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')
```