A data-driven analysis for car accident characteristics in U.S. using Apache Spark

Gabriele Favia

Big Data 2019/2020 – Politecnico di Bari

Contents

The context

The next frontier of mobility driven by information exchange

In a world more and more interconnected, the sector "transportation" is uninterruptedly fusing with computer technology and services, indeed, the car industry is now advertising their new products by pressing the audience attention towards connectivity features [1][2], which can be divided in three big macro-areas of usage:

- Entertainment and road indications: music streaming, instant messaging software.
- Real-time positioning: access and download of updated maps with traffic alerts, current parking position.
- Car conditions checking: integration of specifically designed smartphone apps to control functionalities such as pre-heating/cooling, keyless door opening/closing, scheduling the service/ maintenance and range status.

An extended way to get advantage of these "online" functionality is the implementation of acquired remote data with the driving mechatronics systems of the car: network protocols such as V2x – Vehicle to Everything (Infrastructure, Grid, Vehicle) [3] - are one of the keys on which the Fully Automated Autonomous Vehicle investors are heavily relying.

For example, on January 2020, Seat presented the new Leon, a segment C car equipped with a Predictive Adaptive Cruise Control [4] which mixes sensors' data with GPS information provided by Here [5] in order to, for instance, slow down before taking a curve automatically, recreating this way the road layout in a more precise manner, while Toyota announced on February 2020, the use of Big Data on their cars to make it possible to disable the accelerator when the sensors find discrepancies between the type of the road (with all its characteristics and historical data related, such as average speed and traffic conditions) and the driving style [6]. This feature called "accelerator suppression function", wants to avoid accidents caused by confusion between brake and accelerators that often occurs when driving an automatic-geared car (especially in aged people); in the same month, Hyundai Motor Group released to the press the ICT Connected Shift System [7] which uses Big Data taken from the users' driving habits, traffic conditions and road events, mixing them with local input sensors (lidar, radar to reconstruct the 3D scene) in order to make the gear switching automatic using a "smart" approach.

Car accidents are reportedly indicated by the statistics as one of the primary causes of death in the world, in fact, according to the Centers for Disease Control and Prevention [8], the worldwide number of fatalities on the road have been estimated as 1.35 million every year, in particular

- Every day, almost 3700 people are killed in road traffic crashes involving cars, buses, motorcycles, bicycles, trucks, or pedestrians.
  More than half of those killed are pedestrians, motorcyclists, and cyclists.
- Road traffic injuries are estimated to be the eighth leading cause of death globally for all age groups and the leading cause of death for children and young people 5–29 years of age.
  More people now die in road traffic crashes than from HIV/AIDS [9].

Looking at the United States, the situation is not very different since over 37000 people die in road crashes each year and an additional 2.35 million are injured or disabled [10].

Plus, regarding the economic side, road crashes cost the U.S. $ 230.6 billion per year or an average of $ 820 per person.

Assuming a SIM integrated inside the vehicle, it would be possible for the car piloting systems to set precise dynamics conditions (hardness of the suspensions, speed, reactiveness of the steering wheels…) by simply downloading information about the route it is taking in that moment (in real time or got along with the destination indication, when setting the navigator).

The prospective of automatic cars able to slow down along the most threatful road traits or at least sending a real time warning to the driver, inspired this work to design a model useful to predict the danger level of the current area in which the vehicle is transiting, based on the severity of the accidents.

Factors of risk in car crashes

The principal causes of accidents can be categorized as internal factors driven or external factors driven: while the internal are the ones attributable both to demographics [11] and to the psychophysical conditions of the drivers, the externals are the entities that influence the driving behaviour as a reaction to the environmental status.

Some of the external factors are:

- Slick roads.
- Glare.
- View obstructions.
- Other highway-related conditions.
- Fog, rain and/or snow.
- Other weather conditions.
- Signs and/or signals.
- Road design.

The expected output

The objective of this project is the result of the following steps:

1) Analysing the dataset to get an idea of the external condition influencing the accidents, using this information to pre-process data for the model implementation.
2) Using the MLlib library provided by Apache Spark to develop a data-driven model that takes into account external factors found in the first part of the project as parameters and predicting the road danger assuming direct correlation with damage rate ("Severity").

Performance comparison and evaluation between different Machine Learning models (speed, accuracy) trained on the same dataset, will complete the study.

The Dataset

The dataset to exploit comes from Kaggle [12] under CC BY-NC-SA 4.0 license, composed of 49 features and over 3 million of entries covering the period February 2016 to December 2019, in US.

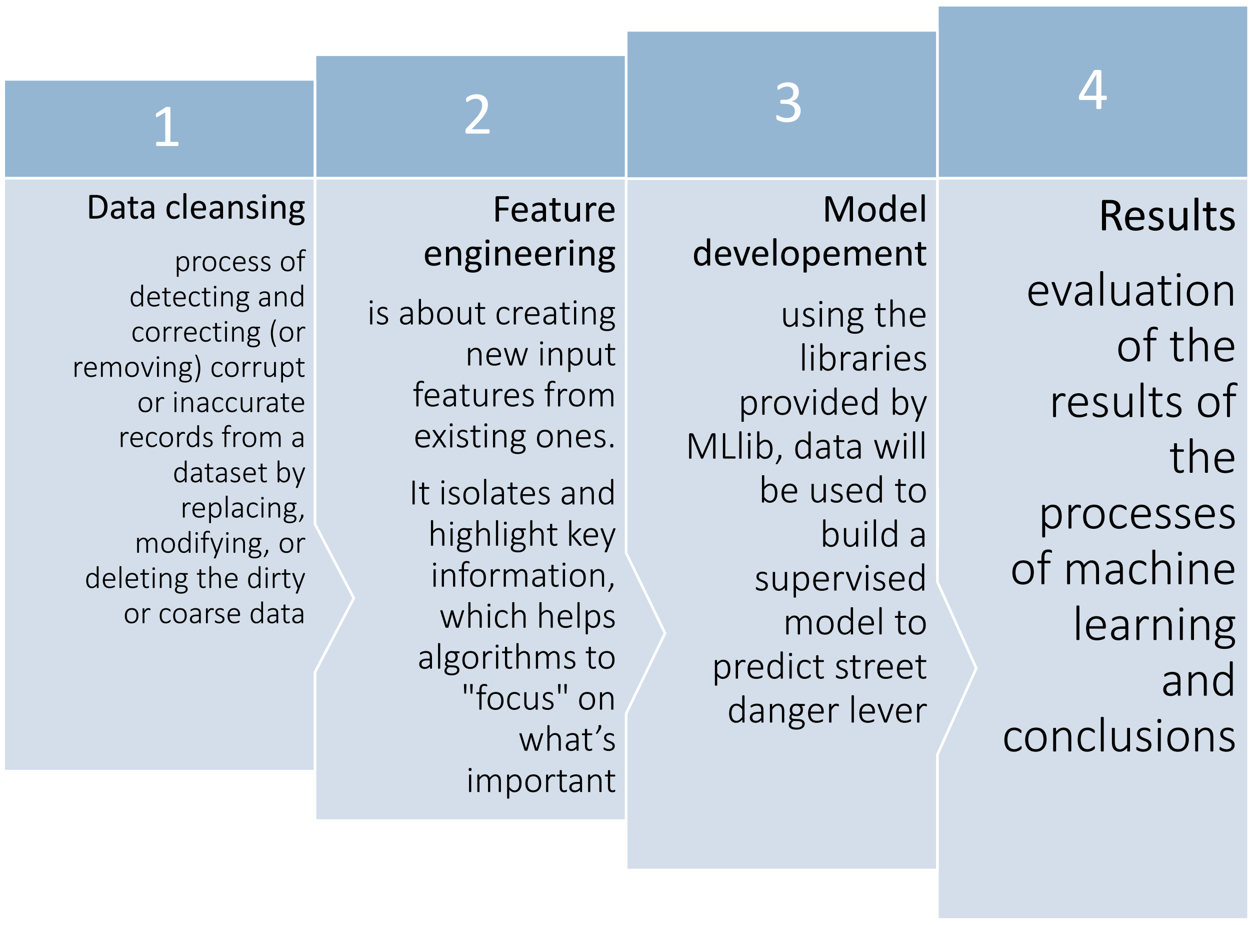
It comes with the Data dictionary displayed in *Table 1,* where “Severity” is the feature to predict.

ID
This is a unique identifier of the accident record.
Source
Indicates source of the accident report (i.e. the API which reported the accident.).
TMC
A traffic accident may have a Traffic Message Channel (TMC) code which provides more detailed description of the event.
Severity
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).
Start_Time
Shows start time of the accident in local time zone.
End_Time
Shows end time of the accident in local time zone.
Start_Lat
Shows latitude in GPS coordinate of the start point.
Start_Lng
Shows longitude in GPS coordinate of the start point.
End_Lat
Shows latitude in GPS coordinate of the end point.
End_Lng
Shows longitude in GPS coordinate of the end point.
Distance(mi)
The length of the road extent affected by the accident.
Description
Shows natural language description of the accident.
Number
Shows the street number in address field.
Street
Shows the street name in address field.
Side
Shows the relative side of the street (Right/Left) in address field.
City
Shows the city in address field.
County
Shows the county in address field.
State
Shows the state in address field.
Zipcode
Shows the zipcode in address field.
Country
Shows the country in address field.
Timezone
Shows timezone based on the location of the accident (eastern, central, etc.).
Airport_Code
Denotes an airport-based weather station which is the closest one to location of the accident.
Weather_Timestamp
Shows the time-stamp of weather observation record (in local time).
Temperature(F)

Shows the temperature (in Fahrenheit).
Wind_Chill(F)
Shows the wind chill (in Fahrenheit).
Humidity(%)
Shows the humidity (in percentage).
Pressure(in)
Shows the air pressure (in inches).
Visibility(mi)
Shows visibility (in miles).
Wind_Direction
Shows wind direction.
Wind_Speed(mph)
Shows wind speed (in miles per hour).
Precipitation(in)
Shows precipitation amount in inches, if there is any.
Weather_Condition
Shows the weather condition (rain, snow, thunderstorm, fog, etc.)
Amenity
A POI annotation which indicates presence of amenity in a nearby location.
Bump
A POI annotation which indicates presence of speed bump or hump in a nearby location.
Crossing
A POI annotation which indicates presence of crossing in a nearby location.
Give_Way
A POI annotation which indicates presence of give_way in a nearby location.
Junction
A POI annotation which indicates presence of junction in a nearby location.
No_Exit
A POI annotation which indicates presence of no_exit in a nearby location.
Railway
A POI annotation which indicates presence of railway in a nearby location.
Roundabout
A POI annotation which indicates presence of roundabout in a nearby location.
Station
A POI annotation which indicates presence of station in a nearby location.
Stop
A POI annotation which indicates presence of stop in a nearby location.
Traffic_Calming
A POI annotation which indicates presence of traffic_calming in a nearby location.
Traffic_Signal
A POI annotation which indicates presence of traffic_signal in a nearby location.
Turning_Loop
A POI annotation which indicates presence of turning_loop in a nearby location.
Sunrise_Sunset
Shows the period of day (i.e. day or night) based on sunrise/sunset.
Civil_Twilight
Shows the period of day (i.e. day or night) based on civil twilight.
Nautical_Twilight
Shows the period of day (i.e. day or night) based on nautical twilight.
Astronomical_Twilight
Shows the period of day (i.e. day or night) based on astronomical twilight.

Table 1- List of columns available in the dataset

The workflow



1

Data cleansing

process of detecting and correcting (or removing) corrupt or inaccurate records from a dataset by replacing, modifying, or deleting the dirty or coarse data

2

Feature engineering

is about creating new input features from existing ones.

It isolates and highlight key information, which helps algorithms to "focus" on what's important

3

Model developement

using the libraries provided by MLlib, data will be used to build a supervised model to predict street danger lever

4

Results

evaluation of the results of the processes of machine learning and conclusions

Data cleansing

- The first level of filtering comes with the option mode “DROPMALFORMED” when reading the .csv file from the storage:

```
temp_df = spark.read.csv("US_Accidents_Dec19.csv",
                header=True, mode="DROPMALFORMED")
```

that *ignores the whole corrupted records* [13].

- Drops duplicated rows from the dataset

```
temp_df = temp_df.dropDuplicates()
```

Feature Engineering

Feature selection

Not all of the features are relevant for the study: these are a burden for the processing and data management, therefore, in the *Table 2* are listed the discarded features with the reason of the removal:

Discarded feature - name	Description	Removal reason
Id	This is a unique identifier of the accident record.	Not relevant.
Source	Indicates source of the accident report (i.e. the API which reported the accident.).	Not relevant, although it would be interesting the comparison of data quality between different platforms
TMC [14]	A traffic accident may have a Traffic Message Channel (TMC) code which provides more detailed description of the event.	Gives clues about the traffic status but not about the accident di per se, moreover the dataset provides one TMC per row: the codes related report events so heterogeneous that can happen together.
Number	Shows the street number in address field.	Not useful for the analysis, this information is not available in the places far from houses.
Street	Shows the street name in address field.	
Side	Shows the relative side of the street (Right/Left) in address field.	
City	Shows the city in address field.	These can be used in further analysis to get and correlate information about the density of the area (high density - downtown or far periphery) and accidents.
Zipcode	Shows the zipcode in address field.	
Country	Shows the country in address field.	Is taken for granted that the entire dataset is about US accidents.
Airport_Code	Denotes an airport-based weather station which is the closest one to location of the accident.	The distance between the airport and the accident can be so long to result not influent.
Amenity	A POI annotation which indicates presence of amenity in a nearby location.	Too much generic as a definition.

Traffic_Calming	A POI annotation which indicates presence of traffic_calming in a nearby location.	
Station	A POI annotation which indicates presence of station in a nearby location.	Not indicating the type of station (train, bus, metro, police etc).
Turning_Loop	A POI annotation which indicates presence of turning_loop in a nearby location.	It's a rare event among the total length of the roads, therefore, it can be discarded.
Sunrise_Sunset	Shows the period of day (i.e. day or night) based on sunrise/sunset.	Choosing, instead, the Nautical twilight because it's the point of the day in which the darkness is more present and the artificial lights tend to be turned on [15].
Civil_Twilight	Shows the period of day (i.e. day or night) based on civil twilight.	
Astronomical_Twilight	Shows the period of day (i.e. day or night) based on astronomical twilight.	
End_Lat	Shows latitude in GPS coordinate of the end point.	This study focuses on the area in which the accident happened (not a particular point)
End_Lng	Shows longitude in GPS coordinate of the end point.	
Distance(mi)	The length of the road extent affected by the accident.	Distance is a consequence of the accident, not a factor, therefore it's discarded.
Description	Shows natural language description of the accident.	Extracting insights from the description can be tricky for this analysis, so it's discarded
End_Time	Shows end time of the accident in local time zone.	End_Time is a consequence of the accident, not a factor, therefore it's discarded. Moreover, the end of the accident it strictly depends by many factors not easily predictable (traffic, the report to the authorities delay, distance from the nearest utilities station etc.)
Wind_Direction	Shows wind direction.	It is not helpful knowing the wind direction if there is no precise knowledge about the driving direction at the moment of the impact
Weather_Timestamp	Shows the time-stamp of weather observation record (in local time).	Redundancy given by the already known Start_time of the accident. Moreover the time will be condensed in macroblocks, so extreme coherency between the timestamp of the sampling and the accident is not needed.

Table 1- List of features discarded

- Drops some columns from the dataframe

```
columns_to_drop = ['id', 'Source', 'TMC', 'Number', 'Street', 'Side',
                   'City', 'Zipcode', 'Country', 'Airport_Code', 'Amenity',
                   'Traffic_Calming', 'Station', 'Turning_Loop',
                   'Sunrise_Sunset', 'Civil_Twilight', 'Astronomical_Twilight',
                  'End_Lat', 'End_Lng', 'Distance(mi)', 'Description',
                   'End_Time', 'Wind_Direction', 'Weather_Timestamp'];
temp_df = temp_df.drop(*columns_to_drop)
```

Splitting and bucketing

Division of the field datetime of “Start_Time”

In order to deal with date and time separately. The new fields are called “Accident_Date” and “Accident_Time”.

- Splits the field "Start_Time" in date and time

```
split_col = F.split(temp_df['Start_Time'], ' ')
```

- Creates the feature “Accident_Date”

```
df_nodatetime = temp_df.withColumn('Accident_Date', split_col.getItem(0))
```

- Creates the feature “Accident_Time”

```
df_nodatetime = df_nodatetime.withColumn('Accident_Time',
                split_col.getItem(1))
```

where “F” is the namespace related to the library function of Spark SQL.

Binning/bucketing for the field “Accident_Time”

According to the US Office of Human Resources Management [16], the typical working hour is 8:30 am to 5 pm, so it’s a good idea using the bucketing technique to divide the hours of the day in six bins of 4 hours each.

- Bin 1: 0:00 am – 3:59 am / 0:00 – 3:59
- Bin 2: 4:00 am – 7:59 am / 4:00 – 7:59
- Bin 3: 8:00 am – 11:59 am / 8:00 – 11:59
- Bin 4: 12:00 am – 3:59 pm / 12:00 – 15:59
- Bin 5: 4:00 pm – 7:59 pm / 16:00 – 19:59
- Bin 6: 8:00 pm – 11:59 pm / 20:00 – 23:59

This step is done because of the different amount of traffic during the day, naturally influenced by the work commuting.

- Defines of the hour according to the reference table

```
time_intervals = ["0:00 - 3:59", "4:00 - 7:59", "8:00 - 11:59", "12:00 -
15:59", "16:00 - 19:59", "20:00 - 23:59"];
```

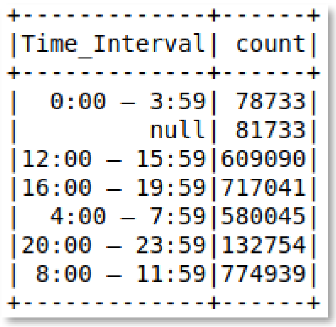
- Defines the function to transfer the hour in a bin

```
def time_categorizer(argument):
    if argument != None:
        hour = (int)(argument.split(':')[0])
        if 0 <= hour < 4:
            return time_intervals[0]
        elif 4 <= hour < 8:
            return time_intervals[1]
        elif 8 <= hour < 12:
            return time_intervals[2]
        elif 12 <= hour < 16:
            return time_intervals[3]
        elif 16 <= hour < 20:
            return time_intervals[4]
        elif 20 < hour:
            return time_intervals[5]
```

- Defines a User Defined Function and executes the bucketing

```
bin_time_udf = udf(time_categorizer, StringType())
df_timebinned = df_nodatetime.withColumn('Time_Interval',
bin_time_udf('Accident_Time'))
```

The results describe the lack of information about time for more than 80000 records (*Table 3*)



```
+-------------+------+
|Time_Interval| count|
+-------------+------+
|   0:00 – 3:59| 78733|
|         null| 81733|
|12:00 – 15:59|609090|
|16:00 – 19:59|717041|
|   4:00 – 7:59|580045|
|20:00 – 23:59|132754|
|  8:00 – 11:59|774939|
+-------------+------+
```

Table 3 – Quantity of observations for each time bin

The creation of another category as suggested in [17] and many other resource, it's more convenient than discarding the entire record with null on that feature; therefore the 'null' value will be replaced by 'Missing', in order to keep it as an information.

```
df_timebinned = df_timebinned.fillna('Missing', subset=['Time_Interval'])
```

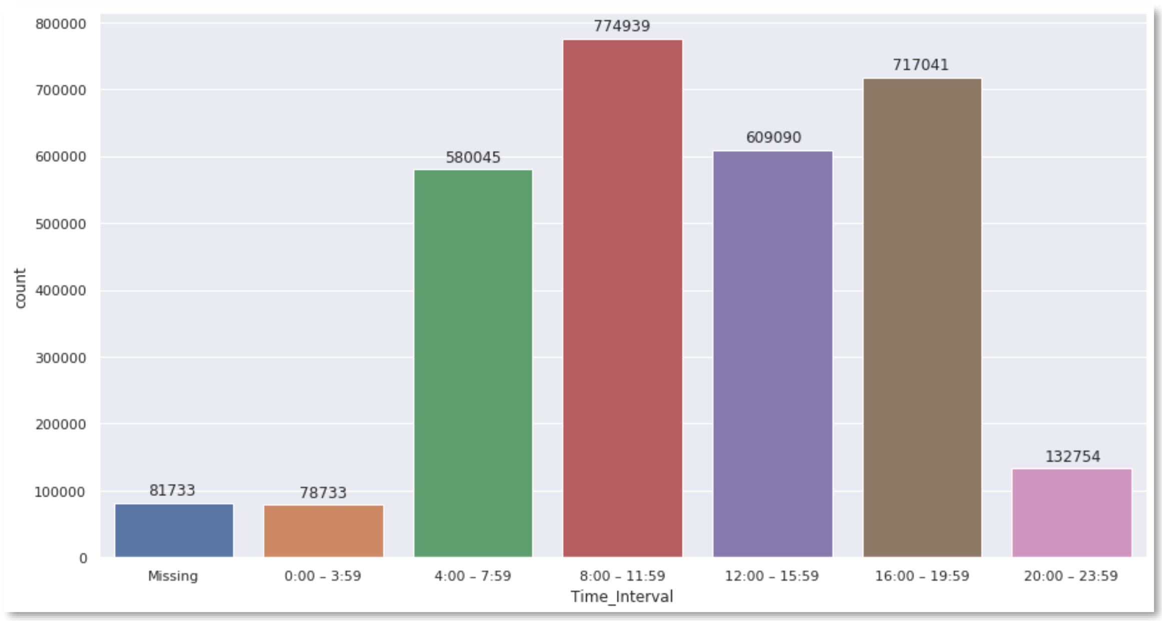


Figure 1 – Number of accidents per time interval

It's pretty visible on the *Figure 1* how the majority of the crashes happen during the time slots typically regarding the commuting to/from the workplace (8:00-11:59; 16:00-19:59), which involves higher traffic intensity, the third peak 12:00/15:59 could be also a consequence of working breaks (lunch etc.) from the working place.

Conversion of number in months of the "Accident_Date"

- Mapping of the months

```
dictionary_months = {1: "Jan",
            2: "Feb",
            3: "Mar",
            4: "Apr",
            5: "May",
            6: "Jun",
            7: "Jul",
            8: "Aug",
            9: "Sep",
            10: "Oct",
            11: "Nov",
            12: "Dec"
        }
```

- Defines the function to categorize the month

```
def month_categorizer(argument):
    if argument != None:
        month = (int)(argument.split('-')[1])
        switcher_month = dictionary_months
    return str(switcher_month.get(month))
```

- Defines a User Defined Function and executes the categorization

```
bin_month_udf = udf(month_categorizer, StringType())
df_monthbinned = df_timebinned.withColumn('Month',
                 bin_month_udf('Accident_Date'))
```

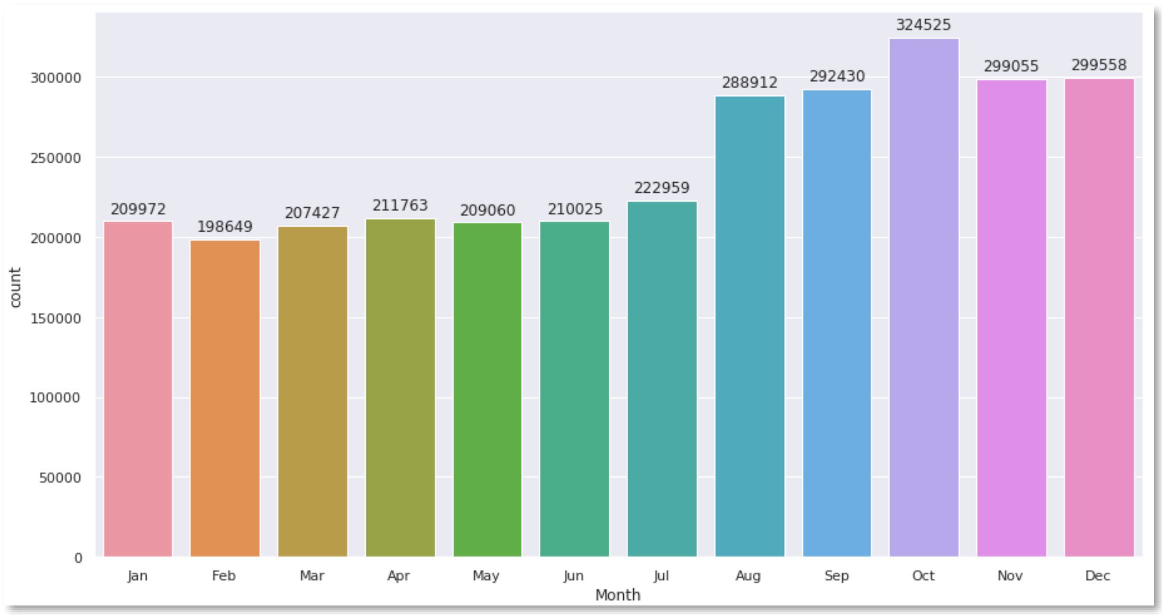


Figure 2 – Number of accidents per month (Feb '16 – Dec '19)

Figure 2 shows no presence of "Missing" values, and that the biggest amount of accidents happened in the second half of the year, with a peak in October and with the biggest absolute discards among following month between December and January (89586 units of difference) and between July and August (65953).

May the accidents in the second half of the year rose due to the increasing traffic local tourism traffic (summer/winter vacation) or for heavier atmosphere events?

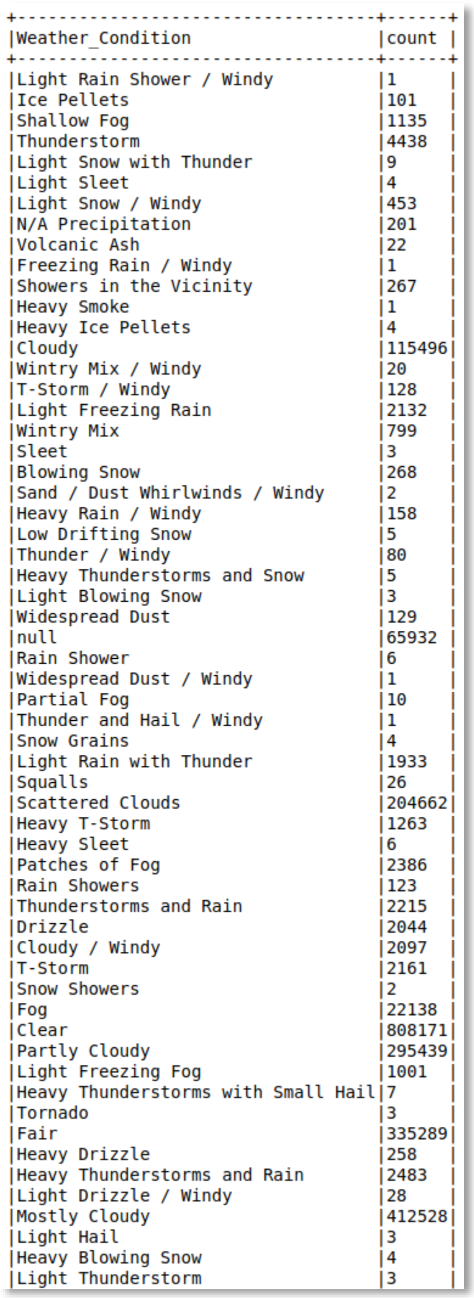
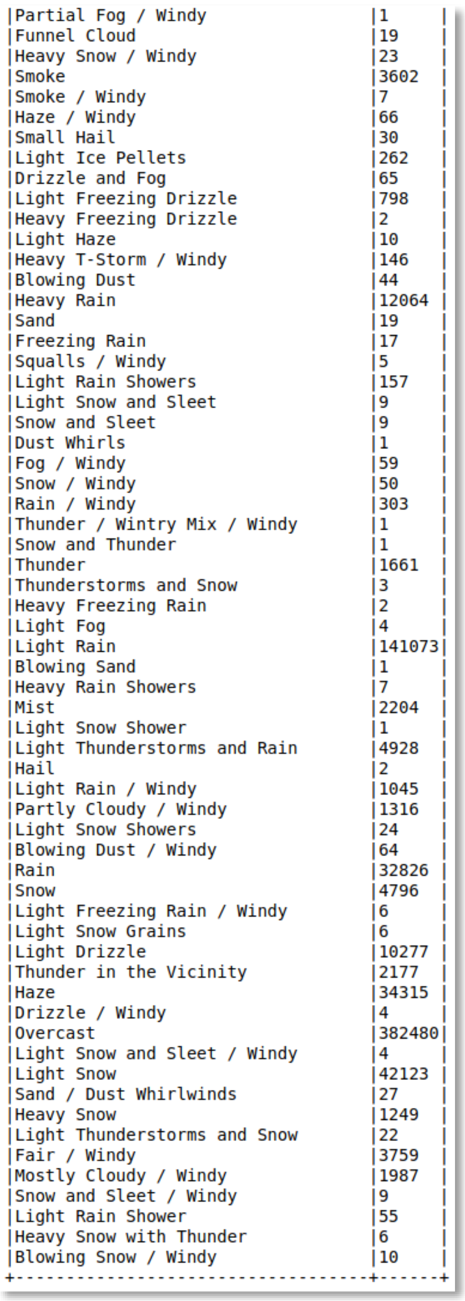
Moreover, the dataset under exam starts from February 2016 instead of January, entailing a possibly small bias in the counting of accidents reported in January.

Binning/bucketing for the months of the “Weather_Condition”:

a brief looking at the dataset shows a lot of different weather conditions (*Tables 4*) ; this steps wants to evaluate if it is possible to bucket them to get a small but still significant varieties.

- Lists all the value with the “groupBy” function

```
frequencies_weather = df_monthbinned.groupBy('Weather_Condition').count()
frequencies_weather.show(frequencies_weather.count(), truncate = False)
```

Weather_Condition	count
Light Rain Shower / Windy	1
Ice Pellets	101
Shallow Fog	1135
Thunderstorm	4438
Light Snow with Thunder	9
Light Sleet	4
Light Snow / Windy	453
N/A Precipitation	201
Volcanic Ash	22
Freezing Rain / Windy	1
Showers in the Vicinity	267
Heavy Smoke	1
Heavy Ice Pellets	4
Cloudy	115496
Wintry Mix / Windy	20
T-Storm / Windy	128
Light Freezing Rain	2132
Wintry Mix	799
Sleet	3
Blowing Snow	268
Sand / Dust Whirlwinds / Windy	2
Heavy Rain / Windy	158
Low Drifting Snow	5
Thunder / Windy	80
Heavy Thunderstorms and Snow	5
Light Blowing Snow	3
Widespread Dust	129
null	65932
Rain Shower	6
Widespread Dust / Windy	1
Partial Fog	10
Thunder and Hail / Windy	1
Snow Grains	4
Light Rain with Thunder	1933
Squalls	26
Scattered Clouds	204662
Heavy T-Storm	1263
Heavy Sleet	6
Patches of Fog	2386
Rain Showers	123
Thunderstorms and Rain	2215
Drizzle	2044
Cloudy / Windy	2097
T-Storm	2161
Snow Showers	2
Fog	22138
Clear	808171
Partly Cloudy	295439
Light Freezing Fog	1001
Heavy Thunderstorms with Small Hail	7
Tornado	3
Fair	335289
Heavy Drizzle	258
Heavy Thunderstorms and Rain	2483
Light Drizzle / Windy	28
Mostly Cloudy	412528
Light Hail	3
Heavy Blowing Snow	4
Light Thunderstorm	3
Partial Fog / Windy	1
Funnel Cloud	19
Heavy Snow / Windy	23
Smoke	3602
Smoke / Windy	7
Haze / Windy	66
Small Hail	30
Light Ice Pellets	262
Drizzle and Fog	65
Light Freezing Drizzle	798
Heavy Freezing Drizzle	2
Light Haze	10
Heavy T-Storm / Windy	146
Blowing Dust	44
Heavy Rain	12064
Sand	19
Freezing Rain	17
Squalls / Windy	5
Light Rain Showers	157
Light Snow and Sleet	9
Snow and Sleet	9
Dust Whirls	1
Fog / Windy	59
Snow / Windy	50
Rain / Windy	303
Thunder / Wintry Mix / Windy	1
Snow and Thunder	1
Thunder	1661
Thunderstorms and Snow	3
Heavy Freezing Rain	2
Light Fog	4
Light Rain	141073
Blowing Sand	1
Heavy Rain Showers	7
Mist	2204
Light Snow Shower	1
Light Thunderstorms and Rain	4928
Hail	2
Light Rain / Windy	1045
Partly Cloudy / Windy	1316
Light Snow Showers	24
Blowing Dust / Windy	64
Rain	32826
Snow	4796
Light Freezing Rain / Windy	6
Light Snow Grains	6
Light Drizzle	10277
Thunder in the Vicinity	2177
Haze	34315
Drizzle / Windy	4
Overcast	382480
Light Snow and Sleet / Windy	4
Light Snow	42123
Sand / Dust Whirlwinds	27
Heavy Snow	1249
Light Thunderstorms and Snow	22
Fair / Windy	3759
Mostly Cloudy / Windy	1987
Snow and Sleet / Windy	9
Light Rain Shower	55
Heavy Snow with Thunder	6
Blowing Snow / Windy	10

Table 4 – Types of weather registered

The number of conditions found is particularly high due to the detailed level of the descriptive words. Since the dataset already provides the quantitative values for precipitations and wind speed, it would be a good idea to bin all the values related to the rain and wind phenomena.

Table 5 shows how all these 121 values have been binned in six macro-blocks: Rain, Fog/Smoke, No precipitations, Snow/Frost, Extreme wind, Dust/Sand/Ash.
The value "null" has been included into "No_precipitation" since it is assumed that the weather_condition parameter is skipped if the operator thinks there is nothing special to report.
The values with thunderstorm only, have been included in "No_precipitation" as well since a lightning doesn't involve precipitation but only a short visual effect.

Category assigned	Value
Rain	Light Rain Shower / Windy
	Freezing Rain / Windy
	Showers in the Vicinity
	Rain Shower
	Light Freezing Rain
	Light Rain with Thunder
	Heavy Rain / Windy
	Rain Showers
	Thunderstorms and Rain
	Drizzle
	Freezing Rain
	Heavy Drizzle
	Heavy Thunderstorms and Rain
	Light Drizzle / Windy
	Light Freezing Drizzle
	Light Rain Showers
	Heavy Freezing Drizzle
	Heavy Rain
	Rain / Windy
	Heavy Freezing Rain
	Light Rain
	Heavy Rain Showers
	Light Thunderstorms and Rain
	Light Rain / Windy
	Rain
	Light Freezing Rain / Windy
	Light Drizzle
	Drizzle / Windy
	Light Rain Shower
Snow/Frost	Ice Pellets
	Light Snow with Thunder
	Light Sleet
	Light Snow / Windy
	Wintry Mix / Windy
	Wintry Mix
	Sleet
	Blowing Snow
	Low Drifting Snow
	Heavy Thunderstorms and Snow
	Light Blowing Snow
	Thunder and Hail / Windy
	Snow Grains
	Heavy Sleet
	Snow Showers
	Heavy Thunderstorms with Small Hail
	Light Hail

	Light Ice Pellets
	Heavy Blowing Snow
	Small Hail
	Heavy Snow / Windy
	Light Snow and Sleet
	Snow and Sleet
	Snow / Windy
	Snow and Thunder
	Thunder / Wintry Mix / Windy
	Thunderstorms and Snow
	Light Snow Shower
	Hail
	Light Snow Showers
	Snow
	Light Snow Grains
	Light Snow and Sleet / Windy
	Light Snow
	Heavy Snow
	Light Thunderstorms and Snow
	Snow and Sleet / Windy
	Heavy Snow with Thunder
	Blowing Snow / Windy
	Heavy Ice Pellets
Fog/Smoke	Shallow Fog
	Heavy Smoke
	Patches of Fog
	Fog
	Partial Fog
	Light Freezing Fog
	Smoke / Windy
	Light Haze
	Haze / Windy
	Partial Fog / Windy
	Smoke
	Drizzle and Fog
	Fog / Windy
	Light Fog
	Mist
	Haze
No precipitation	N/A Precipitation
	Cloudy
	null
	Clear
	Thunder / Windy
	Scattered Clouds
	Cloudy / Windy
	T-Storm / Windy

	T-Storm
	Heavy T-Storm
	Partly Cloudy
	Mostly Cloudy
	Fair
	Light Thunderstorm
	Heavy T-Storm / Windy
	Thunder
	Partly Cloudy / Windy
	Thunder in the Vicinity
	Overcast
	Fair / Windy
	Mostly Cloudy / Windy
Extreme Wind	Funnel Cloud
	Squalls
	Squalls / Windy
	Tornado
Dust/Sand/Ash	Sand / Dust Whirlwinds / Windy
	Volcanic Ash
	Widespread Dust / Windy
	Widespread Dust
	Sand
	Dust Whirls
	Blowing Dust
	Blowing Sand
	Blowing Dust / Windy
	Sand / Dust Whirlwinds

Table 5 – Weather bins organization

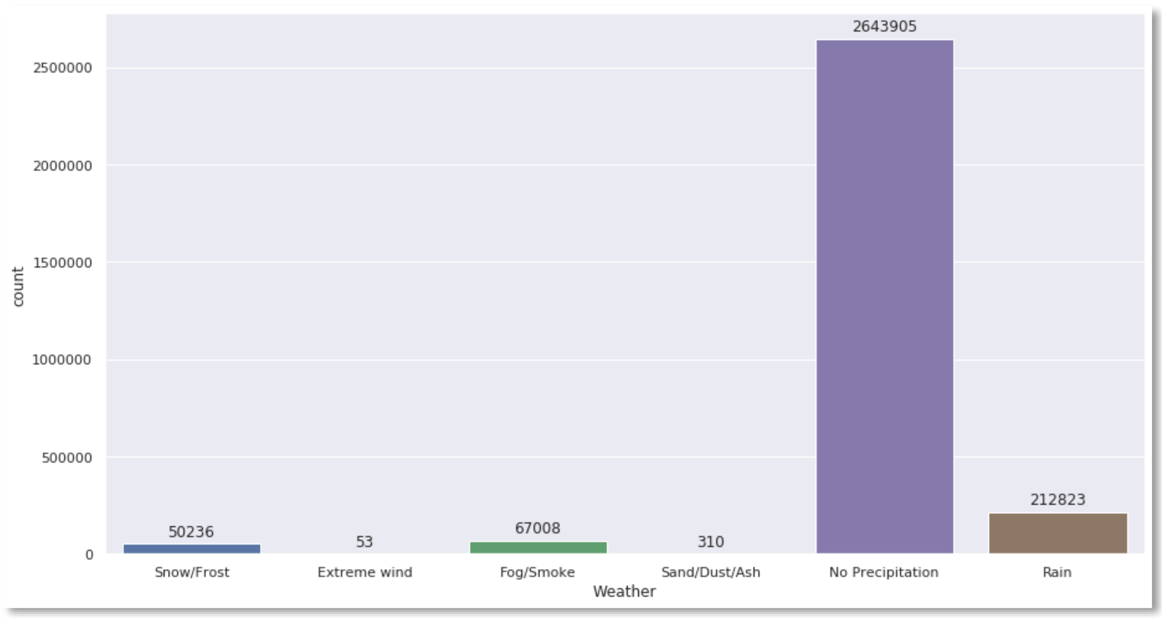


Figure 3 – Number of accidents per weather condition

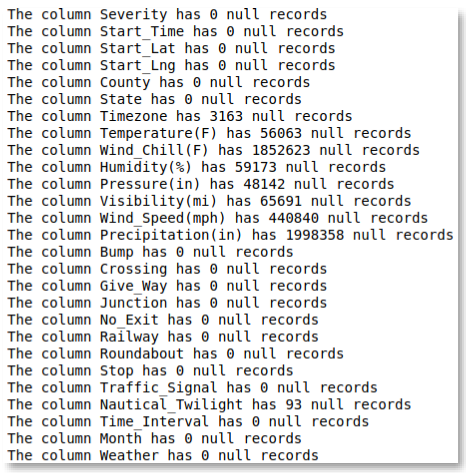
As it's seen in *Figure 3*, the number of accidents is not strictly dependant by the adverse weather condition, in fact the greatest amount of crashes happen during fine weather days; maybe in adverse conditions the driving is more prudent.

Dealing with uncomplete data

The next step is dealing with the remaining uncomplete data within the features of the dataset, as already did with the categoric variables related to the time of the accident time interval, where hasn't been possible to categorize all the dataset.

- Reveals what features still have some null values

```
columns = df_weatherbinned.columns
for column_df in columns:
    print("The column " + column_df + " has " +
  str(df_weatherbinned.filter(df_weatherbinned[column_df].isNull()).count())
  + " null records")
```



```
The column Severity has 0 null records
The column Start_Time has 0 null records
The column Start_Lat has 0 null records
The column Start_Lng has 0 null records
The column County has 0 null records
The column State has 0 null records
The column Timezone has 3163 null records
The column Temperature(F) has 56063 null records
The column Wind_Chill(F) has 1852623 null records
The column Humidity(%) has 59173 null records
The column Pressure(in) has 48142 null records
The column Visibility(mi) has 65691 null records
The column Wind_Speed(mph) has 440840 null records
The column Precipitation(in) has 1998358 null records
The column Bump has 0 null records
The column Crossing has 0 null records
The column Give_Way has 0 null records
The column Junction has 0 null records
The column No_Exit has 0 null records
The column Railway has 0 null records
The column Roundabout has 0 null records
The column Stop has 0 null records
The column Traffic_Signal has 0 null records
The column Nautical_Twilight has 93 null records
The column Time_Interval has 0 null records
The column Month has 0 null records
The column Weather has 0 null records
```

Figure 4 – List of features with quantity of null values

One important categorical variable in this problem is “Nautical_Twilight” which can result useful to be included in the model because of its clues about road visibility (“Day” or “Night” are the possible options). In *Figure 4* is noticeable that 93 records of the dataset have no “Nautical_Twilight” value set:
the parameters needed to deduct these are the geographical coordinates and the time of the crash with the related timezone.

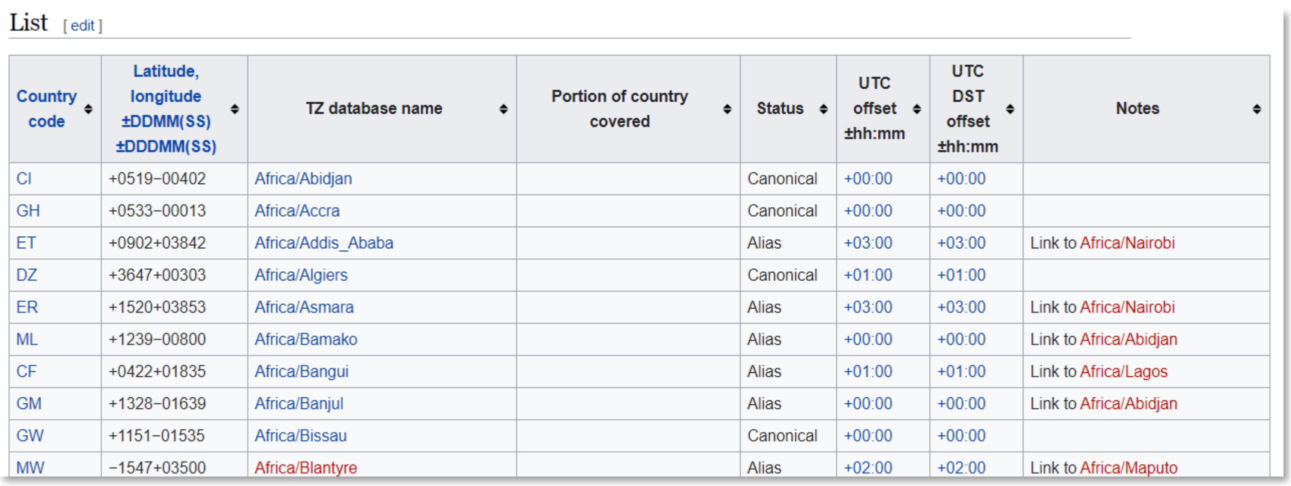
Unfortunately, the “Timezone” field is not always valued since 3163 records don’t have this information available; but thanks to the library “TimezoneFinder” [18] is it possible to determine it.

- Defines the function to get the timezone, give the coordinates of the accident and executes it with a User Defined Function

```
def getTimezone(latitude_input, longitude_input, timezone_input):
    if timezone_input == None:
        tf = TimezoneFinder()
        latitude, longitude = float(latitude_input), float(longitude_input)
        return tf.timezone_at(lng=longitude, lat=latitude)
    else:
        return timezone_input

get_timezone_udf = udf(getTimezone, StringType())
df_timezoned = df_weatherbinned.withColumn('Timezone', get_timezone_udf('Start_Lat',
'Start_Lng', 'Timezone'))
```

Once all the timezones have been found, the next step is finding the twilight condition, using the library Astropy [19]. Timezone is necessary because Astropy gives always back the UTC (Coordianted Universal Time [20]) as output regardless of geographical location, so, hours/minutes offset will be added or subtracted.

One simple idea to determine the offset is scraping it from Wikipedia [24] (*Figure 5*) using Beautifulsoup [21] webscraping library, with the foresight to take in account the change of time along the different seasons (Daylight Saving Time).



List [edit]

Country code	Latitude, longitude ±DDMM(SS) ±DDDMM(SS)	TZ database name	Portion of country covered	Status	UTC offset ±hh:mm	UTC DST offset ±hh:mm	Notes
CI	+0519−00402	Africa/Abidjan		Canonical	+00:00	+00:00	
GH	+0533−00013	Africa/Accra		Canonical	+00:00	+00:00	
ET	+0902+03842	Africa/Addis_Ababa		Alias	+03:00	+03:00	Link to Africa/Nairobi
DZ	+3647+00303	Africa/Algiers		Canonical	+01:00	+01:00	
ER	+1520+03853	Africa/Asmara		Alias	+03:00	+03:00	Link to Africa/Nairobi
ML	+1239−00800	Africa/Bamako		Alias	+00:00	+00:00	Link to Africa/Abidjan
CF	+0422+01835	Africa/Bangui		Alias	+01:00	+01:00	Link to Africa/Lagos
GM	+1328−01639	Africa/Banjul		Alias	+00:00	+00:00	Link to Africa/Abidjan
GW	+1151−01535	Africa/Bissau		Canonical	+00:00	+00:00	
MW	−1547+03500	Africa/Blantyre		Alias	+02:00	+02:00	Link to Africa/Maputo

Figure 5 – Table of Wikipedia reporting the time offsets for each timezone

An overview of the workflow to know the “Nautical_twilight” is therefore depicted in the *Figure 6*

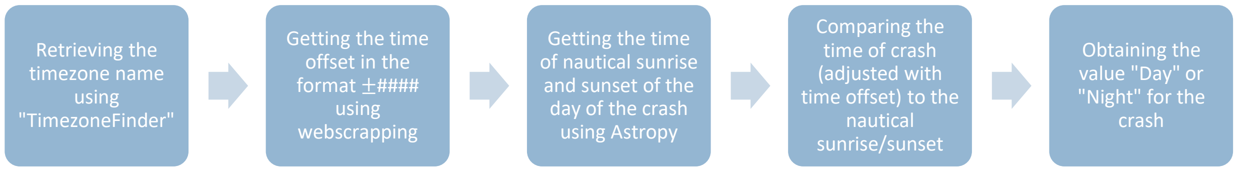
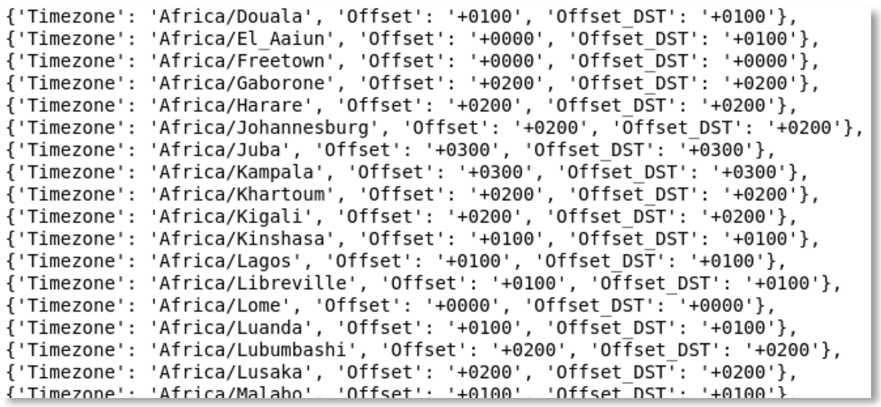

Retrieving the timezone name using "TimezoneFinder"
Getting the time offset in the format ±#### using webscrapping
Getting the time of nautical sunrise and sunset of the day of the crash using Astropy
Comparing the time of crash (adjusted with time offset) to the nautical sunrise/sunset
Obtaining the value "Day" or "Night" for the crash

Figure 6 – Step process to get the “Day” or “Night” value

- Webscrapes the Wikipedia page reguarding the timezones

```
#calls and gets the wikipedia page
URL = 'https://en.wikipedia.org/wiki/List_of_tz_database_time_zones'
page = requests.get(URL)

#parses the page
soup = BeautifulSoup(page.content, 'html.parser')

#gets the table with the timestamps
table_timezones = soup.find_all('table', class_='wikitable')[0]
#collects all the rows of the table
rows_of_table = table_timezones.find_all('tr')
#discards the first row (which is empty)
rows_of_table.pop(0)
timeZonesArray = []
for row in rows_of_table:
    tds = row.find_all("td")
    #gets the standard time
    offset = tds[5].text.strip()
    offset = offset.replace('−', '-').replace(':', '')
    #gets the Daylight Saving Time - DST
    offset_dst = tds[6].text.strip()
    offset_dst = offset_dst.replace('−', '-').replace(':', '')
    timeZonesArray.append({'Timezone': tds[2].text.strip(),'Offset': offset,
'Offset_DST': offset_dst})
```



```
{'Timezone': 'Africa/Douala', 'Offset': '+0100', 'Offset_DST': '+0100'},
{'Timezone': 'Africa/El_Aaiun', 'Offset': '+0000', 'Offset_DST': '+0100'},
{'Timezone': 'Africa/Freetown', 'Offset': '+0000', 'Offset_DST': '+0000'},
{'Timezone': 'Africa/Gaborone', 'Offset': '+0200', 'Offset_DST': '+0200'},
{'Timezone': 'Africa/Harare', 'Offset': '+0200', 'Offset_DST': '+0200'},
{'Timezone': 'Africa/Johannesburg', 'Offset': '+0200', 'Offset_DST': '+0200'},
{'Timezone': 'Africa/Juba', 'Offset': '+0300', 'Offset_DST': '+0300'},
{'Timezone': 'Africa/Kampala', 'Offset': '+0300', 'Offset_DST': '+0300'},
{'Timezone': 'Africa/Khartoum', 'Offset': '+0200', 'Offset_DST': '+0200'},
{'Timezone': 'Africa/Kigali', 'Offset': '+0200', 'Offset_DST': '+0200'},
{'Timezone': 'Africa/Kinshasa', 'Offset': '+0100', 'Offset_DST': '+0100'},
{'Timezone': 'Africa/Lagos', 'Offset': '+0100', 'Offset_DST': '+0100'},
{'Timezone': 'Africa/Libreville', 'Offset': '+0100', 'Offset_DST': '+0100'},
{'Timezone': 'Africa/Lome', 'Offset': '+0000', 'Offset_DST': '+0000'},
{'Timezone': 'Africa/Luanda', 'Offset': '+0100', 'Offset_DST': '+0100'},
{'Timezone': 'Africa/Lubumbashi', 'Offset': '+0200', 'Offset_DST': '+0200'},
{'Timezone': 'Africa/Lusaka', 'Offset': '+0200', 'Offset_DST': '+0200'},
{'Timezone': 'Africa/Malabo', 'Offset': '+0100', 'Offset_DST': '+0100'}
```

Figure 7 – Result of the webscraping process

- Function to determine whether the time is DST in that precise date of the year, using the library pytz.

```
def is_dst(datetime=None, timezone="UTC"):
    if datetime is None:
        datetime = datetime_lib.utcnow()
    timezone = pytz.timezone(timezone)
    timezone_aware_date = timezone.localize(datetime, is_dst=None)
    return timezone_aware_date.tzinfo._dst.seconds != 0
```

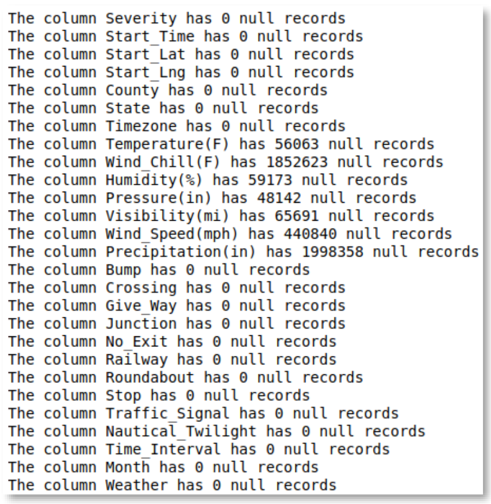
- Astropy function to adjust the time offset

```
def setOffset(datetime_astropy, timezone, offset_licteral="+0000"):
    #gets the date time without milliseconds
    date_string = Time(datetime_astropy, format='iso',
scale='utc').value.split('.')[0]
    #adds the refernce offset to convert from
    date_string = date_string + " " + offset_licteral;
    dt = parse(date_string)
    #converts to the time of the timezone
    localtime = dt.astimezone (pytz.timezone(timezone))
    #returns in a Astropy format
    return Time.strptime(localtime.strftime ("%Y-%m-%d %H:%M:%S"), '%Y-%m-%d
%H:%M:%S', format='jd')
```

- Defines the function that generates the value for “Nautical_twilight” and executes it with a UDF

```
def twilight_categorizer(latitude, longitude, datetime, timezone, Nautical_Twilight):

    if Nautical_Twilight != 'Day' and Nautical_Twilight != 'Night':
        observer = astroplan.Observer(longitude=float(longitude)*u.deg,
latitude=float(latitude)*u.deg, timezone=timezone)

        #generates the time 00:00:01 of the beginning of the day in order to get the
sunrise and sunset of that day
        firstSecondOfTheDay = getFirstSecondOfTheDay(datetime)
        time_of_sunrise = observer.twilight_morning_nautical(firstSecondOfTheDay,
which='next')
        time_of_sunset = observer.twilight_evening_nautical(firstSecondOfTheDay,
which='next')

        time_of_crash = Time.strptime(datetime, '%Y-%m-%d %H:%M:%S', format='jd')
        #adds proper offset based on timezone
        time_of_sunrise = setOffset(time_of_sunrise, timezone)
        time_of_sunset = setOffset(time_of_sunset, timezone)

        if time_of_sunrise < time_of_crash < time_of_sunset:
            return 'Day'
        else:
            return 'Night'
    else:
        return Nautical_Twilight;
```

Figure 8 shows how all the fields “Timestamp” and “Nautical_Twilight” have been filled.



```
The column Severity has 0 null records
The column Start_Time has 0 null records
The column Start_Lat has 0 null records
The column Start_Lng has 0 null records
The column County has 0 null records
The column State has 0 null records
The column Timezone has 0 null records
The column Temperature(F) has 56063 null records
The column Wind_Chill(F) has 1852623 null records
The column Humidity(%) has 59173 null records
The column Pressure(in) has 48142 null records
The column Visibility(mi) has 65691 null records
The column Wind_Speed(mph) has 440840 null records
The column Precipitation(in) has 1998358 null records
The column Bump has 0 null records
The column Crossing has 0 null records
The column Give_Way has 0 null records
The column Junction has 0 null records
The column No_Exit has 0 null records
The column Railway has 0 null records
The column Roundabout has 0 null records
The column Stop has 0 null records
The column Traffic_Signal has 0 null records
The column Nautical_Twilight has 0 null records
The column Time_Interval has 0 null records
The column Month has 0 null records
The column Weather has 0 null records
```

Figure 8 – List of features with quantity of null values after “Nautical_Twilight” filling

Switching to the data related to continues atmospheric values, we have the following fields to deal with:

- *Precipitation(in)*
  The easiest way to reduce the number of null values is using the information in “Weather” produced earlier: hence, the rows with a weather which is different from “Rain” will see their “Precipitation(in)” value set to zero.

  - Sets to zero all the values of “Precipitation(in)” where the corresponding weather is different from the category “Rain”

    ```
    df_weather_fillprec = df_twilightcat.where(df_twilightcat['Weather'] !=
    'Rain').na.fill({'Precipitation(in)': 0})
    ```

- *Temperature(F), Wind_Chill(F), Humidity(%), Pressure(in), Visibility(mi), Wind_Speed(mph)*

  For these numerical variables, the technique chosen is to replace the null value with the average number obtained from the processing of data from the geographical surroundings.

  Since it would be computationally heavy to circumscribe the source data starting from the coordinates of the crash and setting a buffer area of, for example 20mi for each one of the required rows, but at the same time using the overall averages for each State has not been felt as a highly robust way (a state can be so big to have many different subclimates); the script will get the averages of each county (a much smaller area than a State, but not punctiform as a coordinate) for each Month, and average them.

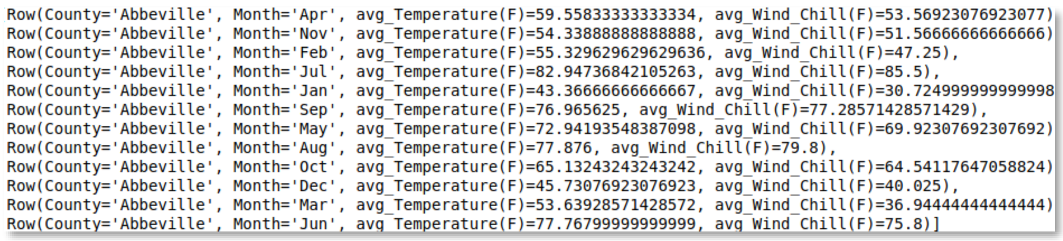
  As a result, each null value will get an average number based on the belonging county and the month of the year, at the same time.

.

- Computes the averages inserting them in an ad-hoc table using Spark SQL

```
table_averages = df_weather_fillprec\
.select('County', 'Month', 'Temperature(F)', 'Wind_Chill(F)',
'Humidity(%)',\
        'Pressure(in)', 'Visibility(mi)', 'Wind_Speed(mph)')\
.groupby('County', 'Month')\
.agg(avg(F.col('Temperature(F)')).alias('avg_Temperature(F)'),\
     avg(F.col('Wind_Chill(F)')).alias('avg_Wind_Chill(F)'),\
     avg(F.col('Humidity(%)')).alias('avg_Humidity(%)'),\
     avg(F.col('Pressure(in)')).alias('avg_Pressure(in)'),\
     avg(F.col('Visibility(mi)')).alias('avg_Visibility(mi)'),\
     avg(F.col('Wind_Speed(mph)')).alias('avg_Wind_Speed(mph)'))\
.orderBy('County')
```

Figure 9 presents a piece of the target table, which shows for the county "Abbeville" the averages of temperature and wind heat for the 12 months of the year.



```
Row(County='Abbeville', Month='Apr', avg_Temperature(F)=59.55833333333334, avg_Wind_Chill(F)=53.56923076923077)
Row(County='Abbeville', Month='Nov', avg_Temperature(F)=54.33888888888888, avg_Wind_Chill(F)=51.56666666666666)
Row(County='Abbeville', Month='Feb', avg_Temperature(F)=55.329629629629636, avg_Wind_Chill(F)=47.25),
Row(County='Abbeville', Month='Jul', avg_Temperature(F)=82.94736842105263, avg_Wind_Chill(F)=85.5),
Row(County='Abbeville', Month='Jan', avg_Temperature(F)=43.36666666666667, avg_Wind_Chill(F)=30.724999999999998
Row(County='Abbeville', Month='Sep', avg_Temperature(F)=76.965625, avg_Wind_Chill(F)=77.28571428571429),
Row(County='Abbeville', Month='May', avg_Temperature(F)=72.94193548387098, avg_Wind_Chill(F)=69.92307692307692)
Row(County='Abbeville', Month='Aug', avg_Temperature(F)=77.876, avg_Wind_Chill(F)=79.8),
Row(County='Abbeville', Month='Oct', avg_Temperature(F)=65.13243243243242, avg_Wind_Chill(F)=64.54117647058824)
Row(County='Abbeville', Month='Dec', avg_Temperature(F)=45.73076923076923, avg_Wind_Chill(F)=40.025),
Row(County='Abbeville', Month='Mar', avg_Temperature(F)=53.63928571428572, avg_Wind_Chill(F)=36.94444444444444)
Row(County='Abbeville', Month='Jun', avg_Temperature(F)=77.76799999999999, avg_Wind_Chill(F)=75.8)]
```

Figure 9 – Example of parameters got by averaging the values of "Abbeville"

Followed by the substitution of the null values with the new ones got from the generated table, using a conditional join function.

- Sets the average values in the target dataframe

```
df_avg_filled = df_weather_joined.withColumn('Temperature(F)',
F.when(F.isnull(df_weather_joined['Temperature(F)']),
F.col('avg_Temperature(F)')).otherwise(df_weather_joined['Temperature(F)']))

df_avg_filled = df_avg_filled.withColumn('Wind_Chill(F)',
F.when(F.isnull(df_avg_filled['Wind_Chill(F)']),
F.col('avg_Wind_Chill(F)')).otherwise(df_avg_filled['Wind_Chill(F)']))

df_avg_filled = df_avg_filled.withColumn('Humidity(%)',
F.when(F.isnull(df_avg_filled['Humidity(%)']),
F.col('avg_Humidity(%)')).otherwise(df_avg_filled['Humidity(%)']))

df_avg_filled = df_avg_filled.withColumn('Pressure(in)',
F.when(F.isnull(df_avg_filled['Pressure(in)']),
F.col('avg_Pressure(in)')).otherwise(df_avg_filled['Pressure(in)']))

df_avg_filled = df_avg_filled.withColumn('Visibility(mi)',
F.when(F.isnull(df_avg_filled['Visibility(mi)']),
F.col('avg_Visibility(mi)')).otherwise(df_avg_filled['Visibility(mi)']))

df_avg_filled = df_avg_filled.withColumn('Wind_Speed(mph)',
F.when(F.isnull(df_avg_filled['Wind_Speed(mph)']),
F.col('avg_Wind_Speed(mph)')).otherwise(df_avg_filled['Wind_Speed(mph)']))
```

This improvement regarding the amount of null values is better displayed with a barplot (*Figure 10*) using logarithmic scale:

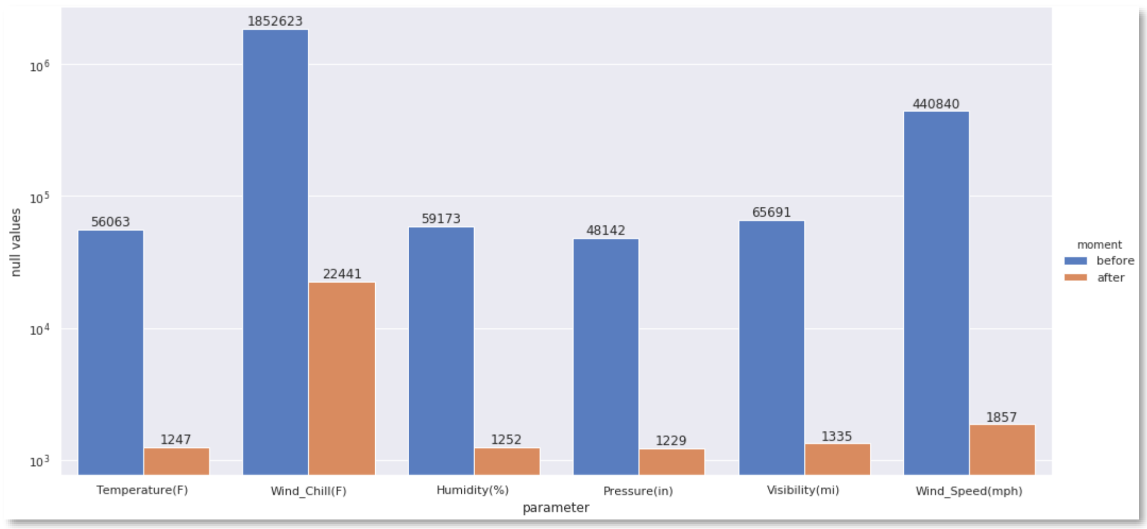
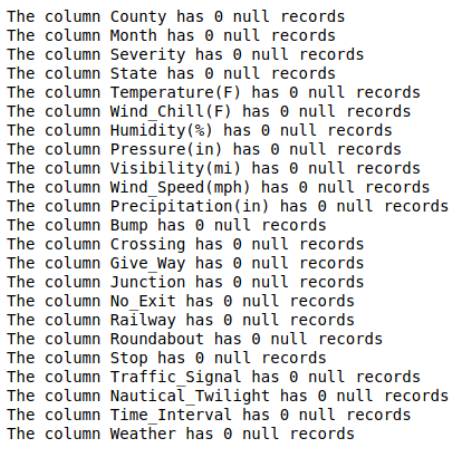


Figure 10 – Comparison between null values before and after the averaging treatment

The rows affected by null values of these parameters are counted as 19227 of the total (one row can have more than one parameter set as null): the 0.69% circa; therefore, to prepare data for the next step, it is felt as not a big loss deleting them entirely from the dataset, obtaining a final dataset free of null values (*Figure 11*).



```
The column County has 0 null records
The column Month has 0 null records
The column Severity has 0 null records
The column State has 0 null records
The column Temperature(F) has 0 null records
The column Wind_Chill(F) has 0 null records
The column Humidity(%) has 0 null records
The column Pressure(in) has 0 null records
The column Visibility(mi) has 0 null records
The column Wind_Speed(mph) has 0 null records
The column Precipitation(in) has 0 null records
The column Bump has 0 null records
The column Crossing has 0 null records
The column Give_Way has 0 null records
The column Junction has 0 null records
The column No_Exit has 0 null records
The column Railway has 0 null records
The column Roundabout has 0 null records
The column Stop has 0 null records
The column Traffic_Signal has 0 null records
The column Nautical_Twilight has 0 null records
The column Time_Interval has 0 null records
The column Weather has 0 null records
```

Figure 11 – Verification of the null free values in the dataset

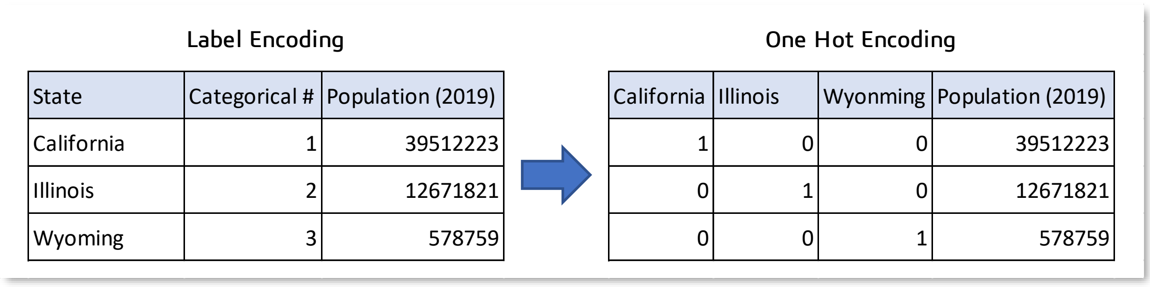
Dataframe preparation for training

Indexing and One Hot Encoding

To give a proper dataset to the train algorithms, it’s necessary to pass from categorical variables (essentially the one described by strings, like “State”), to a numerical one.

Relying only on Indexing (known as Label Encoding), gives a unique id to each of the values of the same class (e.g. Apple = 1, Banana = 2 etc.) but results in many cases not the ideal practice since the training algorithm could get the increasing value as an “increase of importance”, which is a misleading approach for those strings not having a pure ordinal meaning (an ordinal situation may be the instruction level: middle_school = 1, high_school = 2 etc.).

One Hot Encoding [23] transforms the Label Encoded column (or feature) in a series of binary 0 – 1 column for each class (*Figure 12*), as shown in the following example made with three values (California, Illinois and Wyoming) of the class “State”



Label Encoding

State	Categorical #	Population (2019)
California	1	39512223
Illinois	2	12671821
Wyoming	3	578759

One Hot Encoding

California	Illinois	Wyonming	Population (2019)
1	0	0	39512223
0	1	0	12671821
0	0	1	578759

Figure 12 – Conversion of categorical variables in indexed and then one hot encoded

- Converts categorical variables of the dataset into indexed labels using StringIndexer Spark function

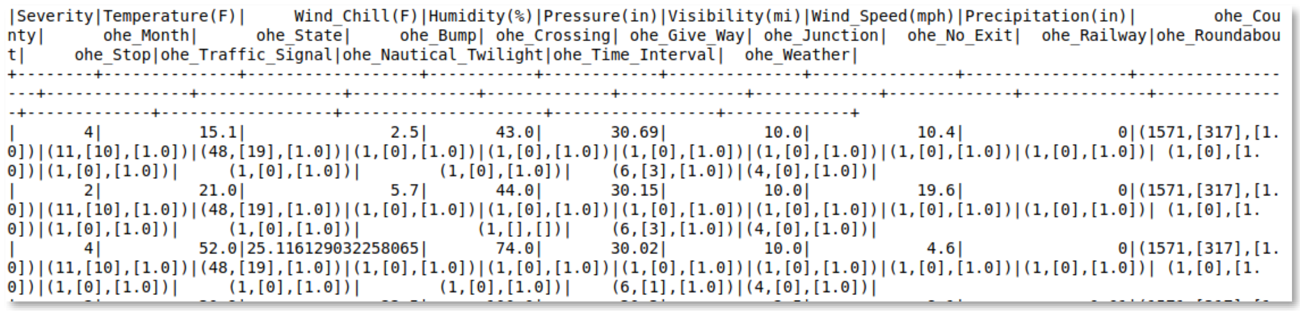
```
for column_to_index in columns_to_index:
    indexer = StringIndexer(inputCol=column_to_index,
    outputCol='indexed_' + column_to_index)
    indexer_model = indexer.fit(df_indexed)
    df_indexed = indexer_model.transform(df_indexed)
```

- Excecutes One Hot Encoding on the indexed feature

```
for column_to_index in columns_to_index:
    ohe = OneHotEncoder(inputCols=["indexed_" + column_to_index],
    outputCols=["ohe_" + column_to_index])
    ohe_model = ohe.fit(df_ohe)
    df_ohe = ohe_model.transform(df_ohe)
```

Figure 13 depicts a compact way to memorize One Hot Encoded features: sparse vector.

In the vector “(1571, [317], [1.0])”, *1571* counts the total number of distinct element in that class (there are 1572 different counties – 1571 + 1), 317 is the index of the county of the list, and *1.0* is the value given to that *317* object – the rest are zero due to OHE mechanism.



```
|Severity|Temperature(F)|     Wind_Chill(F)|Humidity(%)|Pressure(in)|Visibility(mi)|Wind_Speed(mph)|Precipitation(in)|        ohe_Cou
nty|      ohe_Month|       ohe_State|      ohe_Bump| ohe_Crossing| ohe_Give_Way| ohe_Junction|  ohe_No_Exit|  ohe_Railway|ohe_Roundabou
t|     ohe_Stop|ohe_Traffic_Signal|ohe_Nautical_Twilight|ohe_Time_Interval|  ohe_Weather|
+--------+--------------+------------------+-----------+------------+--------------+---------------+-----------------+---------------
---+--------------+--------------+------------+------------+------------+------------+------------+------------+------------+------------
-+------------+-----------------+--------------------+----------------+------------+
|       4|          15.1|               2.5|       43.0|       30.69|          10.0|           10.4|                0|(1571,[317],[1.
0])|(11,[10],[1.0])|(48,[19],[1.0])|(1,[0],[1.0])|(1,[0],[1.0])|(1,[0],[1.0])|(1,[0],[1.0])|(1,[0],[1.0])|(1,[0],[1.0])| (1,[0],[1.
0])|(1,[0],[1.0])|     (1,[0],[1.0])|        (1,[0],[1.0])|    (6,[3],[1.0])|(4,[0],[1.0])|
|       2|          21.0|               5.7|       44.0|       30.15|          10.0|           19.6|                0|(1571,[317],[1.
0])|(11,[10],[1.0])|(48,[19],[1.0])|(1,[0],[1.0])|(1,[0],[1.0])|(1,[0],[1.0])|(1,[0],[1.0])|(1,[0],[1.0])|(1,[0],[1.0])| (1,[0],[1.
0])|(1,[0],[1.0])|     (1,[0],[1.0])|            (1,[],[])|    (6,[3],[1.0])|(4,[0],[1.0])|
|       4|          52.0|25.116129032258065|       74.0|       30.02|          10.0|            4.6|                0|(1571,[317],[1.
0])|(11,[10],[1.0])|(48,[19],[1.0])|(1,[0],[1.0])|(1,[0],[1.0])|(1,[0],[1.0])|(1,[0],[1.0])|(1,[0],[1.0])|(1,[0],[1.0])| (1,[0],[1.
0])|(1,[0],[1.0])|     (1,[0],[1.0])|        (1,[0],[1.0])|    (6,[1],[1.0])|(4,[0],[1.0])|
```

Figure 13 – Some lines of the result of one hot encoding

Splitting of the dataset

Before turning to the next step, the “Severity” feature (the value to predict) has been renamed to “label” due to an unexpected error thrown by some training algorithm that had problem to recognise the parameter “labelCol”, indeed, “label” is the default name assigned to “labelCol”.

- Renames the the “Severity” feature into “label”

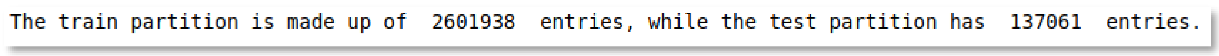
```
df_ohe_dropped = df_ohe_dropped.withColumnRenamed("Severity", "label")
```

The splitting ratio chosen is 0.95-05: more than 100 000 rows will be reserved for testing, while the remaining rows will be used for the training.

- Splits the dataset in train and test

```
df_train, df_test = df_ohe_dropped.randomSplit([0.95, 0.05], seed=12345)
```

Figure 14 shows result of the operation as a split between 2 601 938 and 137 061 rows.



```
The train partition is made up of  2601938  entries, while the test partition has  137061  entries.
```

Figure 14 – Number of observations for each block

Pipelining and training

Pipeline [24] is an API implemented in Spark useful to concatenate different data transformation and machine learning functions in a stage-after-stage workflow fashion, as depicted in *Figure 15*.

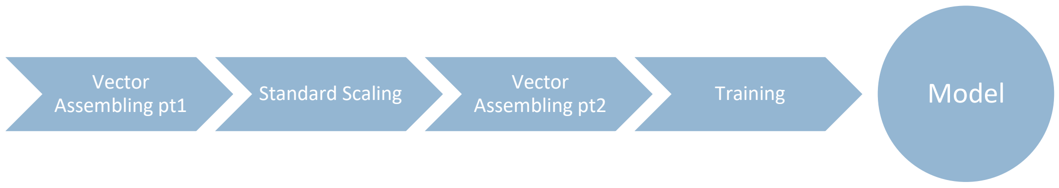


Figure 15 – Pipeline adopted pointing toward the model (the goal)

Conventionally every step of the pipeline is

Vector assembling pt1

VectorAssembler is a function that gets multiple features and aggregates and transforms them in order to create a sparse vector, just like the one saw with the One Hot Encoding output.

Using sparse vector is fundamental since is the kind of input many machine learning training algorithms accept as valid.

In this step, all the features involving atmospheric conditions (decimal numbers) will be aggregated and transformed to make possible the normalization process in the next stage.

- Assembles a sparse vector of atmospheric parameters

```
stage_1 = VectorAssembler(inputCols=columns_parameters,
outputCol='vector_parameters')
```

Standard Scaling

The standardization procedure [25] is used to process the variables of a distribution in the way that these can assume a normal gaussian distribution (mean 0 and standard deviation 0) and it's calculated with the formula in *Figure 16.*

$$x' = \frac{x - \bar{x}}{\sigma}$$

Figure 16 – Formula for standardization

Where $\bar{x}$ is the average, and σ is the standard distribution.

Standardizing the parameters will minimize their difference in terms of magnitude.

For example, in this dataset, it's important to avoid that the value of Humidity 100% can be determined as much more important than Visibility of 7 miles by the machine learning algorithm just because of the different natural dimension (range and absolute min-max values) of the numbers for that particular parameter.

- Standard scaling of atmospheric parameters

```
stage_2 = StandardScaler(withMean=True, withStd=True, inputCol='vector_parameters',
outputCol='scaled_parameters')
```

Vector assembling pt2

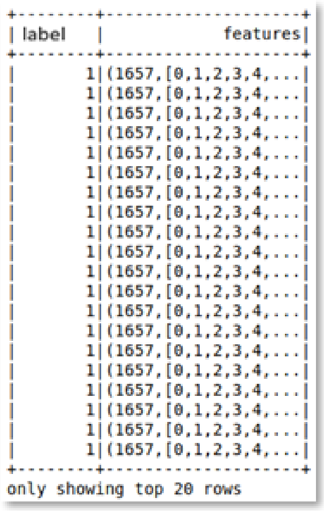
This stage simply involves assembly the vector made with the scaled atmospheric parameters with the columns obtained with One Hot Encoding in order to have the final vector to give to the train algorithms, named “features”.

- Aggregates al the columns and executes the assembly

```
columns_to_assemble = ['scaled_parameters'] + ["ohe_" + columns for columns in
columns_to_index]
stage_3 = VectorAssembler(inputCols=columns_to_assemble, outputCol='features',
handleInvalid='skip')
```

Training

The goal of the training is to develop a model that can predict the level of “Severity” given the of atmospheric situation, the road conditions and the geographical position (State, County etc) all incapsulated into the “features” vectorized column, as reported in *Figure 17*.



```
+--------+--------------------+
|label   |            features|
+--------+--------------------+
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
|       1|(1657,[0,1,2,3,4,...|
+--------+--------------------+
only showing top 20 rows
```

Figure 17 – A sample from the transformed “df_train” dataframe

The “Severity” column contains integer numbers from 1 to 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).

The value of severity reflects a precise category, hence, the problem involves classification algorithms.

The algorithms chosen from the Spark library are Decision Tree, Random Forest and One-vs-Rest.

For each training session, a parameter grid has been used: it’s a faster way to try different parameters without a manual re-training intervention; plus, the output model is the one with the best parameters combination.

Training with Decision Tree

Decision Tree [26] is an algorithm based on a tree-like structure in which each node has two or more branches that represent the result of the assessment done in their parent node, as depicted in the example in *Figure 18*.

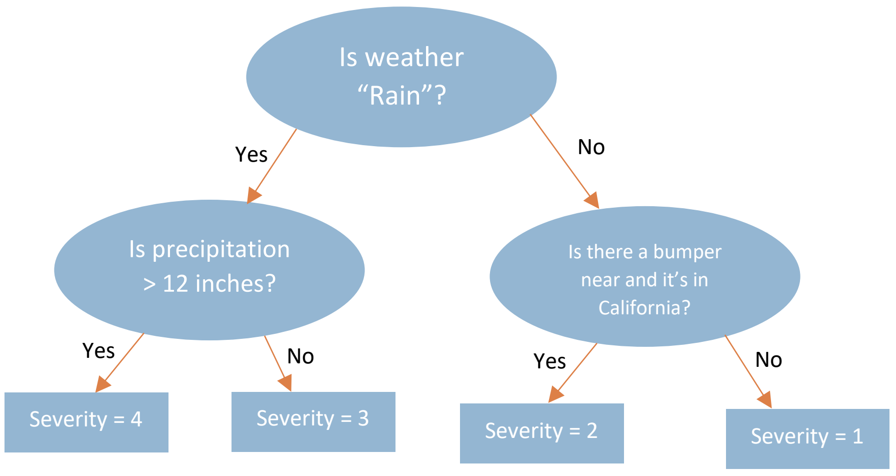


Figure 18 – Example of Decision Tree behaviour

- Instances a Decision Tree Classifier
- Creates a parameter grid with maxDepth (the number of "leaves" children of the tree) set to 3 and 6
- Instances an evaluator to measure the accuracy
- Creates a cross validator that runs the pipeline with two folds
- Starts the training with Decision Tree algorithm

```
stage_4 = DecisionTreeClassifier()
paramGrid = ParamGridBuilder().addGrid(stage_4.maxDepth, [3, 6]).build()
pipeline_dataPrep = Pipeline(stages=[stage_1, stage_2, stage_3, stage_4])
evaluator_multiClass = MulticlassClassificationEvaluator(metricName="accuracy")
crossval = CrossValidator(estimator=pipeline_dataPrep, estimatorParamMaps=paramGrid,
evaluator = evaluator_multiClass, numFolds=2)
cvModel_DTree = crossval.fit(df_train)
```

Training with Random Forest

Random Forest algorithm [27] is an ensemble of Decision Trees: the value of each Decision Tree result would be averaged to get the final value *Figure 19*.

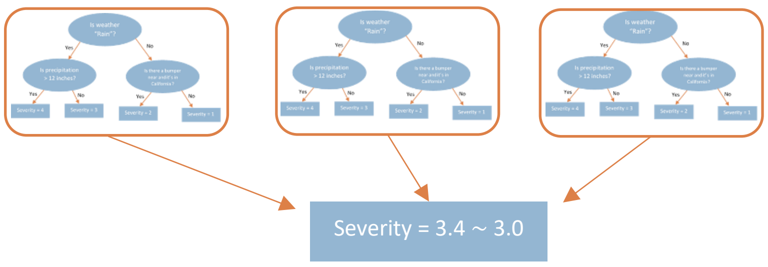


Figure 19 – Example of Random Forest behaviour with three trees

- Instances a Random Forest Classifier
- Creates a parameter grid with numberTrees (the number of trees to use) set to 2 and 5 and maxDepth to 5 and 10.
- Instances an evaluator to measure the accuracy
- Creates a cross validator that runs the pipeline with two folds
- Starts the training with Random Forest algorithm

```
stage_4 = RandomForestRegressor(labelCol="label", featuresCol="features")
pipeline_dataPrep = Pipeline(stages=[stage_1, stage_2, stage_3, stage_4])
paramGrid = ParamGridBuilder().addGrid(stage_4.numTrees, [5,
10]).addGrid(stage_4.maxDepth, [5, 10]).build()
evaluator_multiClass = MulticlassClassificationEvaluator(metricName="accuracy")
crossval = CrossValidator(estimator=pipeline_dataPrep, estimatorParamMaps=paramGrid,
evaluator = evaluator_multiClass, numFolds=2)
cvModel_RNFS = crossval.fit(df_train)
```

Training with One-vs-Rest

One-vs-Rest [28] is a technique implemented in Spark MLlib for classification problems that exploit algorithms which are typically used for binary classification (the result can be A or B only). In practice, it keeps one of the values to be classified as A while assuming all the others as a unique B class; in an iterative fashion.

In this case, the logistic regression classifier has been used as basis.

- Instances a Logistic Regression classifier
- Creates a One-vs-Rest instance
- Creates a parameter grid with maxIter (number of maximum likelihood iterations to find the curve of the logistic regression) set to 2 and 5 and fitIntercept to True and False.
- Instances an evaluator to measure the accuracy
- Creates a cross validator that runs the pipeline with two folds
- Starts the training with One-vs-Rest algorithm

```
lr = LogisticRegression()
stage_4 = OneVsRest(classifier=lr)
pipeline_dataPrep = Pipeline(stages=[stage_1, stage_2, stage_3, stage_4])
paramGrid = ParamGridBuilder().addGrid(lr.maxIter, [5, 10]).addGrid(lr.fitIntercept,
[True, False]).build()
evaluator_multiClass = MulticlassClassificationEvaluator(metricName="accuracy")
crossval = CrossValidator(estimator=pipeline_dataPrep, estimatorParamMaps=paramGrid,
evaluator = evaluator_multiClass, numFolds=2)
cvModel_1vsRst_lr = crossval.fit(df_train)
```

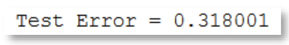
Evaluation of the results

Model error

The models developed have been evaluated using the “df_test” dataframe

Decision Tree

```
df_result_DTree = cvModel_DTree.transform(df_test).select("label", "prediction")
df_result_DTree = df_result_DTree.withColumn("prediction",
F.round(df_result_DTree["prediction"]))
evaluator = MulticlassClassificationEvaluator(labelCol="label",
predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(df_result_DTree)
print("Test Error = %g " % (1.0 - accuracy))
```

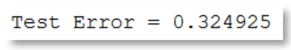


```
Test Error = 0.318001
```

Figure 19 – Error with Decision Tree

Random forest

```
df_result_RNFS = cvModel_RNFS.transform(df_test)
df_result_rndFst = df_result_rndFst.withColumn("prediction",
F.round(df_result_rndFst["prediction"]))
evaluator = MulticlassClassificationEvaluator(labelCol="label",
predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(df_result_rndFst)
print("Test Error = %g" % (1.0 - accuracy))
```

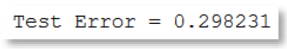


```
Test Error = 0.324925
```

Figure 20 – Error with Random Forest

One-vs-Rest

```
df_result_1vsRst_lr = cvModel_1vsRst_lr.transform(df_test)
df_result_1vsRst_lr = df_result_1vsRst_lr.withColumn("prediction",
F.round(df_result_1vsRst_lr["prediction"]))
evaluator = MulticlassClassificationEvaluator(labelCol="label",
predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Test Error = %g" % (1.0 - accuracy))
```



```
Test Error = 0.298231
```

Figure 21 – Error with One-vs-Rest and Logistic Regression

Performances and costs

Oddly, Decision Tree algorithm with its conservative parameters performed slightly better than Random Forest with a 31.80 % error against 32.49 % circa (*Figures 19* and *20*) while the best result is given by the One-vs-Rest technique with an error score of 29.82 % circa.

Another point of view to score the best algorithm amid the ones proposed is taking into account the time spent to run the training session: a fast and naïve way to measure the performance of the algorithm is dividing the accuracy by the time taken to train the dataset, as shown in *Table 6* under the column “Performance Index”.

All the training sessions have been made through Microsoft Azure HDInsight cloud service with two head nodes and one worker node, each one equipped with 4 vCPU and 28 GB of RAM (named D12 v2 instances) and supported be Apache Spark 2.4.5 with Hadoop + Yarn 3.1.1.

	Seconds	Error	Accuracy	Performance Index
Decision Tree	4986	0.318001	0.681999	0.000136783
Random Forest	3703	0.324925	0.675075	0.000182305
One-vs-Rest	2909	0.298231	0.701769	0.000241241

Table 6 – Performance results for each algorithm

To explicitly compare the algorithms each other, it's possible to consider the one with the best performance index (One-vs-Rest) as the reference for a normalized comparison (*Figure 22*).

$$Normalized\ Performance\ Index = \frac{Performance\ Index_{algorithm}}{Performance\ Index_{One-vs-Rest}} * 100$$

Figure 22 – Normalization of Perf. Index with One-vs-Rest basis

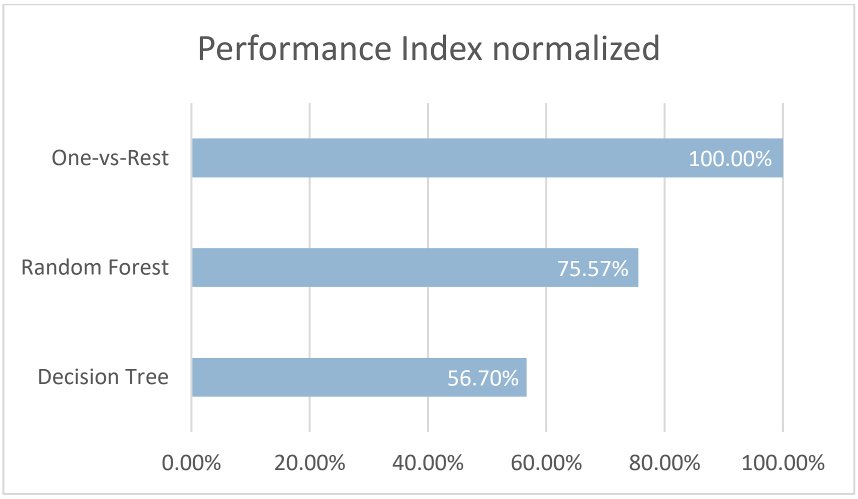


Figure 23 – Relative performance to One-vs-Rest

Figure 23 shows that Random Forest is almost 25% less performing than One-vs-Rest followed by Decision Tree 44% circa less performing.

For what concerns costs, at April 2020, Azure requires $ 0.374/hour for each D12 v2 instance: taking into account the head and the two workers, the total rises to $ 1.222/hour.

In the *Figure 24* is displayed the cost for each training session, excluding the other costs (storage and general setup).

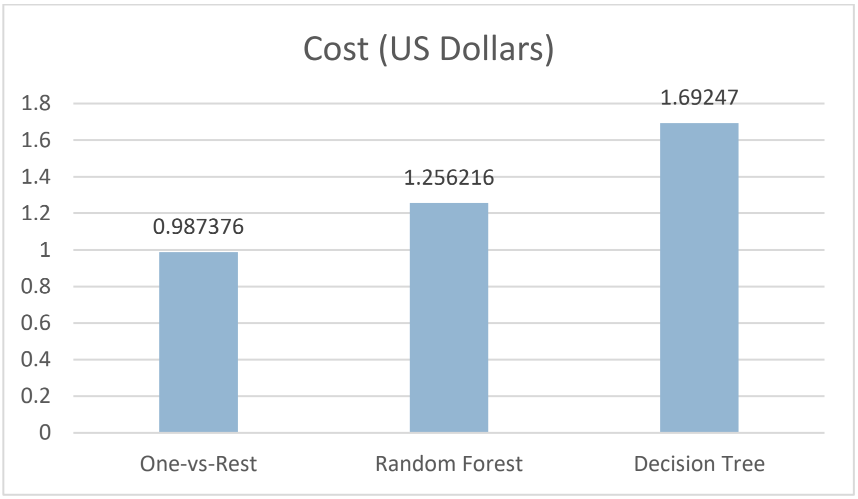


Figure 23 – Relative performance to One-vs-Rest

Conclusions

As seen in *Figure 22*, the best algorithm among the three tried, turned out to be the One-vs-Rest incapsulating Logistic Regression; at the same time *Figure 23* demonstrates that it is also the cheapest in the training session.

Unfortunately, an error overpassing 29% is not acceptable if applied as a single technology to an automatic decision system onboard of a car, due to high risk of impacting on passenger, pedestrian and other driver; but it could be used in a passive way by taking its result as a suggestion to the driver, or in conjunction to the rest of the sensors and processing systems to evaluate if the risk of increasing/decreasing the average speed is manageable.

Limitations

The correlation between accident severity and danger level of a road is a pure assumption since the occurring of a crash can be determined by several other factor such as the absolute and relative speed of the vehicles and the alteration status of the driver(s); nevertheless, it is reasonable to premise that the probability of road fatalities follows the condition of the road trait itself (behind-the-corner-visibility, crossroads, old/worn stradal signs etc).

The presupposition of assigning “null” weather to the “No_precipitation” bin, has been decided due to the fact that a weather can’t physically be “null”, although it is possible that for laziness or for lack of the proper field in the accident form to be compiled, the person must have avoided it; or for a meteorological station no being able to exploit this data.

Margins of improvement

The big issue regarding the possible steps in the enhancement of the work is having access to the instantaneous position of the vehicle, indeed, many other extended functionalities can be used to improve the accuracy of the prediction such as:

- Wind direction and speed: as explained in *Table 1*, the wind direction and speed have been dropped from the dataset since there is no information about which direction the vehicle was facing at, therefore, maintaining these elements alone would be a biasing element for the analysis.
  Knowing the wind direction and speed related to the direction of the vehicle can be an interesting clue: [29] and [30] demonstrate that wind speed starts to affect cars lateral stability starting from 24 m/s with a moderate and steady driving style.
- Changing the paradigm of geo-spatial analysis switching from county as smallest area block to buffers centred on the car accident, clustering the area and assigning to the surroundings a proper number of risk index; so the vehicle coming into this area can have a more precise warning.

  Obviously, the problem of having access to real time position of each car involves dealing with privacy limitations (different for each country) and with the huge traffic generate by moving vehicles as well as the robust infrastructure behind it.

A further future-proof workflow is gathering more road characteristics just using the integrated camera placed behind the windshield of the vehicle, reading the scene and labelling the objects it sees: this way it will possible to intersect, for example, this data with the description of the accident in that area (reported in the column “Description” of the dataset).

In order to extend the experimentation, pressing on the One-vs-Rest algorithm using different classification techniques (due to the good performance reached in comparison to the other two) and parametrizing them with a higher variety of hyperparameters can be useful.

Bibliography

[1] "Tesla App Support," *Tesla, Inc*, 23-Mar-2020. [Online]. Available: https://www.tesla.com/support/tesla-app?redirect=no. [Accessed: 21-Apr-2020]

[2] "InControl: Connectivity: Jaguar USA," *InControl | Connectivity | Jaguar USA*. [Online]. Available: https://www.jaguarusa.com/incontrol/incontrol/connectivity/index.html. [Accessed: 21-Apr-2020]

[3] "Vehicle-to-everything," *Wikipedia*, 31-Mar-2020. [Online]. Available: https://en.wikipedia.org/wiki/Vehicle-to-everything. [Accessed: 21-Apr-2020]

[4] M. Allan, "2020 Seat Leon - all-new hatchback is a high-tech hybrid," *Edinburgh News*, 30-Jan-2020. [Online]. Available: https://www.edinburghnews.scotsman.com/inews-lifestyle/cars/2020-seat-leon-all-new-hatchback-is-a-high-tech-hybrid-1380718. [Accessed: 21-Apr-2020]

[5] "Here Technologies," *Wikipedia*, 03-Apr-2020. [Online]. Available: https://en.wikipedia.org/wiki/Here_Technologies. [Accessed: 21-Apr-2020]

[6] "Toyota uses big data to guard against accelerator-brake mix-up," *Reuters*, 03-Feb-2020. [Online]. Available: https://www.reuters.com/article/us-toyota-adas/toyota-uses-big-data-to-guard-against-accelerator-brake-mix-up-idUSKBN1ZX0FR. [Accessed: 21-Apr-2020]

[7] "Kia and Hyundai develop ICT Connected Shift System," *Autonomous Vehicle International*, 24-Feb-2020. [Online]. Available: https://www.autonomousvehicleinternational.com/news/software/kia-and-hyundai-develop-ict-connected-shift-system.html. [Accessed: 21-Apr-2020]

[8] "Road Traffic Injuries & Deaths: A Global Problem," *Centers for Disease Control and Prevention*, 18-Dec-2019. [Online]. Available: https://www.cdc.gov/injury/features/global-road-safety/index.html. [Accessed: 21-Apr-2020]

[9] "Global status report on road safety 2018," *World Health Organization*. [Online]. Available: https://www.who.int/violence_injury_prevention/road_safety_status/2018/en. [Accessed: 21-Apr-2020]

[10] "Road Safety Facts," *Association for Safe International Road Travel.* [Online]. Available: http://www.asirt.org/safe-travel/road-safety-facts/. [Accessed: 21-Apr-2020]

[11] Lankarani, Kamran & Heydari, Seyed & Aghabeigi, Mohammad & Moafian, Ghasem & Hoseinzadeh, Amin & Vossoughi, Mehrdad. (2013). The impact of environmental factors on traffic accidents in Iran. Journal of injury & violence research. 6. 10.5249/jivr.v6i2.318.

[12] S. Moosavi, "US Accidents (3.0 million records)," Kaggle, 17-Jan-2020. [Online]. Available: http://www.kaggle.com/sobhanmoosavi/us-accidents. [Accessed: 21-Apr-2020]

[13] "pyspark.sql module — PySpark 2.4.5 documentation", *Spark.apache.org*, 2020. [Online]. Available: https://spark.apache.org/docs/2.4.5/api/python/pyspark.sql.html#pyspark.sql.DataFrameReader. [Accessed: 25- Apr-2020]

[14] "TMC/Event Code List - OpenStreetMap Wiki", *Wiki.openstreetmap.org*, 2020. [Online]. Available: https://wiki.openstreetmap.org/wiki/TMC/Event_Code_List. [Accessed: 25- Apr- 2020]

[15] "Twilight", *En.wikipedia.org*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Twilight#Nautical_dawn_and_dusk. [Accessed: 25- Apr- 2020]

[16] "Alternative work schedules", *U.S. Department of Commerce*, 2020. [Online]. Available: https://www.commerce.gov/hr/employees/work-life-balance/aws. [Accessed: 25- Apr- 2020]

[17] "Missing Values for Categorical Variables?", *ResearchGate*, 2020. [Online]. Available: https://www.researchgate.net/post/Missing_Values_for_Categorical_Variables. [Accessed: 25- Apr- 2020]

[18] "timezonefinder", *PyPI*, 2020. [Online]. Available: https://pypi.org/project/timezonefinder/. [Accessed: 25- Apr- 2020]

[19] "Astropy", *Astropy.org*, 2020. [Online]. Available: https://www.astropy.org/. [Accessed: 25- Apr- 2020]

[20] "Coordinated Universal Time", *En.wikipedia.org*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Coordinated_Universal_Time. [Accessed: 25- Apr- 2020]

[21] "List of tz database time zones", *En.wikipedia.org*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/List_of_tz_database_time_zones. [Accessed: 25- Apr- 2020]

[22] "beautifulsoup4", *PyPI*, 2020. [Online]. Available: https://pypi.org/project/beautifulsoup4/. [Accessed: 25- Apr- 2020]

[23] "Categorical encoding using Label-Encoding and One-Hot-Encoder", *Medium*, 2020. [Online]. Available: https://towardsdatascience.com/categorical-encoding-using-label-encoding-and-one-hot-encoder-911ef77fb5bd. [Accessed: 25- Apr- 2020]

[24] "ML Pipelines - Spark 2.4.5 Documentation", *Spark.apache.org*, 2020. [Online]. Available: https://spark.apache.org/docs/latest/ml-pipeline.html. [Accessed: 25- Apr- 2020]

[25] "Standard score", *En.wikipedia.org*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Standard_score. [Accessed: 25- Apr- 2020]

[26] "Classification and regression - Spark 2.4.5 Documentation", *Spark.apache.org*, 2020. [Online]. Available: https://spark.apache.org/docs/latest/ml-classification-regression.html#decision-trees. [Accessed: 25- Apr- 2020]

[27] "Classification and regression - Spark 2.4.5 Documentation", *Spark.apache.org*, 2020. [Online]. Available: https://spark.apache.org/docs/latest/ml-classification-regression.html#random-forests. [Accessed: 25- Apr- 2020]

[28] "Classification and regression - Spark 2.4.5 Documentation", *Spark.apache.org*, 2020. [Online]. Available: https://spark.apache.org/docs/latest/ml-classification-regression.html#one-vs-rest-classifier-aka-one-vs-all. [Accessed: 25- Apr- 2020]

[29] S. Glaser, S. Mammar and D. Dakhlallah, "Lateral wind force and torque estimation for a driving assistance", 2020.

[30] D. Liu, Z. Shi and W. Ai, "An Improved Car-Following Model Accounting for Impact of Strong Wind", 2020.