Machine Learning Model to Predict of Traffic Accident Severity in Seattle, WA

1. INTRODUCTION

1.1. Background

According to the Centers for Disease Control and Prevention (CDC), traffic accidents are a leading cause of death in the US with over 100 people dying every day. The National Highway Traffic Safety of the US Department of Transportation calculated that the total economic cost was $242 billion in 2010 alone. When quality-of-life valuations such as pain and decreased quality of life are considered, the total value of societal harm jumped to $836 billion - in addition to the immeasurable burden on the victims' families and friends.

1.2. Business Problem

The advent of advanced electronic, computer, and communication technologies provides an opportunity for seeking new remedies that can help drivers avoid traffic accidents - thereby preventing accident-related injuries and saving both lives and money. Historical data sources, such as police, hospital, and emergency medical service (EMS) records, are publicly available and can be used to get a better picture of each accident.

The goal of this study is to understand the risk factors and to predict the severity of traffic accidents using Machine Learning with Python.

1.3. Target Audience

This study may help public traffic and safety officials improve traffic policies, install traffic signs, and/or update public facilities such as street lighting at various locations.

This study may also help general population to understand risk factors of collisions leading to injuries and take the necessary precautions.

2. DATA

2.1. Data Source

We use the collision dataset published by the City of Seattle. It is a public data and has more than 200k data points from 2004 to present with 40 attributes. The metadata can be found here.

2.2. Data Understanding and Transformation

In the dataset, there are several attributes for accident severity:

Attribute	Description
SEVERITYCODE	A code that corresponds to the severity of the collision
SEVERITYDESC	A detailed description of the severity of the collision
INJURIES	The number of total injuries in the collision.
SERIOUSINJURIES	The number of serious injuries in the collision.
FATALITIES	The number of fatalities in the collision.
PERSONCOUNT	The total number of people involved in the collision.
PEDCOUNT	The number of pedestrians involved in the collision.
PEDCYLCOUNT	The number of bicycles involved in the collision.
VEHCOUNT	The number of vehicles involved in the collision.

Specifically, both SEVERITYCODE and SEVERITYDESC attributes refer to the same severity level. We select SEVERITYCODE as the dependent variable (target) that we are going to predict.

```
# Check SEVERITYCODE values
my_rawdata['SEVERITYCODE'].value_counts()

1     137776
2      58842
0      21656
2b      3111
3        352
Name: SEVERITYCODE, dtype: int64
```

```
# Check SVERITYDESC values
my_rawdata['SEVERITYDESC'].value_counts()

Property Damage Only Collision    137776
Injury Collision                   58842
Unknown                            21657
Serious Injury Collision            3111
Fatality Collision                   352
Name: SEVERITYDESC, dtype: int64
```

Attribute INCDATE shows the date of the incident and INCDTTM the date and time of the incident. However, many of the INCDTTM data only has the date component.

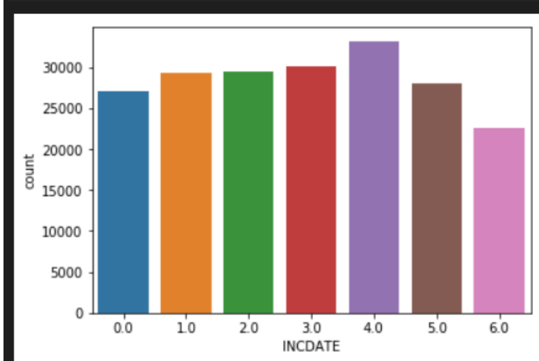
```
# Check INCDTTM values
my_rawdata['INCDTTM'].value_counts()

11/2/2006                103
10/8/2004                 98
10/3/2008                 92
11/5/2005                 85
1/2/2004                  80
                        ...
12/23/2006 7:40:00 PM      1
7/23/2008 6:30:00 PM       1
6/26/2013 7:52:00 PM       1
10/11/2006 4:57:00 PM      1
8/23/2011 8:00:00 PM       1
Name: INCDTTM, Length: 169669, dtype: int64
```

We convert the INCDATE data to the day of the week (Monday = 0, Tuesday = 1, etc.) and review the frequency of the accidents. There is no need to group INCDATE to weekdays and weekends.

```
# Convert the dates in INCDATE columnn to day of the week
my_rawdata['INCDATE'] = pd.to_datetime(my_rawdata['INCDATE'],format='%Y/%m/%d')
my_rawdata['INCDATE'] = my_rawdata['INCDATE'].dt.dayofweek
my_rawdata['INCDATE'] = my_rawdata['INCDATE'].astype(float)
```

```
# Check INCDATE values
ax = sns.countplot(x="INCDATE", data=my_rawdata)
```



Attribute UNDERINFL have four values: Y/N and 0/1. We believe there is a confusion how the data is collected. Some used “Y” when the driver involved in the accident was under the influence of drugs or alcohol, some other used 1. Some used “N” when the driver involved in the accident was not under the influence of drugs or alcohol, some other used 0. We convert the categorical values of the UNDERINFL data to 0 and 1.

```
my_rawdata['UNDERINFL'].value_counts()

N    104000
0     81676
Y      5399
1      4230
Name: UNDERINFL, dtype: int64
```

```
# Convert the values in UNDERINFL columnn: assign NaN and N to 0 and Y to 1
my_rawdata['UNDERINFL'] = my_rawdata['UNDERINFL'].replace(['Y', 'N'], ['1','0'])
my_rawdata['UNDERINFL'] = my_rawdata['UNDERINFL'].astype(float)
my_rawdata['UNDERINFL'].value_counts()

0.0    185676
1.0      9629
Name: UNDERINFL, dtype: int64
```

Attribute SPEEDING and INATTENTIONID only have “Y” and null. We believe null means “N” and we convert the values to 0 for null and 1 for “Y”.

```
# Check SPEEDING values
my_rawdata['SPEEDING'].value_counts()

Y    9936
Name: SPEEDING, dtype: int64
```

```
# Check INATTENTIONIND values
my_rawdata['INATTENTIONIND'].value_counts()

Y    30188
Name: INATTENTIONIND, dtype: int64
```

```
# Convert the values in INATTENTIONIND columnn: assign Y to 1 and else to 0
my_rawdata['INATTENTIONIND'] = my_rawdata['INATTENTIONIND'].apply(lambda x: 0 if x != "Y" else 1)
my_rawdata['INATTENTIONIND'] = my_rawdata['INATTENTIONIND'].astype(float)

# Convert the values in SPEEDING columnn: assign Y to 1 and else to 0
my_rawdata['SPEEDING'] = my_rawdata['SPEEDING'].apply(lambda x: 0 if x != "Y" else 1)
my_rawdata['SPEEDING'] = my_rawdata['SPEEDING'].astype(float)
```

We review other attributes such as WEATHER, ROADCOND, etc. and convert the categorical independent variables into numerical variables. We assign 99 for the “Unknown” and “Other” values, so that we can easily remove these data in the future.

```
# Convert the categorical values in WEATHER columnn: assign Unknown and Other to 99
my_rawdata['WEATHER'] = my_rawdata['WEATHER'].replace(['Clear', 'Raining', 'Overcast', 'Snowing', 'Fog/Smog/Smoke', 'Sleet/Hail/
Freezing Rain', 'Blowing Sand/Dirt', 'Severe Crosswind', 'Partly Cloudy', 'Blowing Snow', 'Unknown', 'Other'], [1,2,3,4,5,6,7,8,9,10,
99,99])
my_rawdata['WEATHER'].value_counts()

1.0     114807
2.0      34038
3.0      28556
99.0     15991
4.0        919
5.0        577
6.0        116
7.0         56
8.0         26
9.0         10
10.0         1
Name: WEATHER, dtype: int64
```

For our initial dataset, we select these nine attributes below as the independent variables (features):

LIGHTCOND: the light condition during the collision		
Original Values	*New Values*	*Frequency*
Daylight	1	119555
Dark - Street Lights On	2	50139
blank		26730
Unknown	99	13533
Dusk	3	6085
Dawn	4	2609
Dark - No Street Lights	5	1580
Dark - Street Lights Off	6	1239
Other	99	244
Dark - Unknown Lighting	7	24

ROADCOND: the condition of the road during the collision		
Original Values	*New Values*	*Frequency*
Dry	1	128660
Wet	2	48737
(blank)		26560
Unknown	99	15139
Ice	3	1232
Snow/Slush	4	1014
Other	99	136
Standing Water	5	119
Sand/Mud/Dirt	6	77
Oil	7	64

WEATHER: description of the weather conditions during the time of the collision.		
Original Values	*New Values*	*Frequency*
Clear	1	114807
Raining	2	34038
Overcast	3	28556
blank		26641
Unknown	99	15131
Snowing	4	919
Other	99	860
Fog/Smog/Smoke	5	577
Sleet/Hail/Freezing Rain	6	116
Blowing Sand/Dirt	7	56
Severe Crosswind	8	26
Partly Cloudy	9	10
Blowing Snow	10	1

JUNCTIONTYPE: category of junction at which collision took place		
Original Values	*New Values*	*Frequency*
Mid-Block (not related to intersection)	1	101823
At Intersection (intersection related)	2	69317
Mid-Block (but intersection related)	3	24412
blank		11979
Driveway Junction	4	11497
At Intersection (but not related to intersection)	5	2499
Ramp Junction	6	190
Unknown	99	21

INATTENTIONIND: whether or not collision was due to inattention		
Original Values	*New Values*	*Frequency*
Y	1	30188
blank	0	191550

UNDERINFL: whether or a driver involved was under the influence of drugs or alcohol.		
Original Values	*New Values*	*Frequency*
0	0	81676
1	1	4230
N	0	104002
Y	1	5399
blank	0	26431

SPEEDING: whether or not speeding was a factor in the collision		
Original Values	*New Values*	*Frequency*
Y	1	9936
blank	0	211802

INCDATE: the date of the incident		
Original Values	New Values	Frequency
Monday	0	30160
Tuesday	1	32583
Wednesday	2	32986
Thursday	3	33603
Friday	4	36556
Saturday	5	30932
Sunday	6	24918

ADDRTYPE: collision address type		
Original Values	*New Values*	*Frequency*
Alley	0	879
Block	1	145118
Intersection	2	72027
blank		3714

2.3. Create Initial Dataset

We first remove the records with SEVERITYCODE 0 (description "Unknown").

```
# Drop "Unknown" value from SEVERITYDESC column
my_rawdata = my_rawdata[my_rawdata.SEVERITYDESC != "Unknown"]

# Group all injuries to SEVERITYCODE=2
my_rawdata['SEVERITYCODE']= my_rawdata['SEVERITYCODE'].replace(['2b', '3'], '2')
my_rawdata['SEVERITYCODE'] = my_rawdata['SEVERITYCODE'].astype(float)
my_rawdata['SEVERITYCODE'].value_counts()
```

```
1.0    137776
2.0     62305
Name: SEVERITYCODE, dtype: int64
```

We create the initial dataset and remove the null values.

```
# Select attributes to be included in the datamodel
my_data = my_rawdata[['INATTENTIONIND', 'JUNCTIONTYPE', 'ROADCOND', 'WEATHER', 'UNDERINFL', 'INCDATE', 'LIGHTCOND', 'ADDRTYPE',
'SPEEDING', 'SEVERITYCODE']]
my_data = my_data.dropna()
my_data.shape
```

```
(188346, 10)
```

```
my_data.head()
```

	INATTENTIONIND	JUNCTIONTYPE	ROADCOND	WEATHER	UNDERINFL	INCDATE	LIGHTCOND	ADDRTYPE	SPEEDING	SEVERITYCODE
0	0.0	2.0	1.0	1.0	0.0	6.0	1.0	2.0	0.0	1.0
1	1.0	1.0	2.0	2.0	0.0	0.0	3.0	1.0	0.0	1.0
2	0.0	1.0	1.0	1.0	0.0	6.0	2.0	1.0	0.0	2.0
3	0.0	2.0	2.0	2.0	0.0	0.0	2.0	2.0	0.0	2.0
4	0.0	1.0	3.0	1.0	0.0	4.0	2.0	1.0	1.0	2.0

We know now that we have an imbalance dataset: 68% of the accidents are property damage only (no injuries involved).

```
# check if the dataset is balance
my_data['SEVERITYCODE'].value_counts()
```

```
1.0    127609
2.0     60737
Name: SEVERITYCODE, dtype: int64
```

2.4. Attribute (Feature) Selection

Some of the nine attributes in the initial dataset may not contribute to the accuracy of the model or may even decrease the model accuracy. In this step, we want to identify those unneeded/irrelevant attributes and remove them from the final dataset. This will also reduce training time because the dataset will be smaller.

We use Univariate Feature Selection method with SelectKBest from scikit-learn library.

```
# Feature Selection with Univariate Statistical Tests
from numpy import set_printoptions
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif

# load data
array = my_data.values
X = array[:,0:9]
Y = array[:,9]

# feature extraction
test = SelectKBest(score_func=f_classif, k=4)
fit = test.fit(X, Y)

# summarize scores
set_printoptions(precision=3)
print(fit.scores_)
features = fit.transform(X)

# summarize selected features
print(features[0:5,:])
```

```
[ 243.689 1377.797 3800.628 3811.59   469.701   53.431 3642.913 7484.274
  354.116]
[[1. 1. 1. 2.]
 [2. 2. 3. 1.]
 [1. 1. 2. 1.]
 [2. 2. 2. 2.]
 [3. 1. 2. 1.]]
```

We select the four best attributes based on its high scores and build the imbalanced dataset imb_df.

Attribute		Score
INATTENTIONIND		243.689
JUNCTIONTYPE		1377.797
ROADCOND	3rd	3800.628
WEATHER	2nd	3811.59
UNDERINFL		469.701
INCDATE		53.431
LIGHTCOND	4th	3642.913
ADDRTYPE	1st	7484.274
SPEEDING		354.116

```
imb_df = my_data[['ADDRTYPE', 'WEATHER', 'ROADCOND', 'LIGHTCOND', 'SEVERITYCODE']]
imb_df.head()
```

	ADDRTYPE	WEATHER	ROADCOND	LIGHTCOND	SEVERITYCODE
0	2.0	1.0	1.0	1.0	1.0
1	1.0	2.0	2.0	3.0	1.0
2	1.0	1.0	1.0	2.0	2.0
3	2.0	2.0	2.0	2.0	2.0
4	1.0	1.0	3.0	2.0	2.0

```
imb_df.shape
```

```
(188346, 5)
```

2.5. Create Final Dataset

To resolve the imbalance problem, we use the "undersampling" method where we randomly delete some of the data from the majority class in order to match the numbers with the minority class (60737 in this case).

bal_df:

```
# Shuffle dataset
shuffled_df = imb_df.sample(frac=1,random_state=52)

# Put all severity code = 2 class in a separate dataset
sev2_df = shuffled_df[shuffled_df['SEVERITYCODE'] == 2]


# Randomly select 62305 observations from the majority class (severity code = 1)
sev1_df = shuffled_df[shuffled_df['SEVERITYCODE'] == 1].sample(n=60737,random_state=25)

# Concatenate both dataframes again
bal_df = pd.concat([sev1_df, sev2_df])

#plot the dataset after the undersampling
plt.figure(figsize=(8, 3))
sns.countplot('SEVERITYCODE', data=bal_df)
plt.title('Balanced Classes')
plt.show()
```

We will also use Balanced Bagging Classifier, which is a Bagging Classifier with an additional step to randomly selecting samples from the majority class and deleting them from the training dataset until a more balanced distribution is reached.

2.6. Create Train and Test Dataset

We assign 80% of the entire data for training and the 20% for testing.

```
# Train/test dataset
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: (150676, 4) (150676,)
Test set: (37670, 4) (37670,)
```

3. MODELING

In this study, we consider several machine learning algorithms to build and train a model using historical accident records and to classify factors leading to injury/non-injury accidents.

3.1. Support Vector Machine (SVM)

```
clf = svm.SVC(kernel='rbf', gamma='auto')
clf.fit(x_train, y_train)
yhat = clf.predict(x_test)
```

3.2. K-Nearest Neighbor (KNN)

```
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)
```

3.3. Decision Tree

```
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics

DecTree2 = DecisionTreeClassifier(criterion="entropy", max_depth = 10)
DecTree2.fit(X_train,Y_train)
predTree2 = DecTree2.predict(X_test)
```

3.4. Logistic Regression

```
xLR = np.asarray(x).astype('float')
yLR = np.asarray(y).astype('float')

# Normalize the dataset
X = preprocessing.StandardScaler().fit(xLR).transform(xLR)

# Train/test dataset
from sklearn.model_selection import train_test_split
xLR_train, xLR_test, yLR_train, yLR_test = train_test_split( xLR, yLR, test_size=0.2, random_state=4)

# Modeling

LR = LogisticRegression(C=0.01, solver='liblinear').fit(xLR_train,yLR_train)

yLRhat = LR.predict(xLR_test)
yLRhat_prob = LR.predict_proba(xLR_test)
```

3.5. Balanced Bagging Classifier

```
# bagged decision trees on an imbalanced classification problem
from imblearn.ensemble import BalancedBaggingClassifier
from sklearn.metrics import balanced_accuracy_score
#Create an object of the classifier.
bbc = BalancedBaggingClassifier(base_estimator=DecisionTreeClassifier(),
                                sampling_strategy='all',
                                replacement=False,
                                random_state=133)


#Train the classifier.
bbc.fit(x_train, y_train)
preds = bbc.predict(x_test)
```

4. EVALUATION

We will evaluate the effectiveness of each model using several measures, but we will use F1-score and Jaccard index to select the best model. We will also check how imbalanced datasets affect the accuracy result.

```
# F1 score for accuracy
f1_score(y_test, yhat, average='weighted')
```

```
# Jaccard index for accuracy
jaccard_score(y_test, yhat, average='weighted')
```

5. RESULTS AND DISCUSSION

We learn that standard accuracy measurement does not work because the disproportionate ratio of observations in each class and we show it in our Jupyter notebook. The 0.54 F1-score of SVM algorithm with imbalanced dataset for example, is quite close to the 0.59 of SVM with Undersampling. However, this result with imbalanced dataset is misleading because it is skewed to one class only (F1-score of class 2 is zero). This is not the case with SVM with Undersampling – the F1-score of class 2 is 0.55!

SVM with imbalanced dataset

```
print (classification_report(y_test, yhat))
              precision    recall  f1-score   support

         1.0       0.68      1.00      0.81     25429
         2.0       0.00      0.00      0.00     12241

    accuracy                           0.68     37670
   macro avg       0.34      0.50      0.40     37670
weighted avg       0.46      0.68      0.54     37670
```

SVM with Undersampling

```
print (classification_report(Y_test, Yhat))
              precision    recall  f1-score   support

         1.0       0.58      0.73      0.64     12097
         2.0       0.64      0.48      0.55     12198

    accuracy                           0.60     24295
   macro avg       0.61      0.60      0.59     24295
weighted avg       0.61      0.60      0.59     24295
```

The table below shows the results of each algorithm with the balanced datasets:

Algorithm	F1-score		Jaccard index	
	BBC	UND	BBC	UND
SVM	0.650	0.595	0.494	0.425
KNN	0.585	0.569	0.469	0.398
Decision Tree	0.649	0.594	0.492	0.424
Log Regression	0.650	0.593	0.494	0.424

From F1-scores, we see that all algorithms show significant improvement (~ +10% increase) using Balanced Bagging Classifier compared to Undersampling, with the exception of KNN (only 3% increase). Interestingly, all algorithm's Jaccard-indices also improve significantly (~ +16% increase) with KNN being the most (+18% increase).

With Balanced Bagging Classifier, there is almost no difference in F1-score and Jaccard-indices for SVC, Decision Tree and Log Regression. We will now use balanced_accuracy_score from sklear.metrics that deal with imbalanced datasets.

```
# bagged decision trees on an imbalanced classification problem
from imblearn.ensemble import BalancedBaggingClassifier
from sklearn.metrics import balanced_accuracy_score
from sklearn.svm import SVC
#Create an object of the classifier.
bbc = BalancedBaggingClassifier(base_estimator=SVC(),
                                sampling_strategy='all',
                                replacement=False,
                                random_state=125)


#Train the classifier.
bbc.fit(x_train, y_train)
preds = bbc.predict(x_test)

print("SVC's Accuracy after balancing the dataset: ", balanced_accuracy_score(y_test, preds))
SVC's Accuracy after balancing the dataset:  0.6035814026999652
```

```
#Create an object of the classifier.
bbc = BalancedBaggingClassifier(base_estimator=DecisionTreeClassifier(),
                                sampling_strategy='all',
                                replacement=False,
                                random_state=3)


#Train the classifier.
bbc.fit(x_train, y_train)
preds = bbc.predict(x_test)

print("Decision Trees's Accuracy after balancing the dataset: ", balanced_accuracy_score(y_test, preds))
Decision Trees's Accuracy after balancing the dataset:  0.6034016219585482
```

```
#Create an object of the classifier.
bbc = BalancedBaggingClassifier(base_estimator=LogisticRegression(),
                                sampling_strategy='all',
                                replacement=False,
                                random_state=13)


#Train the classifier.
bbc.fit(x_train, y_train)
preds = bbc.predict(x_test)

print("Log Regression's Accuracy after balancing the dataset: ", balanced_accuracy_score(y_test, preds))
Log Regression's Accuracy after balancing the dataset:  0.6038704818051587
```

We see here that the Balanced Bagging Classifier with Logistic Regression as base estimator has a slightly better performance relative to SVC and Decision Tree. Similar to F1-score and Jaccard-index, KNN's balanced accuracy score is significantly lower than the other three algorithms.

```python
# bagged decision trees on an imbalanced classification problem
from imblearn.ensemble import BalancedBaggingClassifier
from sklearn.metrics import balanced_accuracy_score
#Create an object of the classifier.
bbc = BalancedBaggingClassifier(base_estimator=KNeighborsClassifier(),
                                sampling_strategy='all',
                                replacement=False,
                                random_state=133)


#Train the classifier.
bbc.fit(x_train, y_train)
preds = bbc.predict(x_test)

print("KNN's Accuracy after balancing the dataset: ", balanced_accuracy_score(y_test, preds))
```

```
KNN's Accuracy after balancing the dataset:  0.5174144416073909
```

6. CONCLUSION

From the results above, we can conclude that how we handle imbalanced dataset plays a crucial role in the performance of the algorithm. With our datasets, Balanced Bagging Classifier significantly outperforms Undersampling.

There are many other methods to handle imbalanced dataset, such as SMOTE algorithm, Random Forest, cost-sensitive training, etc. Further study should look at those methods.

Another direction for further study is to look at the parameters to fine tune the performance. Within Balanced Bagging Classifier for example, we can adjust the ratio of the number of samples in the minority class over the number of samples in the majority class after resampling.