US Accidents dataset Analysis & Storytelling

Data Analytics Assignment

Ghada NAIT SAID email: **naitsaidghada@gmail.com**
Ahmed NAIT SAID email: **anaitsaid98@gmail.com**
Abdelheq MOKHTARI email: **mokhtari.abdelheq@gmail.com**

Erasmus Students from University of BATNA 2, Algeria

University of Piraeus – Spring 2022

Abstract

This assignment is about processing and analyzing the “US Accidents” dataset available on Kaggle. The dataset is a countrywide car accident dataset, which covers 49 states of the USA. The accidents data presented are collected from February 2016 to Dec 2021. The purpose here is an exploratory data analysis, constructing a “data story", and enhancing any conclusions using Machine Learning models.

Program Structure

The program is written purely in “python” since it has a well-established set of libraries for data manipulation, processing, and graphical visualization. Using libraries such as:

- ✓ Pandas.
- ✓ Plotly.
- ✓ Numpy.
- ✓ IO.
- ✓ Scikit-learn.
- ✓ TensorFlow.
- ✓ Category encoders.
- ✓ Matplotlib.
- ✓ Yellowbrick
- ✓ Statistics.
- ✓ Seaborn.
- ✓ Keras.

The program is divided into different python files, in which each file corresponds to a part of the assignment. All these separate files are executed successively, where each will create the necessary data files and graphs to be used by the next part's file.

Part I: Preprocessing

- We first load the dataset into a data frame named “df”, then we extract the dataset description using the info() method. In order to save the output of this method in a TXT file, we use a buffer.

```
# load dataset into "df" DataFrame
df = pd.read_csv('US_Accidents_Dec21_updated.csv')

buffer = io.StringIO()
df.info(buf=buffer)
info = buffer.getvalue()
with open("infoBefore.txt", "w", encoding="utf-8") as f:
    f.write(info)
df.describe().to_csv('descriptionBefore.csv')
```

- The next thing to do is check for any missing data and sort them out in a CSV file as well.

```
NumNaN = df.isnull().sum().sort_values(ascending=False)
PerNaN = df.isnull().sum().sort_values(ascending=False) / df.shape[0]
with open("missingDataBefore.csv", "w", encoding="utf-8") as f:
    f.write("Feature,Missing Data,Percentage\n")
    for idx in NumNaN.index.values:  # Checking for missing data
        line = idx + "," + str(NumNaN[idx]) + "," + "{:.2%}".format(PerNaN[idx])
        f.write(line + "\n")
```

- Here we start creating a list of features to drop that are deemed irrelevant or too vague:

```
# List for features to drop for dimensionality reduction
# Dropping ID column since it doesn't provide anything useful
# Dropping End_Time column as well due to irrelevance
# US Accidents data, so we are dropping the country column
# 61% of the Street Number is missing, so we are dropping it as well
# Dropping the Description columns because it's too vague ->
len(df['Description'].unique()) == 1174563 (huge variation)
# Dropping location columns because ZipCode will be used to resume them
toDrop = ['ID', 'End_Time', 'Country', 'Number', 'Description', 'Start_Lat',
'Start_Lng', 'End_Lat', 'End_Lng',
          'County', 'City', 'State', 'Street', 'Timezone', 'Airport_Code',
'Weather_Timestamp']
```

- When it comes to zip codes, we can use regular expressions to regulate their form and keep the relevant parts only, after that we only extract the 3 first digits (and cast them to strings) that represent the national area and sectional center; since these would serve as good geographical features (not too vague or too detailed).

```
# Making sure all the Zipcodes follow the correct format (5 numbers - 4 numbers OR
5 numbers)
df['Zipcode'] = df['Zipcode'].str.extract(r'(^\d{5}(?:[-\s]\d{4})?$)',
expand=False)
# We extract the three first digits of the Zipcode -> represent national area and
sectional center
df['Zipcode'] = df['Zipcode'].map(lambda x: str(x[:3]), na_action="ignore")
```

- There are also quantitative features with missing values, these values are best filled with the overall mean not to damage our data, and skew any upcoming results.

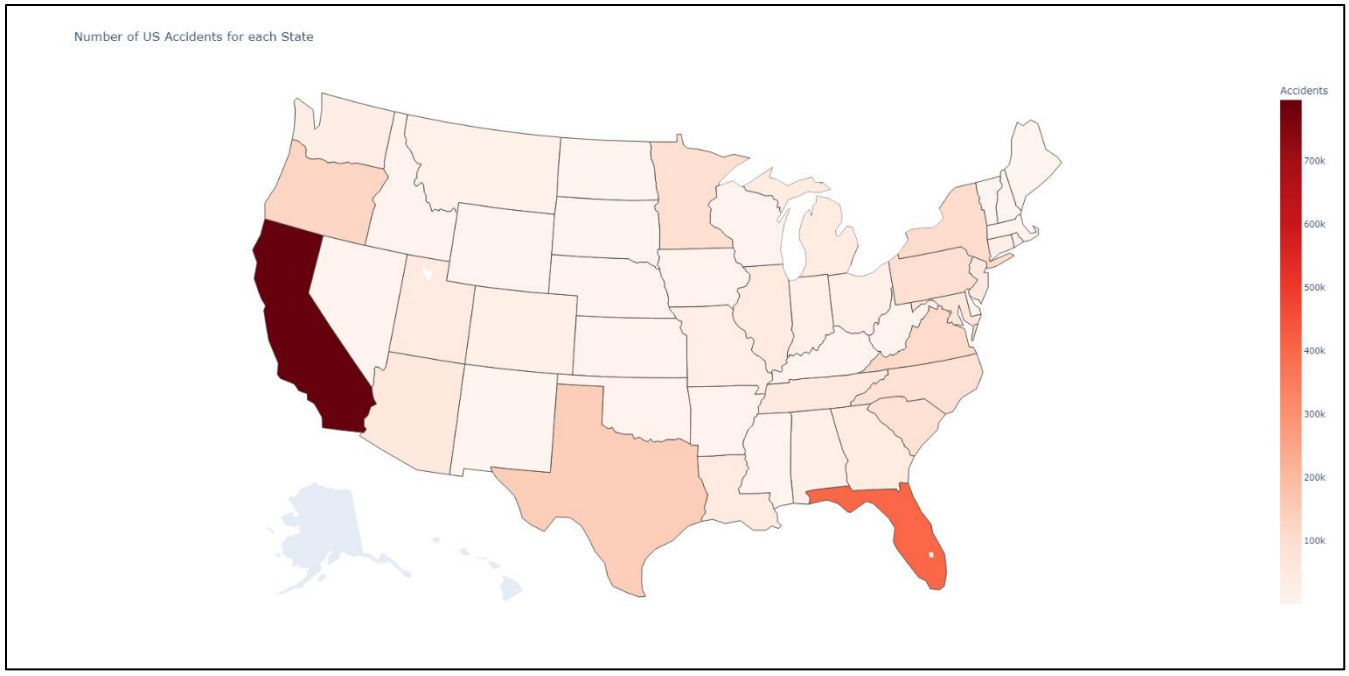
```
# Filling these missing values of numerical data with the mean value not to damage
the data when used
num_data = ['Distance(mi)', 'Temperature(F)', 'Wind_Chill(F)', 'Humidity(%)',
'Pressure(in)', 'Visibility(mi)',
            'Wind_Speed(mph)', 'Precipitation(in)']  # List of numerical data
columns
for elm in num_data:
    df[elm] = df[elm].fillna(df[elm].mean())
```

- And now we will graph the data and fix any other inconsistencies as we do so: Using the Plotly Express, we will easily plot specific features we want to visualize, using the value counts method to count the number of cases. This is basically repeated for all the figures.

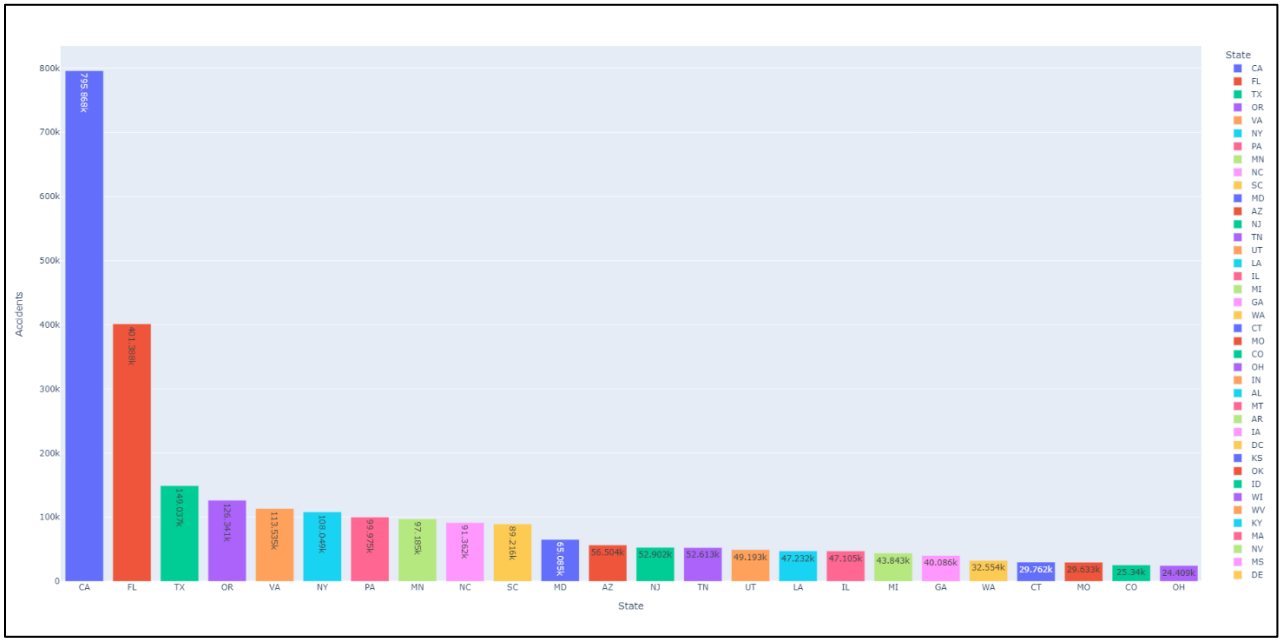
```
tr = []
fl = []
for elm in count_elm:
    count = df[elm].value_counts()
```

```
    indices = count.index.tolist()
    if len(indices) == 2:  # if the truth value varies between True & False
        tr.append(count[True])
        fl.append(count[False])
    else:  # if there's only one truth value we drop the elm as it doesn't offer
valuable info
        toDrop.append(elm)
        count_elm.remove(elm)
fig = go.Figure(data=[go.Bar(name='True', x=count_elm, y=tr),
                      go.Bar(name='False', x=count_elm, y=fl)])

fig.update_layout(barmode='group')
fig.write_html("true_false_fig.html")  # Graph for the true false features

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", colorscale="reds",
colorbar_title="Accidents"))
fig.update_layout(title_text="Number of US Accidents for each State",
geo_scope="usa")
fig.write_html("fig.html")  # Cases per state map

state_counts = state_counts.to_frame()
state_counts.rename(columns={'State': 'Count'}, inplace=True)
state_counts.insert(0, "State", state_counts.index, True)
fig = px.bar(state_counts, x="State", y="Count", color="State", labels={"Count":
"Accidents"}, text_auto=True,
             color_continuous_scale=px.colors.sequential.PuBuGn)
fig.write_html("fig2.html")  # Cases per state histogram
```

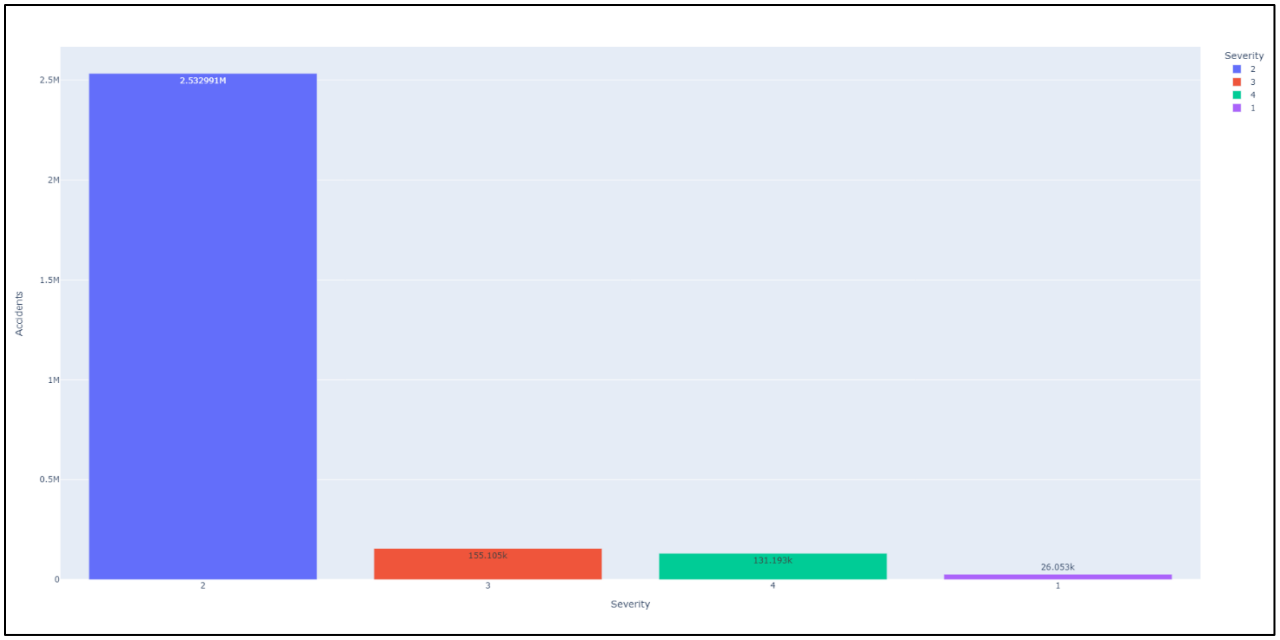
- We repeat it to graph the following figures (these are interactive/dynamic graphs, however in this documentation we are only showing a static view of them):



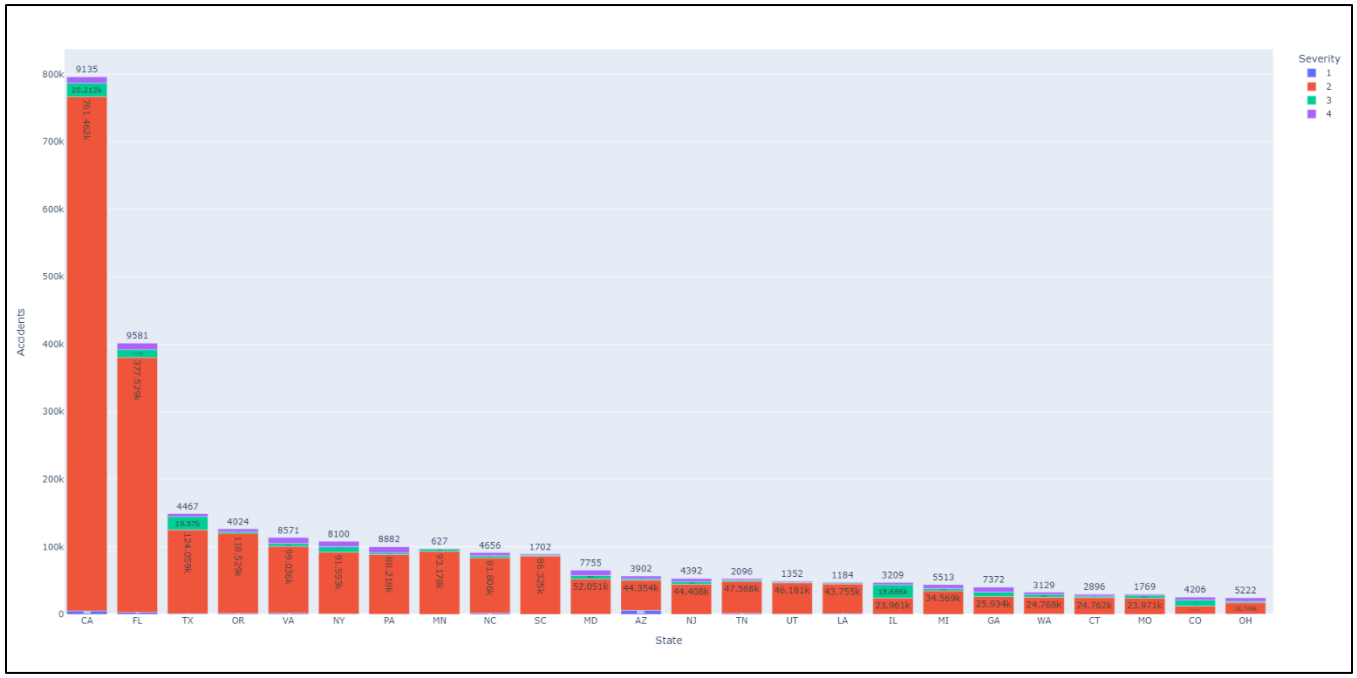
Cases by state Map



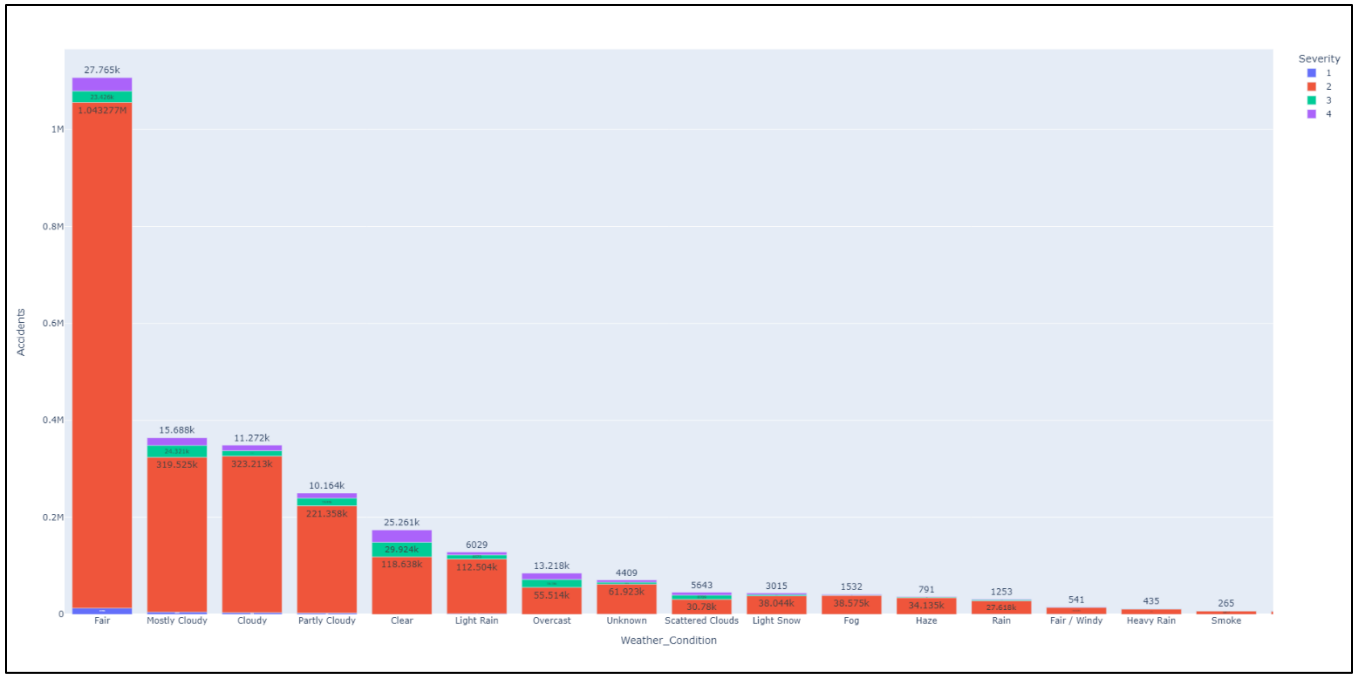
Cases by state Histogram



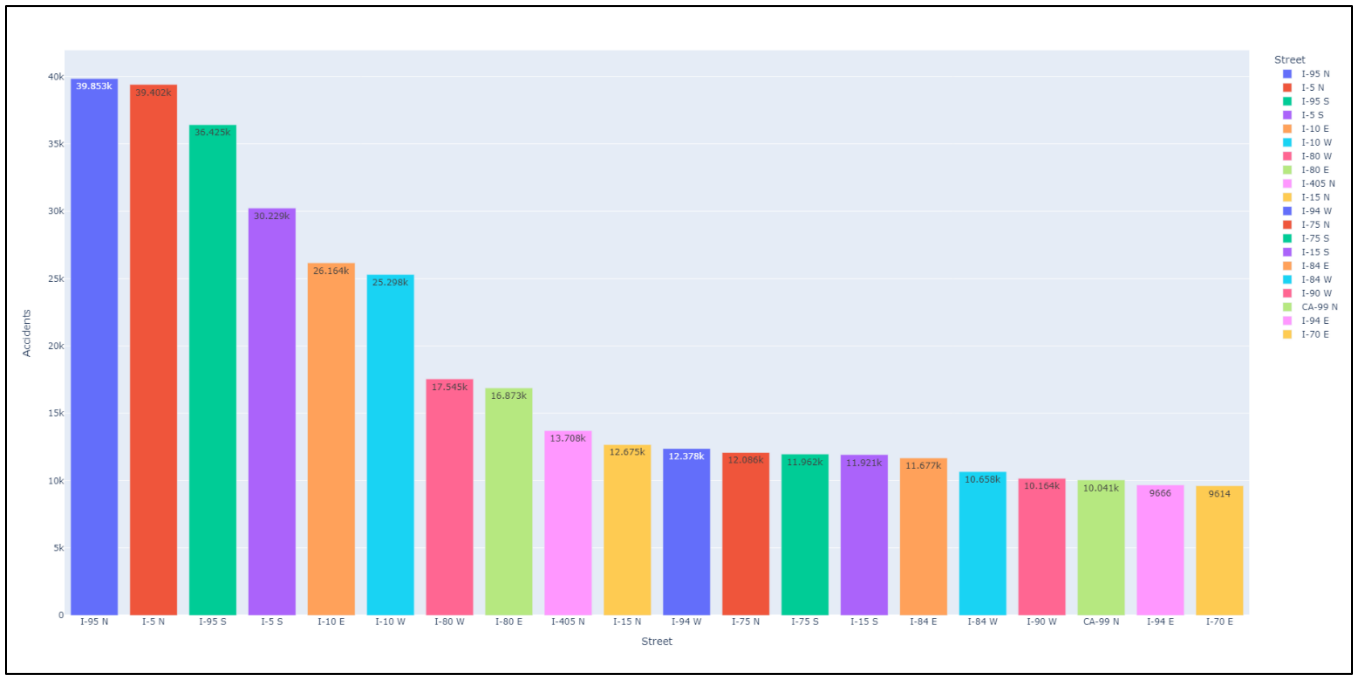
Cases per severity Histogram



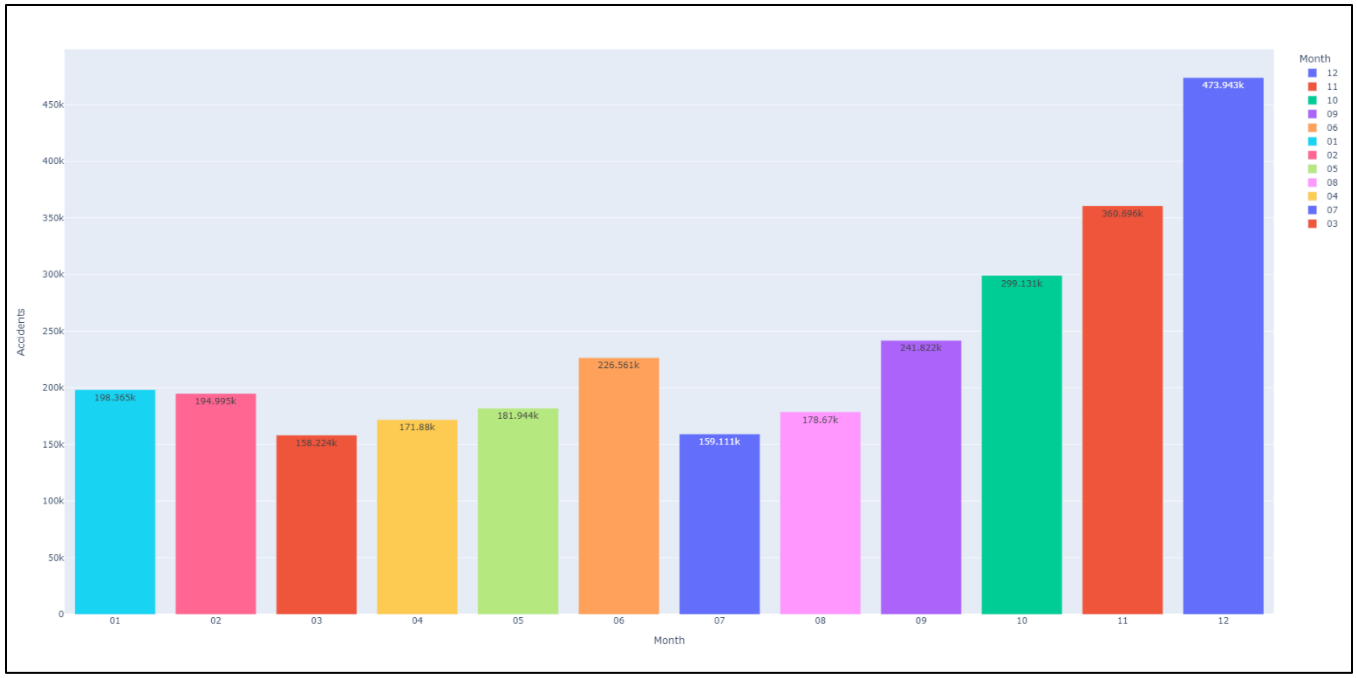
Cases and severity per state Histogram



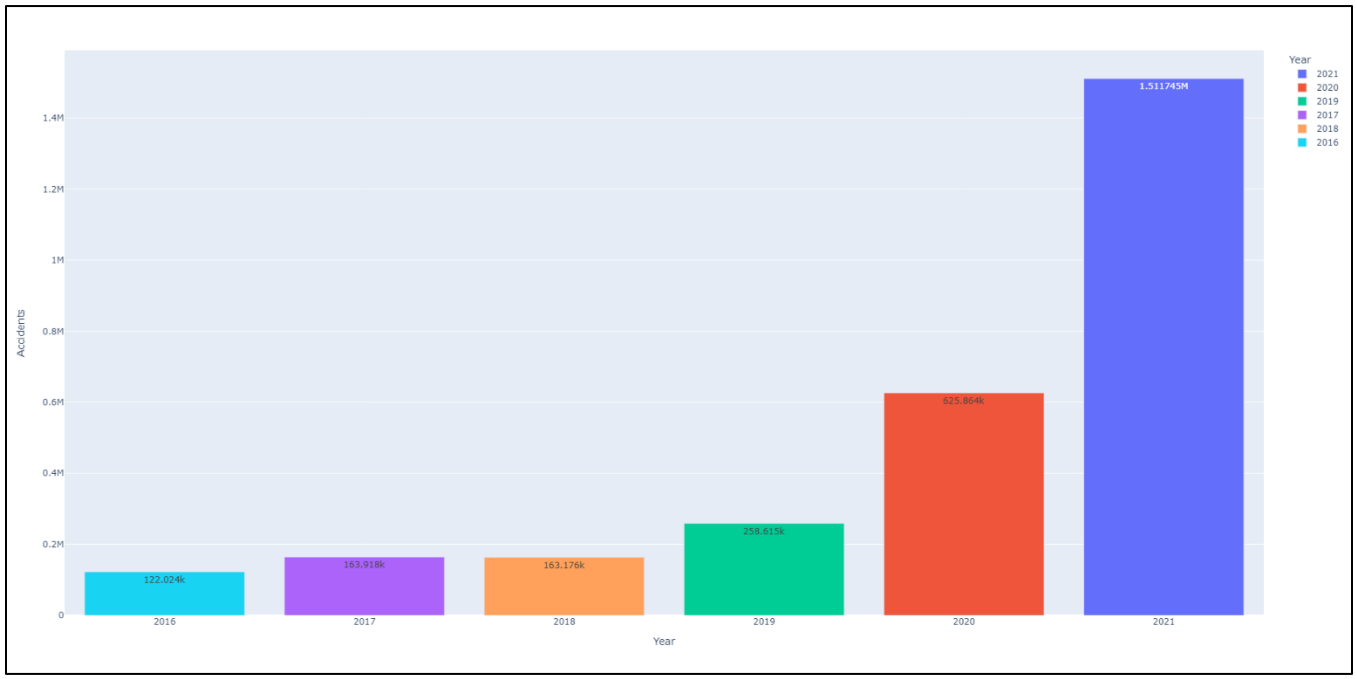
Cases and severity by weather condition



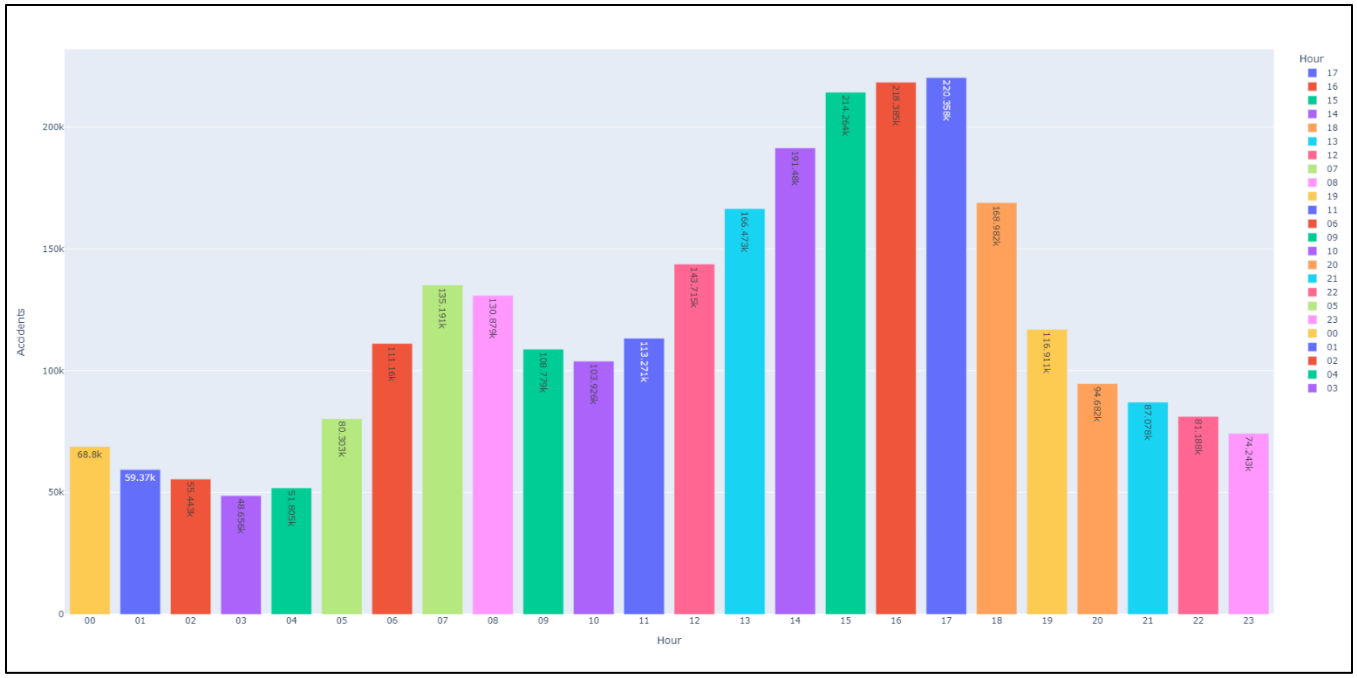
Cases by street histogram



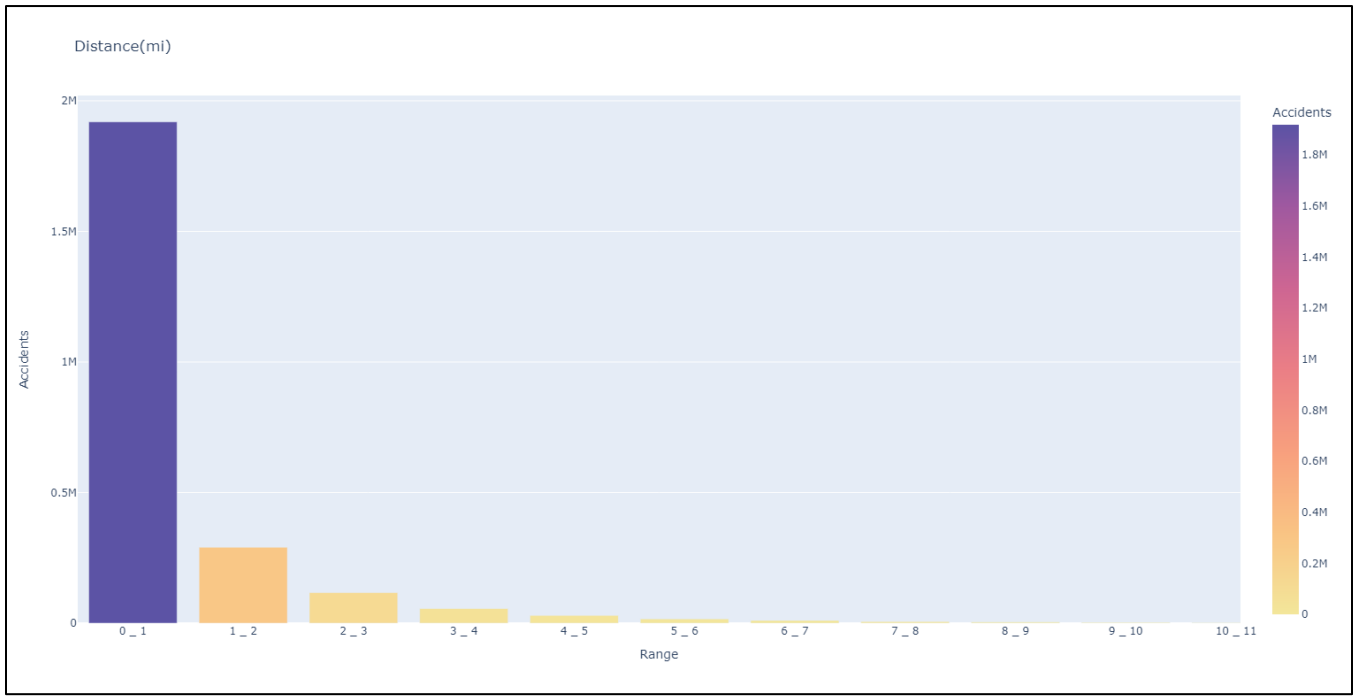
Cases by month Histogram



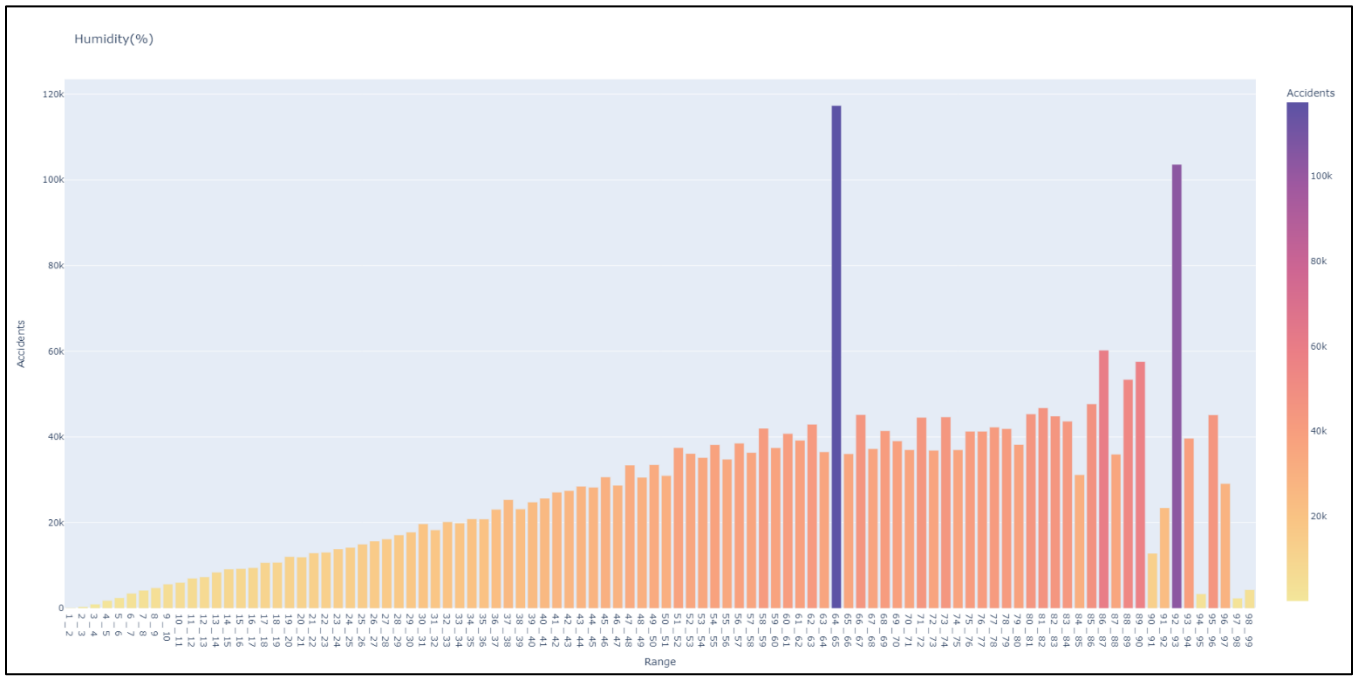
Cases by year Histogram



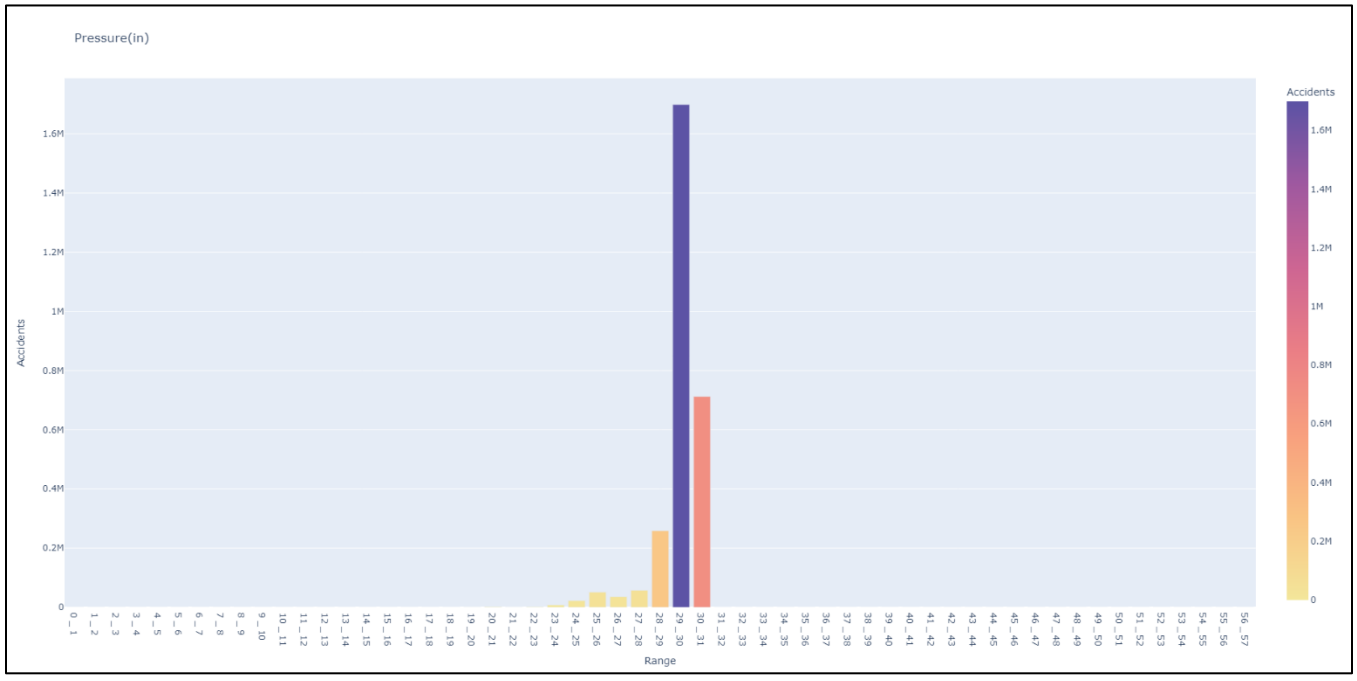
Cases by hour Histogram



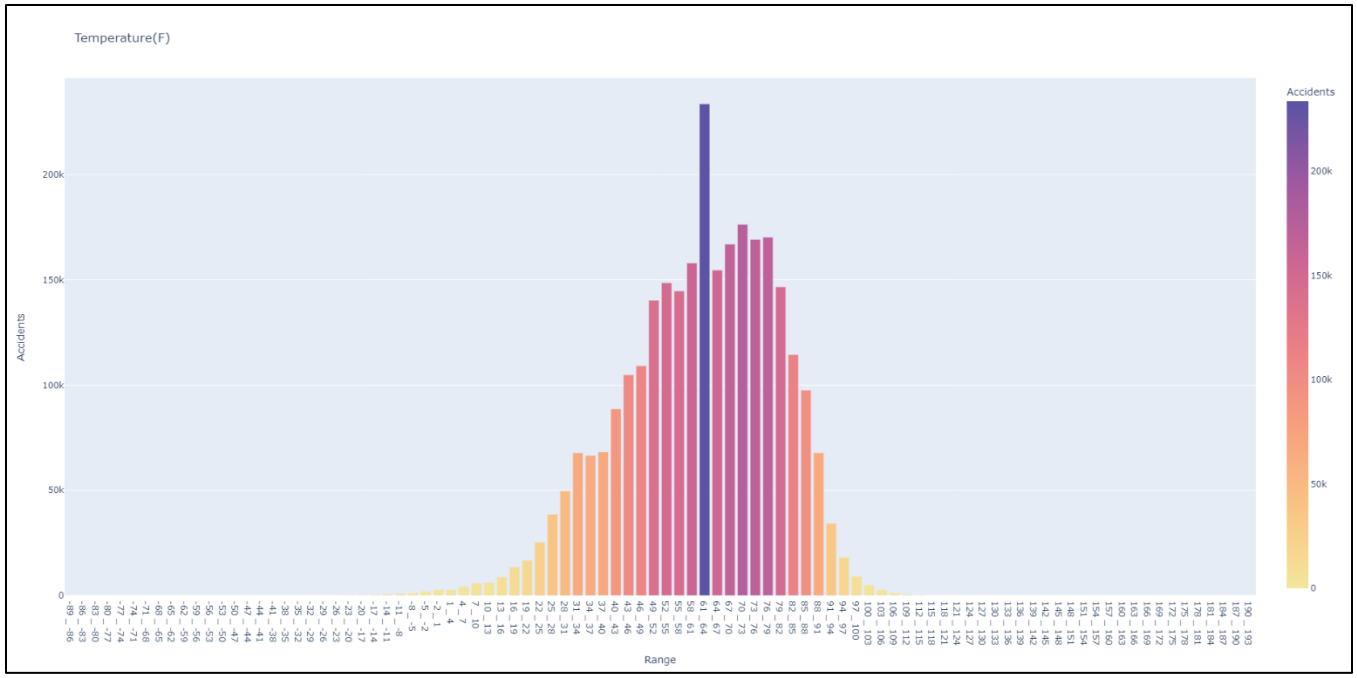
Cases by distance Histogram



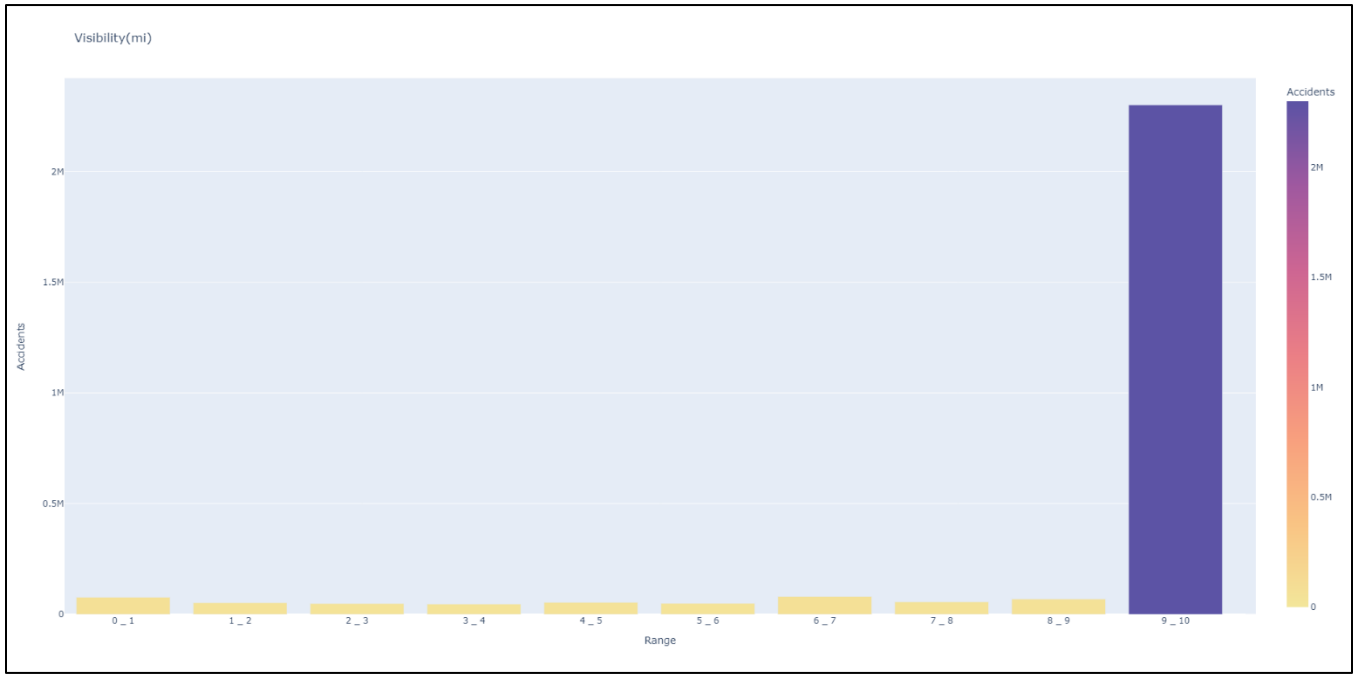
Cases by humidity Histogram



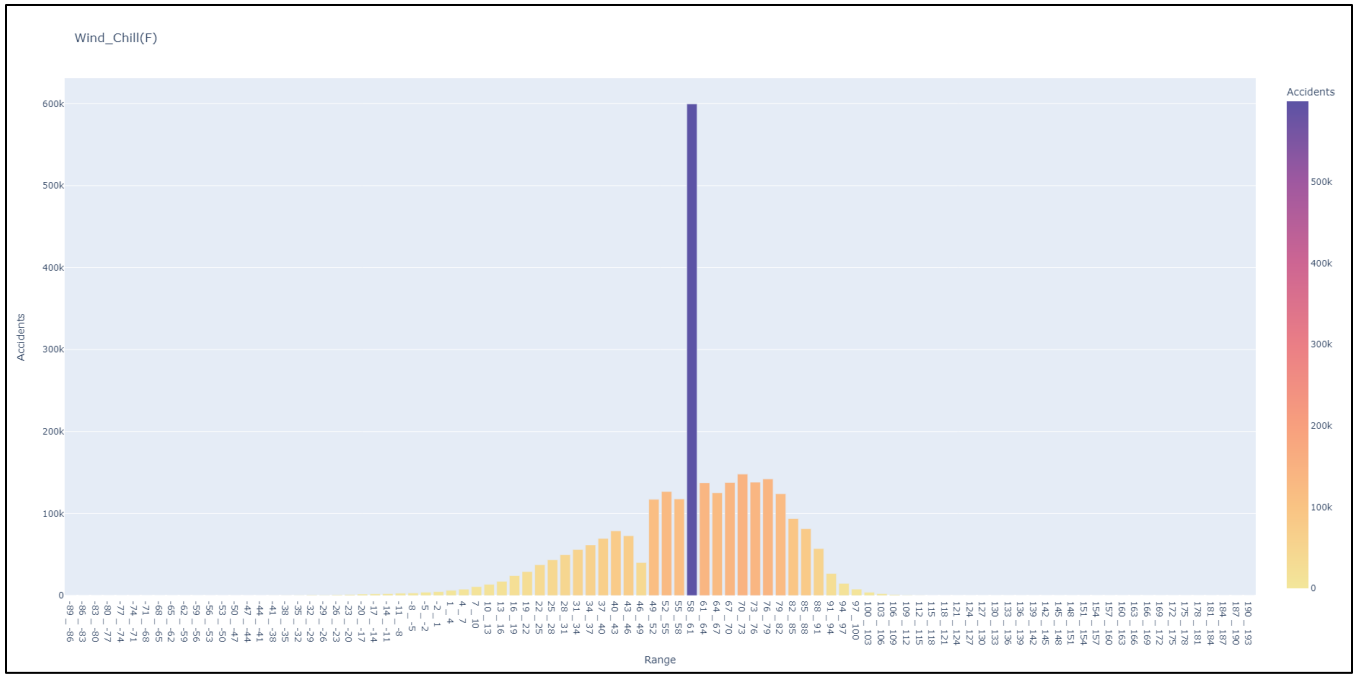
Cases by pressure Histogram



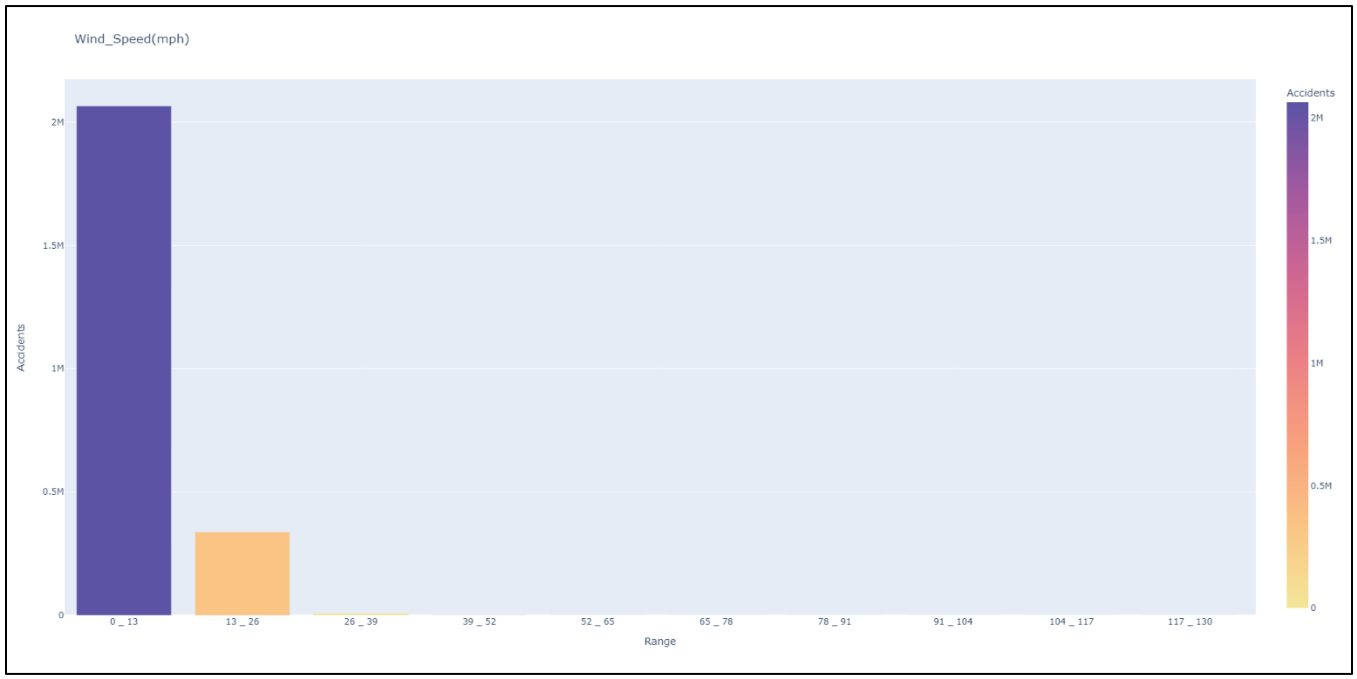
Cases by temperature Histogram



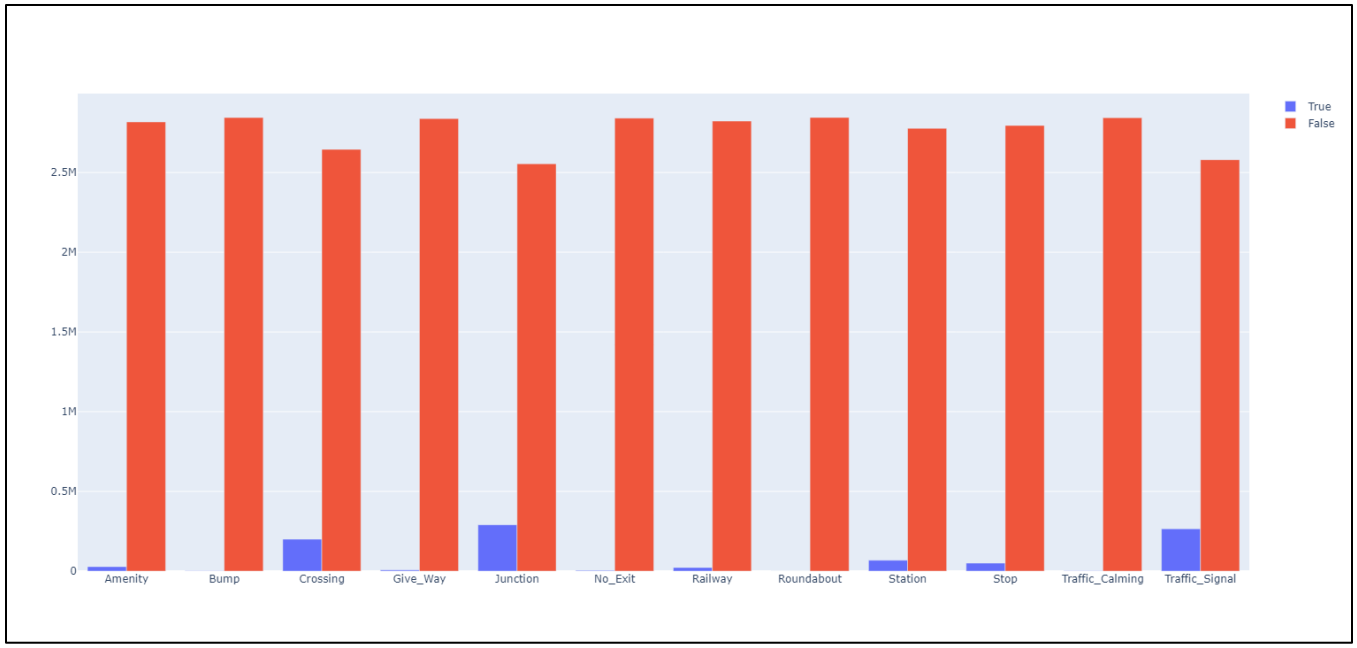
Cases by visibility Histogram



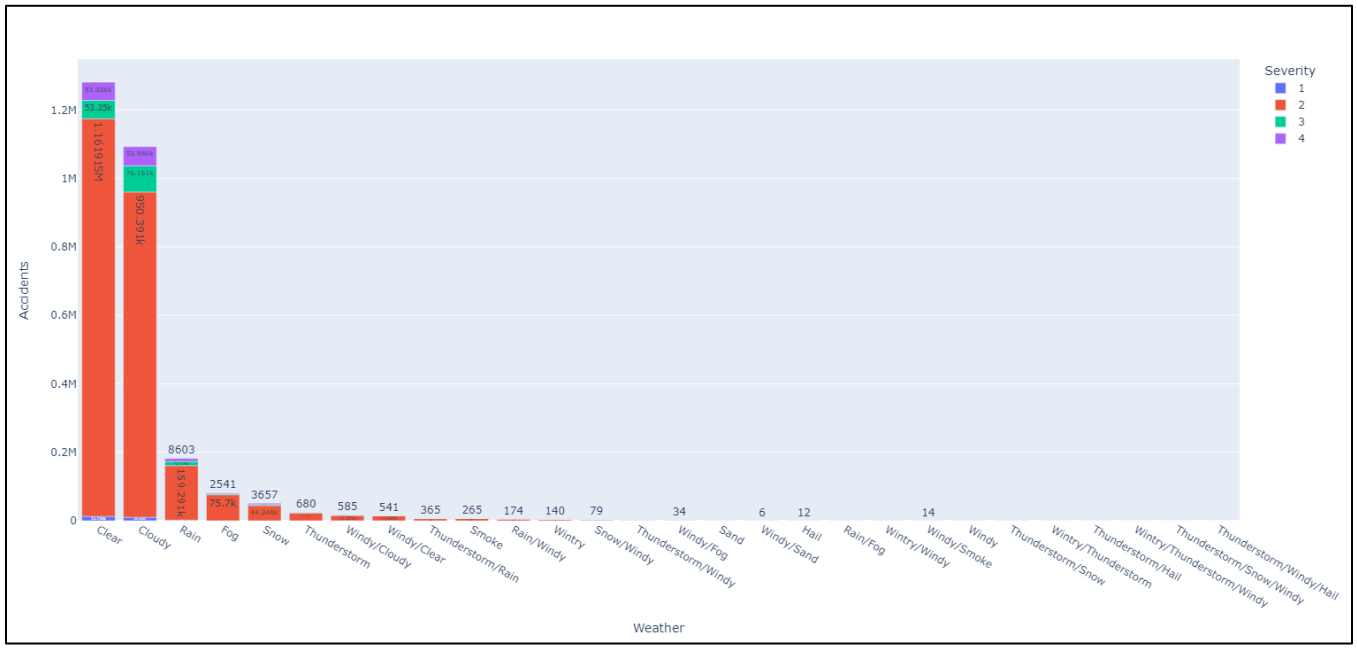
Cases by wind chill Histogram



Cases by wind speed Histogram



POI attributes T:F ratio Histogram



Cases by Weather Condition Histogram

- Here we add to the features to be dropped more low variance data columns. Afterwards, the "Start_Time" feature is separated into Year, Month, Day & Hour and it's itself added to the to drop columns; which are dropped in the next step:

```
# Dropping boolean type data that has a low variance where: False values >>> True
values
toDrop.extend(['Bump', 'Give_Way', 'No_Exit', 'Railway', 'Roundabout',
'Traffic_Calming'])

# from string to datetime -> Get the year, month, day and hour separately from the
Start_Time field and then drop it
df['Start_Time'] = pd.to_datetime(df["Start_Time"])
df['Year'], df['Month'], df['Day'], df['Hour'] = df['Start_Time'].dt.year,
df['Start_Time'].dt.month,\
                                                 df['Start_Time'].dt.day,
df['Start_Time'].dt.hour
toDrop.append('Start_Time')  # no need for this

# Feature Selection -> Dimensionality Reduction
df.drop(toDrop, inplace=True, axis=1)
```

- We update the data description:

```
buffer = io.StringIO()
df.info(buf=buffer)
info = buffer.getvalue()
with open("infoAfterSelection.txt", "w", encoding="utf-8") as f:
    f.write(info)  # new info after selection
```

- Now we do some data cleaning and sampling. First, we get the description summary of all object type data unique values:

```
# Data cleaning & Sampling
 # We get the description of all object type data
 obj_data = ['City', 'Wind_Direction', 'Weather_Condition', 'Side',
 'Sunrise_Sunset',
            'Civil_Twilight', 'Nautical_Twilight', 'Astronomical_Twilight']

 with open("categoricalDataDescriptionBefore.csv", "w", encoding="utf-8") as f:
     f.write('Feature,Count Of Unique Values')
     for elm in obj_data:
         f.write('\n' + elm + ',' + str(len(df[elm].unique())) + ',')
```

- We change the N value found in which side of the road the accident happened to NaN, because this feature is supposed to have only two entries "R" or "L" (as described by the creator of the dataset):

```
# "N" found in "Side" which is neither "R" right nor "L" left, so we change it to
NaN
df.loc[df["Side"] == 'N', "Side"] = np.nan
```

- A lot of variances when it comes to the wind direction data (north has north, north east, and northwest for example), so we will pretty much map all the data to 6 values to group it.

```
# Grouping Wind_Direction data
Variation = ["CALM", "VAR", "East", "North", "South", "West"]
Equivalence = ["Calm", "Variable", "E", "N", "S", "W"]
for var, eq in zip(Variation, Equivalence):
    df.loc[df["Wind_Direction"] == var, "Wind_Direction"] = eq
df["Wind_Direction"] = df["Wind_Direction"].map(lambda x: x if len(x) != 3 else x[-
2:], na_action="ignore")
```

- The same idea applies here to the weather condition, and then we apply a manual One Hot Encoding by creating a feature for every weather condition, assigning 1 to rows with this weather condition, else they're assigned 0, we decided to do it manually since many cases have more than one weather condition (up to three), so the manual approach allowed the precise encoding:

```
# Fixing the Weather_Condition values to more structured values
Variation = ["Wintry", "Thunder|T-Storm|Vicinity|Tornado", "Snow|Sleet",
"Rain|Drizzle", "Wind|Squalls", "Hail|Pellets",
             "Clear|Fair", "Cloud|Overcast", "Mist|Haze|Fog", "Sand|Dust",
"Smoke|Volcanic Ash"]
Equivalence = ["Wintry", "Thunderstorm", "Snow", "Rain", "Windy", "Hail", "Clear",
"Cloudy", "Fog", "Sand", "Smoke"]

df['Weather'] = [[] for _ in range(len(df.index))]
for var, eq in zip(Variation, Equivalence):
    weather = df.loc[df["Weather_Condition"].str.contains(var, na=False),
"Weather"]
    # df.loc[df["Weather_Condition"].str.contains(var, na=False), "Weather"] +=
[eq]
    for val in weather:
        val.append(eq)
df["Weather"] = df["Weather"].str.join('/')
df.loc[df["Weather"] == '', "Weather"] = np.NaN

for weather in Equivalence:  # To create new columns for all different weather
conditions
    df[weather] = 0
for var, eq in zip(Variation, Equivalence):
    count = df["Weather"].str.contains(var, na=False).sum()
    if count > 0:
        df.loc[df["Weather"].str.contains(var, na=False), eq] = 1
```

- And this is a visualization code for the final result of the weather processing (presented previously with other plots).

```
# Weather-Severity-Accidents Count bar chart
weather_severity = df[['Severity', 'Weather']].value_counts()
weather_severity = weather_severity.to_frame()
weather_severity = weather_severity.unstack(level='Severity')
weather_severity.columns = weather_severity.columns.droplevel()
fig = px.bar(weather_severity, labels={'value': 'Accidents'}, text_auto=True)
fig.update_layout(xaxis={'categoryorder': 'total descending'})
fig.write_html("fig5.html")
```

- Then we check for missing data after:

```
NumNaN = df.isnull().sum().sort_values(ascending=False)
PerNaN = df.isnull().sum().sort_values(ascending=False) / df.shape[0]
with open("missingDataAfter.csv", "w", encoding="utf-8") as f:
    f.write("Feature,Missing Data,Percentage\n")
    for idx in NumNaN.index.values:  # Checking for missing data
        line = idx + "," + str(NumNaN[idx]) + "," + "{:.2%}".format(PerNaN[idx])
        f.write(line + "\n")
```

- As it is illustrated below, we have now our data ready, all we need to do is drop rows with missing data, and the weather condition feature since it is already represented by the encoding:

```
# The rest of the data is well categorized, and we already replaced NaN with the
mean for numerical values
# Now we drop rows with NaN values for none numerical values & Weather_Condition
because we already replaced it with the manual encoding
df.dropna(inplace=True)
df.drop(["Weather_Condition", "Weather"], inplace=True, axis=1)
```

- Since all this cleaning and dropping may cause the appearance of some duplicate rows, we drop these to prevent problems with unsupervised and supervised models to be used next:

```
# Due to previous processing -> duplicates may appear (some dropped or fixed fields
may have been what differentiated rows)
df.drop_duplicates(inplace=True)
df.reset_index(inplace=True, drop=True)
```

- We finally update the description file and save it along with the resultant dataset:

```
with open("categoricalDataDescriptionAfter.csv", "w", encoding="utf-8") as f:
    f.write('Feature,Count Of Unique Values')
```

```
    for elm in obj_data:
        f.write('\n' + elm + ',' + str(len(df[elm].unique())) + ',')
df.describe().to_csv('descriptionAfter.csv')
df.to_csv("US_Accidents_AFTER.csv")
```

- Now that we have the data processed and cleansed, it's still a big data set and imbalanced, we will have to reduce it without ending up being bias towards a certain severity:

```
# Even after all this, our data size is still huge and inconsistent
# Data Sampling to reduce the size dealt with and balance data depending on target
"Severity"
s = len(df[df["Severity"] == 1].index)  # Size of the smallest Severity occurrence
dfSample = df[df["Severity"] == 1]
for sev in range(2, 5):  # Conditional Sampling + Constant Rate Sampling
    dfSample = pd.concat([dfSample, df[df["Severity"] == sev].sample(s * 2,
random_state=42)[::2]], ignore_index=True)
dfSample.to_csv('Sample.csv')
```

- The data values we ended up with is variant, so we normalize it using the MinMaxScaler from the sklearn library:

```
# Normalizing numerical data
dfNorm = dfSample.copy()
num_data.extend(['Year', 'Month', 'Day', 'Hour'])
# define min max scaler
scaler = MinMaxScaler()
toScale = dfSample[num_data]
# transform data
scaled = pd.DataFrame(scaler.fit_transform(toScale), columns=toScale.columns)
# update df
dfNorm.update(scaled)
```

- Then we encode the remaining categorical data, we start with the One Hot Encoding for non-binary categories, then we move to encoding binary categorical data mapping true and false values to 1 and 0. At the end we save the resultant Normalized and Encoded sample:

```
# Finally, we numerize the categorical data
# 'Sunrise_Sunset', 'Civil_Twilight', 'Nautical_Twilight', 'Astronomical_Twilight',
'Side', 'Wind_Direction'-> One Hot Encoding
dfNorm = pd.get_dummies(dfNorm, columns=['Side', 'Wind_Direction'],
drop_first=True)
dfNorm = pd.get_dummies(dfNorm, columns=['Sunrise_Sunset', 'Civil_Twilight',
                                         'Nautical_Twilight',
'Astronomical_Twilight'],
                        prefix=['Sun', 'Civ', 'Nau', 'Ast'], drop_first=True)
```

```
# True & False replaced by 1 & 0
dfNorm = dfNorm.replace([True, False], [1, 0])
# Zipcode binary encoded
binary_encoder = ce.binary.BinaryEncoder()
binZip = binary_encoder.fit_transform(dfNorm['Zipcode'])
dfNorm = pd.concat([dfNorm, binZip], axis=1).drop('Zipcode', axis=1)
dfNorm.to_csv('SampleNormalized&Encoded.csv')
```

PART II: Clustering

For this part, we chose to implement three clustering algorithms: K-Means, Spectral and DBSCAN using the sklearn library.

The steps are as follows: We perform dimensionality reduction on the whole dataset to visualize the clusters at the end, the clustering is performed and then for each algorithm results visualization takes place and three clustering metrics are calculated to assess the quality and efficiency (Silhouette, Calinski Harabasz and Davies Bouldin score).

- After loading the normalized dataset, only the columns with of type float were extracted along with the severity column.

```
# Load data ------------------------------------------------------------------
df = pd.read_csv('SampleNormalized&Encoded.csv', index_col=[0])
X1 = df.iloc[:, 0]
X1 = pd.concat([X1, df.select_dtypes(include=[float])], axis=1)
X = df[['Severity', 'Precipitation(in)', 'Distance(mi)', 'Visibility(mi)']]
```

- For dimensionality reduction, we used the Principal Component Analysis (PCA) algorithm, and reduced the set to 2 and 3 dimensions, with components stored in PCA2D & PCA3D respectively.

```
# Data Dimensionality Reduction for Visualization ----------------------------------
reduced2 = PCA(n_components=2, random_state=42).fit_transform(df)  # Dimensionality
reduction for data in 2D
reduced3 = PCA(n_components=3, random_state=42).fit_transform(df)  # Dimensionality
reduction for data in 3D
PCA2D = pd.DataFrame(reduced2, columns=['pca1', 'pca2'])
PCA3D = pd.DataFrame(reduced3, columns=['pca1', 'pca2', 'pca3'])
```

K-Means Clustering

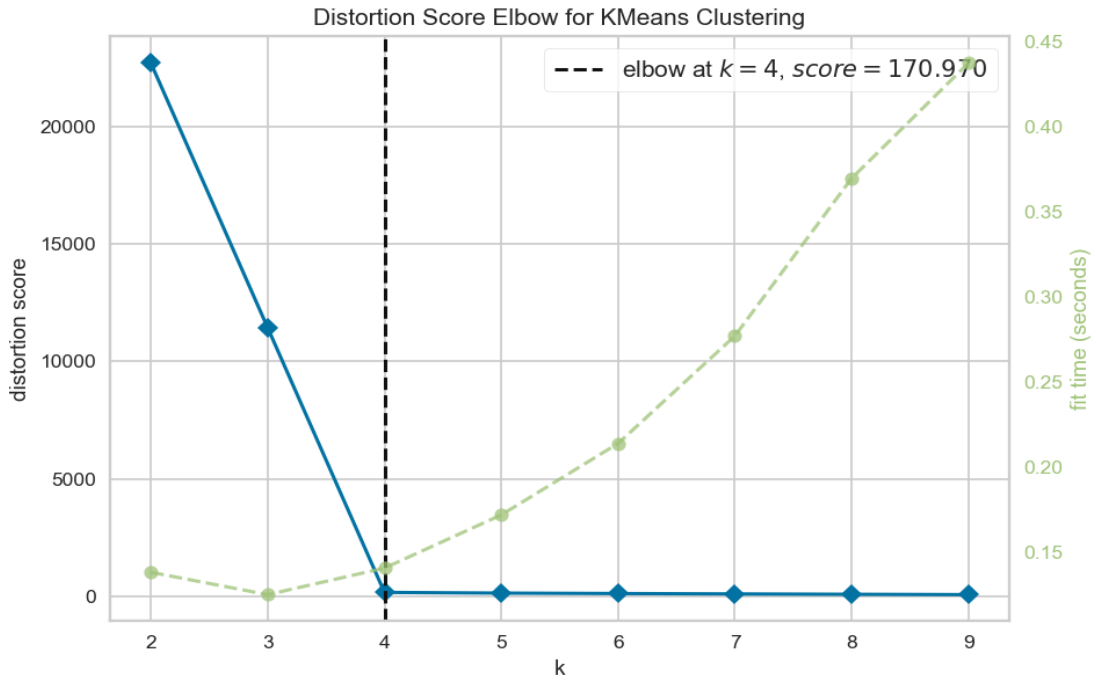
- First, we selected a few features out of the whole set with a predefined progressiveFeatureSelection method inspired by the research paper [1] and available through [2], the selected features are ['Severity', 'Precipitation(in)', 'Distance(mi)', 'Visibility(mi)'].

```
# Feature Selection ----------------------------------------------------------------
def progressiveFeatureSelection(DF, n_clusters=4, max_features=4,):
```

- The we draw the SSE vs number of clusters (k) plot to use the elbow method for selecting the optimal number of clusters then the inertia vs k plot is drawn to accentuate the k value. The elbow is decided at **K=4** after visualization and redrawn for demonstration.

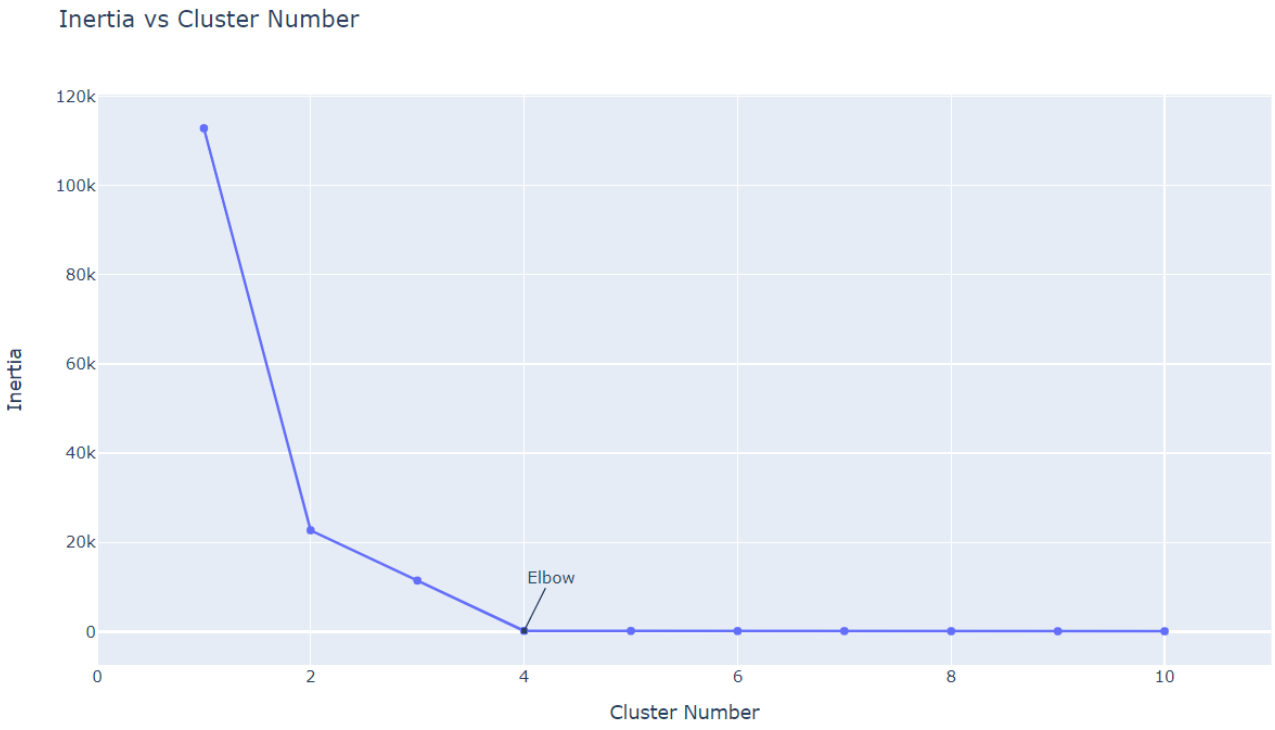
```
# SSE Plot / Elbow Method for finding optimal number of clusters ------------------
km = KMeans(random_state=42)
visualizer = KElbowVisualizer(km, k=(2, 10))
visualizer.fit(X)  # Fit the data to the visualizer
visualizer.show()  # Finalize and render the figure
```

Result 1:




```
# Manual Elbow Method for finding optimal number of clusters ----------------------
inertia = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init="k-means++", n_init=10, random_state=42)
    kmeans.fit(X)
    inertia.append(kmeans.inertia_)
fig = go.Figure(data=go.Scatter(x=np.arange(1, 11), y=inertia))
fig.update_layout(title="Inertia vs Cluster Number", xaxis=dict(range=[0, 11],
title="Cluster Number"),
                  yaxis={'title': 'Inertia'}, annotations=[
        dict(
            x=4,
            y=inertia[3],
            xref="x",
            yref="y",
            text="Elbow",
            showarrow=True,
            arrowhead=7,
            ax=20,
            ay=-40
        )
    ])
fig.write_html('K-MeansElbowPlot.html')
```

Result 2:



- An instance of the model K-Means Clustering is created with the number of clusters 4, and the predicted clusters are stored at the "clusters1" column. Afterwards, the results are plotted in 2D and 3D using the afore-created PCA2D/3D.

```
# Kmeans Clustering ---------------------------------------------------------------
kmeans = KMeans(n_clusters=4, init='k-means++', random_state=0)
df['clusters1'] = kmeans.fit_predict(X)
# 2D & 3D scatter plot
fig = px.scatter(PCA2D, x="pca1", y="pca2", color=df["clusters1"], title="K-means
Clustering | Clusters:4 | PCA: 2D")
fig.write_html('K-MeansClustering_2d.html')
fig = px.scatter_3d(PCA3D, x="pca1", y="pca2", z="pca3", color=df["clusters1"],
title="K-means Clustering | Clusters:4 | PCA: 3D")
fig.write_html('K-MeansClustering_3d.html')
```

- Next, the clustering metrics are calculated, and we start assuming that K-Means has the best and the worst scores (which will be updated if any of the upcoming algorithms is better or worse): Silhouette and Calinski Harabasz Scores the higher the better and Davies Bouldin Score the lower the better.

```
# Clustering Metrics ------------------------------------------------------------
score1 = silhouette_score(X, kmeans.labels_, metric='euclidean')
print('\nK-means Clustering Silhouette Score: %.3f' % score1)
score2 = calinski_harabasz_score(X, kmeans.labels_)
print('K-means Clustering Calinski Harabasz Score: %.3f' % score2)
score3 = davies_bouldin_score(X, kmeans.labels_)
print('K-means Clustering Davies Bouldin Score: %.3f\n' % score3)
# We assume that K-Means Clustering is the best ---------------------------------
Best = dict()
Best["S"] = ("K-Means", score1)
Best["C"] = ("K-Means", score2)
Best["D"] = ("K-Means", score3)
Worst = dict()
Worst["S"] = ("K-Means", score1)
Worst["C"] = ("K-Means", score2)
Worst["D"] = ("K-Means", score3)
```

Spectral Clustering

- An instance of the model Spectral Clustering is created with 4 clusters, and the predicted clusters are stored at the “clusters2” column. Afterwards, the results are plotted in 2D and 3D.

```
# Spectral Clustering -----------------------------------------------------------
spec = SpectralClustering(n_clusters=4, random_state=0,
affinity='nearest_neighbors')
df['clusters2'] = spec.fit_predict(X1)
# 2D & 3D scatter plot
fig = px.scatter(PCA2D, x="pca1", y="pca2", color=df["clusters2"], title="Spectral
Clustering | Clusters:4 | PCA: 2D")
fig.write_html('SpectralClustering_2d.html')
fig = px.scatter_3d(PCA3D, x="pca1", y="pca2", z="pca3", color=df["clusters2"],
title="Spectral Clustering | Clusters:4 | PCA: 3D")
fig.write_html('SpectralClustering_3d.html')
```

- Next, the clustering metrics are calculated, and we update the best and worst clustering metrics:

```
# Clustering Metrics ------------------------------------------------------------
score1 = silhouette_score(X, spec.labels_, metric='euclidean')
print('Spectral Clustering Silhouette Score: %.3f' % score1)
score2 = calinski_harabasz_score(X, spec.labels_)
print('Spectral Clustering Calinski Harabasz Score: %.3f' % score2)
score3 = davies_bouldin_score(X, spec.labels_)
print('Spectral Clustering Davies Bouldin Score: %.3f\n' % score3)

# Checking if Spectral Clustering is better -------------------------------------
if score1 > Best["S"][1]:
    Best["S"] = ("Spectral", score1)
if score2 > Best["C"][1]:
    Best["C"] = ("Spectral", score2)
if score2 < Best["D"][1]:
    Best["D"] = ("Spectral", score3)
# OR Worse ----------------------------------------------------------------------
if score1 < Worst["S"][1]:
```

```
    Worst["S"] = ("Spectral", score1)
if score2 < Worst["C"][1]:
    Worst["C"] = ("Spectral", score2)
if score2 > Worst["D"][1]:
    Worst["D"] = ("Spectral", score3)
```

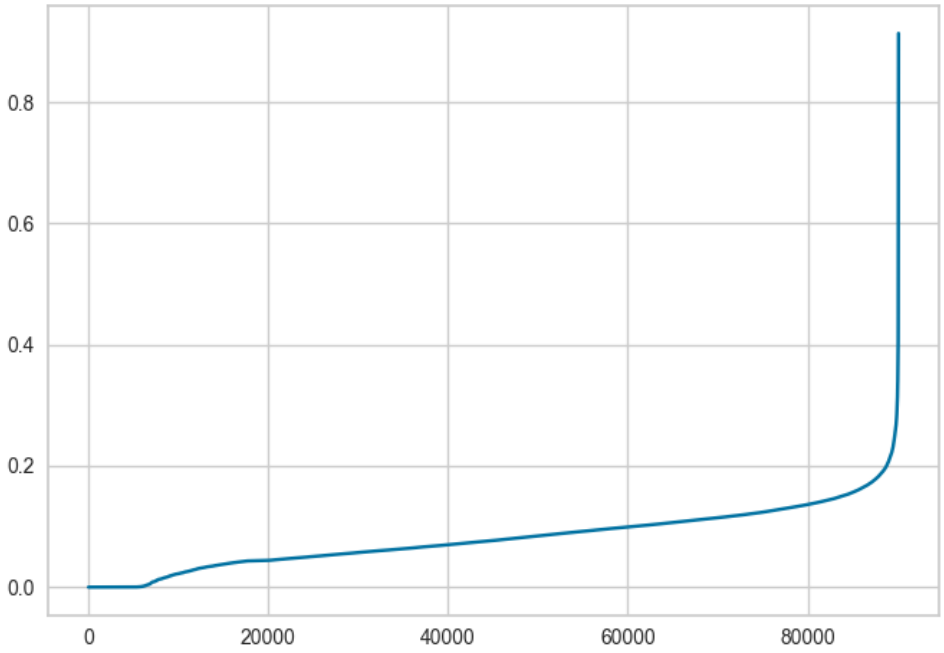
DBSCAN Clustering

- First, we choose the right Epsilon parameter (a fixed value for the maximum distance between two points to be considered of the same cluster) for the DBSCAN clustering with a predefined findOptimalEps method inspired by the research paper [3] and available through [2].

```
# Choosing the right epsilon parameter ----------------------------------------
def findOptimalEps(n_neighbors, data):
```

- The Eps is in the maximum curvature of the graph (the elbow) with form the results below is fixed at about eps=0,2.

Result:



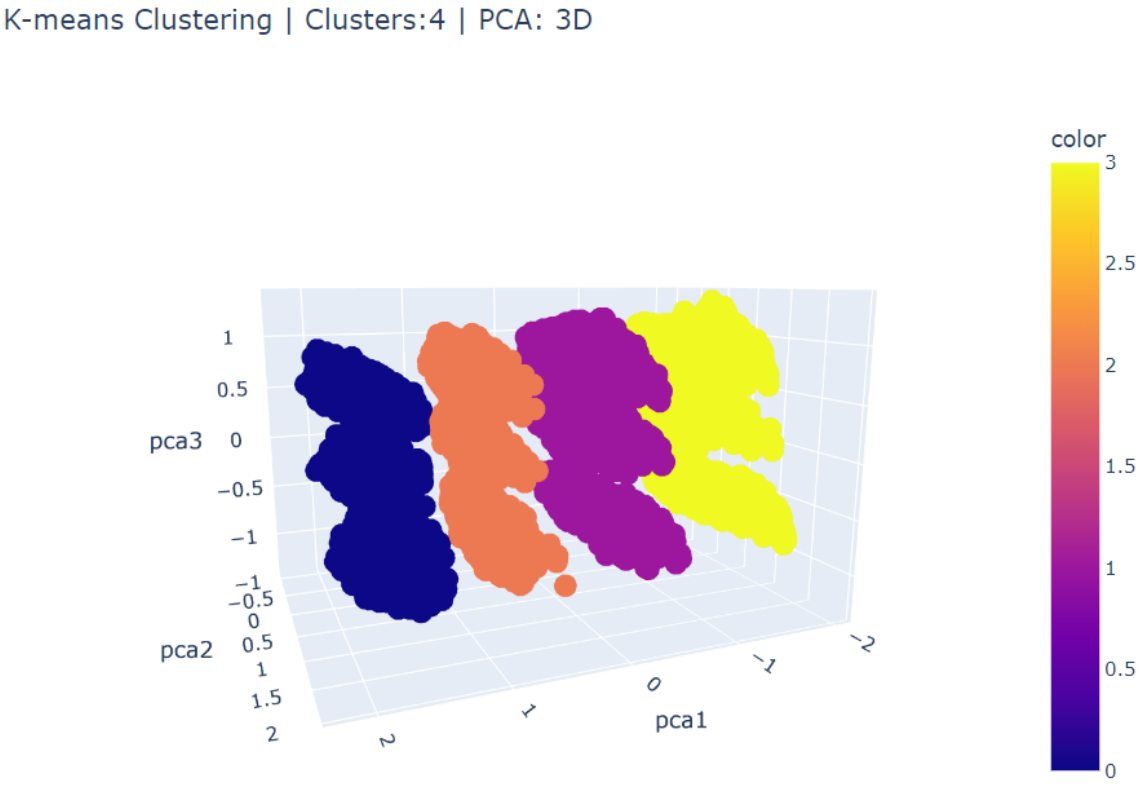
- Just as implemented previously, an instance of the model DBSCAN Clustering is created and the predicted clusters are stored at the “clusters3” column. But unlike the other algorithms, DBSCAN is sensitive to outliers (points with cluster label -1) and determines the number of clusters by itself. Next, the results are plotted in 2D and 3D.

```
# DBSCAN Clustering ------------------------------------------------------------
-----------------------------------
dbscan = DBSCAN(eps=0.2, metric="euclidean")
df['clusters3'] = dbscan.fit_predict(X)
# 2D & 3D scatter plot
fig = px.scatter(PCA2D, x="pca1", y="pca2", color=df["clusters3"], title="DBSCAN
Clustering | PCA: 2D")
fig.write_html('DBSCANClustering_2d.html')
fig = px.scatter_3d(PCA3D, x="pca1", y="pca2", z="pca3", color=df["clusters3"],
title="DBSCAN Clustering | PCA: 3D")
fig.write_html('DBSCANClustering_3d.html')
```

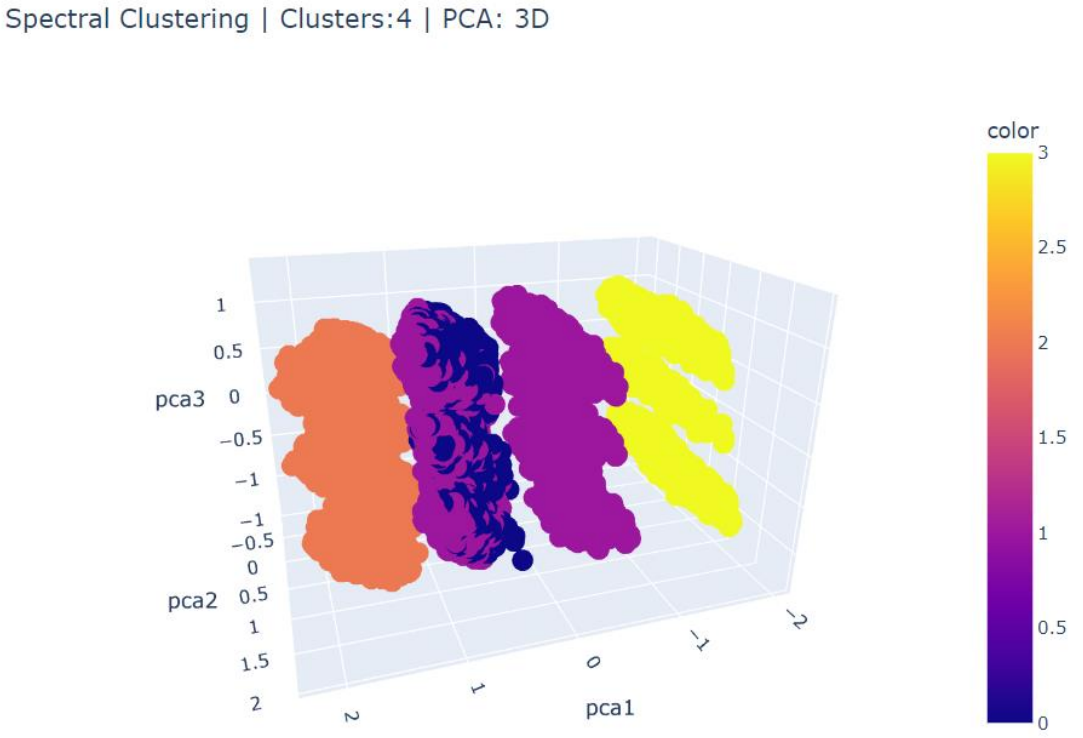
- Next, the clustering metrics are calculated, and we update the best and worst clustering metrics:

```
# Clustering Metrics -----------------------------------------------------------
score1 = silhouette_score(X, dbscan.labels_, metric='euclidean')
print('DBSCAN Clustering Silhouette Score: %.3f' % score1)
score2 = calinski_harabasz_score(X, dbscan.labels_)
print('DBSCAN Clustering Calinski Harabasz Score: %.3f' % score2)
score3 = davies_bouldin_score(X, dbscan.labels_)
print('DBSCAN Clustering Davies Bouldin Score: %.3f\n' % score3)

# Checking if Spectral Clustering is better ------------------------------------
if score1 > Best["S"][1]:
    Best["S"] = ("DBSCAN", score1)
if score2 > Best["C"][1]:
    Best["C"] = ("DBSCAN", score2)
if score2 < Best["D"][1]:
    Best["D"] = ("DBSCAN", score3)
# OR Worse ---------------------------------------------------------------------
if score1 < Worst["S"][1]:
    Worst["S"] = ("DBSCAN", score1)
if score2 < Worst["C"][1]:
    Worst["C"] = ("DBSCAN", score2)
if score2 > Worst["D"][1]:
    Worst["D"] = ("DBSCAN", score3)
```

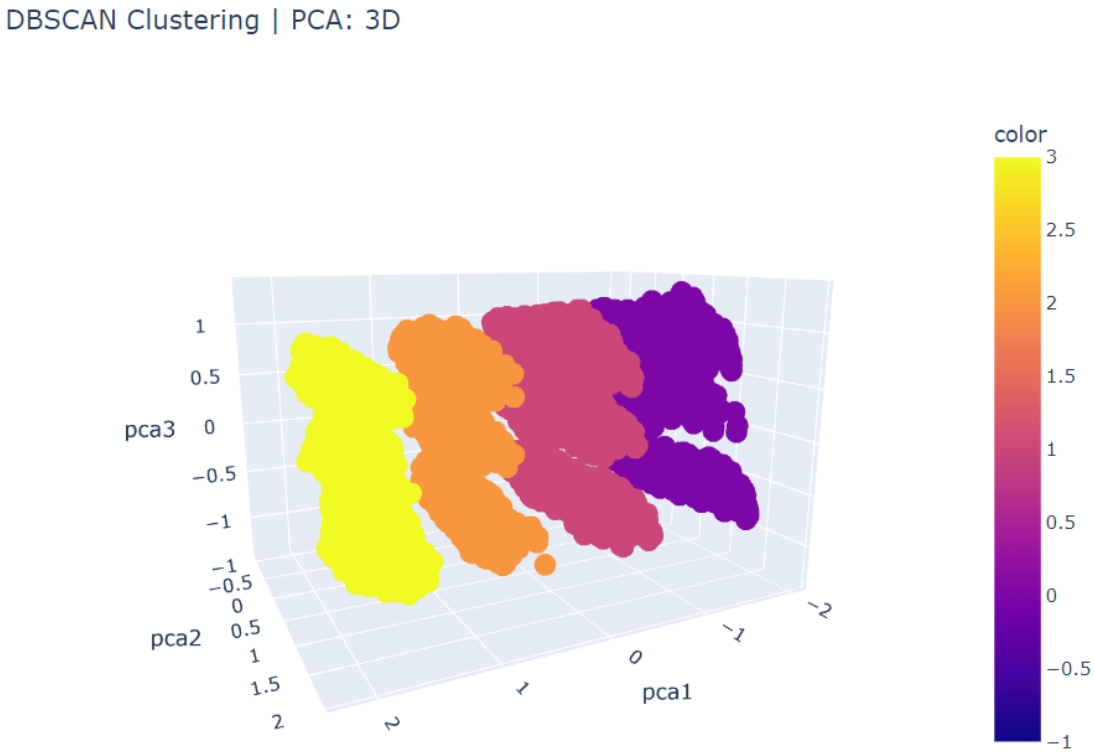
Scatter plots of K-Means Clustering:



Scatter plots of Spectral Clustering:



Scatter plots of BSCAN clustering:

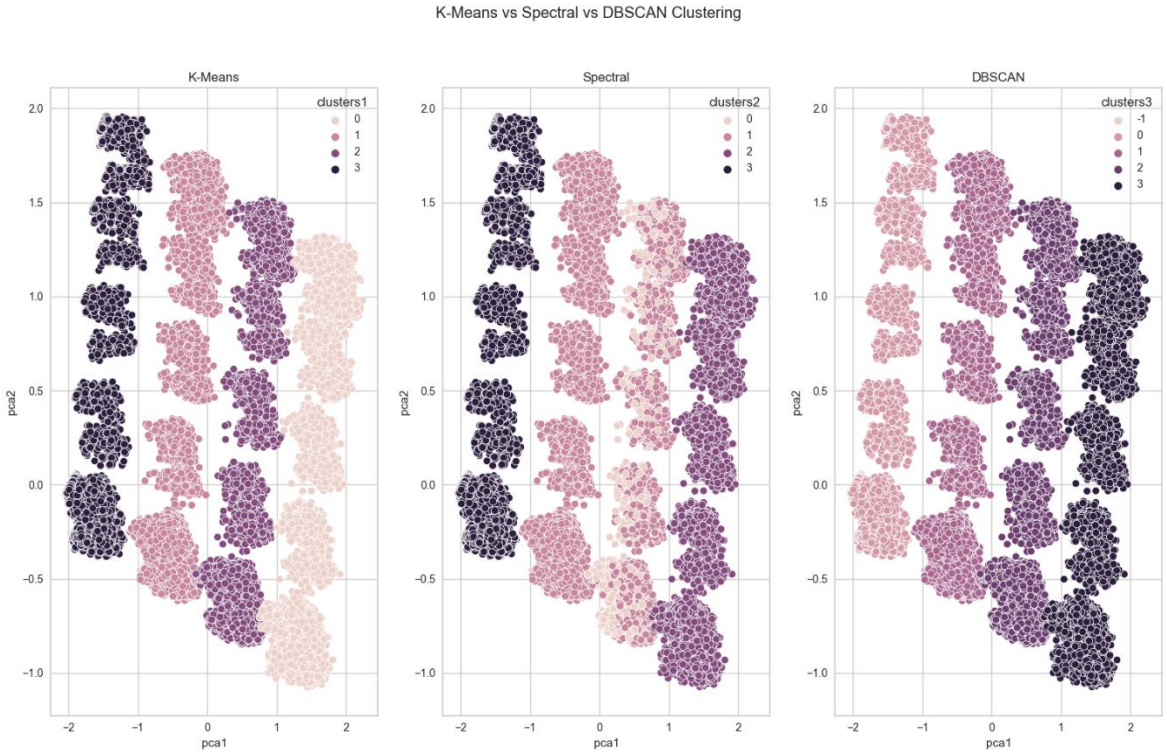


Comparing the Different Clustering Methods

- At the end we draw the comparison scatter plot that contains all the three clusters:

```
# Comparison scatter plot ---------------------------------------------------------
fig1, (ax1, ax2, ax3) = plt.subplots(1, 3)
fig1.set_size_inches(18.5, 10.5)
fig1.suptitle('K-Means vs Spectral vs DBSCAN Clustering')
ax1.set_title('K-Means')
se.scatterplot(x="pca1", y="pca2", hue=df['clusters1'], data=PCA2D, ax=ax1)
ax2.set_title('Spectral')
se.scatterplot(x="pca1", y="pca2", hue=df['clusters2'], data=PCA2D, ax=ax2)
ax3.set_title('DBSCAN')
se.scatterplot(x="pca1", y="pca2", hue=df['clusters3'], data=PCA2D, ax=ax3)
plt.savefig('ClusteringComparisonFigure.png')
```

Result:



- Lastly, we rank the clustering algorithms used depending on the clustering metrics previously calculated:

```
# Best & Worst ---------------------------------------------------------------------
print("Best for Silhouette Score, Calinski Harabasz Score & Davies Bouldin Score
respectively\n", Best)
print("Worst for Silhouette Score, Calinski Harabasz Score & Davies Bouldin Score
respectively\n", Worst)
```

```
Cl = ["K-Means", "Spectral", "DBSCAN"]
print("\nRaking the Clustering methods used from Best to Worst:")
BestList = list(Best.values())
out = [item for t in BestList for item in t]
new = [item for item in out if type(item) == str]
best = mode(new)
print("1. ", best, "Clustering.")
WorstList = list(Worst.values())
out = [item for t in WorstList for item in t]
new = [item for item in out if type(item) == str]
worst = mode(new)
average = "".join([x for x in Cl if x != best and x != worst])
print("2. ", average, "Clustering.")
print("3. ", worst, "Clustering.")
```

Results:

```
C:\Users\naits\Documents\UnivPi\USAccident\Scripts\python.exe C:\Users\naits\Documents\UnivPi\DA\USAccident\Clustering.py

K-means Clustering Silhouette Score: 0.969
K-means Clustering Calinski Harabasz Score: 19790783.735
K-means Clustering Davies Bouldin Score: 0.048

Spectral Clustering Silhouette Score: 0.589
Spectral Clustering Calinski Harabasz Score: 361952.721
Spectral Clustering Davies Bouldin Score: 0.563

DBSCAN Clustering Silhouette Score: 0.969
DBSCAN Clustering Calinski Harabasz Score: 14383734.875
DBSCAN Clustering Davies Bouldin Score: 1.057

Best for Silhouette Score, Calinski Harabasz Score & Davies Bouldin Score respectively
 {'S': ('DBSCAN', 0.9694960370182965), 'C': ('K-Means', 19790783.735476814), 'D': ('K-Means', 0.04750005745767963)}
Worst for Silhouette Score, Calinski Harabasz Score & Davies Bouldin Score respectively
 {'S': ('Spectral', 0.5890580746207702), 'C': ('Spectral', 361952.72061301395), 'D': ('DBSCAN', 1.0566476511956768)}

Raking the Clustering methods used from Best to Worst:
1.  K-Means Clustering.
2.  DBSCAN Clustering.
3.  Spectral Clustering.

Process finished with exit code 0
```

Conclusion

It was obvious from the graphical representations of the clustering methods that the spectral clustering wasn't as accurate as the K-Means and the DBSCAN clustering, and that was proved by the results of the clustering metrics, where it ranked last as the least well-performing method for our dataset. On the other hand, K-Means clustering recorded the best performance and it was followed by the DBSCAN in the 2nd place.

Part III: Regression/ Classification/ Storytelling

There is no model in the Machine Learning Models that works with 100% efficiency with all the datasets, in addition to the Lack of Quality Data, and the knowledge of which model we should implement in a particular problem set. The evaluation and selection model is a step that shows the difference between a model that performs well and a model that performs fine, who predicts well, and who doesn't.

That is why several models must be used to find the best model that fits our problem and solve it as efficiently as possible. Each algorithm or model has its best fit in a particular place or problem set.

For this reason, we'll build a classification of several models to solve our problem set, we'll use various types of classification models to see who's the best to fit our problem, and we'll be using the following classification models in our implementation: Neural network, Random Forest, Logistic Regression, Decision Tree and Naive Baye.

We will use Scikit-Learn and TensorFlow for this task, Scikit-Learn and TensorFlow are libraries that share machine learning implementation.

TensorFlow is a fast numerical computing low-level library for creating Deep Learning models, and constructing large-scale neural networks with many layers, such as Classification, Perception, Understanding, Discovering, and Prediction.

Whilst Scikit-Learn is a high-level library that was made to use an easy-to-use interface for developers, it's used for general Machine Learning Algorithms for both supervised and unsupervised learning such as SVMs, Random Forests, Logistic Regression, and many, many more, it can represent a standard object in a single line or a few lines of code, then use it to fit a set of points or predict a value, it's utilized in training with a wider spectrum of models.

- The first step is about loading the normalized and encoded dataset.

```
df = pd.read_csv('SampleNormalized&Encoded.csv')#load Data
features = df.drop("Unnamed: 0",axis=1)#drop the indicies Column
features.head()
```

- In order to tune and assess the data, we need to split data into a validation set, a training set, and a testing set. The training set will get the majority 80%, the rest will be split between training and validation sets 50% for each (10% of the complete set).

```
       X = features.drop("Severity", axis=1)
Labels = features["Severity"]

train_x,test_x,train_y,test_y = train_test_split(X, Labels,shuffle=True, test_size =
                                                 0.2, random_state=2)
test_x,valid_x,test_y,valid_y = train_test_split(test_x,test_y, test_size =
                                                 0.5,random_state=2)
```

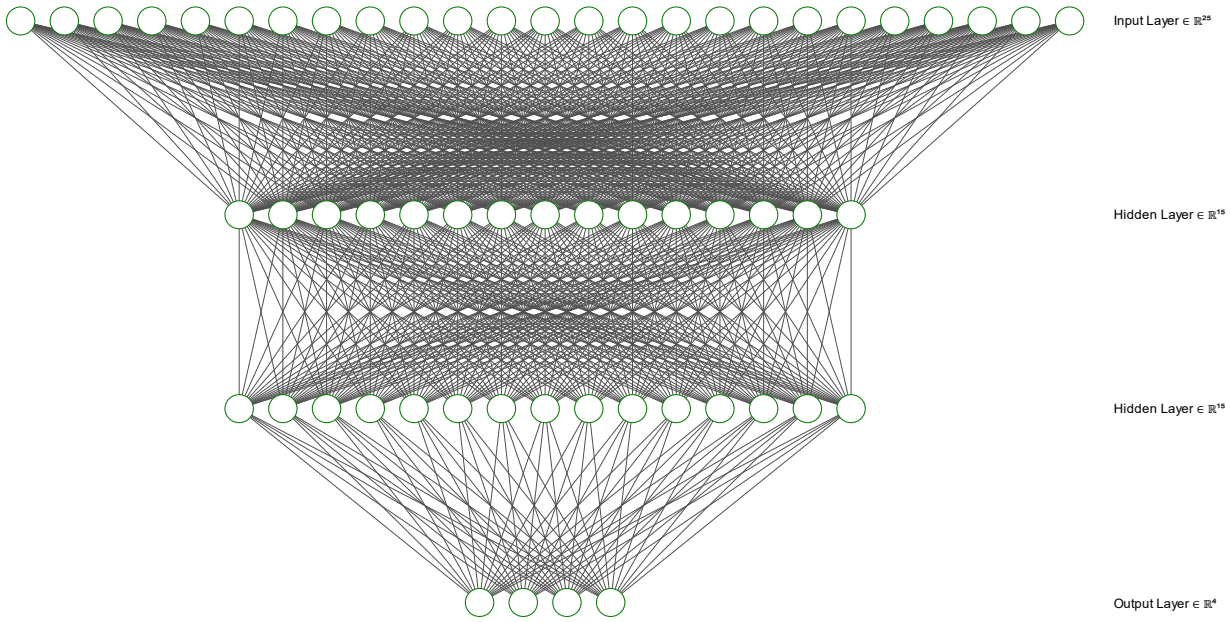
- As mentioned already, we are trying to find the best Model and the best hyperParameter for that model. in order to achieve that we need some variables to store some metrics (F1_score, accuracy), which will be calculated later.

```
accuracy_d = dict()
precision_d = dict()
recall_d = dict()
f1_d = dict()
```

Neural Networks

Is defined as a black box that takes one or multiple inputs and processes them into one or multiple outputs, it consists of many small units called “neurons” grouped into several layers, units of one layer interact with the Neurons of the next layer through "weighted connections" which are connections with a real-valued number attached to them. A neuron takes the value of a connected neuron and multiplies it with its connections weight. The sum of all connected neurons and the neuron's bias value is the “activation

function", which transforms the value before can be passed on to the next neuron. The idea is to find the right weights to get the right results.



- f1 score, recall and precision are not part of keras metrics anymore, so we are going to calculate them manually.

```
def recall(y_true, y_pred):
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    possible_positives = K.sum(K.round(K.clip(y_true, 0, 1)))
    recall = true_positives / (possible_positives + K.epsilon())
    return recall

def precision(y_true, y_pred):
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    predicted_positives = K.sum(K.round(K.clip(y_pred, 0, 1)))
    precision = true_positives / (predicted_positives + K.epsilon())
    return precision

def f1(y_true, y_pred):
    prec = precision(y_true, y_pred)
    rec = recall(y_true, y_pred)
    return 2*((prec*rec)/(prec+rec+K.epsilon()))
```

- our next step will be specifying the architecture of the neural network.

```
model = Sequential()
model.add(Dense(50, activation = 'relu', input_dim = len(features.columns) - 1))
model.add(Dense(15 , activation = 'relu'))
model.add(Dense(15 , activation = 'relu'))
model.add(Dense(4, activation = 'softmax'))
```

- Compile the model with the best optimizer, we tried a lot of them and we choose the best one for our case.

```
model.compile(loss='categorical_crossentropy',optimizer='nadam',
              metrics=['accuracy',f1,precision, recall])
```

- Next, we fit the model to the training set.

```
model.fit(X_train_nn, y_train_nn, epochs = 30,batch_size = 64,
          validation_data = (X_test_nn, y_test_nn))
```

✓ After the train finished, we found that the accuracy of this model is about 72%.

Logistic Regression

Classic statistical model for classification to predict a binary output based on prior observations of a data set, it suppresses the output of the linear model between 0 and 1 so the output is a probability value of whether the inputs belong to class 0 or class 1.

- Select the best HyperParameter using randomizedSearch CV, we initial different HyperParameter and let the model find the best parameter for this Dataset.

```
model = LogisticRegression()
# define search space
space = dict()
space['solver'] = ['newton-cg', 'lbfgs', 'liblinear']
space['penalty'] = [ 'l2']

...
# define evaluation
```

```
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

randm_src = RandomizedSearchCV(model, space, n_iter=3, scoring='accuracy'
, n_jobs=-1, cv=cv, random_state=1)
# execute search
result = randm_src.fit(train_x, train_y)
# summarize result

print('Best Hyperparameters: %s' % result.best_params_)
```

Output:

```
Best Hyperparameters: {'solver': 'newton-cg', 'penalty': 'l2'}
```

- We use this HyperParameter to fit our model

```
logReg = LogisticRegression(solver = "sag",penalty="l2", random_state=2)
logReg.fit(train_x,train_y)
print("accuracy with best HyperParameters")
score = logReg.score(train_x, train_y)
print("accuracy = {:.2f}%".format(score*100,".2f"))
val_score = logReg.score(valid_x, valid_y)
print("val_accuracy = {:.2f}%".format(val_score*100,".2f"))
```

Output:

```
accuracy with best HyperParamaters
accuracy = 62.90%
val_accuracy = 63.24%
```

- Save the results in the dictionary we initialized previously; we will use it later to compare between different Models.

```
pred_y = logReg.predict(test_x)
print(classification_report(test_y, pred_y))
```

Random Forest

Supervised Machine Learning Algorithm, it builds decision trees at training time on different samples and takes their majority vote for classification and average in case of regression. It can handle the data set containing continuous variables as in the case of regression and categorical variables as in the case of classification. It performs better results for classification problems.

We will do the same steps as Logistic Regression we start by finding the best hyperparameters and then we fit the data into the model with the best hyperparameters we found and the we save the results in dictionaries.

- Just this Time instead of randomizedSearchCV we used GridSearchCV trying to finding

```
ranForest = RandomForestClassifier(random_state=2)
hyperPar = [{"criterion": ["gini", "entropy"],
             "n_estimators": [10,50,100 ],"max_depth": [2,5,10],
             }]
grid = GridSearchCV(ranForest, hyperPar, n_jobs=-1, verbose=1)
result = grid.fit(train_x, train_y)
print('Best Hyperparameters: %s' % result.best_params_)
```

Output:

```
Best Hyperparameters: {'criterion': 'gini', 'max_depth': 10, 'n_estimators': 100}
```

Accuracy Output:

```
accuracy with best HyperParamaters
accuracy = 74.37%
val_accuracy = 72.38%
```

Decision Tree

Supervised statistical, Machine Learning, and Data Mining Algorithm, it builds classification or regression models and organizes a series of roots in a Tree Structure, where the data is continuously split according to a certain parameter. It splits a dataset into smaller subsets, while at the same time an associated decision tree is incrementally developed. The final outcome is a tree with decision nodes (make any decision and have multiple branches) and leaf nodes (the output of those decisions and do not contain any further branches).

Exactly the same steps as RandomForest and Logistic Regression so we'll suffice by putting final results after training.

Best HyperParameter:

```
Best Hyperparameters: {'criterion': 'entropy', 'max_depth': 10, 'max_features': 'auto', 'splitter': 'best'}
```

Accuracy score:

```
accuracy with best HyperParamaters
accuracy = 65.14%
val_accuracy = 63.24%
```

Naïve Bayes:

Supervised probabilistic classification. It's based on applying Bayes' theorem with the "naive" hypothesis of conditional independence between every pair of features given the value of the class variable, for example, finding the probability of A happening, given that B has occurred.

Best HyperParameter:

```
Best Hyperparameters: {'fit_prior': False}
```

Accuracy score:

```
accuracy with best HyperParamaters
accuracy = 48.91%
val_accuracy = 48.62%
```

Model Selection and Evaluation:

Model evaluation: is a method of evaluating the correctness of models on data set for testing. The test data consists of data sets that have not been seen by the model before.

Model selection: is a method for selecting the most suitable or best model after the individual models are evaluated based on the required measures.

When we know which challenge we face, we'll be better equipped to tweak our hyperparameters and make better predictions. In the end, we will display the performance of each model by manipulating the Model of Selection and Evaluation using various metrics:

Confusion Matrix:

A set of predictions on a dataset displays the number of trial points accurately and inaccurately classified.

- We used plotly to plot the confusion Matrix for all the models.

```
y = ["Actual Severity 1", "Actual Severity 2",
     "Actual Severity 3", "Actual Severity 4"]
x =  ["Predicted Severity 1", "Predicted Severity 2",
      "Predicted Severity 3", "Predicted Severity 4"]

# change each element of z to type string for annotations
z_text = [[str(y) for y in x] for x in cf_matrix]

# set up figure
fig = ff.create_annotated_heatmap(cf_matrix, x=x, y=y, annotation_text=z_text,
                                  colorscale='Viridis')

# add title
fig.update_layout(title_text='<i><b>Confusion matrix</b></i>',
                  #xaxis = dict(title='x'),
                  #yaxis = dict(title='x')
                 )

# add custom xaxis title
fig.add_annotation(dict(font=dict(color="black",size=14),
                        x=0.5,
                        y=-0.5,
                        showarrow=False,
                        text="Predicted value",
                        xref="paper",
                        yref="paper"))

# add custom yaxis title
fig.add_annotation(dict(font=dict(color="black",size=14),
                        x=-0.5,
                        y=0.5,
                        showarrow=False,
                        text="Real value",
                        #textangle=-90,
                        xref="paper",
                        yref="paper"))
```

```
# adjust margins to make room for yaxis title
fig.update_layout(yaxis = dict(categoryorder = 'category descending'))

# add colorbar
fig['data'][0]['showscale'] = True
fig.show()
fig.write_html("Confusion_Matrix_Random_Forest.html")
```

Confusion matrices for all models:

Confusion Matrix Random Forest

Confusion Matrix Logistic Regression

Confusion Matrix Decision Tree

Confusion Matrix Naïve Bayes

Accuracy, F1-score, Recall, Precision:

Basic predictive implementation measures.

- Accuracy Is the fraction of samples that were classified correctly, divided by the total number of test cases.
- Precision identifies the correctness of classification.
- Recall number of positive cases is correctly classified out of the total number of positive cases.
- F1-score the harmonic mean of precision and recall

The code to calculate F1-score, Recall and Precision is mentioned above.

AUC-ROC Curves:

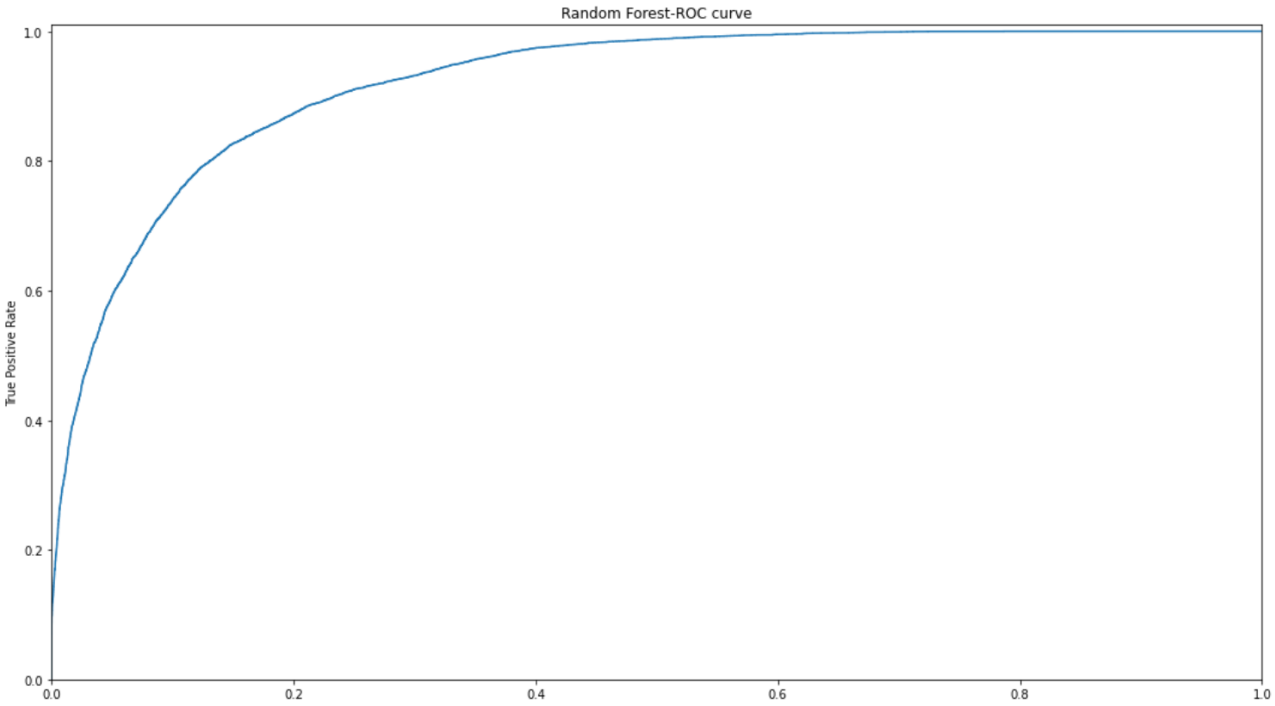
a performance metric for the classification problems at different point sets. It shows the capability of a model in distinguishing between classes.

- ROC (Receiver Operating Characteristics): Is a probability curve. It's a plot of True Positive rate on the y-axis vs the False Positive rate on the x-axis below the code to draw Roc.

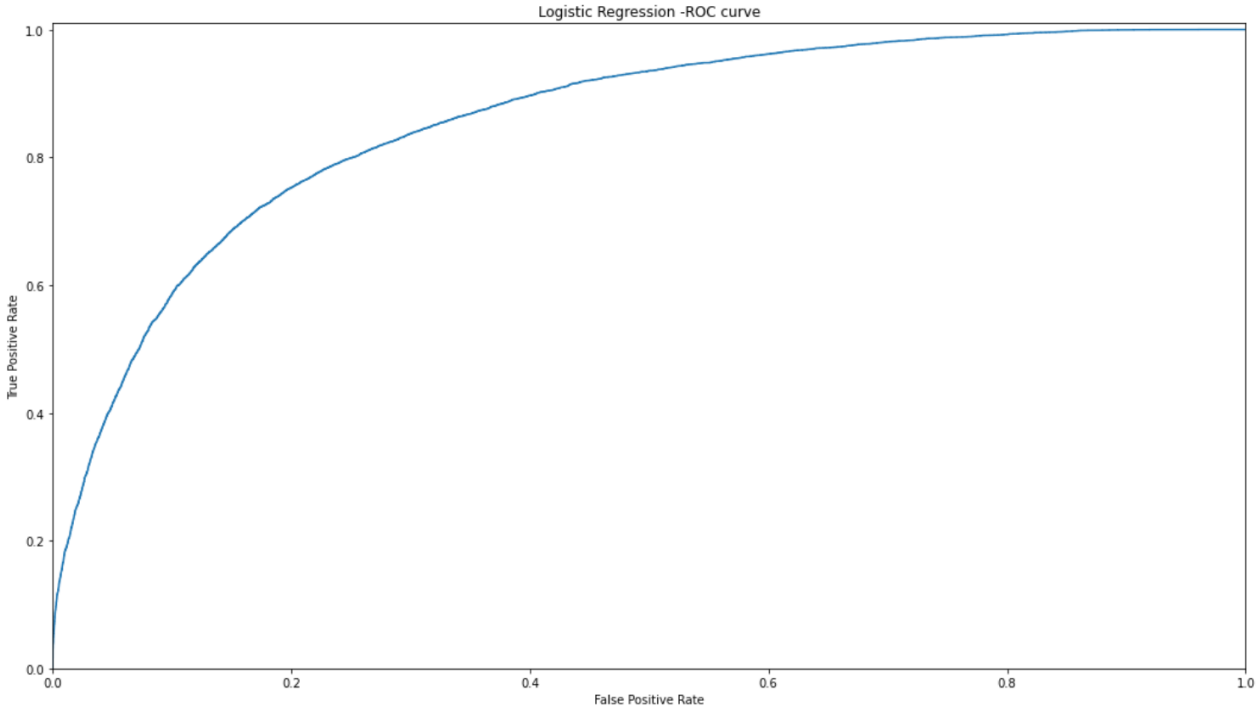
```
plt.figure(figsize=(18, 10))
plt.step(False_pos_Ratio["Random Forest"], True_pos_Ratio["Random Forest"],
where="post")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Random Forest-ROC curve")
plt.xlim([0, 1])
plt.ylim([0, 1.01])
plt.show()
```

- AUC (Area Under the receiving operating Curve): Is the degree or measure of separability. A standard performance measure of classification.

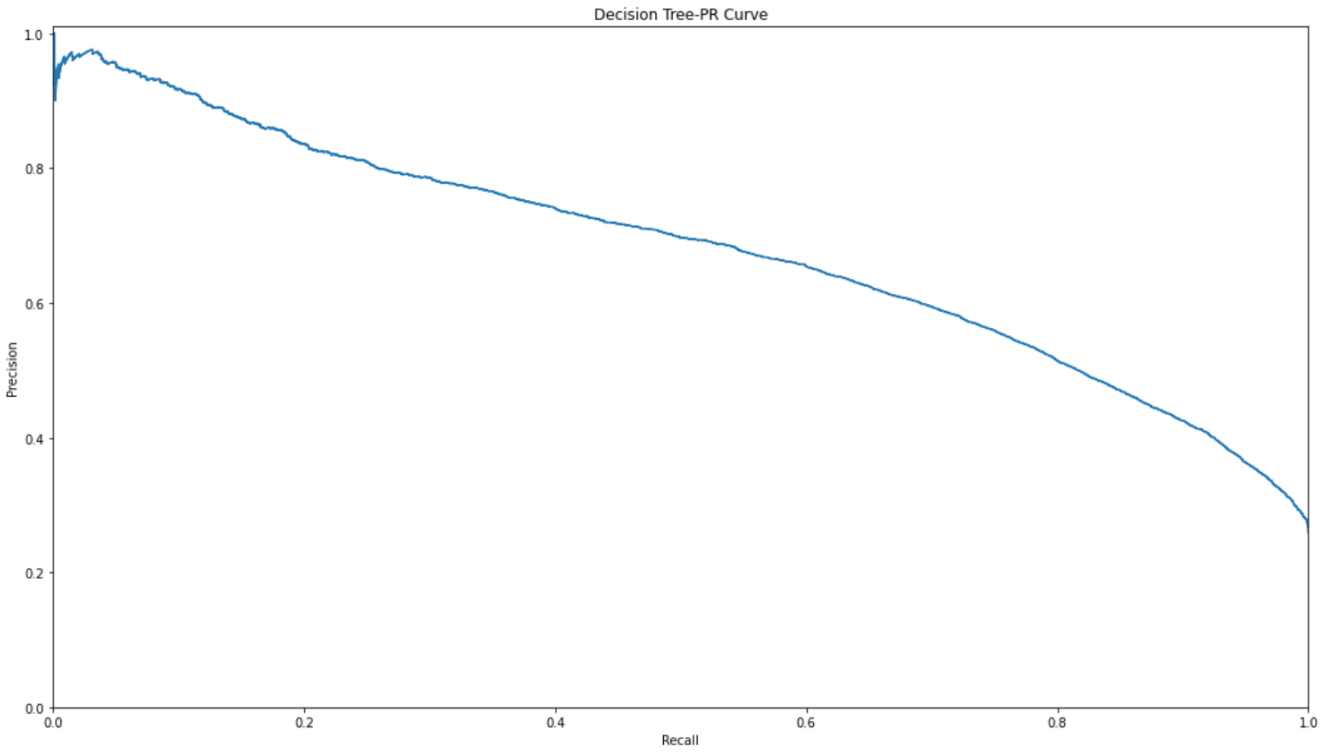
ROC and PR curves of the different models



Random Forest-ROC curve



Logistic Regression-ROC curve



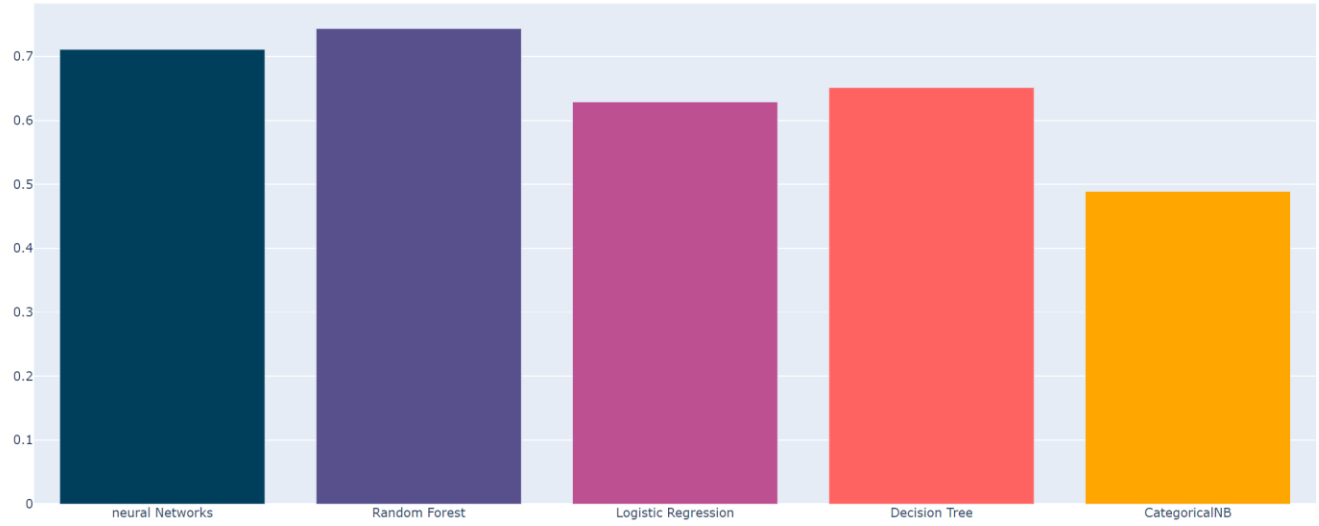
Decision Tree-PR curve

Model Selection:

- First, we calculate accuracy on the training set for all models to compare and find the one with better accuracy, to do that we used plotly to create a fig and see the results.

```
colors = ['#003f5c','#58508d','#bc5090','#ff6361','#ffa600']
fig = go.Figure()
fig.add_trace(go.Bar(x=list(accuracy_d.keys()),
y=list(accuracy_d.values()),marker_color=colors))
fig.show()
```

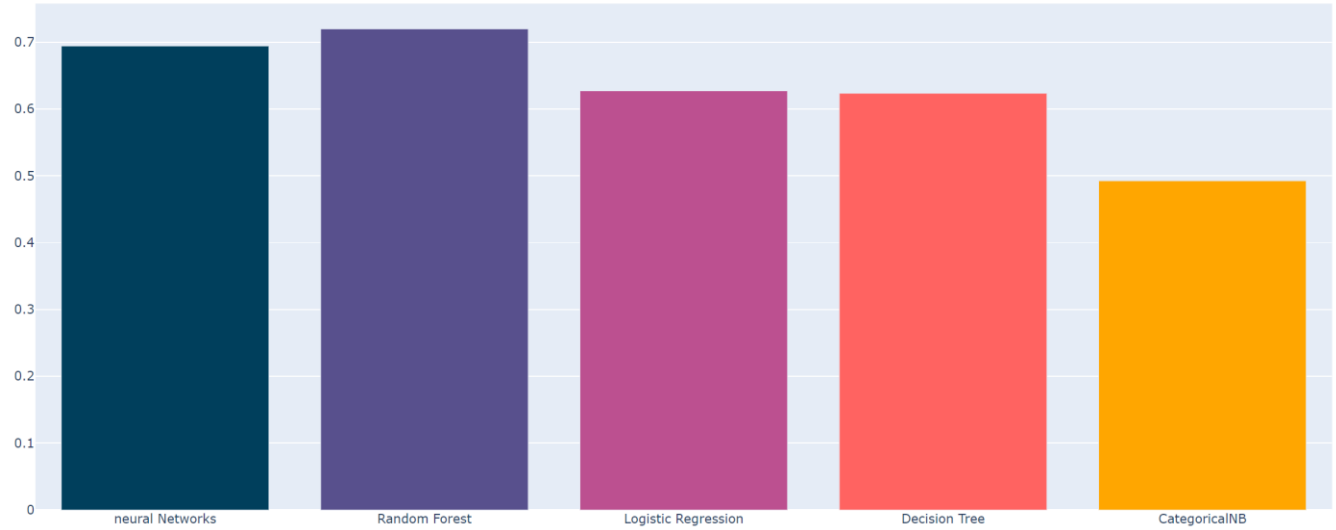
Output:



- Now, we calculate the F1_score for all on training set.

```
colors = ['#003f5c','#58508d','#bc5090','#ff6361','#ffa600']
fig = go.Figure()
fig.add_trace(go.Bar(x=list(f1_d.keys()), y=list(f1_d.values()),marker_color=colors))
fig.show()
```

Output:



Conclusion:

According to the evaluation of these models, we have found out that Random Forest with its parameters (**criterion = 'gini', max_dept' = 10, n_estimators = 100**) is the most suitable classification model for our problem.

References

[1] Feature Selection for Clustering. Through: https://perma.cc/3HQR-RL27

[2] Clustering with Scikit-Learn in Python. Through: https://programminghistorian.org/en/lessons/clustering-with-scikit-learn-in-python

[3] Determination of Optimal Epsilon (Eps) Value on DBSCAN Algorithm to Clustering Data. Through: https://iopscience.iop.org/article/10.1088/1755-1315/31/1/012012/pdf

[4] Heatmaps in Python. Through: https://plotly.com/python/heatmaps/

[5] Model Selection. Through: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.model_selection