classification model. Depending on different problems, we use different methods. Commonly used methods are: accuracy score, confusion matrix, ROC curve, Area Under the Curve, Precision and Recall, F1 score, Top R error, etc.

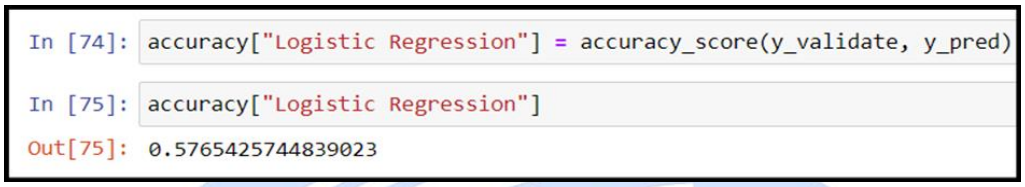
4.1.2. Selected reasons

Accuracy is the fraction of successful predictions with respect to the total number of records.

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

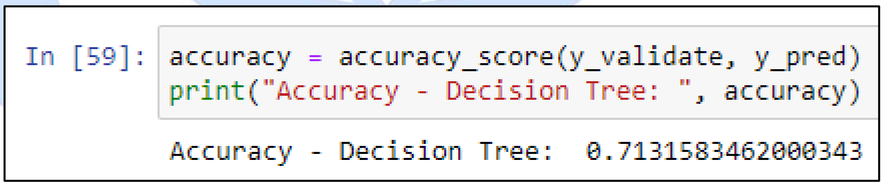
This is the most common performance metric, because it gives us a clear understanding of how often our model is true. It’s very useful when you give the same importance to 0 and 1, but you have to be careful when you use it, because if the dataset is unbalanced, the class with the highest number of records will dominate the fraction and it’s not often a good idea. A very high number of 0s will create a bias over the 1s and biases are not what we need.

4.1.3. Calculate Accuracy Score

- Logistic Regression Algorithm
  - Calculate accuracy for the Logistic Regression model on the validation dataset
  - 

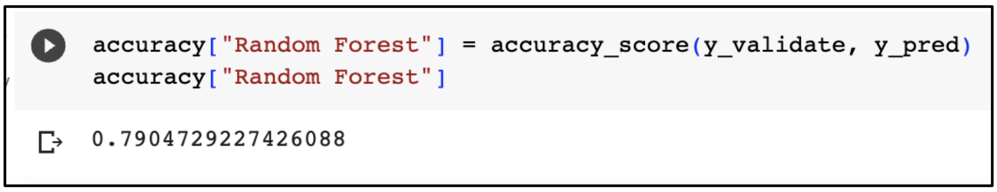
```
In [74]: accuracy["Logistic Regression"] = accuracy_score(y_validate, y_pred)

In [75]: accuracy["Logistic Regression"]
Out[75]: 0.5765425744839023
```

    - The result 0.577 is the accuracy score of a logistic regression model on a validation set.
    - The accuracy score suggests that the model is able to correctly predict the labels for approximately 57.7% of the samples in the validation set.
- Decision Tree
  - Calculate accuracy for the Decision Tree model on the validation dataset



```
In [59]: accuracy = accuracy_score(y_validate, y_pred)
         print("Accuracy - Decision Tree: ", accuracy)

         Accuracy - Decision Tree:  0.7131583462000343
```

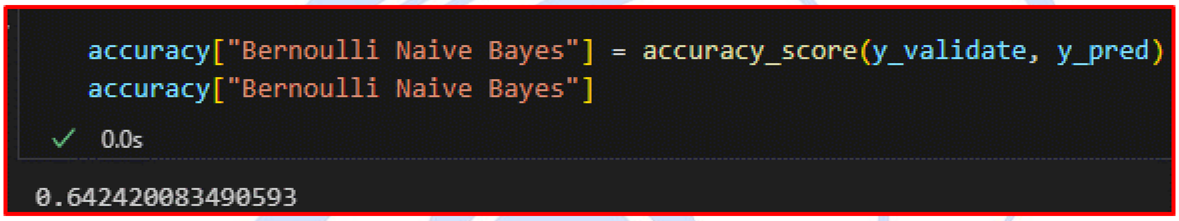
    - The result 0.713 is the accuracy score of a Decision Tree model on a validation set.
    - The accuracy score suggests that the model is able to correctly predict the labels for approximately 71.3% of the samples in the validation set.
- Random Forest
  - Calculate accuracy for the Random Forest model on the validation dataset



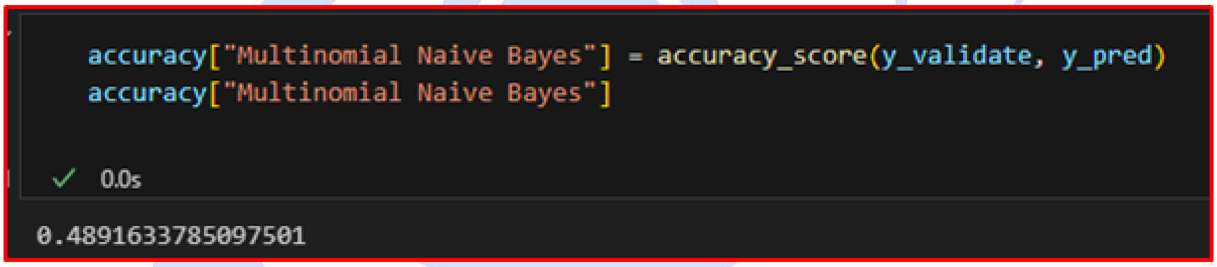
```
accuracy["Random Forest"] = accuracy_score(y_validate, y_pred)
accuracy["Random Forest"]

0.7904729227426088
```

- ○
  - ▪ The result 0.79 is the accuracy score of a Random Forest model on a validation set.
  - ▪ The accuracy score suggests that the model is able to correctly predict the labels for approximately 79% of the samples in the validation set.

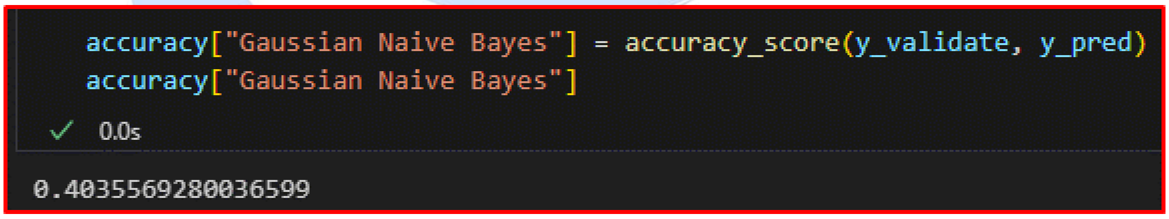
- Native Bayes
  - ○ Calculate accuracy for the Naïve Bayes model on the validation dataset



```
accuracy["Bernoulli Naive Bayes"] = accuracy_score(y_validate, y_pred)
accuracy["Bernoulli Naive Bayes"]
✓ 0.0s
0.642420083490593
```



```
accuracy["Multinomial Naive Bayes"] = accuracy_score(y_validate, y_pred)
accuracy["Multinomial Naive Bayes"]

✓ 0.0s
0.4891633785097501
```



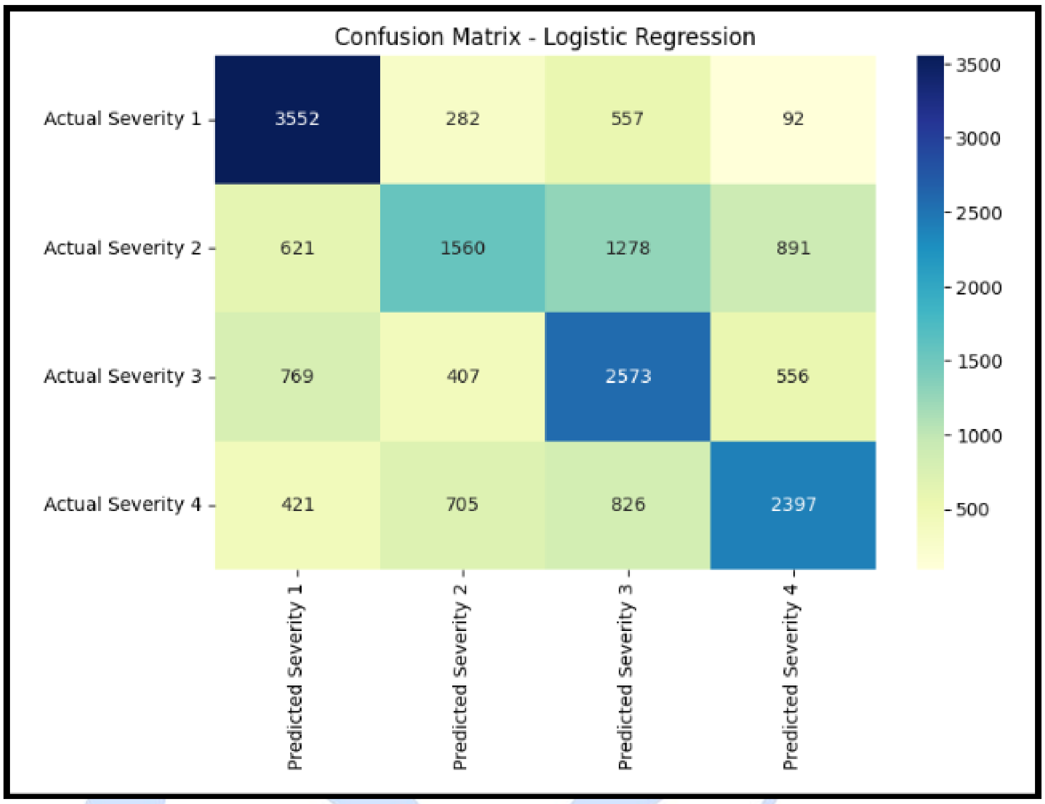
```
accuracy["Gaussian Naive Bayes"] = accuracy_score(y_validate, y_pred)
accuracy["Gaussian Naive Bayes"]
✓ 0.0s
0.4035569280036599
```

  - ▪ The result 0.404, 0.489, 0.642 is the accuracy score of a Naïve Bayes model types on a validation set.
  - ▪ The accuracy score suggests that the model is able to correctly predict the labels for approximately 40.4%, 48.9%, 64.2% of the samples in the validation set.

4.2. Macro F1-Score evaluation metrics

4.2.1. Macro F1-Score evaluation metrics introduction

F1-Score assesses the classification model’s performance starting from the confusion matrix.

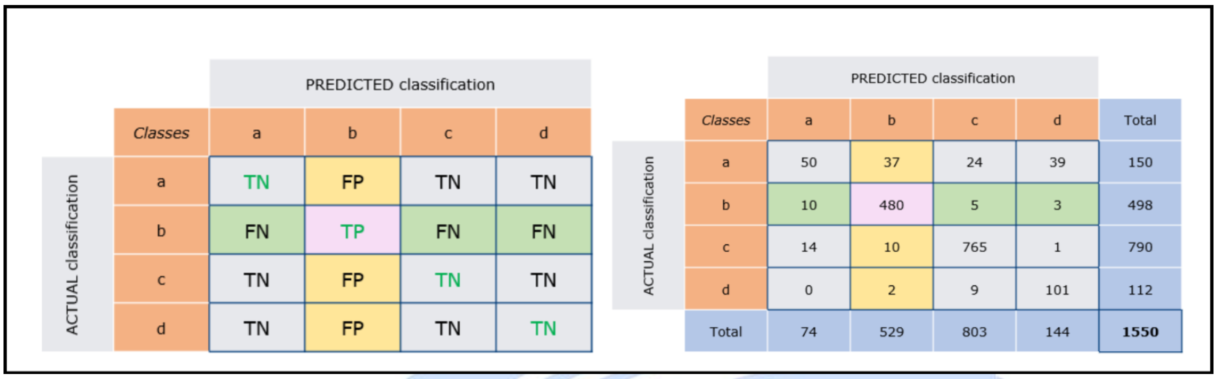


It is the harmonic mean of precision and recall, calculated for a single class in a binary or multi-class classification problem.

In the case of multi-class classification, F1-Score should take into account all classes. To achieve this, we need to use multi-class measures of precision and recall, which can be combined using the harmonic mean. There are two different specifications of these metrics, which lead to two distinct evaluation metrics: Micro F1-Score and Macro F1-Score.

Here, we use Macro F1-Score to evaluate the performance of the classification model.

The Macro approach takes into account all classes as fundamental elements in the calculation: each class is given equal weight in the average, regardless of its size, thereby avoiding any distinction between highly and poorly populated classes. In order to obtain Macro F1-Score, two intermediate steps are required: Macro-Precision and Macro-Recall. Macro-Precision is obtained by averaging the precision values for each predicted class, while Macro-Recall is obtained by averaging the recall values for each actual class.



		PREDICTED classification			
	Classes	a	b	c	d
ACTUAL classification	a	TN	FP	TN	TN
	b	FN	TP	FN	FN
	c	TN	FP	TN	TN
	d	TN	FP	TN	TN

		PREDICTED classification				
	Classes	a	b	c	d	Total
ACTUAL classification	a	50	37	24	39	150
	b	10	480	5	3	498
	c	14	10	765	1	790
	d	0	2	9	101	112
	Total	74	529	803	144	**1550**

The image above focuses on class b as the reference class and demonstrates how to compute its Precision and Recall. To perform these calculations, we use the Confusion Matrix, where each tile is labeled according to its corresponding class. Specifically, we only consider:

- True Positive (TP) as the only correctly classified units for our class
- False Positive (FP) and False Negative (FN) are the wrongly classified elements on the column and the row of the class respectively.
- True Negative (TN) are all the other tiles, as shown in picture where we are considering the class "b" as reference focus. [4]

When we switch from one class to another, we repeat the calculations for Precision and Recall using the new reference class, and update the labels for the tiles in the Confusion Matrix accordingly.

The Precision and Recall values for each class are calculated using the same equations as in the binary classification setting, as described earlier.

$$\boxed{Precision_k = \frac{TP_k}{TP_k + FP_k}}$$ [4]

$$\boxed{Recall_k = \frac{TP_k}{TP_k + FN_k}}$$ [4]

The Macro Average Precision and Recall are calculated as the simple average of the Precision and Recall values for individual classes.

$$MacroAveragePrecision = \frac{\sum_{k=1}^{K} Precision_k}{K}$$

$$MacroAverageRecall = \frac{\sum_{k=1}^{K} Recall_k}{K}$$ [4]

Finally, the Macro F1-Score is obtained by calculating the harmonic mean of the Macro-Precision and Macro-Recall values:

$$Macro\,\text{F1-Score} = 2 * \left(\frac{MacroAveragePrecision * MacroAverageRecall}{MacroAveragePrecision^{-1} + MacroAverageRecall^{-1}}\right)$$

A PR curve is simply a graph with Precision values on the y-axis and Recall values on the x-axis. In other words, the PR curve contains TP/(TP+FP) on the y-axis and TP/(TP+FN) on the x-axis.

- It is important to note that Precision is also called the Positive Predictive Value (PPV).
- Recall is also called Sensitivity, Hit Rate or True Positive Rate (TPR). [5]

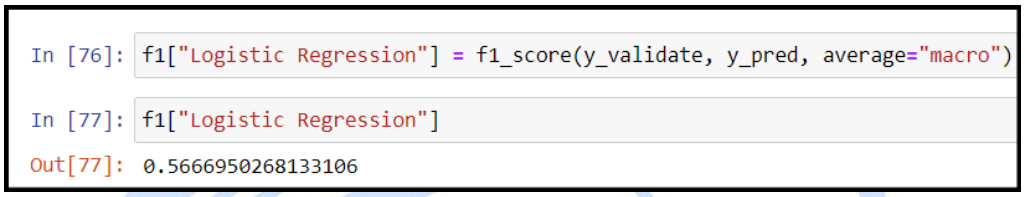
4.2.2. Selected reasons

It is possible to derive some intuitions from the equation.

Macro-Average methods tend to calculate an overall mean of different measures, because the numerators of Macro Average Precision and Macro Average Recall are composed by values in the range [0, 1]. There is no link to the class size, because classes with different size are equally weighted at the numerator. This implies that the effect of the biggest classes have the same importance as small ones have. The obtained metric evaluates the algorithm from a class standpoint: high Macro-F1 values indicate that the algorithm has good performance on all the classes, whereas low Macro-F1 values refers to poorly predicted classes. [4]

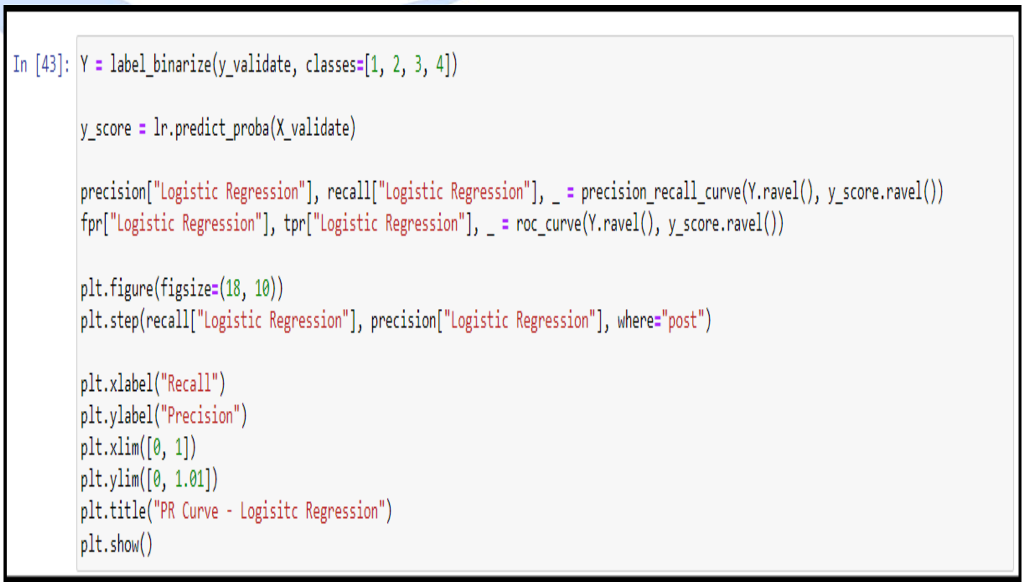
4.2.3. Calculate Macro Precision, Recall and F1-Score

- Logictis Regression Algorithm
  - Calculate F1-Score for the Logictis Regression model on the validation dataset

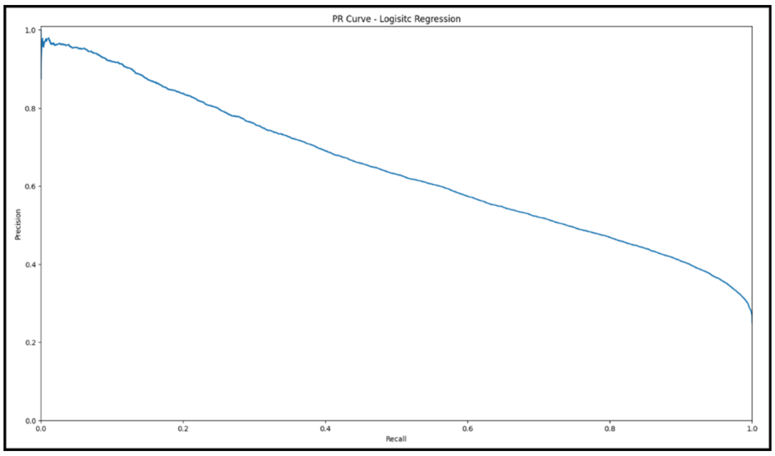


```
In [76]: f1["Logistic Regression"] = f1_score(y_validate, y_pred, average="macro")

In [77]: f1["Logistic Regression"]
Out[77]: 0.5666950268133106
```

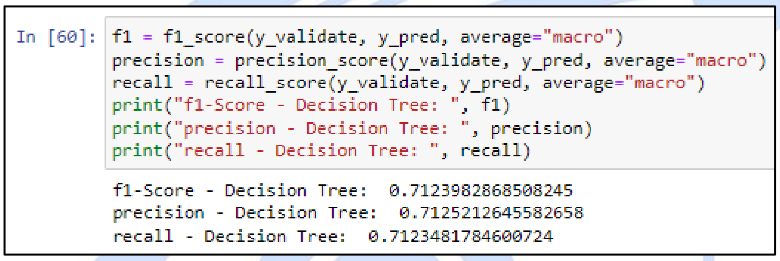
    - The result 0.567 is the accuracy score of a logistic regression model on a validation set.
    - The accuracy score suggests that the model is able to correctly predict the labels for approximately 56.7% of the samples in the validation set.
  - Draw a Precision-Recall (PR) Curve for the Logistic Regression model on the validation dataset, which is a way to evaluate the performance of the classification model.



```
In [43]: Y = label_binarize(y_validate, classes=[1, 2, 3, 4])

         y_score = lr.predict_proba(X_validate)

         precision["Logistic Regression"], recall["Logistic Regression"], _ = precision_recall_curve(Y.ravel(), y_score.ravel())
         fpr["Logistic Regression"], tpr["Logistic Regression"], _ = roc_curve(Y.ravel(), y_score.ravel())

         plt.figure(figsize=(18, 10))
         plt.step(recall["Logistic Regression"], precision["Logistic Regression"], where="post")

         plt.xlabel("Recall")
         plt.ylabel("Precision")
         plt.xlim([0, 1])
         plt.ylim([0, 1.01])
         plt.title("PR Curve - Logisitc Regression")
         plt.show()
```

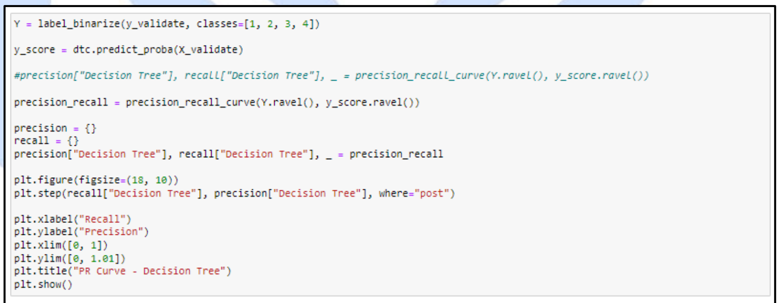


- Decision Tree
    - Calculate F1-Score, precision and recall for the Decision Tree model on the validation dataset

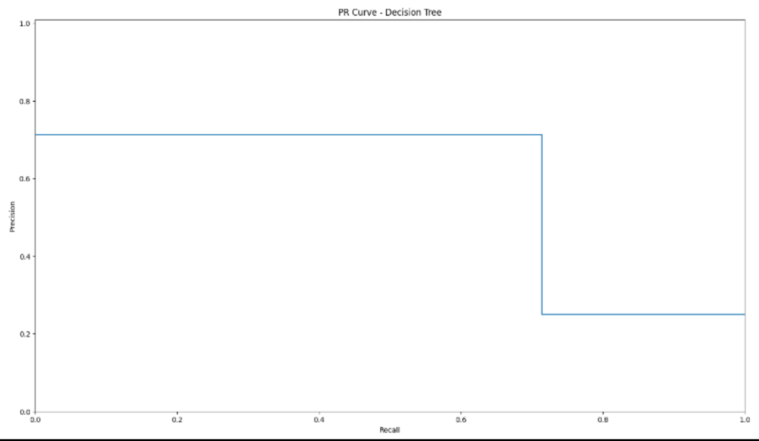


```
In [60]: f1 = f1_score(y_validate, y_pred, average="macro")
         precision = precision_score(y_validate, y_pred, average="macro")
         recall = recall_score(y_validate, y_pred, average="macro")
         print("f1-Score - Decision Tree: ", f1)
         print("precision - Decision Tree: ", precision)
         print("recall - Decision Tree: ", recall)

         f1-Score - Decision Tree:  0.7123982868508245
         precision - Decision Tree:  0.7125212645582658
         recall - Decision Tree:  0.7123481784600724
```

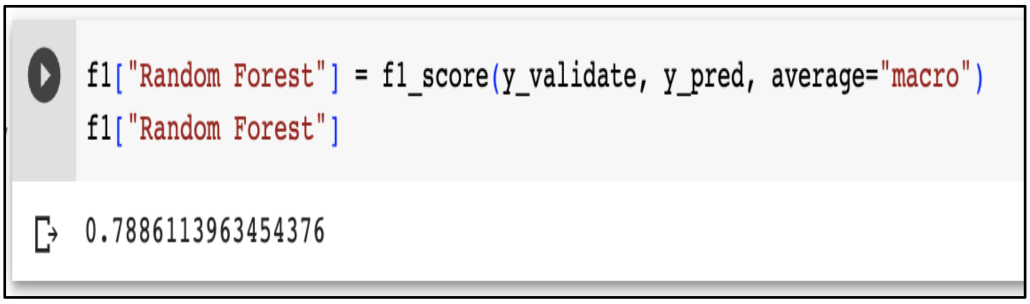
    - Draw a Precision-Recall (PR) Curve for the Logistic Regression model on the validation dataset, which is a way to evaluate the performance of the classification model.



```
Y = label_binarize(y_validate, classes=[1, 2, 3, 4])

y_score = dtc.predict_proba(X_validate)

#precision["Decision Tree"], recall["Decision Tree"], _ = precision_recall_curve(Y.ravel(), y_score.ravel())

precision_recall = precision_recall_curve(Y.ravel(), y_score.ravel())

precision = {}
recall = {}
precision["Decision Tree"], recall["Decision Tree"], _ = precision_recall

plt.figure(figsize=(18, 10))
plt.step(recall["Decision Tree"], precision["Decision Tree"], where="post")

plt.xlabel("Recall")
plt.ylabel("Precision")
plt.xlim([0, 1])
plt.ylim([0, 1.01])
plt.title("PR Curve - Decision Tree")
plt.show()
```



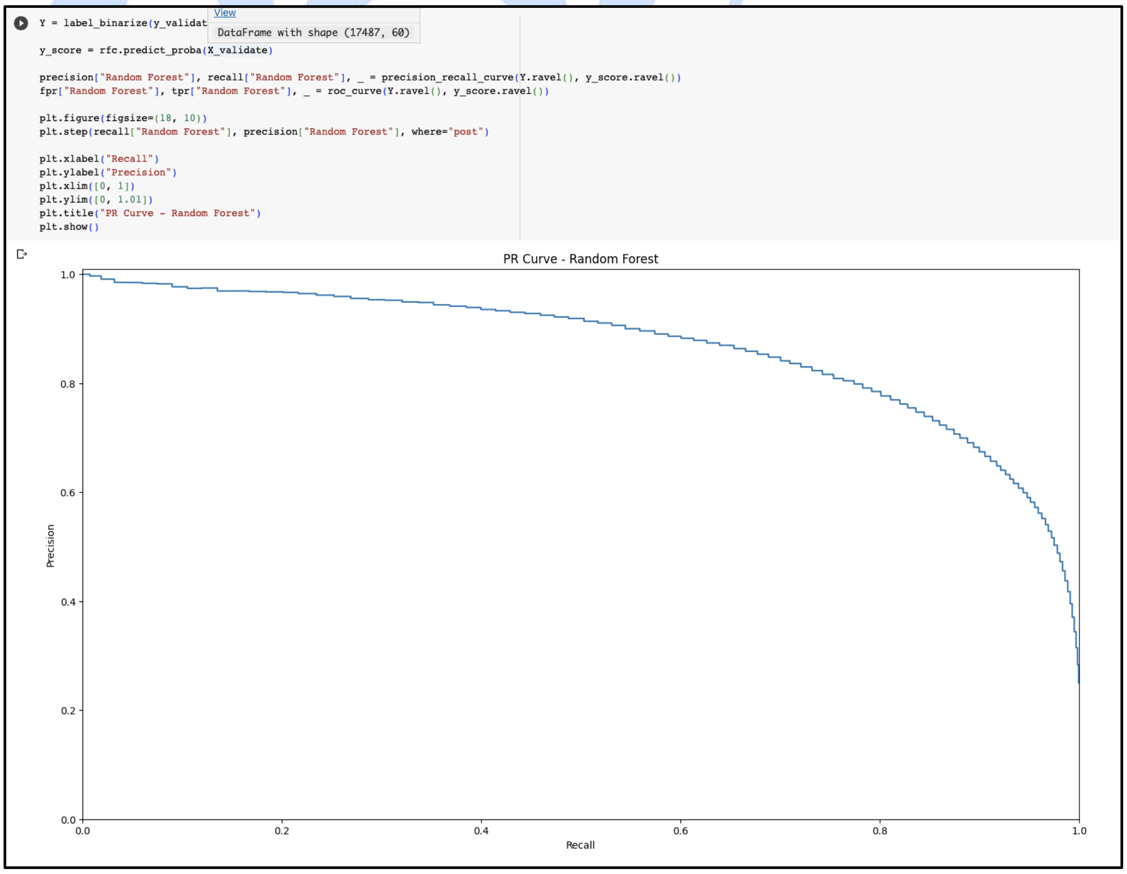
- Random Forest
  - Calculate F1-Score for the Random Forest model on the validation dataset



```
f1["Random Forest"] = f1_score(y_validate, y_pred, average="macro")
f1["Random Forest"]
```

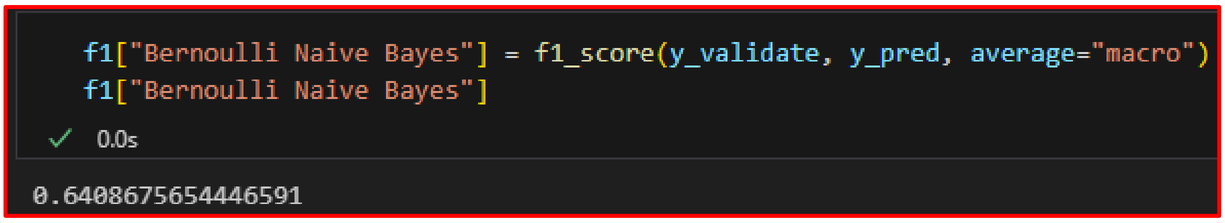
```
0.7886113963454376
```

    - The result 0.789 is the accuracy score of a Random Forest model on a validation set.
    - The accuracy score suggests that the model is able to correctly predict the labels for approximately 78,9% of the samples in the validation set.
  - Draw a Precision-Recall (PR) Curve for the Random Forest model on the validation dataset, which is a way to evaluate the performance of the classification model.



```
Y = label_binarize(y_validat
                              View
                              DataFrame with shape (17487, 60)
y_score = rfc.predict_proba(X_validate)

precision["Random Forest"], recall["Random Forest"], _ = precision_recall_curve(Y.ravel(), y_score.ravel())
fpr["Random Forest"], tpr["Random Forest"], _ = roc_curve(Y.ravel(), y_score.ravel())

plt.figure(figsize=(18, 10))
plt.step(recall["Random Forest"], precision["Random Forest"], where="post")

plt.xlabel("Recall")
plt.ylabel("Precision")
plt.xlim([0, 1])
plt.ylim([0, 1.01])
plt.title("PR Curve - Random Forest")
plt.show()
```

- Native Bayes
  - Calculate F1-Score for the Naïve Bayes model on the validation dataset



```
f1["Bernoulli Naive Bayes"] = f1_score(y_validate, y_pred, average="macro")
f1["Bernoulli Naive Bayes"]
✓ 0.0s

0.6408675654446591
```

- The result 0.641 is the accuracy score of a Naïve Bayes model on a validation set.
- The accuracy score suggests that the model is able to correctly predict the labels for approximately 64,1% of the samples in the validation set.

4.3. ROC Curve Algorithm Evaluation

4.3.1. ROC Curve Algorithm Evaluation introduction

An ROC curve (receiver operating characteristic curve) is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters:

- True Positive Rate
- False Positive Rate

True Positive Rate (TPR) is a synonym for recall and is therefore defined as follows:

$$TPR = \frac{TP}{TP + FN}$$

And False Positive Rate (FPR) is defined as follows:

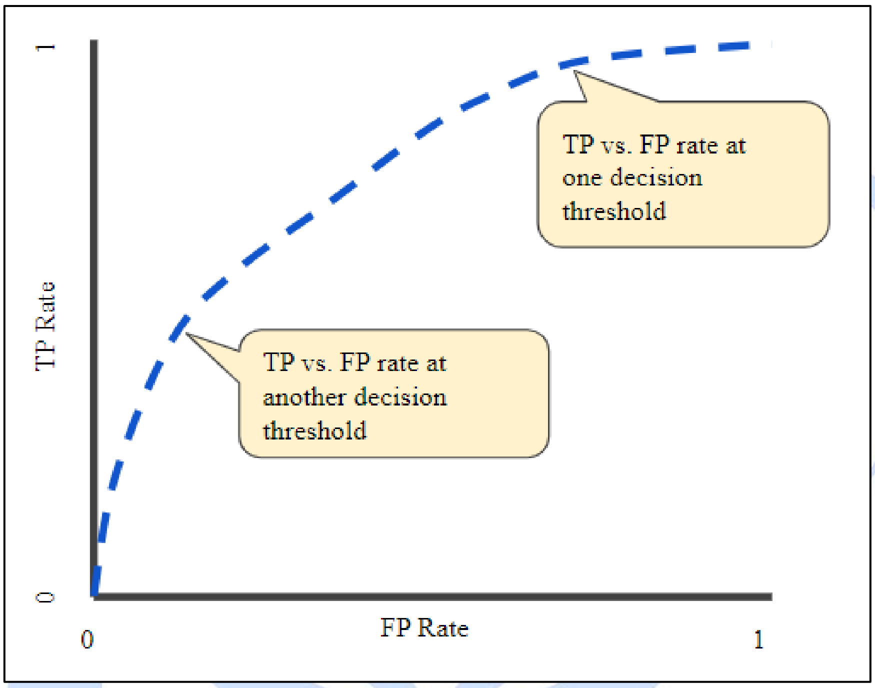
$$FPR = \frac{FP}{FP + TN}$$

Where:

- TP: True Positive (test positive in actually positive cases)

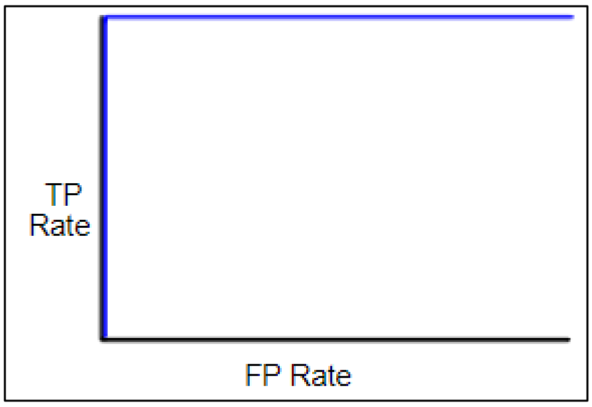
- FN: False Negative (test negative in actually positive cases)
- FP: False Positive (test positive in actually negative cases)
- TN: True Negative (test negative in actually negative cases) [6]

An ROC curve plots TPR vs. FPR at different classification thresholds. Lowering the classification threshold classifies more items as positive, thus increasing both False Positives and True Positives. The following figure shows a typical ROC curve.



[7]

ROC curves typically feature true positive rate on the Y axis, and false positive rate on the X axis. This means that the top left corner of the plot is the “ideal” point - a false positive rate of zero, and a true positive rate of one.



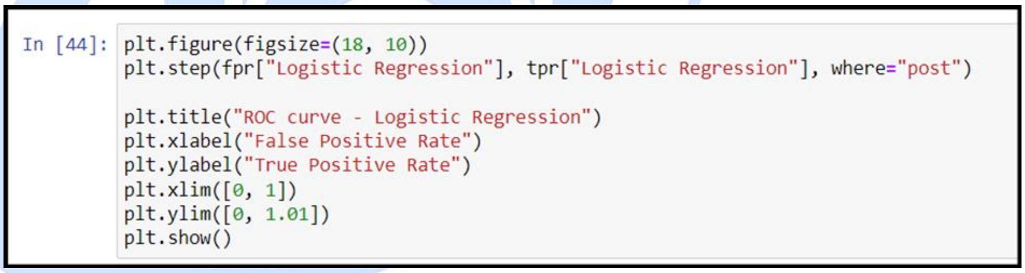
This is not very realistic, but it does mean that a larger area under the curve (AUC) is usually better. [8]

4.3.2. Selected reasons

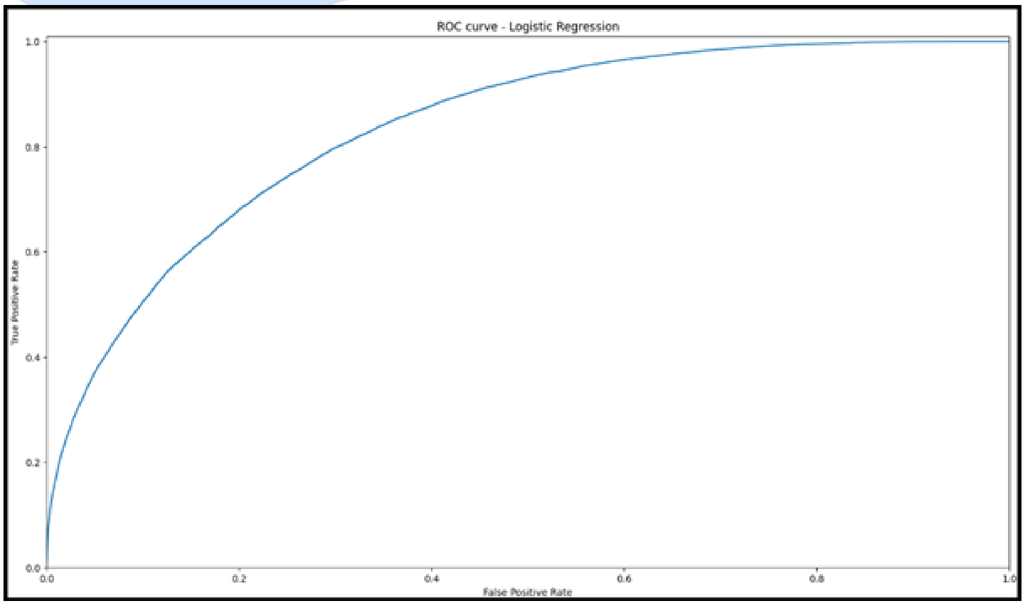
ROC curves are typically used in binary classification to study the output of a classifier. In order to extend ROC curve and ROC area to multi-label classification, it is necessary to binarize the output. One ROC curve can be drawn per label, but one can also draw a ROC curve by considering each element of the label indicator matrix as a binary prediction (micro-averaging). [8]

4.3.3. Calculate ROC Curve

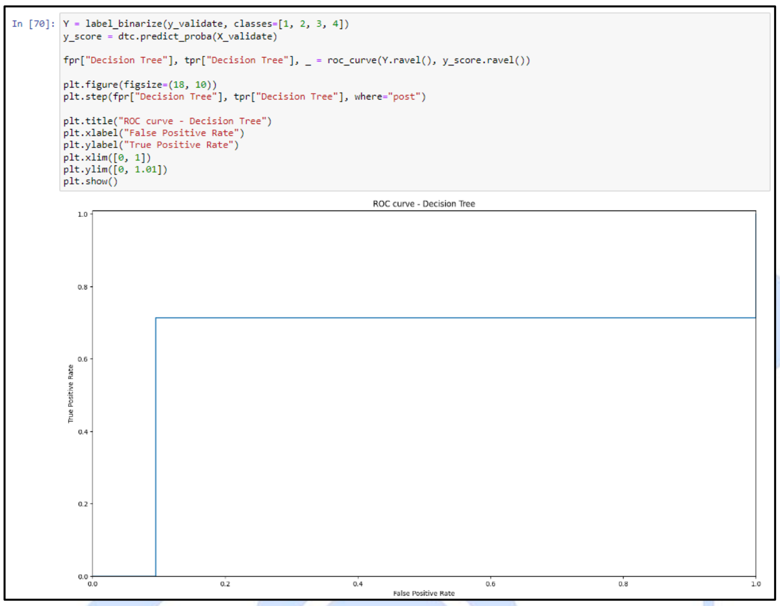
- Logistic Regression Algorithm
    - Creates an ROC (Receiver Operating Characteristic) curve for a Logistic Regression model. This curve is used to evaluate the model's predictive ability and displays the ratio between True Positive Rate and False Positive Rate.



```
In [44]: plt.figure(figsize=(18, 10))
         plt.step(fpr["Logistic Regression"], tpr["Logistic Regression"], where="post")

         plt.title("ROC curve - Logistic Regression")
         plt.xlabel("False Positive Rate")
         plt.ylabel("True Positive Rate")
         plt.xlim([0, 1])
         plt.ylim([0, 1.01])
         plt.show()
```

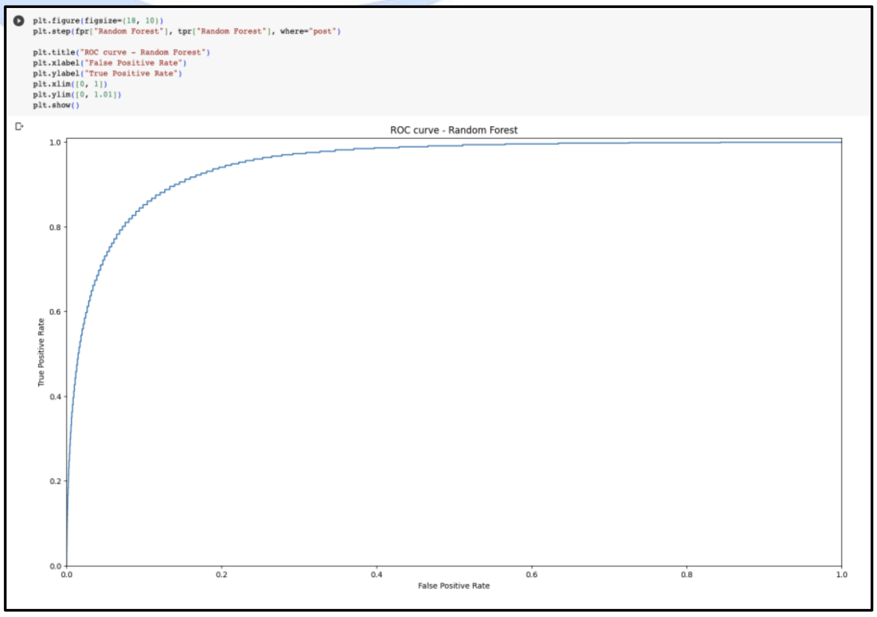


- Decision Tree
    - Creates an ROC (Receiver Operating Characteristic) curve for a Decision Tree model. This curve is used to evaluate the model's predictive ability and displays the ratio between True Positive Rate and False Positive Rate.



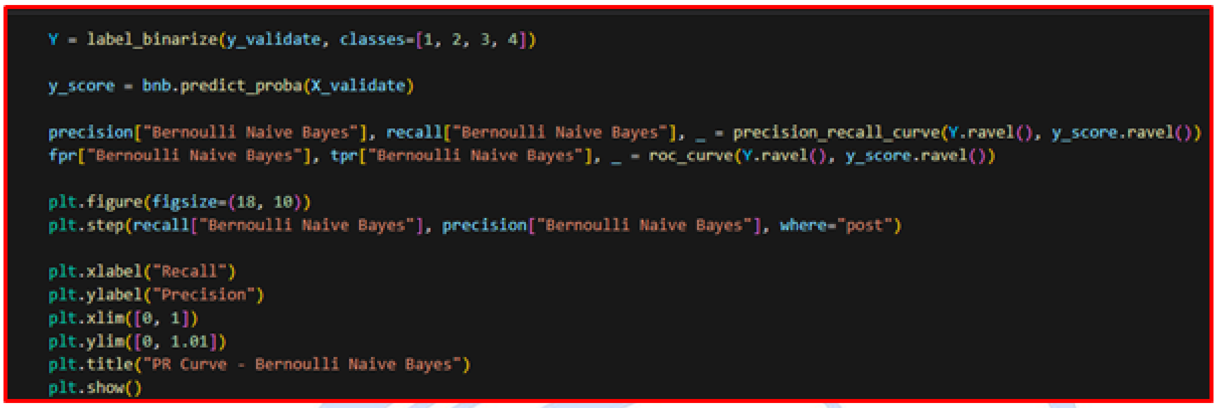
```
In [70]: Y = label_binarize(y_validate, classes=[1, 2, 3, 4])
         y_score = dtc.predict_proba(X_validate)

         fpr["Decision Tree"], tpr["Decision Tree"], _ = roc_curve(Y.ravel(), y_score.ravel())

         plt.figure(figsize=(18, 10))
         plt.step(fpr["Decision Tree"], tpr["Decision Tree"], where="post")

         plt.title("ROC curve - Decision Tree")
         plt.xlabel("False Positive Rate")
         plt.ylabel("True Positive Rate")
         plt.xlim([0, 1])
         plt.ylim([0, 1.01])
         plt.show()
```

- Random Forest
    - Generates a plot that visualizes the ROC curve for the Random Forest classifier. The ROC curve is a graphical representation of the trade-off between the true positive rate and the false positive rate for different classification thresholds.



```
plt.figure(figsize=(18, 10))
plt.step(fpr["Random Forest"], tpr["Random Forest"], where="post")

plt.title("ROC curve - Random Forest")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.xlim([0, 1])
plt.ylim([0, 1.01])
plt.show()
```

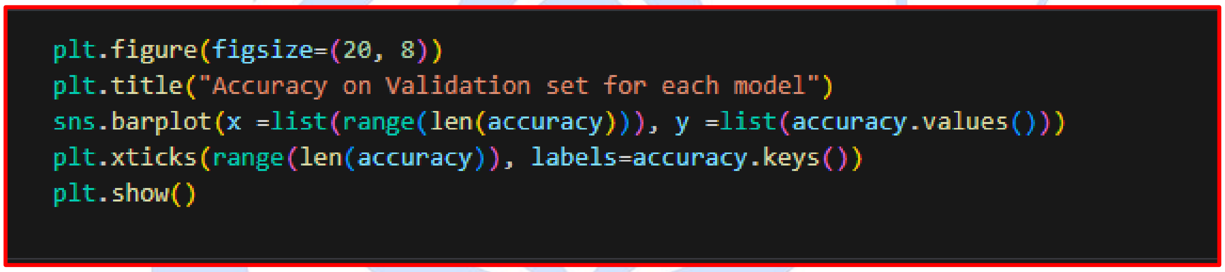
- Native Bayes
    - Creates an ROC (Receiver Operating Characteristic) curve for a Logistic Regression model. This curve is used to evaluate the model's predictive ability and displays the ratio between True Positive Rate and False Positive Rate



```
Y = label_binarize(y_validate, classes=[1, 2, 3, 4])

y_score = bnb.predict_proba(X_validate)

precision["Bernoulli Naive Bayes"], recall["Bernoulli Naive Bayes"], _ = precision_recall_curve(Y.ravel(), y_score.ravel())
fpr["Bernoulli Naive Bayes"], tpr["Bernoulli Naive Bayes"], _ = roc_curve(Y.ravel(), y_score.ravel())

plt.figure(figsize=(18, 10))
plt.step(recall["Bernoulli Naive Bayes"], precision["Bernoulli Naive Bayes"], where="post")

plt.xlabel("Recall")
plt.ylabel("Precision")
plt.xlim([0, 1])
plt.ylim([0, 1.01])
plt.title("PR Curve - Bernoulli Naive Bayes")
plt.show()
```

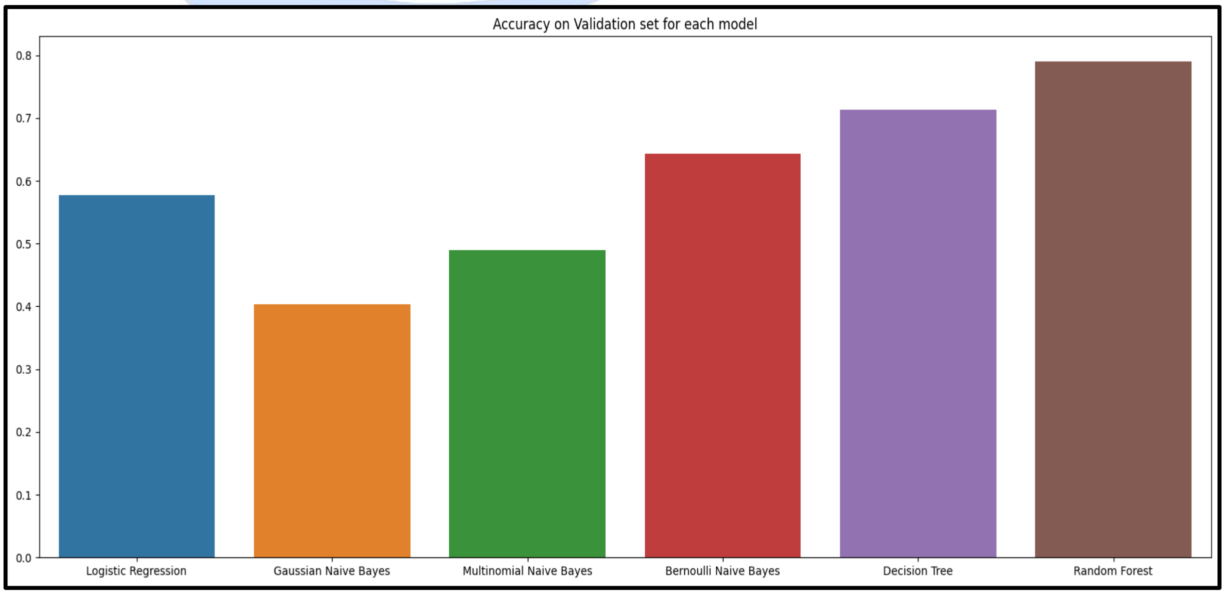
4.4. Predicting

Measure the accuracy on validation set for each model:

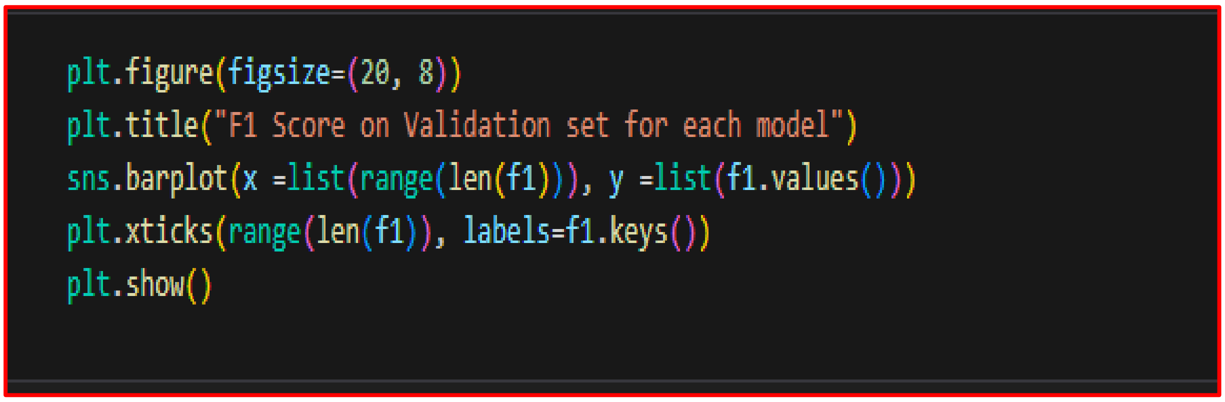


```
plt.figure(figsize=(20, 8))
plt.title("Accuracy on Validation set for each model")
sns.barplot(x =list(range(len(accuracy))), y =list(accuracy.values()))
plt.xticks(range(len(accuracy)), labels=accuracy.keys())
plt.show()
```

Accuracy graph of algorithms:

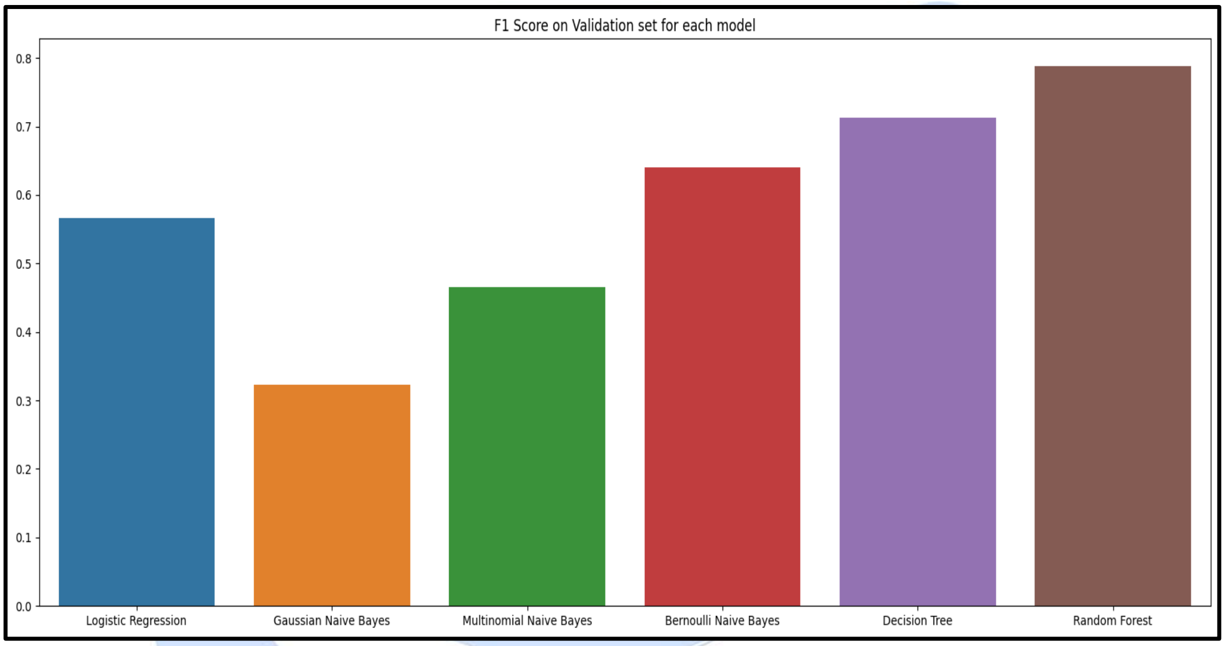


Measure F1 Score on Validation set for each model:

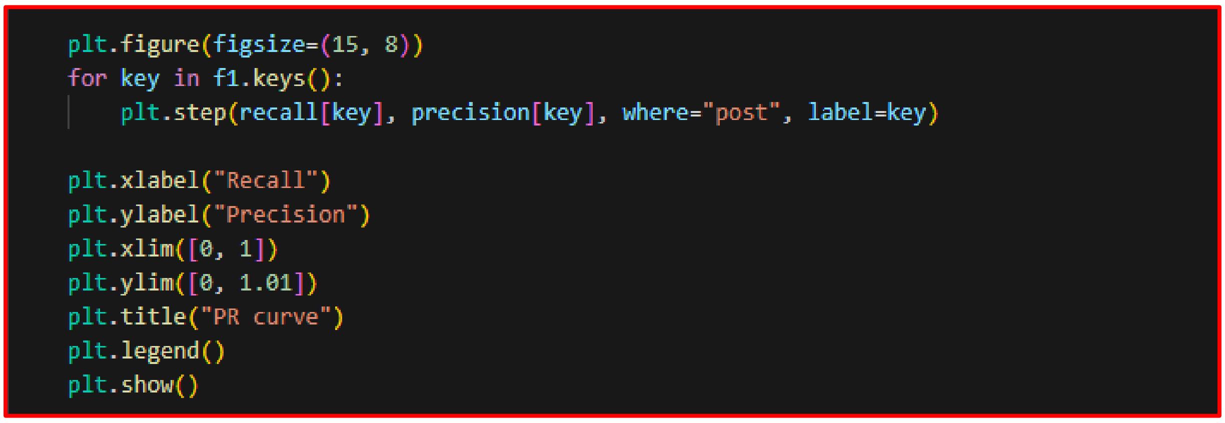


```python
plt.figure(figsize=(20, 8))
plt.title("F1 Score on Validation set for each model")
sns.barplot(x =list(range(len(f1))), y =list(f1.values()))
plt.xticks(range(len(f1)), labels=f1.keys())
plt.show()
```

F1 Score graph of algorithms:

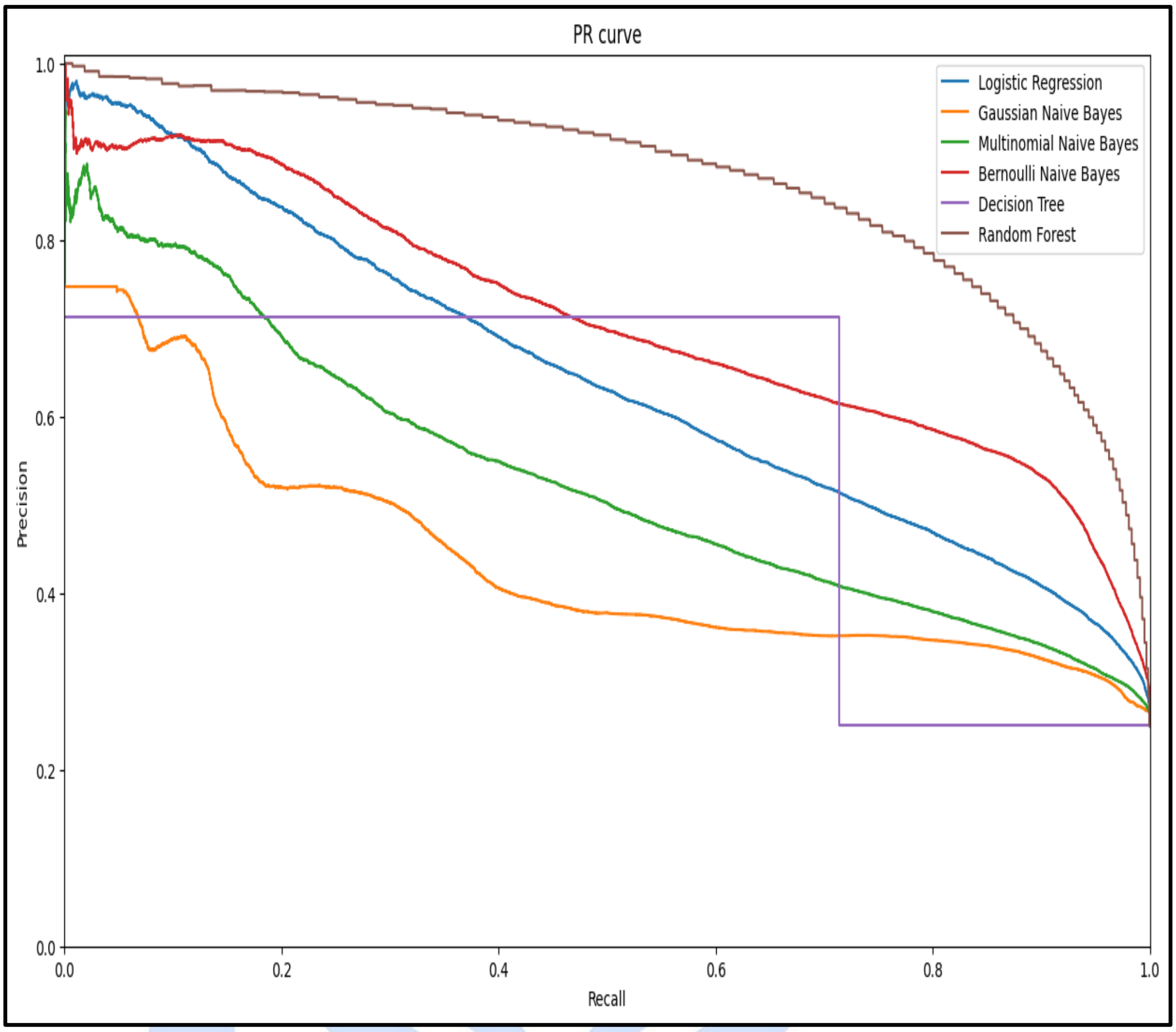


Show PR curve graph to represent the precision and recall values at different probability thresholds for a binary classification model to evaluate the performance of machine learning models:

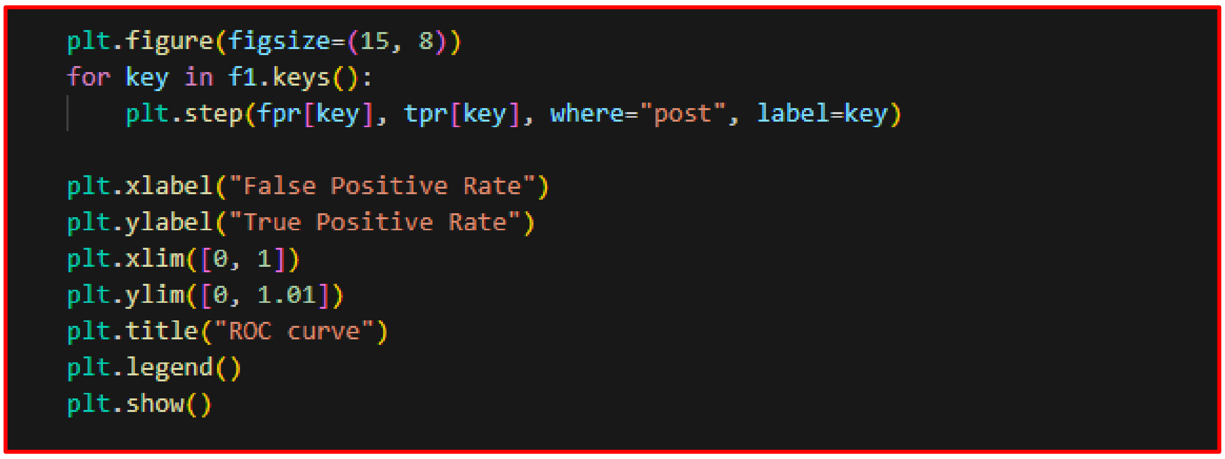


```python
plt.figure(figsize=(15, 8))
for key in f1.keys():
    plt.step(recall[key], precision[key], where="post", label=key)

plt.xlabel("Recall")
plt.ylabel("Precision")
plt.xlim([0, 1])
plt.ylim([0, 1.01])
plt.title("PR curve")
plt.legend()
plt.show()
```

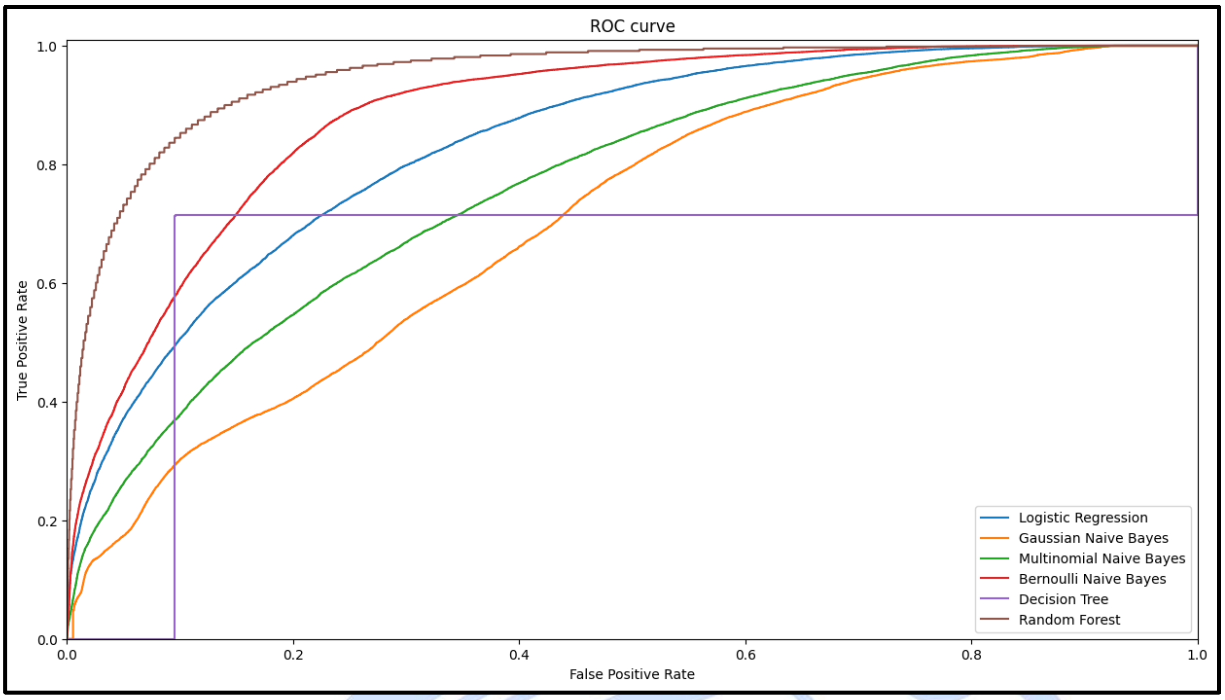
PR curve graph of algorithms:



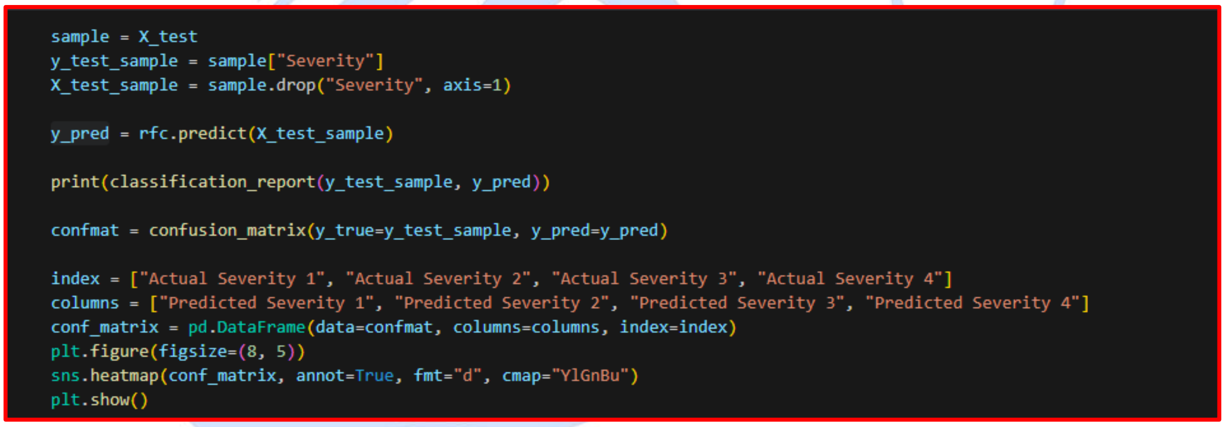
Show ROC curve graph to evaluate the performance of a binary classification model to make informed decisions about threshold selection, and compare the performance of different models:



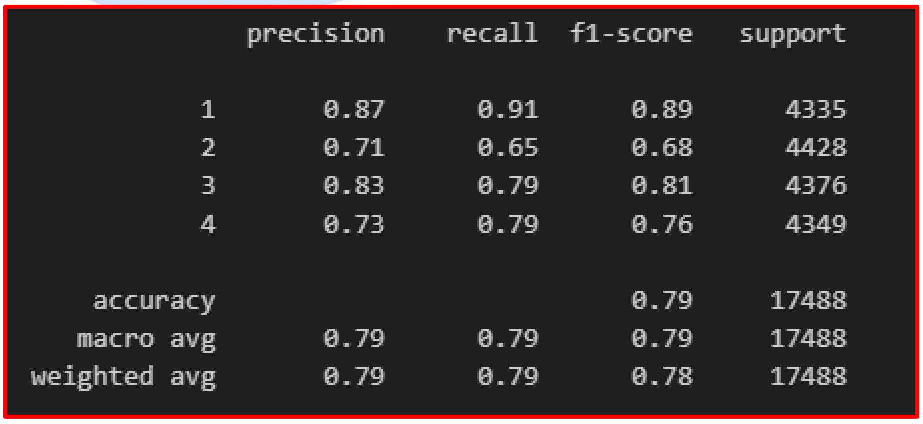
```
plt.figure(figsize=(15, 8))
for key in f1.keys():
    plt.step(fpr[key], tpr[key], where="post", label=key)

plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.xlim([0, 1])
plt.ylim([0, 1.01])
plt.title("ROC curve")
plt.legend()
plt.show()
```



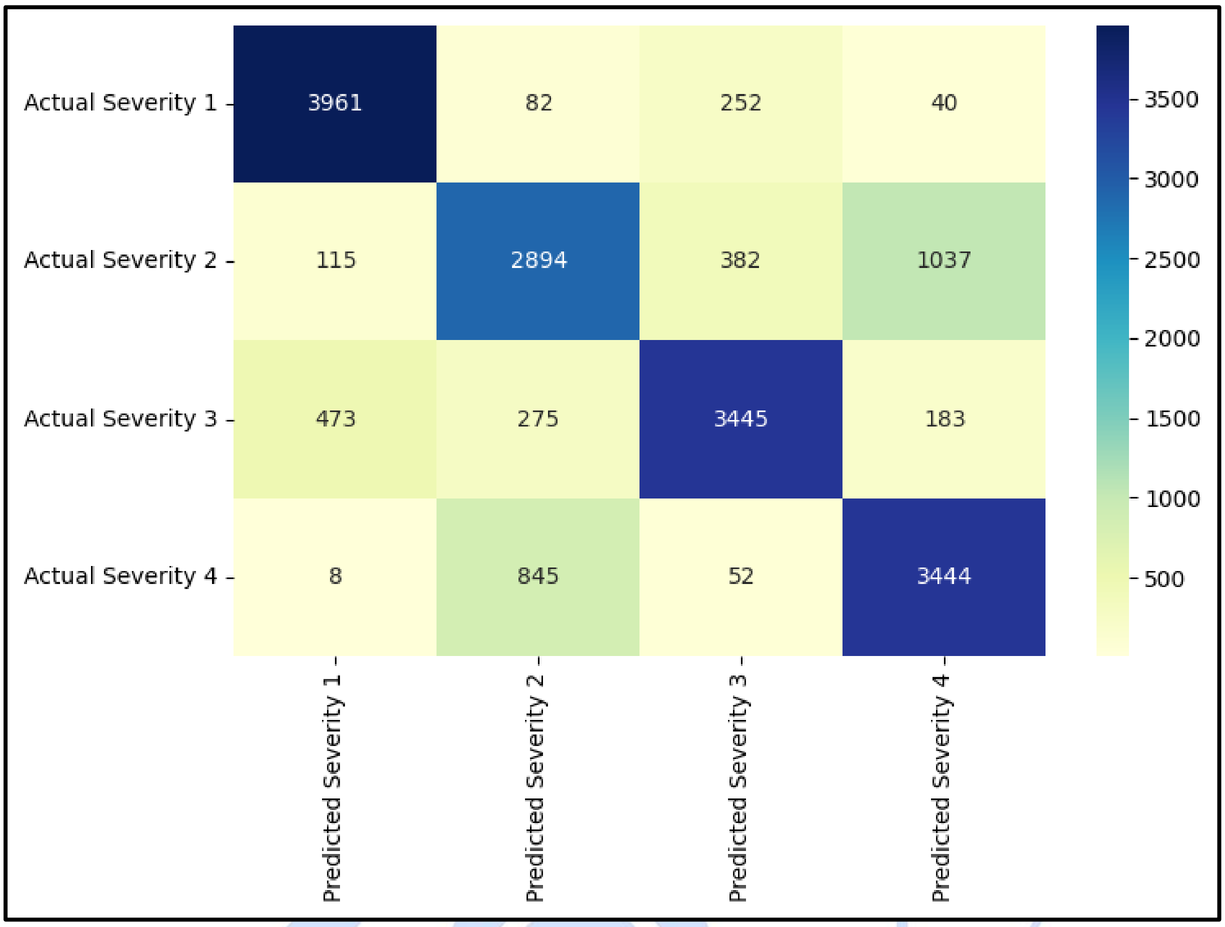
From the above graphs and results, we can derive that Random Forest is the best and most suitable algorithm for our dataset.



```
sample = X_test
y_test_sample = sample["Severity"]
X_test_sample = sample.drop("Severity", axis=1)

y_pred = rfc.predict(X_test_sample)

print(classification_report(y_test_sample, y_pred))

confmat = confusion_matrix(y_true=y_test_sample, y_pred=y_pred)

index = ["Actual Severity 1", "Actual Severity 2", "Actual Severity 3", "Actual Severity 4"]
columns = ["Predicted Severity 1", "Predicted Severity 2", "Predicted Severity 3", "Predicted Severity 4"]
conf_matrix = pd.DataFrame(data=confmat, columns=columns, index=index)
plt.figure(figsize=(8, 5))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="YlGnBu")
plt.show()
```



```
              precision    recall  f1-score   support

           1       0.87      0.91      0.89      4335
           2       0.71      0.65      0.68      4428
           3       0.83      0.79      0.81      4376
           4       0.73      0.79      0.76      4349

    accuracy                           0.79     17488
   macro avg       0.79      0.79      0.79     17488
weighted avg       0.79      0.79      0.78     17488
```



4.5. Conclusion

4.5.1. Advantages and disadvantages

Advantages:

- The application of many classification algorithms helps us to have a more diverse view of the data and helps us to classify better and more accurately.
- The classification algorithms easily allow to evaluate the performance of the model through measures such as accuracy, precision, recall, F1-score, ROC curve, PR curve, confusion matrix,

Disadvantages:

- Unbalanced data: Dataset US Accidents can suffer from data imbalance problems, i.e. uneven proportions of classes (severity levels). This can affect the predictability and performance of the classification algorithms. Therefore, it takes more steps to preprocess the dataset.

- The US Accidents dataset has a large size and a large number of input variables, some classification algorithms may require a lot of computational resources and time to train and predict.

4.5.2. Results achieved

- During the implementing the project, the team gained a lot of knowledge related to data mining on a specific data set: pre-processing, layering, prediction using learned algorithms. After agreeing on opinions, planning, dividing the work that can be done into a complete project with the model.
- Helps us gain a deeper understanding of how existing datasets are handled and how the classification algorithm works.

4.5.3. Development orientation

In the future, the team plans to make the following improvements:

- Research algorithms to reduce data width to select the right column that gives the algorithm the highest accuracy
- Improve the speed of retrieving forecast results
- Better research and data preprocessing to reduce computational resources and training and prediction time.

REFERENCES

[1] L. Rokach and O. Maimon, “Decision Trees,” in *The Data Mining and Knowledge Discovery Handbook*, 2005, pp. 165–192. doi: 10.1007/0-387-25465-X_9.

[2] “1.10. Decision Trees,” *scikit-learn*. https://scikit-learn/stable/modules/tree.html (accessed Jun. 23, 2023).

[3] “Classification Algorithms - NaÃ¯ve Bayes.” https://www.tutorialspoint.com/machine_learning_with_python/classification_algorithms_naive_bayes.htm (accessed Jun. 23, 2023).

[4] M. Grandini, E. Bagli, and G. Visani, “Metrics for Multi-Class Classification: an Overview.” arXiv, Aug. 13, 2020. Accessed: Jun. 23, 2023. [Online]. Available: http://arxiv.org/abs/2008.05756

[5] “Precision-Recall Curve | ML,” *GeeksforGeeks*, Jul. 19, 2019. https://www.geeksforgeeks.org/precision-recall-curve-ml/ (accessed Jun. 23, 2023).

[6] S. H. Park, J. M. Goo, and C.-H. Jo, “Receiver Operating Characteristic (ROC) Curve: Practical Review for Radiologists,” *Korean J Radiol*, vol. 5, no. 1, p. 11, 2004, doi: 10.3348/kjr.2004.5.1.11.

[7] “Classification: ROC Curve and AUC | Machine Learning,” *Google for Developers*. https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc (accessed Jun. 23, 2023).

[8] “Receiver Operating Characteristic (ROC),” *scikit-learn*. https://scikit-learn/stable/auto_examples/model_selection/plot_roc.html (accessed Jun. 23, 2023).