Data Science Cornerstone Report

by Jörg Bergmann

as of 15th October 2020, Darmstadt, Germany

Published on website:

www.herbfrisch.de

Git Hub link:

https://github.com/herbfrisch/jbs_cornerstone

1 Introduction Section

In this section we discuss the business problem and who would be interested in this project.

1.1.Introduction/Business Problem Car Accident Severity Prediction

1.1.1.Introduction/Background
This section defines the business problem wth risks in the mobility area. It is about to solve the decision problem, to drive or not to drive with a car under certain known conditions from the current location to a destination at a planned time in realation to the risk for having an accident.

1.1.2.Problem
In the following, this report is to predict the severity of an accident. The scenario could be described as follows: Say, you are driving to another city for work or to visit some friends. It is rainy and windy, and on the way, you come across a terrible traffic jam on the other side of the highway. Long lines of cars barely moving. As you keep driving, police car start appearing from afar shutting down the highway. Oh, it is an accident and there's a helicopter transporting the ones involved in the crash to the nearest hospital. They must be in critical condition for all of this to be happening. Now, wouldn't it be great if there is something in place that could warn you, given the weather and the road conditions about the possibility of you getting into a car accident and how severe it would be, so that you would drive more carefully or even change your travel if you are able to?

1.1.3.Interests
In the following this is handled as a data science problems that targets the car

drivers audience in the first place, but is also meant to help all involved stakeholders to mitigate risk in the mobility, insurance, healthcare area, and at least for the family of the driver.
This is exactly what will be handled in this report: to predict the severity of a possible car accident on the base of available car accident data from the past and current driving conditions.

1.2.Introduction to the Car Accident Severity Data

1.2. Data acquisition and cleaning

Data
In this section the data that will be used to solve the problem is described. It contains the explanation, why the is date is adequate for the problem and is used. In the discussion part, examples of data is provided.

1.2.1 Data sources

To deal with accidents data the shared data for Seattle city is used as an example:
https://s3.us.cloud-object-storage.appdomain.cloud/cf-courses-data/CognitiveClass/DP0701EN/version-2/Data-Collisions.csv
Metadata descriptions:
https://s3.us.cloud-object-storage.appdomain.cloud/cf-courses-data/CognitiveClass/DP0701EN/version-2/Metadata.pdf

1.2.2 Data cleaning

After open the CSV file and check what type of data is contained. The first column colored in
yellow is the labeled data. The remaining columns have different types of attributes. A selection of these attributes are used to train the model. Also most of the observations found are of sufficent quality to train and test the machine learning model to be build in this case.
The label for the data set used is severity, which describes the
fatality of an accident. It shall be notice that the shared data has unbalanced labels. So the data is balanced, otherwise, a biased ML model would be created.

1.2.3 Feature selection

The following is a list of attributes or features that are used. For description of each attribute, it is refered to the web link on the CSV file.
SEVERITYCODE, STATUS,
ADDRTYPE
- Alley
- Block
- Intersection

INTKEY,

with a collision
LOCATION,
SEVERITYCODE,
● 3—fatality
● 2b—serious injury ● 2—injury
● 1—prop damage
● 0—unknown
PERSONCOUNT,
- Collision address type:
- Key that corresponds to the intersection associated
- Description of the general location of the collision
- A code that corresponds to the severity of the collision:
- The total number of people involved in the collision
PEDCOUNT,
This is entered by the state.
PEDCYLCOUNT, entered by the state.
VEHCOUNT,
entered by the state.
INCDATE, INCDTTM, JUNCTIONTYPE, INATTENTIONIND,
- The number of pedestrians involved in the collision.
- The number of bicycles involved in the collision. This is - The number of vehicles involved in the collision. This is
- The date of the incident.
- The date and time of the incident.
- Category of junction at which collision took place
- Whether or not collision was due to inattention. (Y/N)
UNDERINFL, - Whether or not a driver involved was under the influence of drugs or alcohol.
WEATHER,
of the collision.
ROADCOND, LIGHTCOND,
PEDROWNOTGRNT, granted. (Y/N)
SPEEDING, (Y/N)
ST_COLCODE,
collision. For more information about these codes, please see the State Collision Code Dictionary. Codes: 0-5, 10-32, 40-57, 60-67, 71-74, 81-84.
SEGLANEKEY, occurred.
CROSSWALKKEY,
HITPARKEDCAR car. (Y/N)
- A key for the lane segment in which the collision
- A key for the crosswalk at which the collision occurred. - Whether or not the collision involved hitting a parked
- A description of the weather conditions during the time
- The condition of the road during the collision. - The light conditions during the collision.
- Whether or not the pedestrian right of way was not

- Whether or not speeding was a factor in the collision.
- A code provided by the state that describes the

In addition, there is probably need to do some feature engineering to improve the predictability
of the model as follows:
From the ST_COLCODE, a smaller set of cathegories could be defined. From INCDATE the day of the week could be calculated.
From INCDTTM the part of the day could be calculated: morning, afternoon, evening, night.
The target or label columns should be accident " severity" in terms of human fatality, traffic delay, property damage, or any other type of accident bad impact. These terms are cathegories and are constructed from the last attributes listed above.
Then, the built severity data set is applied building a machine learning model.

2 Data Section

2.1 Data

In this section we describe the data that will be used to solve the problem and the source of the data.

2.1.1 Data Analysis

Basic Insight of Dataset
After reading data into Pandas dataframe, we explore the dataset with:

- df.head(10)
- df.tail(10)
- df.shape : (194673, 38) , which is number of data sets by number of columns.

2.1.2 Data Types

In order to better learn about each attribute, it is always good for us to know the data type of each column.
df.dtypes :

```
SEVERITYCODE       int64
X                float64
Y                float64
OBJECTID           int64
INCKEY             int64
COLDETKEY          int64
REPORTNO          object
```

```
STATUS            object
ADDRTYPE          object
INTKEY           float64
LOCATION          object
EXCEPTRSNCODE     object
EXCEPTRSNDESC     object
SEVERITYCODE.1     int64
SEVERITYDESC      object
COLLISIONTYPE     object
PERSONCOUNT        int64
PEDCOUNT           int64
PEDCYLCOUNT        int64
VEHCOUNT           int64
INCDATE           object
INCDTTM           object
JUNCTIONTYPE      object
SDOT_COLCODE       int64
SDOT_COLDESC      object
INATTENTIONIND    object
UNDERINFL         object
WEATHER           object
ROADCOND          object
LIGHTCOND         object
PEDROWNOTGRNT     object
SDOTCOLNUM       float64
SPEEDING          object
ST_COLCODE        object
ST_COLDESC        object
SEGLANEKEY         int64
CROSSWALKKEY       int64
HITPARKEDCAR      object
```

2.1.3 Describe the data

If we would like to get a statistical summary of each column, such as count, column mean value, column standard deviation, etc. we use the describe method.
This method will provide various summary statistics, including NaN (Not a Number) values.
df.describe(include ="all") :

SEVERITYCODE	X	Y	OBJECTID	INCKEY	COLDETKEY	REPORTNO	STATUS	ADDRTYPE	INTKEY	...	ROADCOND	LIGHTCOND	PEDROWNOTGRNT	SDOTCOLNUM	SPEEDING	ST_COLCODE	ST_COLDESC	SEGLANEKEY	CROSSWALKKEY	HITPARKEDCAR	
count	194673.000000	189339.000000	189339.000000	194673.000000	194673.000000	194673.000000	194673	194673	192747	65070.000000	...	189661	189503	4667	1.149360e+05	9333	194655	189769	194673.000000	1.946730e+05	194673
unique	NaN	NaN	NaN	NaN	NaN	NaN	194670	2	3	NaN	...	9	9	1	NaN	1	115	62	NaN	NaN	2

top	NaN	NaN	NaN	NaN	NaN	NaN	1782439	Matched	Block	NaN	...	Dry	Daylight	Y	NaN	Y	32	Oneparked--onemoving	NaN	NaN	N
freq	NaN	NaN	NaN	NaN	NaN	NaN	2	189786	126926	NaN	...	124510	116137	4667	NaN	9333	27612	44421	NaN	NaN	187457
mean	1.2989901	-122.330518	47.619543	108479.364930	141091.456350	141298.811381	NaN	NaN	NaN	37558.450576	...	NaN	NaN	NaN	7.972521e+06	NaN	NaN	NaN	269.401114	9.782452e+03	NaN

std	0.457778	0.029976	0.056157	62649.722558	86634.402737	86986.542110	NaN	NaN	NaN	51745.990273	...	NaN	NaN	NaN	2.553533e+06	NaN	NaN	NaN	3315.776055	7.226926e+04	NaN
min	1.000000	-122.419091	47.495573	1.000000	1001.000000	1001.000000	NaN	NaN	NaN	23807.000000	...	NaN	NaN	NaN	1.007024e+06	NaN	NaN	NaN	0.000000	0.000000e+00	NaN
25%	1.000000	-122.348673	47.575956	54267.000000	70383.000000	70383.000000	NaN	NaN	NaN	28667.000000	...	NaN	NaN	NaN	6.040015e+06	NaN	NaN	NaN	0.000000	0.000000e+00	NaN

50%	1.000000	-122.330224	47.615369	106912.000000	123363.000000	123363.000000	NaN	NaN	NaN	29973.000000	...	NaN	NaN	NaN	8.023022e+06	NaN	NaN	NaN	0.000000	0.000000e+00	NaN
75%	2.000000	-122.311937	47.663664	162272.000000	203319.000000	203459.000000	NaN	NaN	NaN	33973.000000	...	NaN	NaN	NaN	1.015501e+07	NaN	NaN	NaN	0.000000	0.000000e+00	NaN
max	2.000000	-122.238949	47.734142	219547.000000	331454.000000	332954.000000	NaN	NaN	NaN	757580.000000	...	NaN	NaN	NaN	1.307202e+07	NaN	NaN	NaN	525241.000000	5.239700e+06	NaN

11 rows × 38 columns

2.1.4 Dataset Info

Another method you can use to check your dataset is dataframe.info
It provides a concise summary of your DataFrame.Look at the info of "df".

Here we are able to see the information of our dataframe, with the top 30 rows and the bottom 30 rows.
And, it also shows us the whole data frame has 194673 rows and 38 columns in total.

df.info :

```
<bound method DataFrame.info of        SEVERITYCODE          X          Y
OBJECTID  INCKEY  COLDETKEY  \
0              2 -122.323148  47.703140        1    1307       1307
1              1 -122.347294  47.647172        2   52200      52200
2              1 -122.334540  47.607871        3   26700      26700
3              1 -122.334803  47.604803        4    1144       1144
4              2 -122.306426  47.545739        5   17700      17700
...          ...         ...        ...      ...     ...        ...
194668         2 -122.290826  47.565408   219543  309534     310814
194669         1 -122.344526  47.690924   219544  309085     310365
194670         2 -122.306689  47.683047   219545  311280     312640
194671         2 -122.355317  47.678734   219546  309514     310794
194672         1 -122.289360  47.611017   219547  308220     309500

      REPORTNO   STATUS     ADDRTYPE   INTKEY  ... ROADCOND  \
0      3502005  Matched  Intersection  37475.0  ...      Wet
1      2607959  Matched        Block      NaN  ...      Wet
2      1482393  Matched        Block      NaN  ...      Dry
3      3503937  Matched        Block      NaN  ...      Dry
4      1807429  Matched  Intersection  34387.0  ...      Wet
...        ...      ...          ...      ...  ...      ...
194668  E871089  Matched        Block      NaN  ...      Dry
194669  E876731  Matched        Block      NaN  ...      Wet
194670  3809984  Matched  Intersection  24760.0  ...      Dry
194671  3810083  Matched  Intersection  24349.0  ...      Dry
194672  E868008  Matched        Block      NaN  ...      Wet

                      LIGHTCOND PEDROWNOTGRNT  SDOTCOLNUM SPEEDING
ST_COLCODE  \
0                      Daylight          NaN         NaN      NaN         10
1       Dark - Street Lights On          NaN   6354039.0      NaN         11
2                      Daylight          NaN   4323031.0      NaN         32
3                      Daylight          NaN         NaN      NaN         23
4                      Daylight          NaN   4028032.0      NaN         10
...                         ...          ...         ...      ...        ...
194668                 Daylight          NaN         NaN      NaN         24
194669                 Daylight          NaN         NaN      NaN         13
194670                 Daylight          NaN         NaN      NaN         28
194671                     Dusk          NaN         NaN      NaN          5
```

```
194672              Daylight          NaN         NaN      NaN        14

                                        ST_COLDESC  SEGLANEKEY  \
0                                 Entering at angle           0
1       From same direction - both going straight - bo...           0
2                             One parked--one moving           0
3                      From same direction - all others           0
4                                 Entering at angle           0
...                                                 ...         ...
194668    From opposite direction - both moving - head-on           0
194669  From same direction - both going straight - bo...           0
194670  From opposite direction - one left turn - one ...           0
194671                      Vehicle Strikes Pedalcyclist        4308
194672  From same direction - both going straight - on...           0

       CROSSWALKKEY  HITPARKEDCAR
0                 0            N
1                 0            N
2                 0            N
3                 0            N
4                 0            N
...             ...          ...
194668            0            N
194669            0            N
194670            0            N
194671            0            N
194672            0            N

[194673 rows x 38 columns]>
```

In [135]:

2.2 Data Wrangling

2.2.1 Convert "?" to NaN if any

In the dataset, missing data comes sometimes with the question mark "?".
We replace "?" with NaN (Not a Number), which is Python's default missing value marker,
for reasons of computational speed and convenience. Here we would use the function:
replace "?" to NaN

```
df.replace("?", np.nan, inplace = True)
```

2.2.2 Missing Data
Steps for working with missing data:
1. identify missing data
2. deal with missing data
3. correct data format

2.2.2.1 Identify_missing_values
valuating for Missing Data
The missing values are converted to Python's default. We use Python's built-in functions to identify these missing values. There are two methods to detect missing data:
a. .isnull()
b. .notnull()
The output is a boolean value indicating whether the value that is passed into the argument is in fact missing data.
"True" stands for missing value, while "False" stands for not missing value.

```
missing_data = df.isnull()
missing_data.head(5)
```

Count missing values in each column
Using a for loop in Python, we can quickly figure out the number of missing values in each column.
As mentioned above, "True" represents a missing value, "False" means the value is present in the dataset.
In the body of the for loop the method ".value_counts()" counts the number of "True" values.

```
for column in missing_data.columns.values.tolist():
    print(column)
    print (missing_data[column].value_counts())
    print("")
```

Based on the summary above, each column has 205 rows of data, seven columns containing missing data:
1. "X": 5534 missing data
2. "Y": 5534 missing data
3. "ADDRTYPE": 1926 missing data
4. "INTKEY" : 129603 missing data
5. "LOCATION": 2677 missing data
6. "EXCEPTRSNCODE": 109862 missing data
7. "EXCEPTRSNDESC": 189035 missing data
8. "COLLISIONTYPE": 4904 missing data
9. "JUNCTIONTYPE": 6329 missing data

10. "INATTENTIONIND": 164868 missing data
11. "UNDERINFL": 4884 missing data
12. "WEATHER": 5081 missing data
13. "ROADCOND": 5012 missing data
14. "LIGHTCOND": 5170 missing data
15. "PEDROWNOTGRNT": 190006 missing data
16. "SDOTCOLNUM": 79737 missing data
17. "SPEEDING": 185340 missing data
18. "ST_COLCODE": 18 missing data
19. "ST_COLDESC": 4904 missing data

```
df.dropna(axis=0, inplace=False)
df.head()
```

Select the columns we want to use

```
df_all =
df[['SEVERITYCODE','STATUS','ADDRTYPE','INTKEY','LOCATION','COLLISIONTYPE','SEVERITYCODE.1','PERSONCOUNT','PEDCOUNT','PEDCYLCOUNT','VEHCOUNT', 'INCDATE', 'INCDTTM', 'JUNCTIONTYPE', 'INATTENTIONIND', 'UNDERINFL', 'WEATHER', 'ROADCOND', 'LIGHTCOND', 'PEDROWNOTGRNT', 'SPEEDING', 'ST_COLCODE', 'SEGLANEKEY', 'CROSSWALKKEY', 'HITPARKEDCAR']]
```

For KNN: select columns without content as: plain text, koordinates, ids, keys:

```
cdf =
df[['SEVERITYCODE','PERSONCOUNT','PEDCOUNT','PEDCYLCOUNT','VEHCOUNT','WEATHER', 'ROADCOND', 'LIGHTCOND',]]
cdf.head(9)
```

For decision tree: select the accident counts and weather columns:

```
ddf =
df[['SEVERITYCODE','PERSONCOUNT','PEDCOUNT','PEDCYLCOUNT','VEHCOUNT','WEATHER', 'ROADCOND', 'LIGHTCOND',]]
ddf.head(9)
```

2.2.2.2 Deal with missing data

How to deal with missing data

1. Drop data

1. a) Drop the whole row
Whole columns should be dropped only if most entries in the column are empty.

In our dataset, none of the columns are empty enough to drop entirely.
We have some freedom in choosing which method to replace data; however, some methods may seem more reasonable than others. We will apply each method to many different columns.

* 8."COLLISIONTYPE": 4904 missing data, simply delete the whole row.
    * Reason: COLLISIONTYPE is what we need to predict accident fatality. Any data entry without COLLISIONTYPE data cannot be used for prediction; therefore any row now without COLLISIONTYPE data is not useful to us.
* 3."ADDRTYPE": 1926 missing data. See reason as above.

* 12. "WEATHER": 5081 missing data, simply delete the whole row. See reason as above.
* 13. "ROADCOND": 5012 missing data, simply delete the whole row. See reason as above.
* 14. "LIGHTCOND": 5170 missing data, simply delete the whole row. See reason as above.
* 18. "ST_COLCODE": 18 missing data, simply delete the whole row.

Simply drop whole row with NaN in "COLLISIONTYPE" column
cdf.dropna(subset=["COLLISIONTYPE"], axis=0, inplace=True)

Simply drop whole row with NaN in "ADDRTYPE" column
#df.dropna(subset=["ADDRTYPE"], axis=0, inplace=True)

Simply drop whole row with NaN in "WEATHER" column
cdf.dropna(subset=["WEATHER"], axis=0, inplace=True)
ddf.dropna(subset=["WEATHER"], axis=0, inplace=True)

Simply drop whole row with NaN in "ROADCOND" column
cdf.dropna(subset=["ROADCOND"], axis=0, inplace=True)
ddf.dropna(subset=["ROADCOND"], axis=0, inplace=True)

Simply drop whole row with NaN in "LIGHTCOND" column
cdf.dropna(subset=["LIGHTCOND"], axis=0, inplace=True)
ddf.dropna(subset=["LIGHTCOND"], axis=0, inplace=True)

Reset index, because we droped some rows
cdf.reset_index(drop=True, inplace=True)
ddf.reset_index(drop=True, inplace=True)

1. b) Drop the whole column

* 4. "INTKEY" : 129603 missing data, drop the whole column.
    * Reason: The key that corresponds to the intersection associated with a collision is not used herewith as it is redundant to coordinates.

*   5. "LOCATION": 2677 missing data, drop the whole column. See reason as above.
*   6. "EXCEPTRSNCODE": 109862 missing data, drop the whole column. Code not used herewith.
*   7. "EXCEPTRSNDESC": 189035 missing data, drop the whole column. Description not used herewith.
*   9. "JUNCTIONTYPE": 6329 missing data, drop the whole column. Type of Junction not used herewith.
* 10. "INATTENTIONIND": 164868 missing data, drop the whole column. Inattention not used herewith.
* 11. "UNDERINFL": 4884 missing data, drop the whole column. Drugs and alcohol usage is not used herewith.
* 15. "PEDROWNOTGRNT": 190006 missing data, drop the whole column. Pedestrian right not used herewith.
* 16. "SDOTCOLNUM": 79737 missing data, drop the whole column. Sub cathegories of ST_COLCODE not used herewith.
* 17. "SPEEDING": 185340 missing data, drop the whole column. Speeding was a factor not used herewith.
* 19. "ST_COLDESC": 4904 missing data, drop the whole column. Description not used as this is plain text.

2. Replace data:

2. a) Replace it by mean
*   1. "X": 5334 missing data, replace them with mean.
    * Reason: as this is a coordinate of the date set in the Seattle area assumed, mean value might be sufficient
*   2. "Y": 5334 missing data, replace them with mean. See reason as above.

For KNN:

Calculate the mean value for 'PERSONCOUNT' column¶
avg_PERSONCOUNT=cdf['PERSONCOUNT'].astype('float').mean(axis=0)
print("Average of PERSONCOUNT:", avg_PERSONCOUNT)
Replace NaN by mean value
cdf["PERSONCOUNT"].replace(np.nan, avg_PERSONCOUNT, inplace=True)

Calculate the mean value for 'PEDCOUNT' column¶
avg_PEDCOUNT=cdf['PEDCOUNT'].astype('float').mean(axis=0)
print("Average of PEDCOUNT:", avg_PEDCOUNT)
Replace NaN by mean value
cdf["PEDCOUNT"].replace(np.nan, avg_PEDCOUNT, inplace=True)

Calculate the mean value for 'PEDCYLCOUNT' column¶
avg_PEDCYLCOUNT=cdf['PEDCYLCOUNT'].astype('float').mean(axis=0)

print("Average of PEDCYLCOUNT:", avg_PEDCYLCOUNT)
Replace NaN by mean value
cdf["PEDCYLCOUNT"].replace(np.nan, avg_PEDCYLCOUNT, inplace=True)

Calculate the mean value for 'VEHCOUNT' column¶
avg_VEHCOUNT=cdf['VEHCOUNT'].astype('float').mean(axis=0)
print("Average of VEHCOUNT:", avg_VEHCOUNT)
Replace NaN by mean value
cdf["VEHCOUNT"].replace(np.nan, avg_VEHCOUNT, inplace=True)

For decision tree:

Calculate the mean value for 'PERSONCOUNT' column¶
avg_PERSONCOUNT=ddf['PERSONCOUNT'].astype('float').mean(axis=0)
print("Average of PERSONCOUNT:", avg_PERSONCOUNT)
Replace NaN by mean value
ddf["PERSONCOUNT"].replace(np.nan, avg_PERSONCOUNT, inplace=True)

Calculate the mean value for 'PEDCOUNT' column¶
avg_PEDCOUNT=ddf['PEDCOUNT'].astype('float').mean(axis=0)
print("Average of PEDCOUNT:", avg_PEDCOUNT)
Replace NaN by mean value
ddf["PEDCOUNT"].replace(np.nan, avg_PEDCOUNT, inplace=True)

Calculate the mean value for 'PEDCYLCOUNT' column¶
avg_PEDCYLCOUNT=ddf['PEDCYLCOUNT'].astype('float').mean(axis=0)
print("Average of PEDCYLCOUNT:", avg_PEDCYLCOUNT)
Replace NaN by mean value
ddf["PEDCYLCOUNT"].replace(np.nan, avg_PEDCYLCOUNT, inplace=True)

Calculate the mean value for 'VEHCOUNT' column¶
avg_VEHCOUNT=ddf['VEHCOUNT'].astype('float').mean(axis=0)
print("Average of VEHCOUNT:", avg_VEHCOUNT)
Replace NaN by mean value
ddf["VEHCOUNT"].replace(np.nan, avg_VEHCOUNT, inplace=True)

2. b) Replace it by frequency c) replace it based on other functions
not applicable

Calculated averages:

Average of PERSONCOUNT: 2.459566804164004

Average of PEDCOUNT: 0.03810665638517564
Average of PEDCYLCOUNT: 0.0291279570290012
Average of VEHCOUNT: 1.970412544827477

2.2.2.3 Correct Data Format

Data types corrections:

```
cdf.dtypes,
ddf.dtypes

SEVERITYCODE     int64
PERSONCOUNT      int64
PEDCOUNT         int64
PEDCYLCOUNT      int64
VEHCOUNT         int64
WEATHER         object
ROADCOND        object
LIGHTCOND       object
dtype: object
```

As we can see above, some columns are not of the correct data type.
Numerical variables should have type 'float' or 'int',
and variables with strings such as categories should have type 'object'.

For decision tree SEVERITYCODE has to be of String data type.

```
ddf[["SEVERITYCODE"]] = ddf[["SEVERITYCODE"]].astype("object")
```

Value Counts for cathgories

Specific values for categories must be converted to Numbers as the machine learning algorithm
Can not handle text categories.

```
cdf['WEATHER'].value_counts() ,
ddf['WEATHER'].value_counts()

Clear          111008
Raining         33117
Overcast        27681
Unknown         15039
Snowing           901
Other             824
```

```
Fog/Smog/Smoke               569
Sleet/Hail/Freezing Rain       113
Blowing Sand/Dirt              55
Severe Crosswind               25
Partly Cloudy                   5
Name: WEATHER, dtype: int64
```

```
cdf['ROADCOND'].value_counts() ,
ddf['ROADCOND'].value_counts()
```

```
Dry             124300
Wet              47417
Unknown          15031
Ice               1206
Snow/Slush         999
Other              131
Standing Water     115
Sand/Mud/Dirt       74
Oil                 64
Name: ROADCOND, dtype: int64
```

```
cdf['LIGHTCOND'].value_counts() ,
ddf['LIGHTCOND'].value_counts()
```

```
Daylight                    116077
Dark - Street Lights On      48440
Unknown                      13456
Dusk                          5889
Dawn                          2502
Dark - No Street Lights       1535
Dark - Street Lights Off      1192
Other                          235
Dark - Unknown Lighting         11
Name: LIGHTCOND, dtype: int64
```

```
cdf['SEVERITYCODE'].value_counts() ,
ddf['SEVERITYCODE'].value_counts()
```

```
1    132285
2     57052
Name: SEVERITYCODE, dtype: int64
```

2.3 Pre-processing

2.3.1 Pre-processing for KNN:

Get indicator variables and assign it to data frame "dummy_variable_1“:
dummy_variable_1 = pd.get_dummies(cdf["WEATHER"])
dummy_variable_2 = pd.get_dummies(cdf["ROADCOND"])
dummy_variable_3 = pd.get_dummies(cdf["LIGHTCOND"])

Change column names for clarity:
dummy_variable_1.rename(columns={'Unknown':'WeUnknown'}, inplace=True)
dummy_variable_1.rename(columns={'Other':'WeOther'}, inplace=True)
dummy_variable_1.rename(columns={'Fog/Smog/Smoke':'FogSmogSmoke', 'Sleet/Hail/Freezing Rain':'SleetHailFreezingRain'}, inplace=True)
dummy_variable_1.rename(columns={'Blowing Sand/Dirt':'BlowingSandDirt','Severe Crosswind':'SevereCrosswind'}, inplace=True)
dummy_variable_1.rename(columns={'Partly Cloudy':'PartlyCloudy'}, inplace=True)

dummy_variable_2.rename(columns={'Unknown':'RoUnknown'}, inplace=True)
dummy_variable_2.rename(columns={'Snow/Slush':'SnowSlush', 'Other':'RoOther'}, inplace=True)
dummy_variable_2.rename(columns={'Standing Water':'StandingWater', 'Sand/Mud/Dirt':'SandMudDirt'}, inplace=True)

dummy_variable_3.rename(columns={'Dark - Street Lights On':'DarkStreetLightsOn'}, inplace=True)
dummy_variable_3.rename(columns={'Unknown':'LiUnknown'}, inplace=True)
dummy_variable_3.rename(columns={'Dark - No Street Lights':'DarkNoStreetLights'}, inplace=True)
dummy_variable_3.rename(columns={'Dark - Street Lights Off':'DarkStreetLightsOff', 'Other':'LiOther'}, inplace=True)
dummy_variable_3.rename(columns={'Dark - Unknown Lighting':'DarkUnknownLighting'}, inplace=True)

dummy_variable_1.head()
dummy_variable_2.head()
dummy_variable_3.head()
dummy_variable_4.head()

We now have i.e. the value 0 to represent “Dry" and 1 to represent "Wet" in the column "WEATHER" etc.
We will now insert this column back into our original dataset.

Merge data frame "cdf" and "dummy_variable_1"
cdf = pd.concat([cdf, dummy_variable_1], axis=1)

Drop original column "WEATHER" from "cdf"
cdf.drop("WEATHER", axis = 1, inplace=True)

Merge data frame "cdf" and "dummy_variable_2"
cdf = pd.concat([cdf, dummy_variable_2], axis=1)

Drop original column "ROADCOND" from "cdf"
cdf.drop("ROADCOND", axis = 1, inplace=True)

Merge data frame "cdf" and "dummy_variable_3"
cdf = pd.concat([cdf, dummy_variable_3], axis=1)

Drop original column "LIGHTCOND" from "cdf"
cdf.drop("LIGHTCOND", axis = 1, inplace=True)

Merge data frame "cdf" and "dummy_variable_4"
cdf = pd.concat([cdf, dummy_variable_4], axis=1)

Drop original column "SPEEDING" from "cdf"
cdf.drop("SPEEDING", axis = 1, inplace=True)

cdf.head()

2.3.2 Pre-processing for Decision Tree

Remove columns not needed for decision tree.
dtree_df = ddf[['SEVERITYCODE', 'PERSONCOUNT', 'VEHCOUNT', 'WEATHER', 'ROADCOND', 'LIGHTCOND']]
dtree_df[0:5]

We use dtree_data as the accident data read by pandas, declare the following variables:
- X_dtree as the Feature Matrix (data of dtree_data)
- y_dtree as the response vector (target)

We remove the column containing the target name since it doesn't contain numeric values.

```
X_dtree = dtree_df[['PERSONCOUNT', 'VEHCOUNT', 'WEATHER', 'ROADCOND',
'LIGHTCOND']].values
X_dtree[0:5]
```

As we figure out, some features in this dataset are categorical such as WEATHER, ROADCOND or LIGHTCOND.
Unfortunately, Sklearn Decision Trees do not handle categorical variables.
But still we can convert these features to numerical values.
pandas.get_dummies()
Convert categorical variable into dummy/indicator variables.

```
le_weather = preprocessing.LabelEncoder()
le_weather.fit(['Blowing Sand/Dirt', 'Clear', 'Fog/Smog/Smoke', 'Other',
'Overcast',
       'Partly Cloudy', 'Raining', 'Severe Crosswind',
       'Sleet/Hail/Freezing Rain', 'Snowing', 'Unknown'])
X_dtree[:,2] = le_weather.transform(X_dtree[:,2])

le_roadcond = preprocessing.LabelEncoder()
le_roadcond.fit([ 'Dry', 'Ice', 'Oil',
      'Other', 'Sand/Mud/Dirt', 'Snow/Slush', 'Standing Water', 'Unknown', 'Wet'])
X_dtree[:,3] = le_roadcond.transform(X_dtree[:,3])

le_lightcond = preprocessing.LabelEncoder()
le_lightcond.fit([ 'Dark - No Street Lights', 'Dark - Street Lights Off',
      'Dark - Street Lights On', 'Dark - Unknown Lighting', 'Dawn',
      'Daylight', 'Dusk', 'Other', 'Unknown'])
X_dtree[:,4] = le_lightcond.transform(X_dtree[:,4])

X_dtree[0:5]
```

Now we can fill the target variable.

```
y_dtree = dtree_df["SEVERITYCODE"].astype(str)

y_dtree[0:5]
```

3 Methodology Section

3.1 Methodology

This section which represents the main component of the report where we discuss and describe any exploratory data analysis that we did, any inferential statistical testing that we performed, if any, and what machine learnings were used and why.

We will use the following machine learning models for car accident prediction:

- K-Nearest Neighbors (KNN)
- Decision Tree

3.2 K-Nearest Neighbors (KNN)

In this Project we will use K-Nearest Neighbors to predict a data point, whether SERVERITYCODE is 1 or 2.

K-Nearest Neighbors is an algorithm for supervised learning, where the data is 'trained' with
data points corresponding to their classification. Once a point is to be predicted, it takes into account
the 'K' nearest points to it to determine it's classification.

In this case, we have data points of SERVERITYCODE 1 and 2. We want to predict what the star (test data point) is.
If we consider a k value of 3 (3 nearest data points) we will obtain a prediction of class SERVERITYCODE 2 which is the worst case for an traffic accident.
Yet if we consider a k value of 6, we will obtain a prediction of Class SERVERITYCODE 1.

In this sense, it is important to consider the value of k. But hopefully from the resulting diagram,
we should get a sense of what the K-Nearest Neighbors algorithm is.
It considers the 'K' Nearest Neighbors (points) when it predicts the classification of the test point.

A number of required libraries must be loaded.

3.2.1 Feature set

Lets define feature sets, X:
cdf.columns

```
Index(['SEVERITYCODE', 'PERSONCOUNT', 'PEDCOUNT', 'PEDCYLCOUNT', 'VEHCOUNT',
       'BlowingSandDirt', 'Clear', 'FogSmogSmoke', 'WeOther', 'Overcast',
       'PartlyCloudy', 'Raining', 'SevereCrosswind', 'SleetHailFreezingRain',
```

```
       'Snowing', 'WeUnknown', 'Dry', 'Ice', 'Oil', 'RoOther', 'SandMudDirt',
       'SnowSlush', 'StandingWater', 'RoUnknown', 'Wet', 'DarkNoStreetLights',
       'DarkStreetLightsOff', 'DarkStreetLightsOn', 'DarkUnknownLighting',
       'Dawn', 'Daylight', 'Dusk', 'LiOther', 'LiUnknown'],
      dtype='object')
```

To use scikit-learn library, we have to convert the Pandas data frame to a Numpy array:

```
X = cdf[['PERSONCOUNT', 'PEDCOUNT', 'PEDCYLCOUNT', 'VEHCOUNT',
       'Blowing Sand/Dirt', 'Clear', 'Fog/Smog/Smoke', 'Other', 'Overcast',
       'Partly Cloudy', 'Raining', 'Severe Crosswind',
       'Sleet/Hail/Freezing Rain', 'Snowing', 'Unknown', 'Dry', 'Ice', 'Oil',
       'Other', 'Sand/Mud/Dirt', 'Snow/Slush', 'Standing Water', 'Unknown',
       'Wet', 'Dark - No Street Lights', 'Dark - Street Lights Off',
       'Dark - Street Lights On', 'Dark - Unknown Lighting', 'Dawn',
       'Daylight', 'Dusk', 'Other', 'Unknown']] .values.astype(float)
X[0:5]
```

```
array([[2., 0., 0., ..., 0., 0., 0.],
       [2., 0., 0., ..., 0., 0., 0.],
       [4., 0., 0., ..., 0., 0., 0.],
       ...,
       [2., 0., 0., ..., 0., 0., 0.],
       [2., 0., 0., ..., 0., 0., 0.],
       [3., 0., 0., ..., 0., 0., 0.]])
```

What are our labels

```
y = cdf['SEVERITYCODE'].values
y[0:5]
```

```
array([2, 1, 1, 1, 2, 1, 1, 2, 1, 2, 1, 1, 1, 1, 2, 2, 1, 2, 1, 2, 2, 1,
       1, 2, 2, 1, 1, 1, 1, 1])
```

3.2.2 Normalize Data

Data Standardization give data zero mean and unit variance, it is good practice, especially for algorithms such as KNN which is based on distance of cases:

```
X = preprocessing.StandardScaler().fit(X).transform(X.astype(float))
X[0:5]
```

3.2.3 Train Test Split for KNN

Out of Sample Accuracy is the percentage of correct predictions that the model

makes on data that that
the model has NOT been trained on. Doing a train and test on the same dataset will most likely have
low out-of-sample accuracy, due to the likelihood of being over-fit.
It is important that our models have a high, out-of-sample accuracy, because the purpose of any model,
of course, is to make correct predictions on unknown data. So how can we improve out-of-sample accuracy?
One way is to use an evaluation approach called Train/Test Split.
Train/Test Split involves splitting the dataset into training and testing sets respectively,
which are mutually exclusive. After which, you train with the training set and test with the testing set.
This will provide a more accurate evaluation on out-of-sample accuracy because the testing dataset
is not part of the dataset that have been used to train the data.
It is more realistic for real world problems.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2, random_state=4)
print ('Train set:', X_train.shape, y_train.shape)
print ('Test set:', X_test.shape, y_test.shape)
```

```
Train set: (151469, 45) (151469,)
Test set: (37868, 45) (37868,)
```

3.2.4 Classification

K nearest neighbor (KNN)
Import library
Classifier implementing the k-nearest neighbors vote.

3.2.5 Training

Let’s start the algorithm with k=4 for now:

```
k=4
```

Train Model and Predict

```
neigh = KNeighborsClassifier(n_neighbors = k).fit(X_train,y_train)
neigh
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
metric_params=None, n_jobs=None, n_neighbors=4, p=2, weights='uniform')
```

3.2.6 Predicting
We can use the model to predict the test set:

```
yhat = neigh.predict(X_test)
yhat[0:5]

array([1, 1, 1, 1, 1])
```

3.2.7 What about other K?

K in KNN, is the number of nearest neighbors to examine. It is supposed to be specified by the User.
So, how can we choose right value for K? The general solution is to reserve a part of our data
for testing the accuracy of the model.
Then chose k =1, we use the training part for modeling, and calculate the accuracy of prediction using
all samples in your test set. We repeat this process, increasing the k, and see which k is the best for
our model.
We can calculate the accuracy of KNN for different Ks.

```
Ks = 10
mean_acc = np.zeros((Ks-1))
std_acc = np.zeros((Ks-1))
ConfustionMx = [];

for n in range(1,Ks):
    #Train Model and Predict
    neigh = KNeighborsClassifier(n_neighbors = n).fit(X_train,y_train)
    yhat=neigh.predict(X_test)
    mean_acc[n-1] = metrics.accuracy_score(y_test, yhat)
    std_acc[n-1]=np.std(yhat==y_test)/np.sqrt(yhat.shape[0])

mean_acc

array([0.71170909, 0.73885603, 0.69615506, 0.70758952, 0.7018855 ,
       0.73600401, 0.71714904, 0.73478927, 0.72150628])
```

3.2.8 Plot model accuracy for Different number of Neighbors

```
plt.plot(range(1,Ks),mean_acc,'g')
plt.fill_between(range(1,Ks),mean_acc - 1 * std_acc,mean_acc + 1 * std_acc,
alpha=0.10)
```

```
plt.legend(('Accuracy ', '+/- 3xstd'))
plt.ylabel('Accuracy ')
plt.xlabel('Number of Neighbours (K)')
plt.tight_layout()
plt.show()
```

3.3 Decision Tree

3.3.1 Setting up the Decision Tree

We will be using train/test split on our decision tree.

Now train_test_split will return 4 different parameters.
We will name them: X_trainset, X_testset, y_trainset, y_testset
The train_test_split will need the parameters: X, y, test_size=0.3, and random_state=3.
The X and y are the arrays required before the split,
the test_size represents the ratio of the testing dataset, and
the random_state ensures that we obtain the same splits.

```
X_dtree_trainset, X_dtree_testset, y_dtree_trainset, y_dtree_testset = 
train_test_split(X_dtree, y_dtree, test_size=0.3, random_state=3)
```

We print the shape of X_dtree_trainset and y_dtree_trainset. We ensure that the dimensions match

```
X_dtree_trainset.shape
X_dtree_trainset[0:5]

y_dtree_trainset.shape
y_dtree_trainset[0:5]

# We print the shape of X_dtree_testset and y_dtree_testset. We ensure that the dimensions match
X_dtree_testset.shape
X_dtree_testset[0:5]

y_dtree_testset.shape
y_dtree_testset[0:5]
```

3.3.2 Modeling Decision Tree

We will first create an instance of the DecisionTreeClassifier called sevTree. Inside of the classifier, specify criterion="entropy" so we can see the information gain of each node.

```
sevTree = DecisionTreeClassifier(criterion="entropy", max_depth = 4)
```

```
sevTree = DecisionTreeClassifier(criterion="entropy", max_depth = 4,
    class_weight=None, max_features=None, max_leaf_nodes=None,
    min_impurity_decrease=0.0, min_impurity_split=None, min_samples_leaf=1,
    min_samples_split=2, min_weight_fraction_leaf=0.0,
    presort=False, random_state=None, splitter='best')
```

sevTree - it shows the default parameters

Next, we will fit the data with the training feature matrix X_dtree_trainset and training response vector y_dtree_trainset

```
sevTree.fit(X_dtree_trainset,y_dtree_trainset)
```

3.3.3 Prediction

Let's make some predictions on the testing dataset and store it into a variable called predTree.

```
predTree = sevTree.predict(X_dtree_testset)
```

We can print out predTree and y__dtree_testset if we want to visually compare the prediction to the actual values.

```
print (predTree [0:5])
print (y_dtree_testset [0:5])
```

3.3.4 Visualization

To visualize the tree a number of python libraries are needed.

Notice: We might need install the libraries in Environment of the applied Jupiter Notebbook, which is a tedious procedure for beginners.

```
dot_data = StringIO()
```

```
filename = "sevtree.png"
featureNames = dtree_df.columns[0:5]
targetNames = dtree_df["SEVERITYCODE"].unique().tolist()
# targetNames = dtree_df["SEVERITYCODE"].astype('str').unique().tolist()

out=tree.export_graphviz(sevTree,feature_names=featureNames,
out_file=dot_data, class_names= np.unique(y_dtree_trainset), filled=True,
special_characters=True,rotate=False)

graph = pydotplus.graph_from_dot_data(dot_data.getvalue())

graph.write_png(filename)

img = mpimg.imread(filename)

plt.figure(figsize=(100, 200))
plt.imshow(img,interpolation='nearest')
```

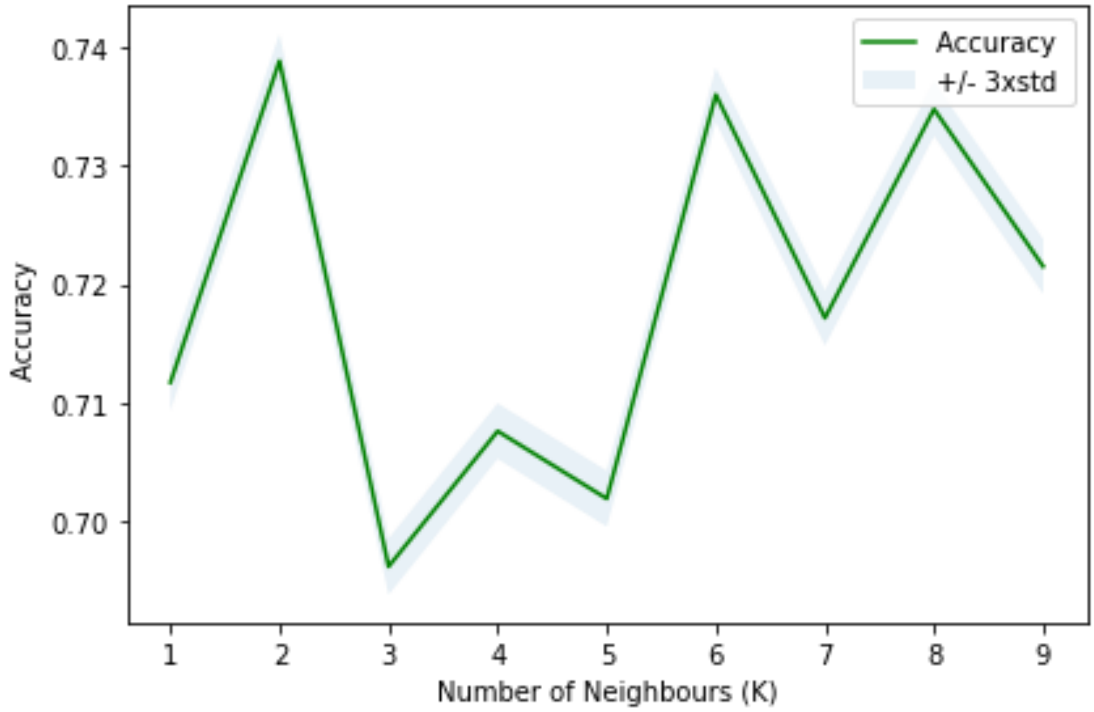
4 Results Section

Evaluation for KNN

Accuracy evaluation for KNN
In multilabel classification, accuracy classification score is a function that computes subset accuracy.
This function is equal to the jaccard_similarity_score function.
Essentially, it calculates how closely the actual labels and predicted labels are matched in the test set.

```
print("Train set Accuracy: ", metrics.accuracy_score(y_train,
neigh.predict(X_train)))
print("Test set Accuracy: ", metrics.accuracy_score(y_test, yhat))

Train set Accuracy:  0.7127332985627423
Test set Accuracy:  0.707589521495722
```



print("The best accuracy was with", mean_acc.max(), "with k=", mean_acc.argmax()+1)

The best accuracy was with 0.7388560261962607 with k= 2

Evaluation for Decision Tree

print("DecisionTrees's Accuracy: ", metrics.accuracy_score(y_dtree_testset, predTree))

DecisionTrees's Accuracy: 0.7446216682511179

So Accuracy for Decision Tree is at 74%

Accuracy classification score computes subset accuracy:
the set of labels predicted for a sample must exactly match the corresponding set of labels in y_true.
In multilabel classification, the function returns the subset accuracy.
If the entire set of predicted labels for a sample strictly match with the true set of labels,
then the subset accuracy is 1.0; otherwise it is 0.0.

Fig. for decision tree available when prerequisite python libraries installed.

5 Discussion Section

In this section we discuss any observations you noted and any recommendations you can make based on the results.

5.1 Selection of machine learning models

We used the following machine learning models for car accident prediction:
- K-Nearest Neighbors (KNN)
- Decision Tree

5.2 Accuracy of the models

The obtained accuracy of both seems to be suitable for resolving the problem of car accident prediction on current available weather observations and conditions for a planned car travel.

5.3 Technical Installation Prerequisites

For the models we need a number of python libraries. To install the right version of these in the development environment without having any python error messages when developing the models was a tedious job.

For KNN the necessary libraries could be installed and a graphical plot could be printed.

For Decision Tree the necessary libraries most could be installed. We experience timeouts when installing the libraries for Decision Tree and a graphical plot could not be printed. This problem needs to be fixed in the near future.

5.4 Unresolved Error Messages
The following error message when preparing the decision tree plot could not be resolved:

InvocationException: GraphViz's executables not found

5.5 Integrated Development Environments IDEs
We used to different IDEs:

- IBM Cloud, Jupyter Notebook

– Anaconda on iMac OSX Jupyter Notebook

Both work fine for the development of the machine learning models

Setup the specific technical environment for Jupyter could be improved for machine learning problems.

Elapsed installation times for prerequisite python libraries in both IDEs are not acceptable long.

5.5 Other machine learning models

5.6 Business Understanding phase
The initial phase to understand the project's objective from the business or application perspective could be resolved.

Translation of this knowledge into a machine learning problem with a preliminary plan to achieve the objectives could be resolved.

5.7 Data understanding phase
Collecting or extracting the dataset from various sources such as csv file or SQL database could be resolved. csv File was used for that task. A SQL database dis not apply.

Determining the attributes (columns) that are useed to train the selected machine learning model could be resolved. Also, assessing the condition of chosen attributes by looking for trends, certain patterns, skewed information, correlations, and so on was resolved initially, but could be improved. At leaset we found only two severity categories, which depend not only on the current weather conditions. These patterns need further studies.

5.8 Data Preparation phase
Data preparation included all the required activities to construct the final dataset which were fed into the selected modeling tools. Data preparation was performed multiple times and it included balancing the labeled data, transformation, filling missing data, and cleaning the dataset. This was until now the major effort of the project: data cleansing. The available data quality needs in general more attention, which can not be assumed without effort. This is also true for the right data available for statistics problems to be resolved with the selected models.

5.9 Modeling phase
In this phase, in general various algorithms and methods can be selected and applied to build the model including supervised machine learning techniques. We selected only two: KNN and decision tree. Furthermore, SVM, XGBoost,

decision tree, or any other techniques could be selected as well. This is for further studies. In general, a single or multiple machine learning models for the same data mining problem could be selected. At least, herewith only two machine learning models were selected. At this phase, stepping back to the data preparation phase was often required. This was also high effort prone.

5.10 Evaluation phase

Before proceeding to the deployment stage, the model needed to be evaluated thoroughly to ensure that the business or the applications' objectives are achieved. Certain metrics could be used for the model evaluation such as accuracy, recall, F1-score, precision, and others. For this project, only accuracy was calculated ans evaluated for the selected machine learning models. Both have acceptable values. Other metrics are for further studies.

5.11 Deployment

In general, as he deployment phase requirements varies from project to project, the report is deployed to a website of the author. As this can be as simple as creating a report, developing interactive visualization, or making the machine learning model available in the production environment, the working files a submitted to the authors' Git hub. In this environment, the possible customers or end-users can utilize the model in different ways such as API, website, or so on. At least, this work is published to everyone interested.

Published as a blog on: www.energing.de

Guthub: https://github.com/herbfrisch/jbs_cornerstone

6 Conclusion Section

In this section we conclude the report.

As the work is still in progress, this conclusion part is of preliminary status.

Accident severity probability prediction is feasible on the basis of weather data for K-Nearest Neighbors (KNN) and Decision Tree machine models.

Additional machine learning models ar for further studies.

In depth assessment the condition of chosen weather and accident attributes by looking for trends, certain patterns, skewed information, correlations, and so on is for further study.