Automobile Accident Severity Predictor

Trent Barron

October 22, 2020

Table of Contents

Introduction: Business Understanding

The purpose of this project is to develop a predictor tool that can predict the severity of a traffic accident based on various scenarios that affect driving conditions, specifically weather, road and light conditions. The tool will provide highway travelers the means to determine if current travel conditions may increase the severity of a traffic accident and help them adjust their travel plans based on their risk profile. This project utilizes the CRISP-DM methodology to analyze data provided by the Transportation Department of the City of Seattle and industry best practice machine learning techniques to model and validate the predictor tool.

Methodology

Data Understanding

The dataset used for this predictor is provided by the Transportation Department of the City of Seattle and was retrieved from the site Seattle GeoData. This dataset provides a range of information on documented collisions that have taken place in the City of Seattle, WA. It is important to review the attributes of this dataset and determine what data is relevant to our predictor. After reviewing the descriptions of the attributes provided in the dataset, it is apparent that SEVERITYCODE, WEATHER, ROADCOND, and LIGHTCOND are the attributes that we should focus on. The SEVERITYCODE provides 5 different codes that correspond to five

different “severities”: 3=fatality, 2b=serious injury, 2=injury, 1=property damage, and 0=unknown. WEATHER provides a description of the weather conditions during the time of the collision, ROADCOND provides a description of the road conditions during the collision, and LIGHTCOND provides the light conditions during the collision. Each one of these attributes will need to be investigated further to ensure there is no questionable, missing, or ambiguous data.

Some key takeaways that came from investigating the data set are that there are only two severity types identified: ‘Property Damage Only Collision and ‘Injury Collision.’ These severity types are identified by severity codes 1 and 2, respectively. This will come in handy for models that do not work well with character variables.

Running Analysis of Variance (ANOVA) was useful in identifying the unique variables for each attribute (weather condition, road conditions, and light condition) and determining their statistical significance (Figure 1). After running this test it is determined that each attribute is significant to ‘severity’ due to large F-scores and P-values near zero.

	WEATHER	SEVERITYCODE
0	Overcast	2
1	Raining	1
3	Clear	1
15	NaN	1
19	Unknown	1
34	Other	1
290	Snowing	1
706	Fog/Smog/Smoke	2
799	Sleet/Hail/Freezing Rain	1
2238	Blowing Sand/Dirt	1
10601	Severe Crosswind	1
180685	Partly Cloudy	1

ANOVA results: F= 14.78219963145612 , P = 7.727373580982942e-22

	ROADCOND	SEVERITYCODE
0	Wet	2
2	Dry	1
15	NaN	1
23	Unknown	1
682	Snow/Slush	1
799	Ice	1
2754	Other	1
3336	Sand/Mud/Dirt	2
3353	Standing Water	2
3942	Oil	2

ANOVA results: F= 30.99357510599123 , P = 1.926098492840346-37

	LIGHTCOND	SEVERITYCODE
0	Daylight	2
1	Dark - Street Lights On	1
13	Dark - No Street Lights	1
15	NaN	1
19	Unknown	1
56	Dusk	2
89	Dawn	2
175	Dark - Street Lights Off	2
197	Other	1
437	Dark - Unknown Lighting	1

ANOVA results: F= 54.74957134205072 , P = 4.866743798662696-57

Figure 1

Data Preparation

The next step is to streamline the original dataset into a data frame (df) that contains the relevant data (Figure 2) and then clean the data to eliminate questionable, missing, or ambiguous data.

```
#Create data frame with only required attributes
df=rawdata[['SEVERITYCODE','WEATHER','ROADCOND','LIGHTCOND']]
df.head()
```

	SEVERITYCODE	WEATHER	ROADCOND	LIGHTCOND
0	2	Overcast	Wet	Daylight
1	1	Raining	Wet	Dark - Street Lights On
2	1	Overcast	Dry	Daylight
3	1	Clear	Dry	Daylight
4	2	Raining	Wet	Daylight

Figure 2

By creating a data frame called “missing_data” that indicates missing data in “df” using the “isnull” function we can identify rows with missing data (Figure 2). We are also able to analyze the amount of data missing by comparing the count of “true”, missing data, to “false”, has data (Figure 3). Analysis of the missing data revealed that SEVERITYCODE, WEATHER, ROADCOND, AND LIGHTCOND all had missing values (Figure 3) that made up approximately 3% of the rows per column.

```
#Determine if there is any missing data
missing_data = df.isnull()
missing_data.head(5)
```

	SEVERITYCODE	WEATHER	ROADCOND	LIGHTCOND
0	False	False	False	False
1	False	False	False	False
2	False	False	False	False
3	False	False	False	False
4	False	False	False	False

Figure 3

```
#True means that ther is missing data.
for column in missing_data.columns.values.tolist():
    print(column)
    print (missing_data[column].value_counts())
    print("")
```

```
SEVERITYCODE
False    194673
Name: SEVERITYCODE, dtype: int64

WEATHER
False    189592
True       5081
Name: WEATHER, dtype: int64

ROADCOND
False    189661
True       5012
Name: ROADCOND, dtype: int64

LIGHTCOND
False    189503
True       5170
Name: LIGHTCOND, dtype: int64
```

Figure 4

Because of the low “true” to “false” ratio for each attribute, there was little chance that removing this data would jeopardize the integrity of the data. The rows with missing values were removed and the results were validated to ensure no missing data (Figure 5).

```
# drop whole row with NaN in "SEVERITYCODE", "WEATHER", "ROADCOND" AND "LIGHTCOND" column and check results
df.dropna(subset=["SEVERITYCODE", "WEATHER", "ROADCOND","LIGHTCOND"], axis=0, inplace=True)
missing_data = df.isnull()
for column in missing_data.columns.values.tolist():
    print(column)
    print (missing_data[column].value_counts())
    print("")
```

```
SEVERITYCODE
False    189337
Name: SEVERITYCODE, dtype: int64

WEATHER
False    189337
Name: WEATHER, dtype: int64

ROADCOND
False    189337
Name: ROADCOND, dtype: int64

LIGHTCOND
False    189337
Name: LIGHTCOND, dtype: int64
```

Figure 5

We now need to determine if there are any attributes that are not really factors .

Pre-processing

Further analysis of the remaining data revealed that WEATHER, ROADCOND, and LIGHTCOND were object type data rather than numerical values. This makes the data difficult to model because the machine learning library that we are going to use for modeling, Scikit-learn, will not handle categorical variables such as those used in these attributes. Fortunately, Scikit-learn provides preprocessing functions that will convert these categorical variables into dummy/indicator variables for modeling. First, we need to identify the unique values used in each of these attributes (Figure 6).

```
print('SEVERITYCODE unique values:', df['SEVERITYCODE'].unique())
print('WEATHER unique values:', df['WEATHER'].unique())
print('ROADCOND unique values:', df['ROADCOND'].unique())
print('LIGHTCOND unique values:', df['LIGHTCOND'].unique())
```

```
SEVERITYCODE unique values: [2 1]
WEATHER unique values: ['Overcast' 'Raining' 'Clear' 'Unknown' 'Other' 'Snowing' 'Fog/Smog/Smoke'
 'Sleet/Hail/Freezing Rain' 'Blowing Sand/Dirt' 'Severe Crosswind'
 'Partly Cloudy']
ROADCOND unique values: ['Wet' 'Dry' 'Unknown' 'Snow/Slush' 'Ice' 'Other' 'Sand/Mud/Dirt'
 'Standing Water' 'Oil']
LIGHTCOND unique values: ['Daylight' 'Dark - Street Lights On' 'Dark - No Street Lights' 'Unknown'
 'Dusk' 'Dawn' 'Dark - Street Lights Off' 'Other'
 'Dark - Unknown Lighting']
```

Figure 6

Because Scikit-learn fit method generally accepts 2 inputs X, the samples matrix, and y, the target values, we need to construct our data accordingly. In this case we want to be able to predict a SEVERITYDESC and SEVERITYCODE based on WEATHER, ROADCOND, and LIGHTCOND so we need to make y = SEVERITYCODE and X equal the matrix containing the features WEATHER, ROADCOND, and LIGHTCOND and their rows or samples (Figure 7).

```
#acceptable inputs for sklearn fit method
X=df[['WEATHER', 'ROADCOND', 'LIGHTCOND']].values
y=df[['SEVERITYCODE']]
print('X, the samples matrix:', X)
print('y, the target vlues:', y)
```

```
X, the samples matrix: [['Overcast' 'Wet' 'Daylight']
 ['Raining' 'Wet' 'Dark - Street Lights On']
 ['Overcast' 'Dry' 'Daylight']
 ...
 ['Clear' 'Dry' 'Daylight']
 ['Clear' 'Dry' 'Dusk']
 ['Clear' 'Wet' 'Daylight']]
y, the target vlues:         SEVERITYCODE
0                       2
1                       1
2                       1
3                       1
4                       2
...                   ...
194668                  2
194669                  1
194670                  2
194671                  2
194672                  1

[189337 rows x 1 columns]
```

Figure 7

Next we need to load the preprocessing functions of Scikit-learn and transform the categorical variables to indicator variables (Figure 8) and validate that they have been changed.

```
from sklearn import preprocessing

le_weather = preprocessing.LabelEncoder()
le_weather.fit(['Overcast','Raining','Clear','Unknown','Other','Snowing','Fog/Smog/Smoke',
                'Sleet/Hail/Freezing Rain','Blowing Sand/Dirt','Severe Crosswind','Partly Cloudy'])
X[:, 0] = le_weather.transform(X[:, 0])

le_roadcond = preprocessing.LabelEncoder()
le_roadcond.fit(['Wet','Dry','Unknown','Snow/Slush','Ice','Other','Sand/Mud/Dirt','Standing Water','Oil'])
X[:, 1] = le_roadcond.transform(X[:, 1])

le_lightcond = preprocessing.LabelEncoder()
le_lightcond.fit(['Daylight','Dark - Street Lights On','Dark - No Street Lights','Unknown','Dusk','Dawn',
                  'Dark - Street Lights Off','Other','Dark - Unknown Lighting'])
X[:, 2] = le_lightcond.transform(X[:, 2])
X[0:5]
```

```
array([[4, 8, 5],
       [6, 8, 2],
       [4, 0, 5],
       [1, 0, 5],
       [6, 8, 5]], dtype=object)
```

Figure 8

We also need to fill the target variable based on these changes for SEVERITYDESC (Figure 9) and SEVERITYCODE (Figure 10). SEVERITYDESC will be useful as the target variable for visualizing the Decision Tree model.

```
y=df['SEVERITYDESC']
y[0:5]
```

```
0                  Injury Collision
1    Property Damage Only Collision
2    Property Damage Only Collision
3    Property Damage Only Collision
4                  Injury Collision
Name: SEVERITYDESC, dtype: object
```

Figure 9

```
y=df['SEVERITYCODE']
y[0:5]
```

```
0    2
1    1
2    1
3    1
4    2
Name: SEVERITYCODE, dtype: int64
```

Figure 10

The final step in data pre-processing is to further separate the dataset into a "train" set and "test" set. The train set will be used to train the model to predict severity given specific indicators. The test set will be used to determine the accuracy of the model by comparing the results of the "trained" model to actual values in the test set. For this model we will be using 80% of the dataset to train the model and the remaining 20% to test the model. We also need to print the shapes of the sets to ensure they match (Figure 11).

```
from sklearn.model_selection import train_test_split
X_trainset, X_testset, y_trainset, y_testset = 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, 3) (151469, 1)
Test set: (37868, 3) (37868, 1)
```

Figure 11

Data Modeling

Now that we have prepared the data, we are ready be build our models. I have decided to use Sklearn's Decision Tree, Logistic Regression, K-nearest Neighbor, and Support Vector Machine algorithms to predict accident severity. I will then determine the most accurate based on F-score and Jaccard index analysis.

Decision Tree

The Decision Tree (DT) algorithm returned an accuracy of 0.67 and I was able to produce a nice visual of the process (Figures 12 & 13).

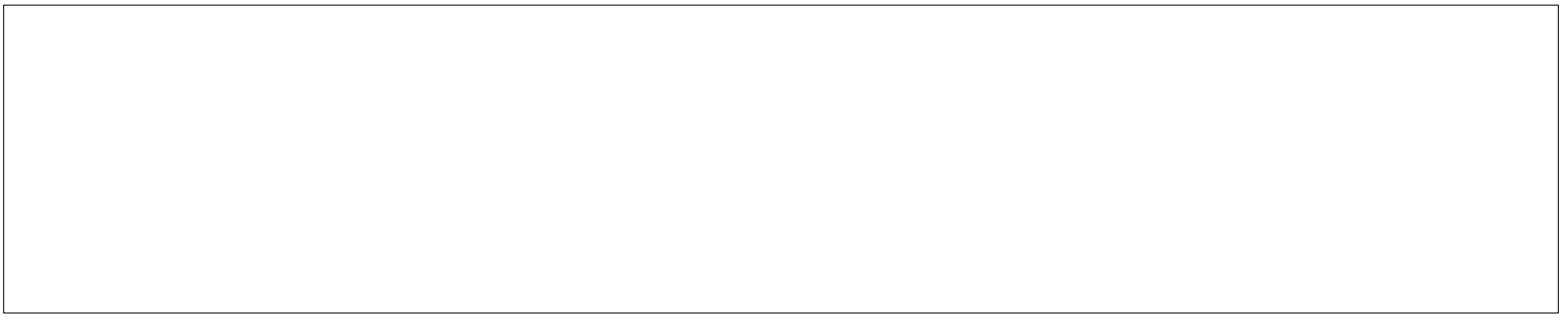

Figure 12

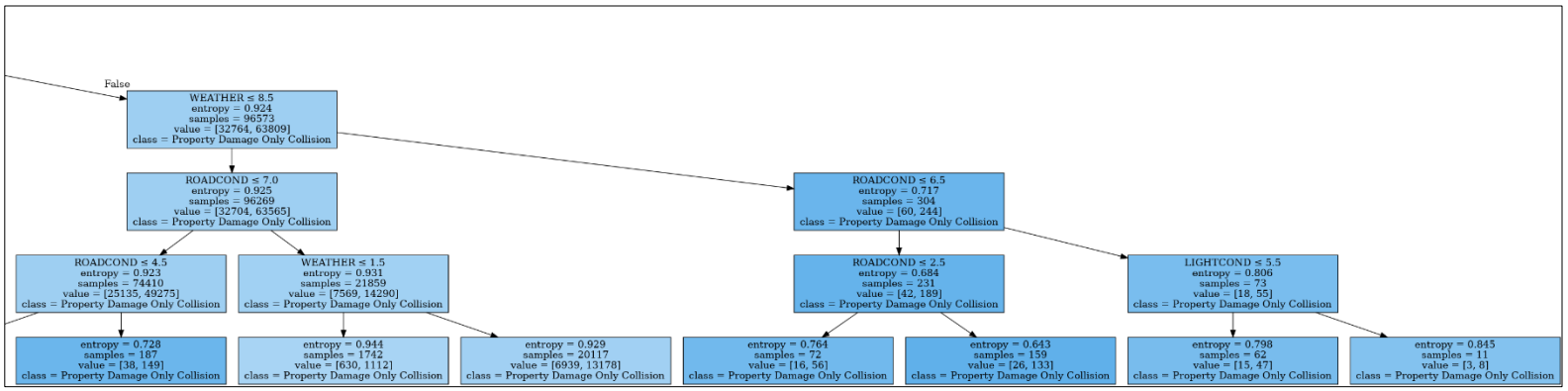

Figure 13

I decided to revise the target to SEVERITYCODE to create a confusion matrix of this model. This will be useful to make comparisons to the other classifiers (Figure 14). This function also returns the F-score which is a measurement of the algorithm's accuracy calculated from the precision and recall of the test. This also provides a better basis with which to compare other models. In the case of the DT model the weighted average F-score is 0.55.

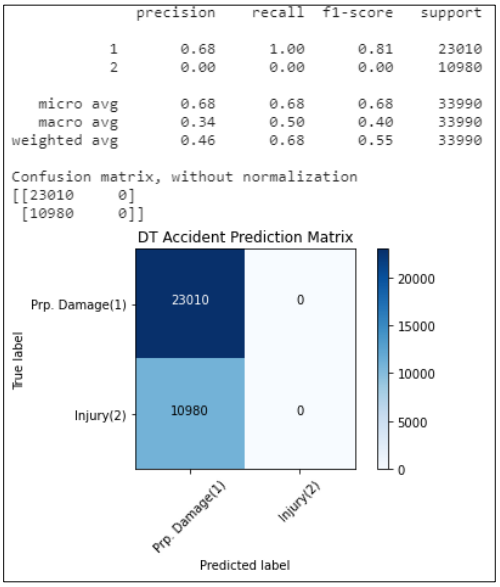

Figure 124

Logistic Regression

The Logistic Regression (LR) algorithm also returned a weighted F-score of 0.55 (Figure 15).

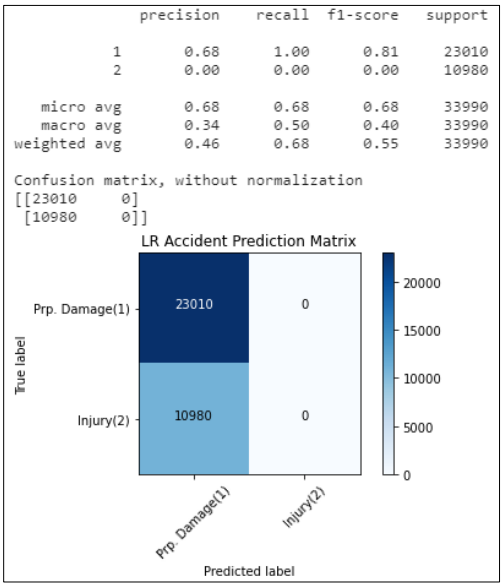

Figure 15

K-nearest Neighbor

The K-nearest Neighbor (KNN) algorithm returned a weighted F-score of 0.57 (Figure 16).

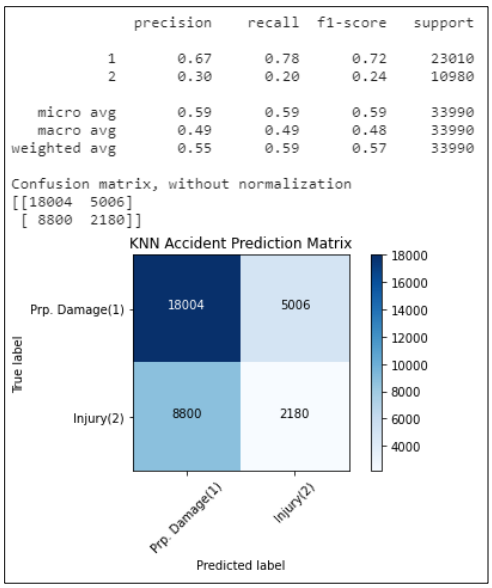

Figure 136

Support Vector Machine

The Support Vector Machine (SVM) algorithm returned a weighted F-score of 0.55 (Figure 17).

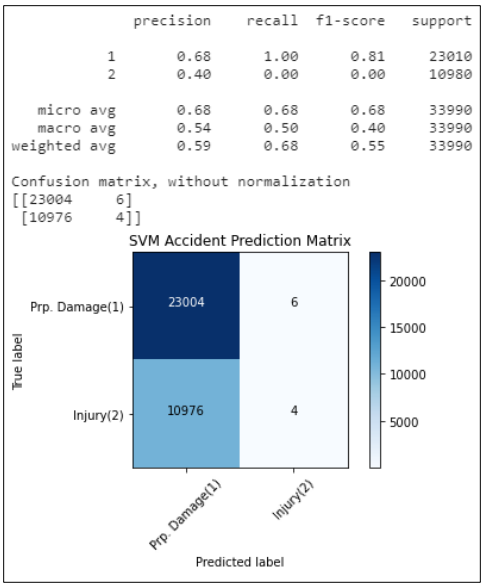

Figure 17

Results

After running confusion matrix functions on each model an obvious winner did not emerge. I reevaluated each model using the Sklearn metrics jaccard and F1-score. The report I was able to produce (Figure 18) revealed that Logistic Regression, SVM and Decision Tree had higher jaccard scores than K-nearest Neighbors and Logistic regression had a higher F-1 score than the other algorithms. I have concluded that for the dataset I produced to predict accident severity, Logistic Regression is the most accurate model to use.

Algorithm	Jaccard	F1-score	LogLoss
KNN	0.593822	0.566874	NA
Decision Tree	0.676964	0.546559	NA
SVM	0.676905	0.546748	NA
LogisticRegression	0.676964	0.676964	23.382

Figure 18