UCDPA – John Lenehan

GitHub URL

https://github.com/jlenehan/UCDPA_JohnLenehan

Abstract

An analysis is conducted on traffic collision data in Chicago from 2013 to present, following which a machine learning model is constructed to predict collisions involving either an injury or a tow, based on factors such as crash type, location, and time at which the crash occurred. This analysis shows that district 8 of Chicago is the most heavily affected by car collisions. Using the data provided, a K Nearest Neighbors classification model is produced which can predict the need for a callout to these collisions to a 74% accuracy.

Introduction

For this project, the use case is defined as a machine learning tool for first responders in the Chicago municipal area to predict the likelihood of collisions requiring a callout. This tool would be beneficial in allocating resources to different parts of the city based on weather conditions and time of the day, to allow for faster response times to the scenes of traffic collisions.

Dataset

The dataset used for this analysis is a merging of data from two sources; a live dataset of traffic collisions in Chicago from 2013 to present [1], and a static dataset of Chicago PD beats [2]. The beats data is used to determine which district the collision took place in, as this information wasn't contained in the original dataset.

Implementation Process

Step 0: Import Libraries

To begin, the necessary libraries for analysis must be imported; these are laid out below:

```
#generic data analysis
import os
import pandas as pd
from datetime import date
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import missingno as msno
```

Additionally, functions from the machine learning module sci-kit learn (sklearn) are imported to build a machine learning engine; these functions are shown below:

```
#Preprocessing
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler

# Models
from sklearn.neighbors import KNeighborsClassifier

# Reporting
from sklearn.model_selection import train_test_split
from sklearn.model_selection import RandomizedSearchCV

#metrics
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
```

Step 1: Import Data

Step 1 is to import the data; the collision data is imported from the web using the pd.read_json() function. The data sources used in this project can be found at the below addresses:

Data name	**Data Source (URL)**	**JSON link**
Collisions Data	https://data.cityofchicago.org/Transportation/Traffic-Crashes-Crashes/85ca-t3if [1]	https://data.cityofchicago.org/resource/85ca-t3if.json?$limit=1000000
Beats Data	https://data.cityofchicago.org/Public-Safety/Boundaries-Police-Beats-current-/aerh-rz74 [2]	Data downloaded and imported as a csv file

Note that the query "?$limit=1000000" is added to the json string of the collision dataset, to increase the json row limit from 1,000 to 1,000,000. This is to ensure that all data will be imported in the data pull.

```
#importing car crash data from chicago data portal
#url to overview page - https://data.cityofchicago.org/Transportation/Traffic-Crashes-Crashes/85ca-t3if
collision_json = r'https://data.cityofchicago.org/resource/85ca-t3if.json?$limit=100000000' #json url

#using chunksize method to read in large datasets
collision_raw = pd.read_json(collision_json) #reading collisions json
```

For the CPD beats data, the data is downloaded and stored as a csv file (included in the zip file for this assignment) and the data is read into the jupyter notebook using the .read_csv() function. This method is used as the beats and district data is static, therefore the data wouldn't benefit from being pulled live. The directory of the notebook is set to the assignment folder using the os.chdir() function:

```
#Setting directory
directory = r'C:\Users\jlenehan\OneDrive\Documents\Data Science\Data Analytics Essentials\UCDPA_JohnLenehan'
os.chdir(directory)
```

From there the file can be read into the read_csv() function locally:

```
#importing beat data to join to main dataset
#source - Chicago Data Portal
#beat data url - https://data.cityofchicago.org/Public-Safety/Boundaries-Police-Beats-current-/aerh-rz74
beat_data=pd.read_csv('PoliceBeatDec2012.csv')
```

Step 2: Merge Data

With all individual dataframes now cleaned and aggregated to show the desired data, these data must now be merged to produce the final amalgamated dataframe for analysis. These dataframes are joined using an inner join on beat number (beat_of_occurrence in the original dataframe, BEAT_NUM in the beats data).

```
#joining collision data to beat data - inner join
collisions = collision_raw.merge(beat_data, how='inner',
                                 left_on='beat_of_occurrence',
                                 right_on='BEAT_NUM'
                                 )
```

The purpose of this is to add information on the district in which the collision took place. Alternatively, we can use a regex expression to take the first 2 digits out of the beat data (noting that the first 2 digits correspond with the district number):

```
#alternatively - can get district from first 2 digits of beat of occurrence
collisions['district'] = collisions['beat_of_occurrence'].astype(str).str.extract(r'^(\d{2})')
```

Step 3: Describe Data

Once the dataset is merged to include the district data, the methods .columns, .shape, and the functions .info, and .describe are used to give an idea of what the dataset looks like.

```
#Describe recent incidents dataset
print(collisions.columns)
print(collisions.info())
print(collisions.describe())
print(collisions.shape)
```

Additionally, a custom function is defined to print all the unique values of each column, along with the number of unique values. This function uses a generator object to speed up iteration across values, and contains a try-except statement to avoid the function failing due to irregular data:

```
#defining custom function to print all unique values and their counts from each column
def print_uniques(df):
    #for large datasets - using generator object for speed
    uniques_generator = ((x, df[x].unique(), df[x].nunique()) for x in df.columns)

    print('\nUnique Values:')
    for x, unique_values, num_unique in uniques_generator:
        #building in try-except statement for possible strange column data
        try:
            print(f"{x}: \n {unique_values} \n ({num_unique} unique values)")
        except Exception as e:
            print(f"Error occurred in column '{x}': {str(e)}")
```

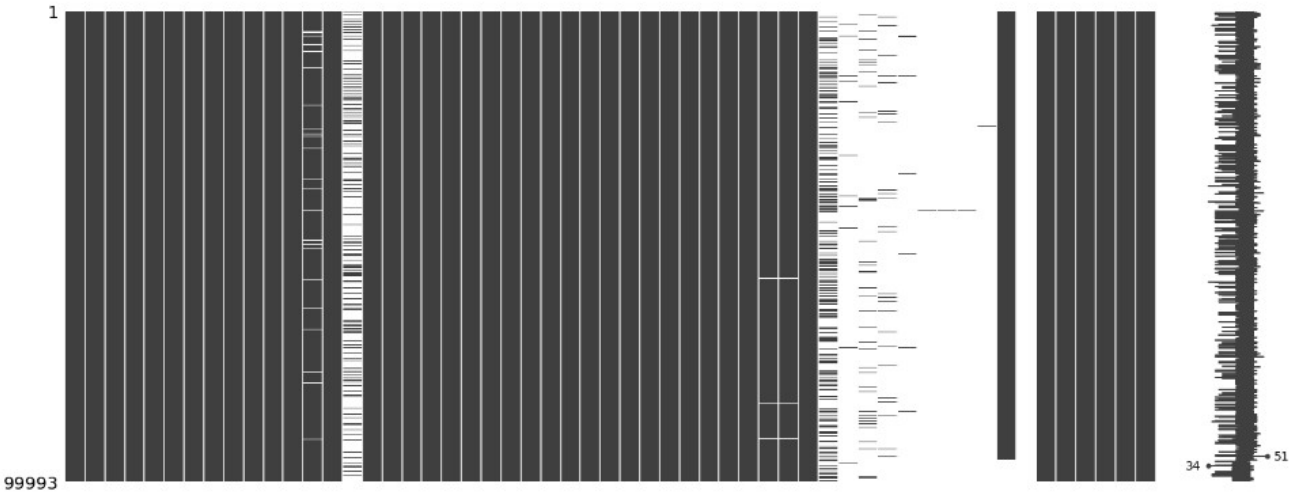
This allows for a better view of the data. Noting that the location data is in a format that would be confused for a dictionary object, the location column is first converted to a string before calling the function:

```
#converting location data to string
collisions['location'] = collisions['location'].astype(str)

#calling custom function on collisions df
print_uniques(collisions)
```

Step 3: Clean and Manipulate Data

The information provided from the .info() function shows a number of columns with sparse data. This can be visualized using the missingno matrix function, called as msno.matrix:

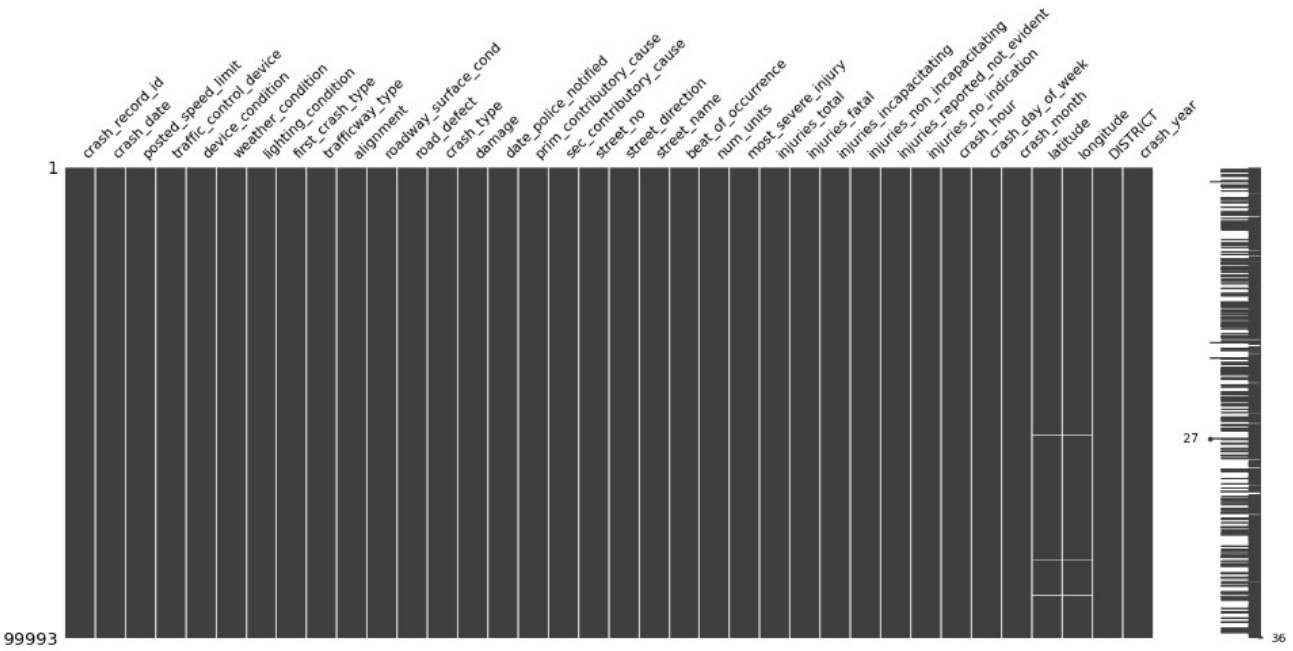
```
#Visualising missing data
#Sorting values by report received date
collisions = collisions.sort_values(by='crash_date',ascending=True)
```



By dropping columns with sparse data, a much cleaner dataset can be extracted; the columns to drop are defined in a list and then removed from the dataset using the .drop() function:

```
#defining unnecessary columns
drop_cols = ['location', 'crash_date_est_i','report_type', 'intersection_related_i',
       'hit_and_run_i', 'photos_taken_i', 'crash_date_est_i', 'injuries_unknown',
       'private_property_i', 'statements_taken_i', 'dooring_i', 'work_zone_i',
       'work_zone_type', 'workers_present_i','lane_cnt','the_geom','rd_no',
            'SECTOR','BEAT','BEAT_NUM']

#dropping columns
collisions=collisions.drop(columns=drop_cols)
```

Following this the msno matrix is reprinted to give a view of the pruned dataset:



Some values are missing from the latitude and longitude columns – as it would be difficult to determine these values through other means, these are dropped from the dataset:

```
#drop all rows with missing latitude/longitude data
collisions.dropna(subset=['longitude','latitude'],inplace=True)
```

Values are also missing from the injuries columns – intuitively this would likely be due to no injuries of those types being reported. These missing values are filled with zeroes using the fillna() function:

```
#replacing all null values in injuries columns with 0
#defining injury columns
num_injury_cols = ['injuries_total','injuries_fatal','injuries_incapacitating',
                   'injuries_non_incapacitating','injuries_reported_not_evident',
                   'injuries_no_indication']
collisions[num_injury_cols] = collisions[num_injury_cols].fillna(0)
```

Additionally there are some zero values for latitude and longitude in the data, based on the output from the .describe() function.

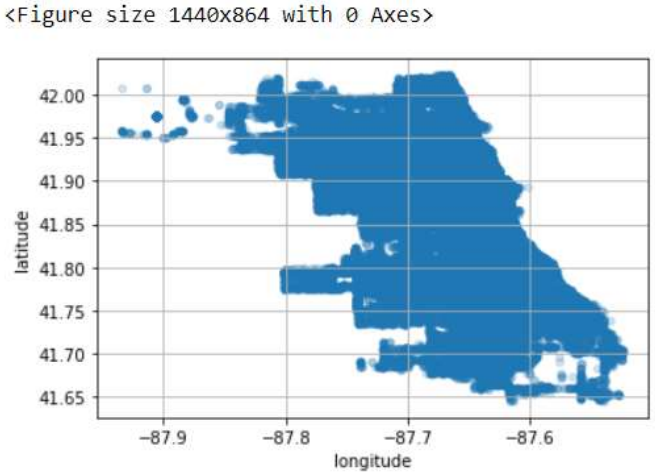
	crash_month	latitude	longitude
count	723127.000000	718458.000000	718458.000000
mean	6.605770	41.854620	-87.673489
std	3.446554	0.331399	0.672950
min	1.000000	0.000000	-87.936193
25%	4.000000	41.781660	-87.721583
50%	7.000000	41.874537	-87.673846
75%	10.000000	41.924142	-87.633086
max	12.000000	42.022780	0.000000

These are removed to avoid skew in the data.

```
#Some incorrect lat/long data - need to remove these rows
collisions = collisions[collisions['longitude']<-80]
collisions = collisions[collisions['latitude']>40]
```

Step 5: Plot Data

the latitude and longitude data is first plotted, to ensure that it looks correct. This can be seen below.



Additionally, plots of collisions by district, crash type and number of total injuries are plotted using the matplotlib library. The code for one of these plots can be seen below.

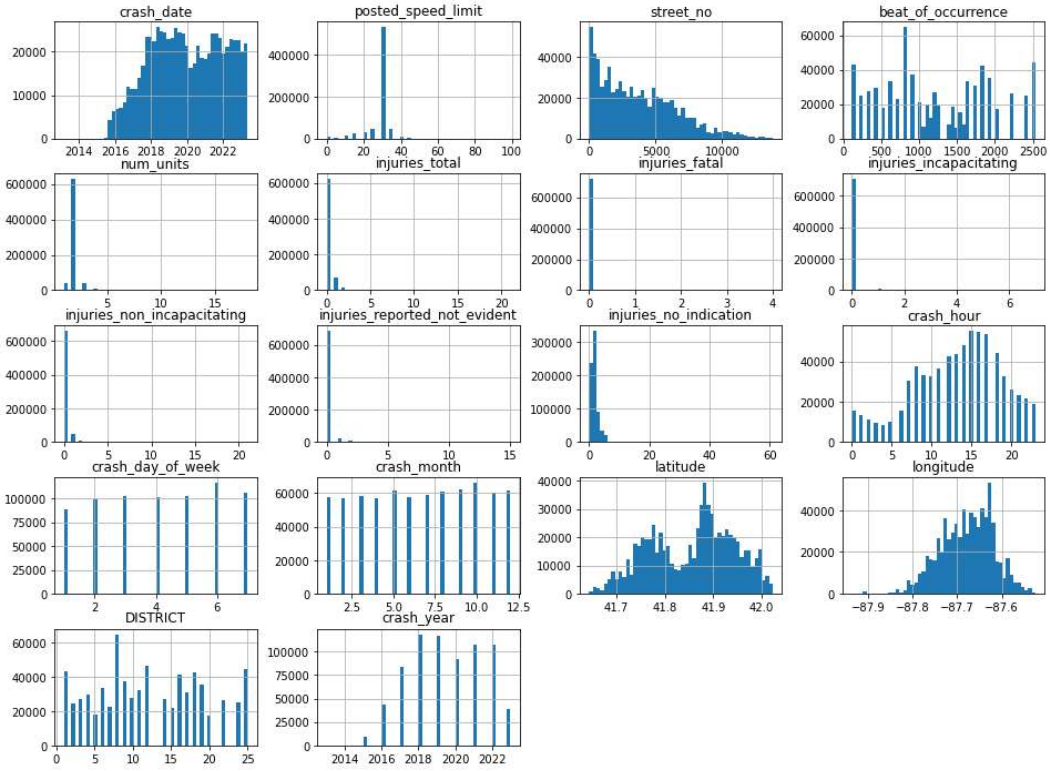
```
#plot graphs of collisions by number of injuries
injuries_total_count = collisions['injuries_total'].value_counts()

injuries_total_count.plot.bar()
plt.xlabel('Total Injuries')
plt.ylabel('Collisions')
plt.title('Collisions by number of Injuries')
plt.show()
```

Step 6: Classification Model

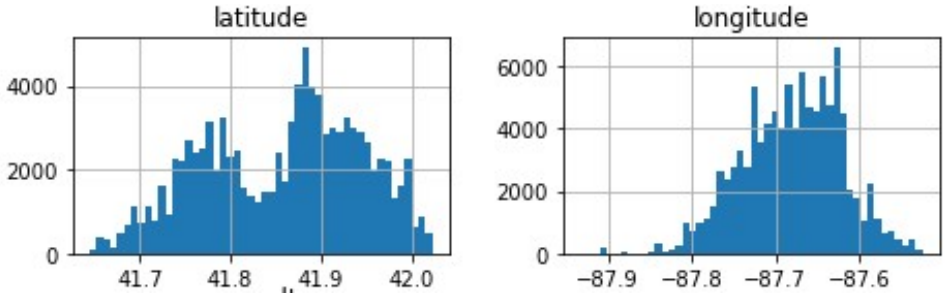
For this project, a K nearest neighbors classification model is used to fit the data. This model is then Scored based on accuracy precision recall and F1.

Before proceeding with the machine learning model some exploratory data analysis needs to be performed – each of the columns of the data frame are plotted on a histogram, with bins of 50 to show the distribution of the data.

```
#plotting histograms of numerical values
collisions.hist(bins=50,figsize=(16,12))
plt.show()
```



A cursory look at the column histograms indicates that the longitude data is rightly skewed, while the latitude data is bimodal. This will need to be standardized so that we can use it better for machine learning purposes.



Additionally, it appears the crash hour column is cyclic in nature - this can be transformed using a sine function.

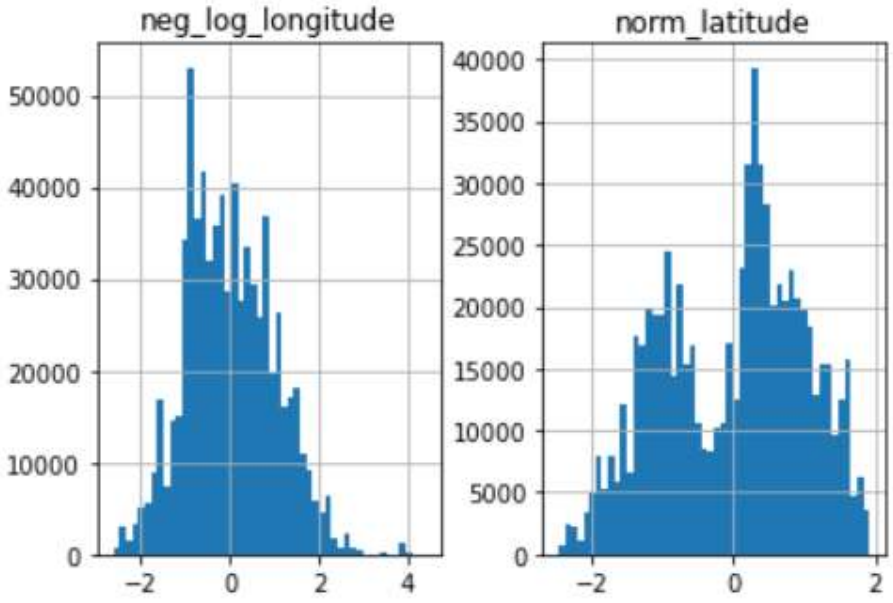
To begin, a copy of the collisions dataset is made containing the columns to be used for analysis. The categorical columns are further separated, then encoded using the LabelEncoder function from sklearn:

```
ml_cols = ['posted_speed_limit','traffic_control_device', 'device_condition', 'weather_condition',
           'lighting_condition', 'first_crash_type', 'trafficway_type','alignment',
            'roadway_surface_cond', 'road_defect', 'crash_type','damage','prim_contributory_cause',
           'sec_contributory_cause','street_direction','num_units', 'DISTRICT',
           'crash_hour','crash_day_of_week','latitude', 'longitude']
cat_cols = ['traffic_control_device', 'device_condition', 'weather_condition', 'DISTRICT',
            'lighting_condition', 'first_crash_type', 'trafficway_type','alignment',
            'roadway_surface_cond', 'road_defect', 'crash_type','damage','prim_contributory_cause',
            'sec_contributory_cause','street_direction','num_units']

collisions_ml = collisions[ml_cols].copy()

#encoding categorical values
label_encoder = LabelEncoder()
for col in collisions_ml[cat_cols].columns:
    collisions_ml[col] = label_encoder.fit_transform(collisions_ml[col])
```

.

Both the latitude and longitude data are reshaped using a standard scaler. firstly the negative of the longitude data is transformed using the log 1P formula. Histograms of the transformed data can be seen below.

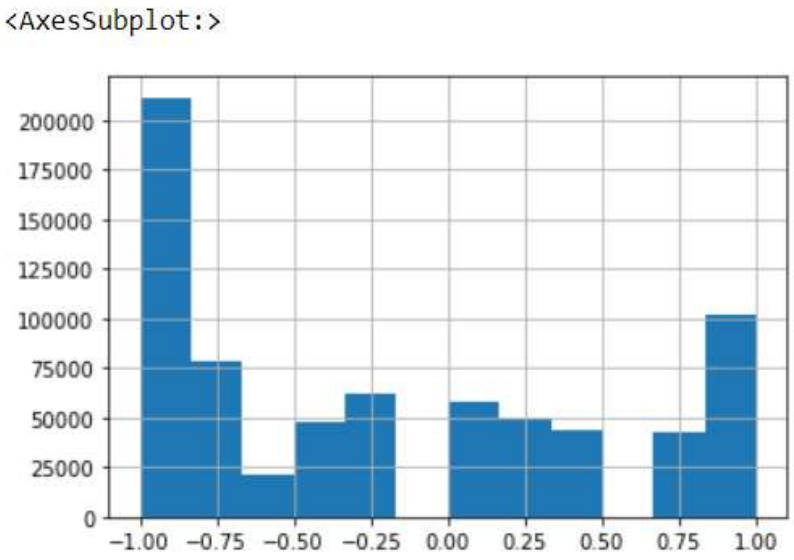
```
#scaling latitude and longitude data
scaler = StandardScaler()

# Logarithmic transformation on longitude
collisions_ml['neg_log_longitude'] = scaler.fit_transform(np.log1p(-collisions_ml['longitude']).values.reshape(-1,1))

# Normalisation on latitude
collisions_ml['norm_latitude'] = scaler.fit_transform(collisions['latitude'].values.reshape(-1, 1))
```



Following this the crash our data is also transformed. As noted earlier the crush hour data is cyclic in nature, and therefore we can use a sine function to transform this column.

```
#transforming crash_hour
#data is cyclic, can be encoded using trig transforms

#trig transformation - sin(crash_hr)
collisions_ml['sin_hr'] = np.sin(2*np.pi*collisions_ml['crash_hour']/24)
```

A histogram of the transformed data can be seen below.



Subsequently the non transformed data is stripped from the data set using the drop function.

```
#drop previous latitude/longitude columns
lat_long_drop_cols = ['longitude','latitude']
collisions_ml.drop(lat_long_drop_cols,axis=1,inplace=True)

#drop crash_hour column
collisions_ml.drop('crash_hour',axis=1,inplace=True)
```

Once all the transformations are complete the data is now ready to be split into the test set and training set. Firstly access defined as every column apart from the crash type column, using the .drop() function. The labels are defined as the crash type. Once this is complete the data can now be separated into training and test set. This is done using the train test split function; a test size of 0.2 is defined along with a random state of 42.

```
#Create test set
#setting X and y values

X = collisions_ml.drop('crash_type', axis=1)
y = collisions_ml['crash_type']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

The first classifier model to be tested is a K neighbors classifier model, where we specify the initial n_neighbors hyperparameter as 3. This is then fit the training data. The classifier is used to predict labels on the training data and test data. These predictions will be compared later to evaluate fit.

```
#Classifier 1 - K Nearest Neighbours
#instantiate KNN Classifier
KNNClassifier = KNeighborsClassifier(n_neighbors=3)

KNNClassifier.fit(X_train,y_train)
```

Once the model has predicted labels for both the training and test set, accuracy scores for both are computed using the accuracy score function. Additionally the F1 score, precision score, and recall score are also computed.

```
#7. Evaluate model
# Calculate the accuracy of the model

#calculating accuracy of model on training data
train_accuracy = accuracy_score(y_train, y_train_pred)

#calculating accuracy of model on test data
test_accuracy = accuracy_score(y_test, y_test_pred)

#computing f1 score,precision,recall
f1 = f1_score(y_test, y_test_pred)
precision = precision_score(y_test,y_test_pred)
recall = recall_score(y_test,y_test_pred)

#comparing performances
print("Training Accuracy:", train_accuracy)
print("Test Accuracy:", test_accuracy)

#print precision score
print("Precision Score:", precision)

#print recall score
print("Recall Score:", recall)

#print f1 score
print("F1 Score:", f1)
```

The metrics of the KNN model can be seen below.

```
Training Accuracy: 0.9310771093759854
Test Accuracy: 0.7960645812310797
Train-Test Accuracy Difference: 0.13501252814490572
Precision Score: 0.8211491599333077
Recall Score: 0.9108692559396785
F1 Score: 0.8636854175097801
```

The model performed well with high precision, recall and F1 score (for context, high precision indicates low false positives, high recall indicates low false negatives, and F1 score is a measure of both). One point of concern is the difference between the training accuracy and the test accuracy; the model is 13.5% more accurate on the training set compared to the test set. This would suggest overfitting, although the high metrics are also positive signs of model performance.

Next hyperparameter tuning is performed on the model, by first defining a parameter grid of n_neighbors, weights and metric parameters for the knn model. This is then inputted to a randomized search cross-validation function, to extract the optimal parameters for the model.

```python
#8. Fine tuning (GridsearchCV/RandomisedSearchCV)
# Define parameter grid
param_grid = {
    'n_neighbors': [3, 7, 10],
    'weights': ['uniform','distance'],
    'metric': ['euclidean','manhattan']
}

# instantiate RandomizedSearchCV
random_search = RandomizedSearchCV(estimator=KNeighborsClassifier(), param_distributions=param_grid, cv=5)

# fit to training data
random_search.fit(X_train, y_train)

# Retrieve best model and performance
best_classifier = random_search.best_estimator_
best_accuracy = random_search.best_score_

print("Best Accuracy:", best_accuracy)
print("Best Model:", best_classifier)
```

The results of this cross validation can then be re-inputted to the knn classifier, in order to generate better performance from the classification model.

Results

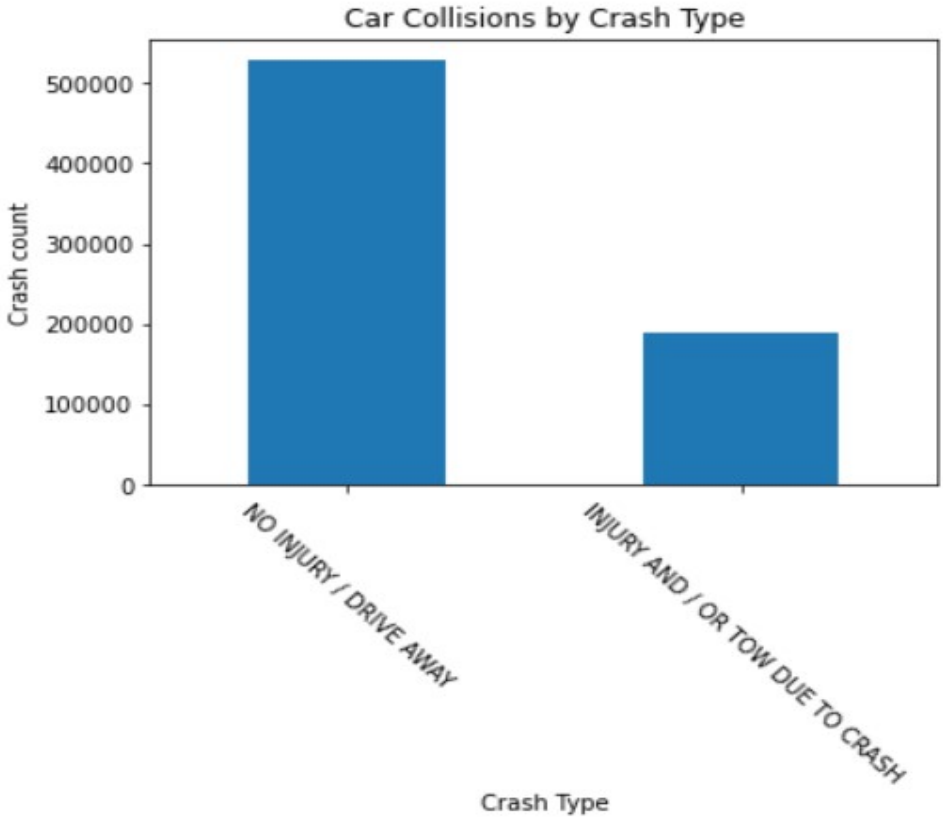


Figure 1: Graph showing total car collisions by crash type

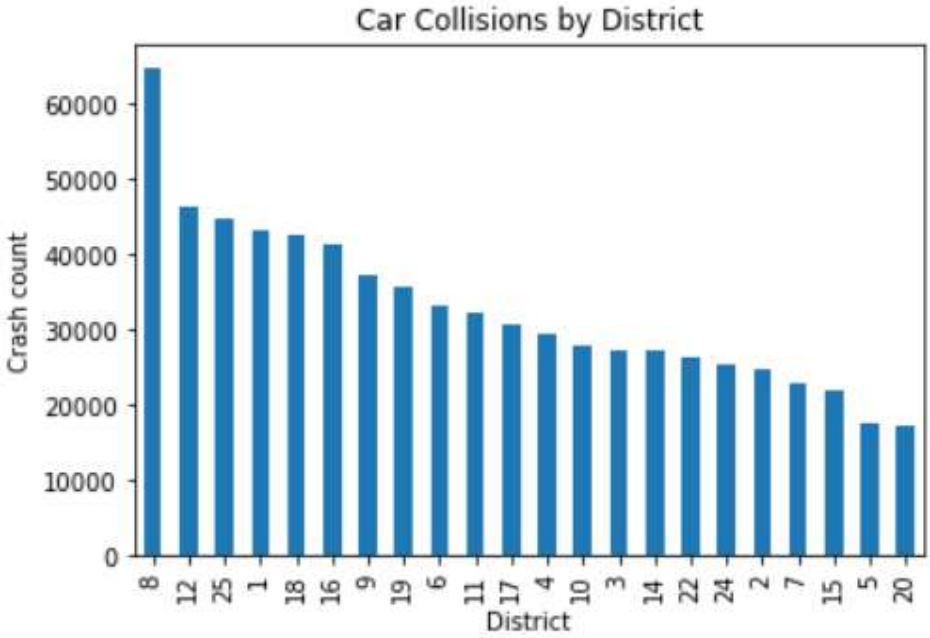


Figure 2: Graph showing total car collisions by District.

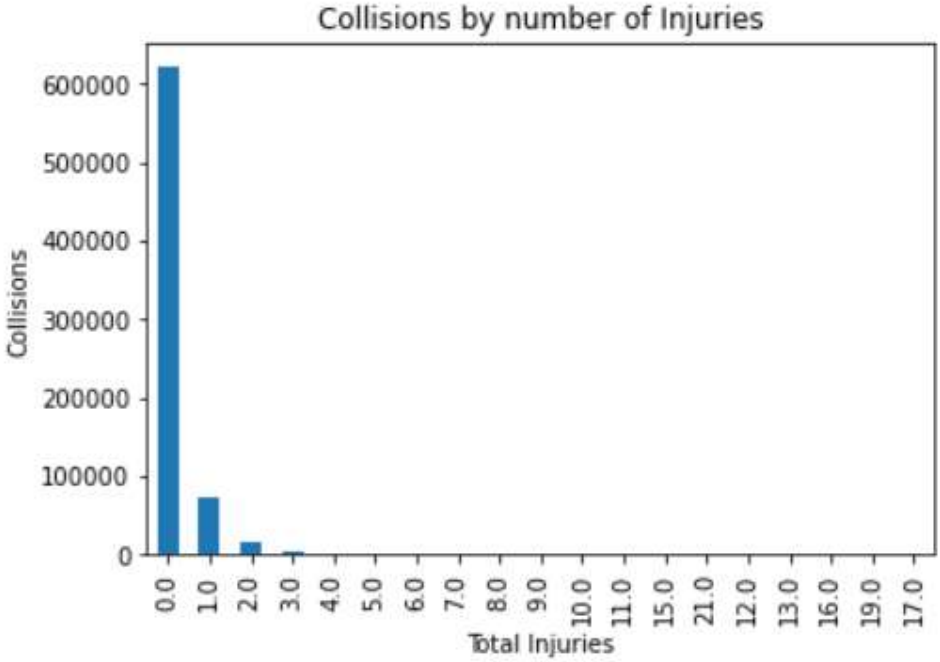


Figure 3: Graph showing collisions by number of total injuries.

```
Best Accuracy: 0.7405552797632174
Best Model: KNeighborsClassifier(metric='manhattan', n_neighbors=10)
```

Figure 4: Results of the RandomizedSearchCV fine tuning, showing the recommended metric and n_neigbors parameters.

```
Training Accuracy: 0.7872649065933373
Test Accuracy: 0.7422805247225025
Train-Test Accuracy Difference: 0.044984381870834755
Precision Score: 0.779477891581314
Recall Score: 0.8878218807796272
F1 Score: 0.8301296973727966
```

Figure 5: the results of the knn classifier when using optimal parameters as recommended by the RandomisedSearchCV function. Note the much reduced difference between training and test accuracies.

Insights

1. The number of car crashes involving either a tow or injury, compared with collisions involving neither, are in the ratio of 2:5 based on the graph in figure 1. This indicates that where collisions have occurred, the occupants of the vehicle will require a callout roughly 30% of the time.

2. Chicago's 8th district is the most heavily affected by car collisions, much higher than the other districts, based on the data shown in Figure 2. This indicates that District 8 will require the most callouts.

3. Of the over 700,000 instances of collisions captured in this dataset, 6/7 collisions involve no injuries as reported by the Chicago police department, as indicated by the graph in Figure 3 – good news. When taken with the data given in Figure 1, this indicates that roughly 1 in 7 collisions requiring callouts are solely for towing the vehicle.

4. Figure 4 shows the results of the randomized search cross validation, indicating high n_neighbors and a Manhattan metric are optimal for this model.

5. The finalized analysis based on the results of the randomized search cross validation show much reduced difference between the training and test accuracies, while maintaining high metrics of precision, recall, and F1 score. This indicates that the optimized model is no longer overfitting the data, while still maintaining strong performance on the metrics.

References

[1] Levy, J. (n.d.). Traffic Crashes - Crashes [Dataset]. Retrieved from Chicago Data Portal: https://data.cityofchicago.org/Transportation/Traffic-Crashes-Crashes/85ca-t3if (Accessed: 14th May 2023).

[2] Chicago Police Department. (n.d.). Boundaries - Police Beats (current) [Data set]. Retrieved from Chicago Data Portal: https://data.cityofchicago.org/Public-Safety/Boundaries-Police-Beats-current-/aerh-rz74 (Accessed: 14th May 2023).