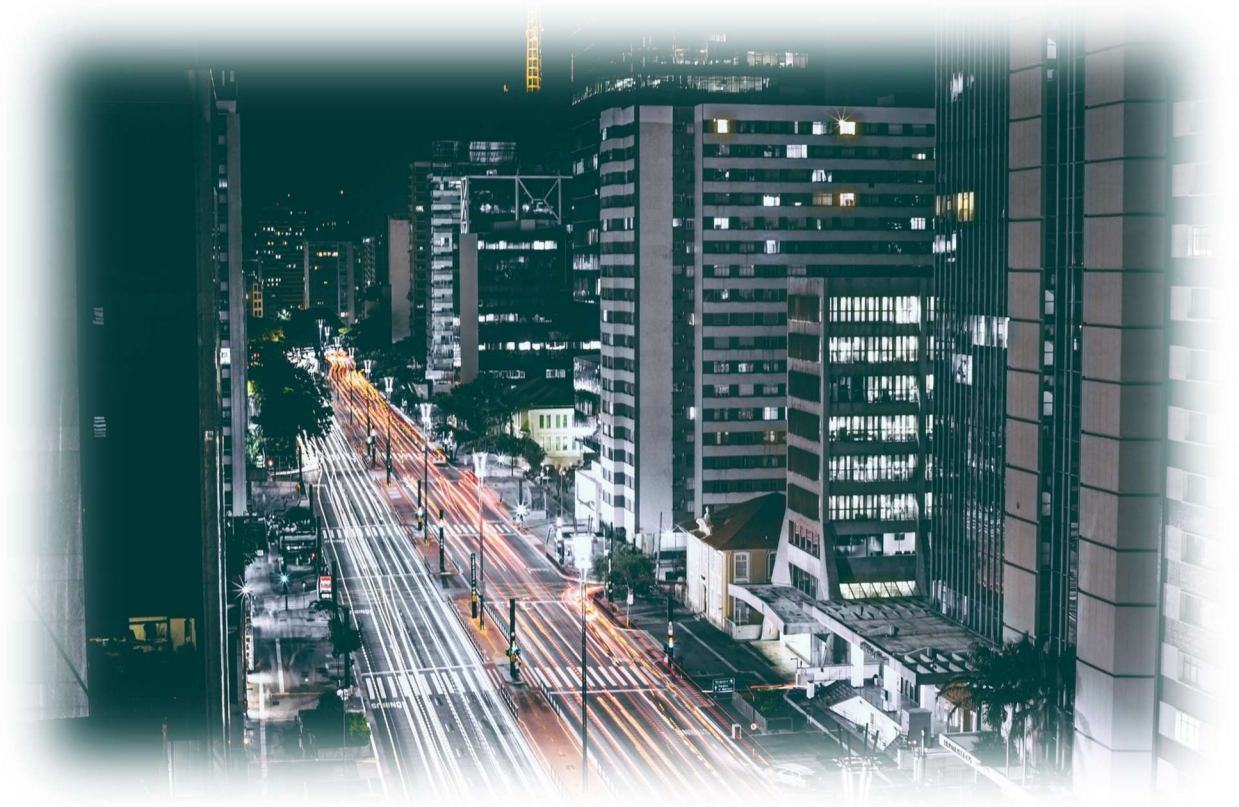
Coursera Capstone Project

Car Accident Severity

2 September 2020

Michael Fretwell

Table of Contents



Introduction

According to the Association for Safe International Road Travel (ASIRD) approximately 38,000 people die in road crashes in the United States each year, with the fatality rate being 12.4 deaths per 100,000 inhabitants. An additional 4.4 million people are injured seriously enough to require medical attention. And road crashes are the leading cause of death in the U.S. for people aged 1-54.

The scope of this project will include identifying factors (such as speed, time of day, and weather conditions) that contribute to accidents. And by use of regression classification modeling, the results will be able to provide measures of probability, given various conditions, that can be considered as risk of accidents. This information will be useful to those in the traffic safety industry for the purpose of implementing actions that would reduce the risks of accidents. It will also be of use to the insurance industry.

Data

Given the short timeframe for the development of this project, two weeks to plan and complete the project, the Seattle "Collisions – All Years" dataset was selected for use. It is produced by the Seattle SDOT Traffic Management Division, Traffic Records Group. Point of contact for the data is the SDOT GIS Analyst at DOT_IT_GIS@seattle.gov.

This dataset is smaller in size with fewer features than comparable datasets that are available (such as the US Accidents Countrywide Traffic Accident dataset by Sobhan Moosavi at the Ohio State University). Later, this project can be expanded upon to include countrywide data and additional features available in other datasets.

The Seattle dataset "Collisions – All Years" consists of 194,673 observations with accident data from January 2004 to 19 May 2020 and it contains 37 attributes. The dataset was prepared by eliminating duplicate or unnecessary columns, and normalizing the values. The label for the models is "SeverityCode" which indicates the severity of the collision ("1" for property damage or "2" for injury). The label contains significantly unbalanced data, so the dataset was balanced to preclude bias in the model.

Methodology

After creating a link to the dataset comma separated values (CSV) file and reading into a Pandas dataframe, duplicate or unnecessary features were removed (see Figure 1).

X	Y	INCKEY
COLDETKEY	REPORTNO	STATUS
SDOTCOLNUM	LOCATION	SEVERITYCODE.1
SEVERITYDESC	ST_COLDESC	EXCEPTRSNCODE
EXCEPTRSNDESC	ST_COLCODE	SDOT_COLCODE
SDOT_COLDESC	INTKEY	INCDATE
OBJECTID	SEGLANEKEY	CROSSWALKKEY

Figure 1: Removed Features

Missing Values.

Next, the dataset was searched for missing values (see Figure 2).

```
In [185]: #Look for Missing Values.

          df.isnull().sum()

Out[185]: SEVERITYCODE          0
          ADDRTYPE           1926
          COLLISIONTYPE      4904
          PERSONCOUNT           0
          PEDCOUNT              0
          PEDCYLCOUNT           0
          VEHCOUNT              0
          INCDTTM               0
          JUNCTIONTYPE       6329
          INATTENTIONIND   164868
          UNDERINFL          4884
          WEATHER            5081
          ROADCOND           5012
          LIGHTCOND          5170
          PEDROWNOTGRNT    190006
          SPEEDING         185340
          HITPARKEDCAR          0
          dtype: int64
```

Figure 2 – Missing Values

An analysis was done for each of the features that had missing values and appropriate steps were applied to resolve the missing fields. In most cases, it was appropriate to

replace the missing values with the most frequently occurring value. For example, the vast majority of the ADDRTYPE values was "Block" (see Figure 3).

```
In [186]: #Evaluate ADDRTYPE.

          df['ADDRTYPE'].value_counts()

Out[186]: Block           126926
          Intersection     65070
          Alley              751
          Name: ADDRTYPE, dtype: int64

In [187]: #Replace the missing ADDRTYPE values by the most frequent.

          df['ADDRTYPE'].replace(np.nan, "Block", inplace=True)
```

Figure 3 – Evaluation of Feature ADDRTYPE

In some cases, as with SPEEDING, it was necessary to convert existing field values to binary values ("0" and "1") (see Figure 4).

```
In [204]: #Evaluate SPEEDING.

          df['SPEEDING'].value_counts()

Out[204]: Y    9333
          Name: SPEEDING, dtype: int64

In [205]: #Update SPEEDING values to "0" and "1".

          df['SPEEDING'].replace(np.nan, 0, inplace=True)
          df['SPEEDING'].replace("Y", 1, inplace=True)
          df['SPEEDING'].replace("N", 0, inplace=True)
```

Figure 4 – Evaluation of Feature SPEEDING

Convert Categorical Features to Numbers.

The next step in dataset processing was to convert categorical features to numbers. Figure 5 shows an example of the process used for feature "COLLISIONTYPE".

```
In [210]: #COLLISIONTYPE

          df['COLLISIONTYPE'].value_counts()

Out[210]: Parked Car    52891
          Angles        34674
          Rear Ended    34090
          Other         23703
          Sideswipe     18609
          Left Turn     13703
          Pedestrian     6608
          Cycles         5415
          Right Turn     2956
          Head On        2024
          Name: COLLISIONTYPE, dtype: int64

In [211]: df['COLLISIONTYPE'].replace(to_replace=['Parked Car', 'Angles', 'Rear Ended', 'Other', 'Sideswipe',\
                                                  'Left Turn', 'Pedestrian', 'Cycles', 'Right Turn', 'Head On',\
                                                  ], value=[0,1,2,3,4,5,6,7,8,9], inplace=True)
```

Figure 5 – Converting Categorical Features to Numbers

Convert Object Data Types to Integers.

Any features that were stored as objects were then converted to integers (see Figure 6).

```
In [223]: #Check Data Types.

          df.dtypes

Out[223]: SEVERITYCODE       int64
          ADDRTYPE           int64
          COLLISIONTYPE      int64
          PERSONCOUNT        int64
          PEDCOUNT           int64
          PEDCYLCOUNT        int64
          VEHCOUNT           int64
          INCDTTM           object
          JUNCTIONTYPE       int64
          INATTENTIONIND     int64
          UNDERINFL         object
          WEATHER            int64
          ROADCOND           int64
          LIGHTCOND          int64
          PEDROWNOTGRNT      int64
          SPEEDING           int64
          HITPARKEDCAR      object
          dtype: object

In [224]: #Convert Objects to Integers.

          df['UNDERINFL'] = pd.to_numeric(df['UNDERINFL'])
          df['WEATHER'] = pd.to_numeric(df['WEATHER'])
          df['HITPARKEDCAR'] = pd.to_numeric(df['HITPARKEDCAR'])
```

Figure 6 – Converting Object Features to Integers

Balance Dataset.

The label, or target, for the models will be “SEVERITYCODE” which indicates the severity of the collision – “1” for property damage, or “2” for injury. The label contained extremely unbalanced data (see Figure 7).

```
In [226]: #Check for Balanaced Dataset.

target_count = df.SEVERITYCODE.value_counts()
print('Class 1:', target_count[1])
print('Class 2:', target_count[2])
print('Proportion:', round(target_count[1] / target_count[2], 2), ': 1')

target_count.plot(kind='bar', title='Count (target)')
plt.ylabel('Count')
plt.xlabel('Severity Code');
```

```
Class 1: 136485
Class 2: 58188
Proportion: 2.35 : 1
```

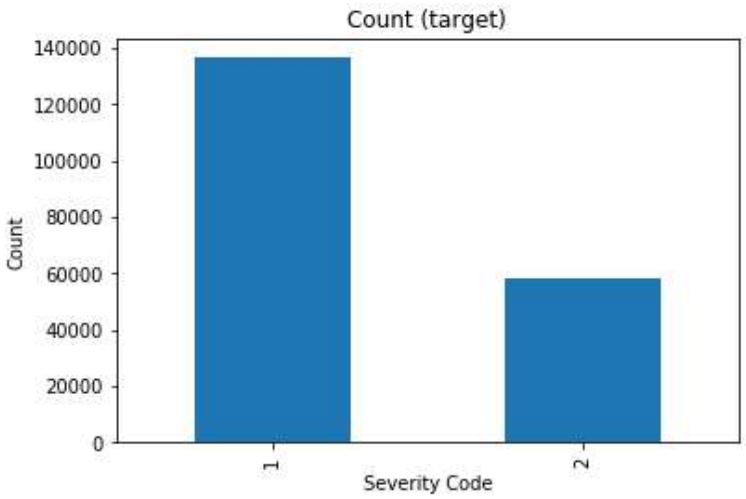


Figure 7 – Checking for Balanced Dataset

Steps were processed to balance the dataset and a final check was done to verify the results (see Figures 8 and 9).

```
In [227]: #Balance Dataset.

          SeverityCode1 = df[df['SEVERITYCODE']==1]
          SeverityCode2 = df[df['SEVERITYCODE']==2]

In [228]: SeverityCode1.shape,SeverityCode2.shape

Out[228]: ((136485, 17), (58188, 17))

In [229]: SeverityCode1 = SeverityCode1.sample(SeverityCode2.shape[0])
          SeverityCode1.shape

Out[229]: (58188, 17)

In [230]: df = SeverityCode2.append(SeverityCode1,ignore_index=True)
          df.shape

Out[230]: (116376, 17)
```

Figure 8 – Balancing the Dataset

```
In [231]: #Recheck for Balanaced Dataset.

          target_count = df.SEVERITYCODE.value_counts()
          print('Class 1:', target_count[1])
          print('Class 2:', target_count[2])
          print('Proportion:', round(target_count[1] / target_count[2], 2), ': 1')

          target_count.plot(kind='bar', title='Count (target)')
          plt.ylabel('Count')
          plt.xlabel('Severity Code');

Class 1: 58188
Class 2: 58188
Proportion: 1.0 : 1
```

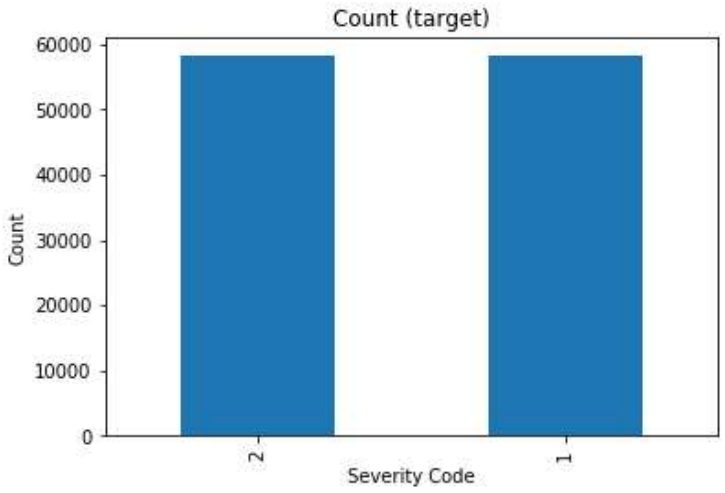


Figure 9 – Rechecking for Balanced Dataset

Feature Engineering.

The next steps involved some feature engineering to attempt to extract more useful information from the dataset. First, the INCDTTM feature was converted from an object to a datetime data type to allow for calculations (see Figure 10).

```
In [232]: #Convert to Date Time Object.

          df['INCDTTM'] = pd.to_datetime(df['INCDTTM'])
          df.head()
```

Out[232]:

	SEVERITYCODE	ADDRTYPE	COLLISIONTYPE	PERSONCOUNT	PEDCOUNT	PEDCYLCOUNT	VEHCOUNT	INCDTTM
0	2	1	1	2	0	0	2	2013-03-27 14:54:00
1	2	1	1	2	0	0	2	2004-01-28 08:04:00
2	2	1	7	3	0	1	1	2020-04-15 17:47:00
3	2	1	1	2	0	0	2	2006-03-20 15:49:00
4	2	0	9	2	0	0	2	2013-03-31 02:09:00

Figure 10 – Converting INCDTTM to Datetime Format

An analysis was done to evaluate the accidents by the day of the week they occurred. The results show mostly minor variations in the number of accidents by day of the week, but there was a noticeable drop in the number of accidents occurring on Saturdays (see Figure 11).

In [612]:
```
#Evaluate Day of Week that Accidents Occured.

df['dayofweek'] = df['INCDTTM'].dt.dayofweek

df_dayofweek = df.groupby(['dayofweek'])['SEVERITYCODE'].value_counts()
df_dayofweek
```

Out[612]:
```
dayofweek  SEVERITYCODE
0          2               7973
           1               7820
1          2               8731
           1               8474
2          2               8757
           1               8433
3          2               9018
           1               8638
4          1               9670
           2               9559
5          1               8287
           2               8047
6          1               6866
           2               6103
Name: SEVERITYCODE, dtype: int64
```

In [613]:
```
#Unstack Dataframe for Plotting.

df_dayofweek = df_dayofweek.unstack()
```

In [614]:
```
#Create Plot.

df_dayofweek.plot(kind='bar', figsize=(6,5))
plt.ylabel('count')
```

Out[614]:
```
Text(0, 0.5, 'count')
```

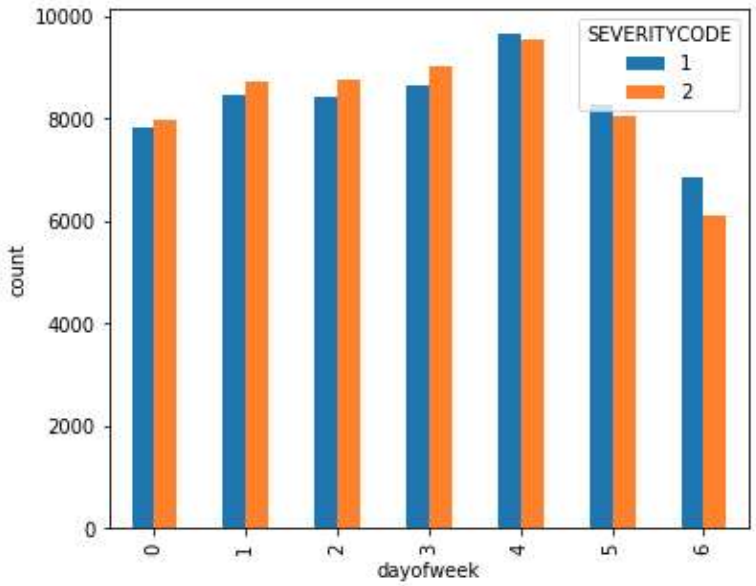


Figure 11 – Accidents by Day of the Week

An analysis was also done to evaluate the number of accidents based on time of day. Six time of day bins were created (see Figure 12).

Time of Day	Bin
0000 to 0400	0
0400 to 0800	1
0800 to 1200	2
1200 to 1600	3
1600 to 2000	4
2000 to 0000	5

Figure 12 – Time of Day Bins

The data was then grouped based on the time of day bins and plotted to graphically display the results. There were slight variations between all the time of day bins, but the time periods 0400 to 0800 and 2000 to 0000 had significantly fewer accident occurrences (see Figure 13).

In [617]:
```
#Evaluate Time of Day that Accidents Occured.

df_timeofday = df.groupby(['timeofday'])['SEVERITYCODE'].value_counts()
df_timeofday
```

Out[617]:
```
timeofday  SEVERITYCODE
0          1               14059
           2               11574
1          2                6352
           1                5794
2          1               10191
           2                9936
3          2               14208
           1               12965
4          2               11867
           1               10346
5          1                4833
           2                4251
Name: SEVERITYCODE, dtype: int64
```

In [618]:
```
#Unstack Dataframe for Plotting.

df_timeofday = df_timeofday.unstack()
df_timeofday
```

Out[618]:

SEVERITYCODE	1	2
timeofday		
0	14059	11574
1	5794	6352
2	10191	9936
3	12965	14208
4	10346	11867
5	4833	4251

In [619]:
```
#Create Plot.

df_timeofday.plot(kind='bar', figsize=(6,5))
plt.ylabel('count')
```

Out[619]:
```
Text(0, 0.5, 'count')
```

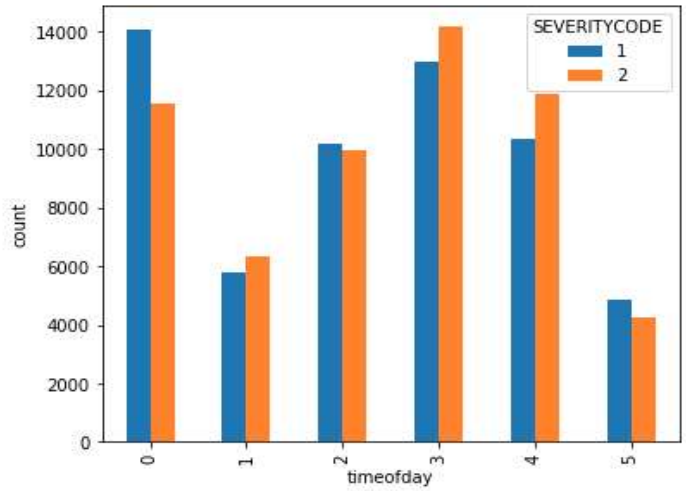


Figure 13 – Accidents by Time of Day

Feature Selection.

Figure 14 shows the features that were selected for use in the models.

ADDRTYPE	COLLISIONTYPE	PERSONCOUNT
PEDCOUNT	PEDCYLCOUNT	VEHCOUNT
JUNCTIONTYPE	INATTENTIONIND	UNDERINFL
WEATHER	ROADCOND	LIGHTCOND
PEDROWNOTGRNT	SPEEDING	HITPARKEDCAR
dayofweek	timeofday	

Figure 14 – Selected Features

Standardize Data / Train Test Split.

Since all the data in the features had been manually converted to numerical data, it was uncertain if using the sklearn.preprocessing package to further standardize the data would have an impact on model accuracy. So, the models were run both with and without applying preprocessing. The variances were minor and are shown in the Model Evaluation section of this report.

The sklearn.model_selection train_test_split package was used to split the dataset into training and testing groups. A test size of 20 percent and random state of 4 were used (see Figure 15).

```
In [243]: 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: (93100, 17) (93100,)
          Test set: (23276, 17) (23276,)
```

Figure 15 – Train Test Split

Modeling

Since the objective of the models will be to predict labels (severity of accident), classification models are appropriate for use. I chose K Nearest Neighbor, Decision Tree, Support Vector Machine, and Logistic Regression. One of the advantages of classification models is they predict values as probabilities which can be interpreted as likelihood or confidence in the accuracy of the prediction.

K Nearest Neighbor (KNN).

The KNN model was initially run with a K value of 7. (see Figure 16).

```
In [88]: #Begin with K = 7.

         k = 7

         #Train Model and Predict.

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

Out[88]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                              metric_params=None, n_jobs=None, n_neighbors=7, p=2,
                              weights='uniform')

In [89]: yhat = neigh.predict(X_test)
         yhat[0:5]

Out[89]: array([2, 2, 2, 1, 2], dtype=int64)

In [90]: 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:  0.7388829215896885
         Test set Accuracy:  0.6796270836913559
```

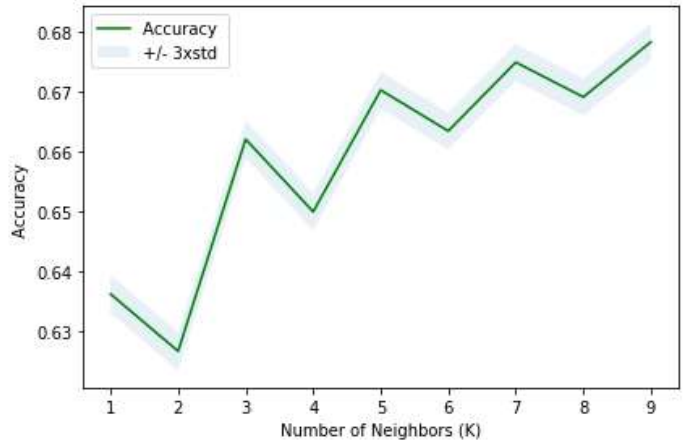
Figure 16 – Initial KNN Model

A for loop was then run to determine the best K value. This check was run numerous times. Sometimes the results validated the value of 7 as producing the best accuracy score other times it produced 9 as the best value. For the purposes of the project, KNN 9 was used (see Figure 17).

In [665]:
```
#Determine Best K.

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
```

Out[665]:
```
array([0.64, 0.63, 0.66, 0.65, 0.67, 0.66, 0.67, 0.67, 0.68])
```

In [666]:
```
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 Neighbors (K)')
plt.tight_layout()
plt.show()
```



In [667]:
```
print( "The best accuracy was with", mean_acc.max(), "with k=", mean_acc.argmax()+1)
```

```
The best accuracy was with 0.678381165148651 with k= 9
```

Figure 17 – Determining Best K Value

The KNN model was then rerun with a K value of 9 (note that, when rounded to two decimal places, there was no difference in the accuracy scores between K=7 and K=9) (see Figure 18).

```
In [170]: #Rerun Model if Required.
          #When data standardization is applied, the best K is 9.
          #When data standardization is not applied, the best K is usually 7 (sometimes it is 9).

          k = 9

          #Train Model and Predict.

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

Out[170]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                               metric_params=None, n_jobs=None, n_neighbors=9, p=2,
                               weights='uniform')

In [171]: #Rerun Prediction if Required.

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

Out[171]: array([1, 1, 2, 1, 2], dtype=int64)

In [172]: #Compute Accuracy Scores.

          KNN_F1 = round(f1_score(y_test, yhat),2)
          KNN_Jaccard = round(jaccard_similarity_score(y_test, yhat),2)
          KNN_LogLoss = round(log_loss(y_test, yhat),2)


          print("K Nearest Neighbor's F1 Score: ", KNN_F1)
          print("K Nearest Neighbor's Jaccard Score: ", KNN_Jaccard)
          print("K Nearest Neighbor's LogLoss Score: ", KNN_LogLoss)

          K Nearest Neighbor's F1 Score:  0.67
          K Nearest Neighbor's Jaccard Score:  0.68
          K Nearest Neighbor's LogLoss Score:  17.31
```

Figure 18 – Rerunning KNN with K=9

Decision Tree.

The decision tree model was initially run with a max depth setting of 6. Comparisons were then run with settings of 5, 7, and 8. A max depth setting of 7 produced slightly better Jaccard scoring, so that is the setting used in the final model (see Figure 19).

```
In [175]: DTree = DecisionTreeClassifier(criterion="entropy", max_depth = 7)

In [176]: #Train Model.

          DTree.fit(X_train,y_train)

Out[176]: DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='entropy',
                                 max_depth=7, 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='deprecated',
                                 random_state=None, splitter='best')

In [177]: #Predict.

          PredTree = DTree.predict(X_test)

In [178]: print (PredTree [0:5])
          print (y_test [0:5])

          [2 1 2 1 2]
          93822     1
          109172    1
          32284     2
          29807     2
          5015      2
          Name: SEVERITYCODE, dtype: int64

In [179]: #Compute Accuracy Scores.

          DecTree_F1 = round(f1_score(y_test, PredTree),2)
          DecTree_Jaccard = round(jaccard_similarity_score(y_test, PredTree),2)
          DecTree_LogLoss = round(log_loss(y_test, PredTree),2)


          print("DecisionTree's F1 Score: ", DecTree_F1)
          print("DecisionTree's Jaccard Score: ", DecTree_Jaccard)
          print("DecisionTree's LogLoss Score: ", DecTree_LogLoss)

          DecisionTree's F1 Score:  0.67
          DecisionTree's Jaccard Score:  0.71
          DecisionTree's LogLoss Score:  17.31
```

Figure 19 – Decision Tree

Support Vector Machine (SVM).

The SVM model was run with a kernel setting of radial basis function (RBF) (see Figure 20).

```
In [181]: clf = svm.SVC(kernel='rbf')
          clf.fit(X_train, y_train)

Out[181]: SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
              decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
              max_iter=-1, probability=False, random_state=None, shrinking=True,
              tol=0.001, verbose=False)

In [182]: yhat = clf.predict(X_test)
          yhat [0:5]

Out[182]: array([1, 1, 2, 1, 2], dtype=int64)

In [183]: #Compute Accuracy Scores.

          SVM_F1 = round(f1_score(y_test, yhat),2)
          SVM_Jaccard = round(jaccard_similarity_score(y_test, yhat),2)
          SVM_LogLoss = round(log_loss(y_test, yhat),2)


          print("Support Vector Machine's F1 Score: ", SVM_F1)
          print("Support Vector Machine's Jaccard Score: ", SVM_Jaccard)
          print("Support Vector Machine's LogLoss Score: ", SVM_LogLoss)

          Support Vector Machine's F1 Score:  0.68
          Support Vector Machine's Jaccard Score:  0.69
          Support Vector Machine's LogLoss Score:  17.31
```

Figure 20 – SVM Model

Logistic Regression.

The logistic regression model was run with a solver setting of "liblinear" (see Figure 21).

```
In [278]: from sklearn.linear_model import LogisticRegression
          LR = LogisticRegression(C=0.01, solver='liblinear').fit(X_train,y_train)
          LR

Out[278]: LogisticRegression(C=0.01, class_weight=None, dual=False, fit_intercept=True,
                             intercept_scaling=1, l1_ratio=None, max_iter=100,
                             multi_class='auto', n_jobs=None, penalty='l2',
                             random_state=None, solver='liblinear', tol=0.0001, verbose=0,
                             warm_start=False)

In [279]: yhat = LR.predict(X_test)
          yhat[0:5]

Out[279]: array([1, 1, 2, 2, 1], dtype=int64)

In [280]: #Compute Accuracy Scores

          LR_F1 = round(f1_score(y_test, yhat),2)
          LR_Jaccard = round(jaccard_similarity_score(y_test, yhat),2)
          LR_LogLoss = round(log_loss(y_test, yhat),2)

          print("Logistic Regression's F1 Score: ", LR_F1)
          print("Logistic Regression's Jaccard Score: ", LR_Jaccard)
          print("Logistic Regression's Log Loss: ", LR_LogLoss)

          Logistic Regression's F1 Score:  0.68
          Logistic Regression's Jaccard Score:  0.67
          Logistic Regression's Log Loss:  17.31
```

Figure 21 – Logistic Regression Model

Results

Accuracy Scoring Metrics.

The accuracy of the models was evaluated using the Jaccard, F-1, and LogLoss scoring methods from the sklearn metrics library.

As mentioned in the Dataset Preprocessing section of this report, the models were trained both with and without the dataset being standardized with the sklearn preprocessing package. The test score summaries are displayed in Figures 22 and 23. As the figures depict, there were minor differences between the two versions of the dataset, with the one with preprocessing showing slightly better results. And, in both cases, the decision tree model had slightly better Jaccard scores (F1 and LogLoss scores were all pretty much the same).

Note that for KNN, most of the time the best K without data standardization was 7 (sometimes it was 9) and with data standardization it was 9. However, the accuracy scores were the same given both scenarios.

Algorithm	Jaccard	F1-Score	LogLoss
KNN	0.68	0.67	17.31
Decision Tree	0.70	0.67	17.31
SVM	0.69	0.67	17.31
Logistic Regression	0.66	0.67	17.31

Figure 22 – Accuracy Scores (without dataset preprocessing standardization)

Algorithm	Jaccard	F1-Score	LogLoss
KNN	0.68	0.67	17.31
Decision Tree	0.71	0.67	17.31
SVM	0.69	0.68	17.31
Logistic Regression	0.67	0.68	17.31

Figure 23 – Accuracy Scores (with dataset preprocessing standardization)

Based on the results of the scoring matrices, we can determine the decision tree model is likely the best for making out of sample predictions.

Confusion Matrices.

For a visual comparison of the models, confusion matrices were created (with dataset preprocessing standardization) to compare the results (see Figure 24).

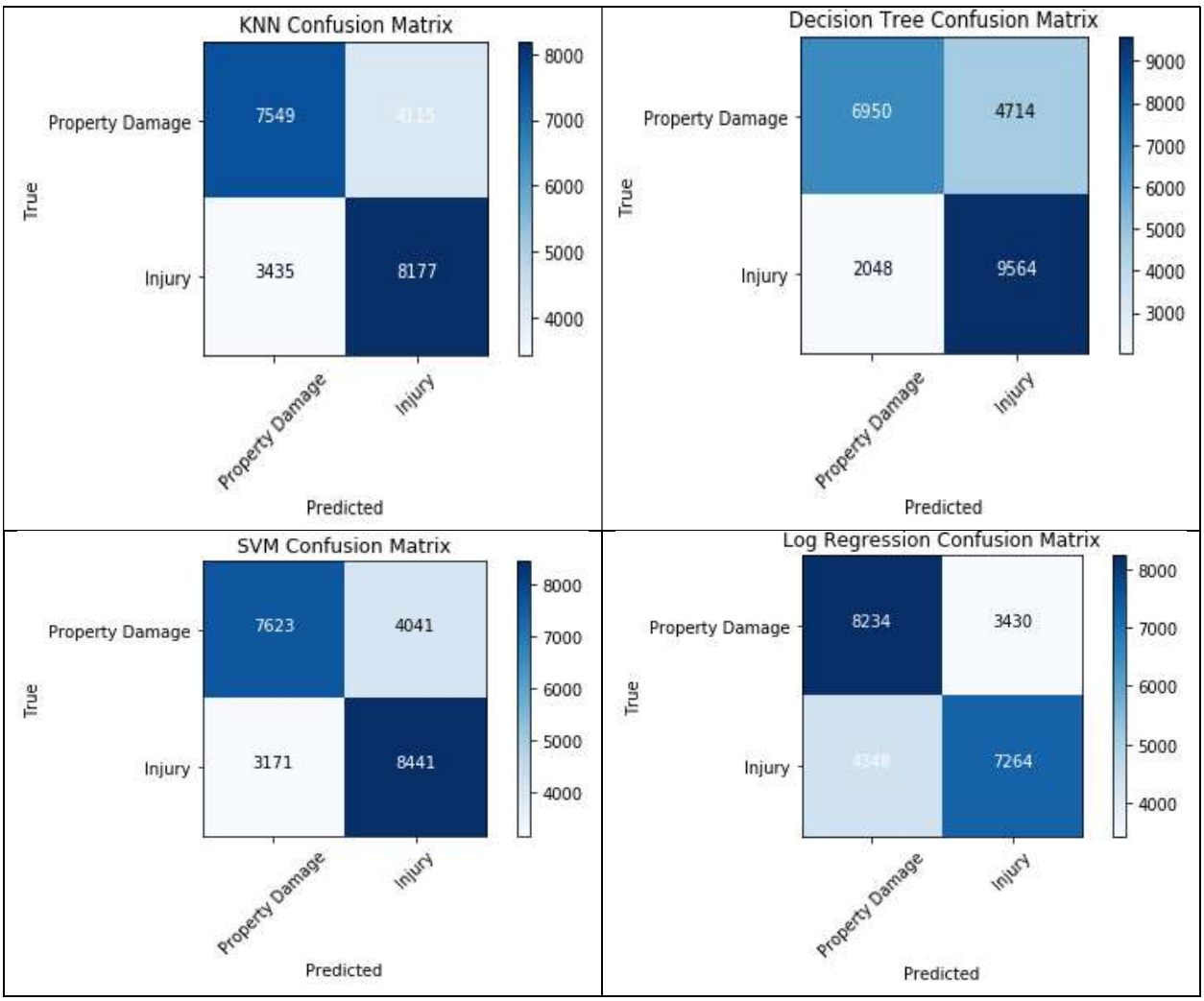


Figure 24 – Confusion Matrices

As with the accuracy scoring, the confusion matrices indicate the decision tree model performed the best. The decision tree model did a better job predicting accidents with injuries while the logistic regression model did a better job predicting accidents with property damage.

Discussion

Using the Seattle accident information dataset, we were able to process the data and build classification models that could predict with reasonable accuracy the accident category. We were able to determine that, while there were minor variations in the number of accidents by day of the week, there was a noticeable drop in the number of accidents occurring on Saturdays. We also determined that, while there were slight variations in the number of accidents occurring based on the time of day, time periods 0400 to 0800 and 2000 to 0000 had significantly fewer accident occurrences.

Conclusion

There is room for improvement in the accuracy of the models. This project essentially used all the relevant features in the Seattle accidents dataset for the models. Future efforts could include the use of OneHotEncoder, Cross-Validation and Pipeline to determine the best mix of features and model optimization for target prediction. Additionally, including datasets with more information such as nationwide data could allow for state by state comparisons and combining a population dataset could allow for comparison based on population densities. The categorical target data could also be expanded to include fatality information.