Implementation of Classification Algorithms to predict Accident Severity

In [ ]:
```
#Import Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import datetime
from scipy import stats

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
from sklearn import svm
from sklearn.dummy import DummyClassifier
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix, recall_score


from imblearn.over_sampling import SMOTE
from collections import Counter
from sklearn import metrics
import time
```

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

In [ ]:
```
#Store the train dataset file in a variable
file_train = "final_trainset.csv"

#Read the dataset with pandas
df_train = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/' + file_train)

#Store the test dataset file in a variable
file_test = "final_testset.csv"

#Read the dataset with pandas
df_test = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/' + file_test)
```

In [ ]:
```
#Number of instances and features in the train dataset
df_train.shape
```

Out[52]:
```
(93978, 35)
```

In [ ]:
```
#Number of instances and features in the test dataset
df_test.shape
```

Out[53]:
```
(23495, 35)
```

In [ ]:
```
#Fetching few records of the dataset
df_train.head()
```

Out[54]:

	Location_Easting_OSGR	Location_Northing_OSGR	Longitude	Latitude	Number_of_Vehicles	Speed_limit	high_winds	Monday	Saturday	Sunday	Thursday	Tuesday	Wednsday	One way street	Roundabout	Single carriageway	Slip road
0	-1.442574	2.079796	-1.556315	2.064596	-1.0	0.0	True	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0
1	0.370081	1.005455	0.398955	1.001380	-1.0	-1.0	False	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
2	-0.698771	-0.202495	-0.706757	-0.194844	-1.0	0.0	False	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0
3	0.560456	-0.225449	0.545707	-0.225843	1.0	-1.0	False	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
4	0.373136	0.722523	0.391836	0.720016	-1.0	3.0	False	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0

In [ ]:
```
#Number of instances and features in the dataset
df_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 93978 entries, 0 to 93977
Data columns (total 35 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Location_Easting_OSGR   93978 non-null  float64
 1   Location_Northing_OSGR  93978 non-null  float64
 2   Longitude               93978 non-null  float64
 3   Latitude                93978 non-null  float64
 4   Number_of_Vehicles      93978 non-null  float64
 5   Speed_limit             93978 non-null  float64
 6   high_winds              93978 non-null  bool
 7   Monday                  93978 non-null  float64
 8   Saturday                93978 non-null  float64
 9   Sunday                  93978 non-null  float64
 10  Thursday                93978 non-null  float64
 11  Tuesday                 93978 non-null  float64
 12  Wednsday                93978 non-null  float64
 13  One way street          93978 non-null  float64
 14  Roundabout              93978 non-null  float64
 15  Single carriageway      93978 non-null  float64
 16  Slip road               93978 non-null  float64
 17  Darkness - No lighting  93978 non-null  float64
 18  Daylight                93978 non-null  float64
 19  Snow                    93978 non-null  float64
 20  Water                   93978 non-null  float64
 21  None                    93978 non-null  float64
 22  Road Defect             93978 non-null  float64
 23  Urban                   93978 non-null  float64
 24  Spring                  93978 non-null  float64
 25  Summer                  93978 non-null  float64
 26  Winter                  93978 non-null  float64
 27  Morning                 93978 non-null  float64
 28  Night                   93978 non-null  float64
 29  Road Hazard             93978 non-null  float64
 30  Fog or mist             93978 non-null  float64
 31  Other                   93978 non-null  float64
 32  Raining                 93978 non-null  float64
 33  Snowing                 93978 non-null  float64
 34  Accident_Severity       93978 non-null  object
dtypes: bool(1), float64(33), object(1)
memory usage: 24.5+ MB
```

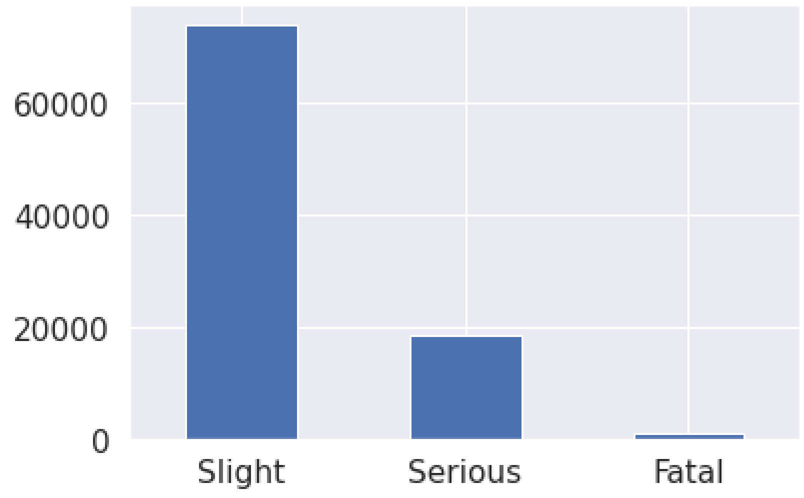
Confirm that the distribution of the variable is similar:

In [ ]:
```
#normalizing the distribution for train dataset
df_train["Accident_Severity"].value_counts(normalize=True)
```

Out[56]:
```
Slight     0.786588
Serious    0.199302
Fatal      0.014110
Name: Accident_Severity, dtype: float64
```

In [ ]:
```
# Visualise this unbalance using a bar chart
df_train["Accident_Severity"].value_counts().plot(kind="bar", rot = 0)
```

Out[57]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f28922688d0>
```



The Target variable seems to be imbalanced in nature

In [ ]:
```
#normalizing the distribution for test dataset
df_test["Accident_Severity"].value_counts(normalize=True)
```

Out[58]:
```
Slight     0.786593
Serious    0.199319
Fatal      0.014088
Name: Accident_Severity, dtype: float64
```

The total records for the training and the test dataset

In [ ]:
```
print(f"There are {df_train.shape[0]} training and {df_test.shape[0]} test instances")
```

```
There are 93978 training and 23495 test instances
```

In [ ]:
```
#assigning the predictors of train data
x_train= df_train.drop("Accident_Severity",axis=1)
```

In [ ]:
```
#assigning the dependent or target variable to train data
y_train = df_train["Accident_Severity"].copy()
```

In [ ]:
```
#assigning the predictors of test data
x_test= df_test.drop("Accident_Severity",axis=1)
```

In [ ]:
```
#assigning the dependent or target variable to test data
y_test = df_test["Accident_Severity"].copy()
```

Baseline Model

A performance baseline provides a minimum score above which a model is considered to have skill on the dataset. It also provides a point of relative improvement for all models evaluated on the dataset.

Calculate the F-score for the majority baseline (every accident severity label is "slight"):

In [ ]:
```
df_train["Accident_Severity"].value_counts()
```

Out[64]:
```
Slight     73922
Serious    18730
Fatal       1326
Name: Accident_Severity, dtype: int64
```

In [ ]:
```
n_slight_severity= df_train["Accident_Severity"].value_counts()["Slight"]
n_instances = df_train.shape[0]
```

In [ ]:
```
# For the "slight_severity" label, the accuracy measures will be:
slight_severity_precision = n_slight_severity/n_instances
slight_severity_recall = n_slight_severity/n_slight_severity
slight_severity_fscore = 2/(1/slight_severity_precision + 1/slight_severity_recall)

# For the "no" label, it will be:
serious_severity_precision = 0.0
serious_severity_recall = 0.0
serious_severity_fscore = 0.0

# The averages of the two classes, i.e. the eventual baseline scores:
p = (slight_severity_precision+serious_severity_precision)/2
r = (slight_severity_recall+serious_severity_recall)/2
f = (slight_severity_fscore+serious_severity_fscore)/2

print(f"Precision: {p:.5}")
print(f"Recall: {r:.5}")
print(f"F-score: {f:.5}")
```

```
Precision: 0.39329
Recall: 0.5
F-score: 0.44027
```

Tree-Based Algorithm

Decision trees often perform well on imbalanced datasets because their hierarchical structure allows them to learn signals from the classes.

In [ ]:
```
# train model
rfc = RandomForestClassifier(n_estimators=10).fit(x_train, y_train)

# predict on test set
rfc_pred = rfc.predict(x_test)

accuracy_score(y_test, rfc_pred)
```

Out[67]:
```
0.7445413917854863
```

In [ ]:
```
# recall score
recall_score(y_test, rfc_pred,average="micro")
```

Out[68]:
```
0.7445413917854863
```

In [ ]:
```
# f1 score
f1_score(y_test, rfc_pred,average="micro")
```

Out[69]:
```
0.7445413917854863
```

In [ ]:
```
#confusion matrix
data = confusion_matrix(y_test, rfc_pred)
df_cm = pd.DataFrame(data, columns=np.unique(y_test),index = np.unique(y_test))
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'
plt.figure(figsize = (10,7))
#sns.set(font_scale=1.4)#for label size
sns.heatmap(df_cm, cmap="Blues", annot=True,fmt='g', annot_kws={"size": 16})# font size
```

Out[70]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f288fbf5f50>
```

The F-score for Baseline using RandomForestClassifier has been improved to 0.75 as comapred to F-score of majority baseline i.e 0.44

Resampling Techniques

SMOTE (Synthetic Minority Oversampling Technique)

This technique is used to create synthetic samples. Here we will use imblearn's SMOTE Technique. SMOTE uses a nearest neighbors algorithm to generate new data which can be used for training our model. With over-sampling methods, the number of samples in a class should be greater or equal to the original number of samples

Also, we need to generate the new samples only in the training set to ensure our model generalizes well to unseen data.

In [ ]:
```
sm = SMOTE(sampling_strategy = {'Slight':80000, 'Serious':50000, 'Fatal':15000}
)
x_test, y_test = sm.fit_sample(x_train, y_train)
```

```
/usr/local/lib/python3.7/dist-packages/imblearn/utils/_validation.py:257: UserWarning: After over-sampling, the number of samples (80000) in class Slight will be larger than the nu
mber of samples in the majority class (class #Slight -> 73922)
  n_samples_majority))
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87: FutureWarning: Function safe_indexing is deprecated; safe_indexing is deprecated in version 0.22 and will be
removed in version 0.24.
  warnings.warn(msg, category=FutureWarning)
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87: FutureWarning: Function safe_indexing is deprecated; safe_indexing is deprecated in version 0.22 and will be
removed in version 0.24.
  warnings.warn(msg, category=FutureWarning)
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87: FutureWarning: Function safe_indexing is deprecated; safe_indexing is deprecated in version 0.22 and will be
removed in version 0.24.
  warnings.warn(msg, category=FutureWarning)
```

In [ ]:
```
pd.Series(np.array(y_test)).value_counts()
```

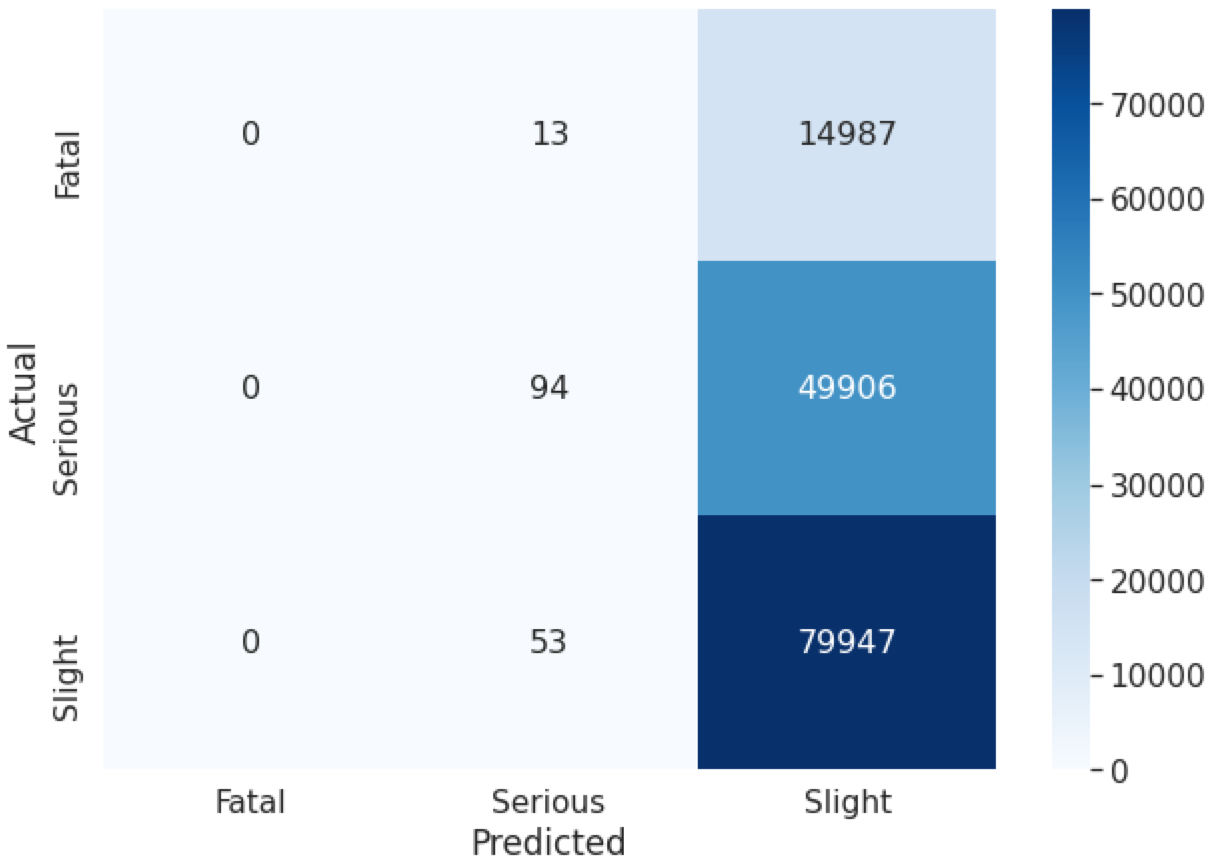
Out[72]:
```
Slight     80000
Serious    50000
Fatal      15000
dtype: int64
```

In [ ]:
```
smote = LogisticRegression(solver='liblinear').fit(x_train, y_train)

smote_pred = smote.predict(x_test)

# Checking accuracy
accuracy_score(y_test, smote_pred)
```

Out[73]:
```
0.5520068965517242
```

In [ ]:
```
#confusion matrix
data = confusion_matrix(y_test, smote_pred)
df_cm = pd.DataFrame(data, columns=np.unique(y_test),index = np.unique(y_test))
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'
plt.figure(figsize = (10,7))
#sns.set(font_scale=1.4)#for label size
sns.heatmap(df_cm, cmap="Blues", annot=True,fmt='g', annot_kws={"size": 16})# font size
```

Out[74]: <matplotlib.axes._subplots.AxesSubplot at 0x7f288fbf19d0>



In [ ]:
```
# f1 score -SMOTE
f1_score(y_test, smote_pred,average="micro")
```

Out[75]: 0.5520068965517242

In [ ]:
```
# recall score
recall_score(y_test, smote_pred,average="micro")
```

Out[76]: 0.5520068965517242

The F-score value of SMOTE is less than the above RandomForestClassifier but better as compared to that of majority Baseline.

Model - Random Forest

In [ ]:
```
rf = RandomForestClassifier()

# specify the hyperparameters and their values
# 3 x 3 x 2 = 18 combinations in the grid
param_grid = {
    'n_estimators': [10, 150, 500],
    'max_depth': [3, 5, 15],
    'min_samples_split': [5, 10],
    'random_state': [7]
}

# we'll use 5-fold cross-validation
grid_search = GridSearchCV(rf, param_grid, cv=5,
                           scoring='f1_macro',
                           return_train_score=True)

start = time.time()
grid_search.fit(x_train, y_train)
end = time.time() - start
print(f"Took {end} seconds")
```

```
Took 1464.672619342804 seconds
```

Construction of Predictive Features

In [ ]:
```
# put them into a separate variable for convenience
feature_importances = grid_search.best_estimator_.feature_importances_

# the order of the features in `feature_importances` is the same as in the Xtrain dataframe,
# so we can "zip" the two and print in the descending order:

for k, v in sorted(zip(feature_importances, x_train.columns), reverse=True):
    print(f"{v}: {k}")
```

```
Location_Northing_OSGR: 0.14733187185238353
Latitude: 0.14305420713093792
Longitude: 0.13579193009135954
Location_Easting_OSGR: 0.13348982048703445
Number_of_Vehicles: 0.0868285296433132
Speed_limit: 0.05367270460940861
Urban: 0.023208205173166498
Single carriageway: 0.01834747855357178
Water: 0.017132635645604445
Night: 0.016881856229914394
Daylight: 0.01558504046523907
Winter: 0.015028331341581014
Summer: 0.01453413710184986
Spring: 0.01407321175820545
Raining: 0.01352116116787012
Monday: 0.013248864643713993
Morning: 0.013025094192801559
Saturday: 0.012795873303922515
Thursday: 0.011451517505647725
Sunday: 0.010965362776596554
Darkness - No lighting: 0.010797502532704808
Wednsday: 0.010696322093664109
Tuesday: 0.010514101481613993
high_winds: 0.009548615733786134
Roundabout: 0.008278986673300941
None: 0.007369147970662991
Other: 0.006926252642062511
Road Defect: 0.006813491803615833
Road Hazard: 0.00543876773824674
Slip road: 0.004624994130385711
Fog or mist: 0.0031590421301260193
One way street: 0.0029740219992906835
Snowing: 0.00158880582311409
Snow: 0.001302113573303201
```

From the predictor importance, we can say that Location_Northing_OSGR and Latitude are predictors having highest importance or significance in predicting the Accident Severity while Number of Vehicles and Speed Limit holds moderate importance and, the weather conditions like Fog or mist, snowing and, road hazards and road defects are some of the variable which plays least importance or significance in predicting the Accident Severity in my data

In [ ]:
```
#finding the best estimator
grid_search.best_estimator_
```

Out[80]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=15, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=5,
                       min_weight_fraction_leaf=0.0, n_estimators=10,
                       n_jobs=None, oob_score=False, random_state=7, verbose=0,
                       warm_start=False)
```

In [ ]:
```
#getting the best score for the model
grid_search.best_score_
```

Out[81]:
```
0.30683513769415915
```

The best hyperparameters prove to be n_estimators=10, max_depth=15 and min_sample_split=5. The achieved F-score is 0.31

Let's record the results of the best model in each split, for future reference.

The best-performing model (rank 1) is in grid_search.cv_results_["rank_test_score"] at index is 12

In [ ]:
```
grid_search.cv_results_["rank_test_score"].tolist().index(1)
```

Out[123]:
```
12
```

In [ ]:
```
rf_split_test_scores = []
for x in range(5):
    # extract f-score of the best model (index=18) from each of the 5 splits
    val = grid_search.cv_results_[f"split{x}_test_score"][12]
    rf_split_test_scores.append(val)
```

Reviewing the scores achieved by all the models in the search grid:

In [ ]:
```
val_scores = grid_search.cv_results_["mean_test_score"]
train_scores = grid_search.cv_results_["mean_train_score"]
params = [str(x) for x in grid_search.cv_results_["params"]]

for val_score, train_score, param in sorted(zip(val_scores, train_scores, params), reverse=True):
    print(val_score, train_score, param)
```

```
0.30683513769415915 0.36881368099220996 {'max_depth': 15, 'min_samples_split': 5, 'n_estimators': 10, 'random_state': 7}
0.30473224890678846 0.34376984499071356 {'max_depth': 15, 'min_samples_split': 10, 'n_estimators': 10, 'random_state': 7}
0.29941496166475606 0.3399385379425592 {'max_depth': 15, 'min_samples_split': 5, 'n_estimators': 150, 'random_state': 7}
0.29910415574360283 0.33746172767377197 {'max_depth': 15, 'min_samples_split': 5, 'n_estimators': 500, 'random_state': 7}
0.299101066281938 1 0.3213998504863946 {'max_depth': 15, 'min_samples_split': 10, 'n_estimators': 500, 'random_state': 7}
0.2987229573145232 0.3226518966834527 {'max_depth': 15, 'min_samples_split': 10, 'n_estimators': 150, 'random_state': 7}
0.2935533138939584 0.293608754889894 {'max_depth': 5, 'min_samples_split': 5, 'n_estimators': 10, 'random_state': 7}
0.2935159817069123 0.2935159817333942 {'max_depth': 3, 'min_samples_split': 5, 'n_estimators': 500, 'random_state': 7}
0.2935159817069123 0.2935159817333942 {'max_depth': 3, 'min_samples_split': 5, 'n_estimators': 150, 'random_state': 7}
0.2935159817069123 0.2935159817333942 {'max_depth': 3, 'min_samples_split': 5, 'n_estimators': 10, 'random_state': 7}
0.2935159817069123 0.2935159817333942 {'max_depth': 3, 'min_samples_split': 10, 'n_estimators': 500, 'random_state': 7}
0.2935159817069123 0.2935159817333942 {'max_depth': 3, 'min_samples_split': 10, 'n_estimators': 150, 'random_state': 7}
0.2935159817069123 0.2935159817333942 {'max_depth': 3, 'min_samples_split': 10, 'n_estimators': 10, 'random_state': 7}
0.2935137591355066 0.2936093182246945 {'max_depth': 5, 'min_samples_split': 10, 'n_estimators': 10, 'random_state': 7}
0.29351375912137806 0.29354398266948123 {'max_depth': 5, 'min_samples_split': 5, 'n_estimators': 150, 'random_state': 7}
0.29351375912137806 0.29353465020493275 {'max_depth': 5, 'min_samples_split': 10, 'n_estimators': 150, 'random_state': 7}
0.29351375912137806 0.2935253165596497 {'max_depth': 5, 'min_samples_split': 5, 'n_estimators': 500, 'random_state': 7}
0.29351375912137806 0.2935253165596497 {'max_depth': 5, 'min_samples_split': 10, 'n_estimators': 500, 'random_state': 7}
```

The performance of Random Forest classifiers varies a bit across the runs, between 0.29 and 0.31. This variation is not much significant.

In particular, we notice that better performance is achieved with greater values of max_depth. However, at higher values of this hyperparameter, we notice some evidence of overfitting: the performance on training parts is considerably better than on the validation part.

Saving the model to disk, so that if in the future we need it, e.g., after re-starting the notebook, we can read the trained model from the disk, instead of re-training it from scratch.

This is useful for models that takes longer time duration to train.

Here, we will use the dump function from the built-in Python module joblib for this

In [ ]:
```
import os
from joblib import dump

# create a folder where all trained models will be kept
if not os.path.exists("models"):
    os.makedirs("models")

dump(grid_search.best_estimator_, 'models/rf-clf.joblib')
```

Out[84]:
```
['models/rf-clf.joblib']
```

Model-Decision Trees

In [ ]:
```
# training a DescisionTreeClassifier
dtree_model = DecisionTreeClassifier(max_depth = 10).fit(x_train, y_train)
dtree_predictions = dtree_model.predict(x_test)

#calculating the accuracy score for Decision trees

#Accuracy is defined as:(fraction of correct predictions): correct predictions / total number of data points
score= accuracy_score(y_test, dtree_predictions)
print(score)
```

```
0.5673034482758621
```

In [ ]:
```
# recall score
recall_score(y_test, dtree_predictions,average="micro")
```

Out[86]:
```
0.5673034482758621
```

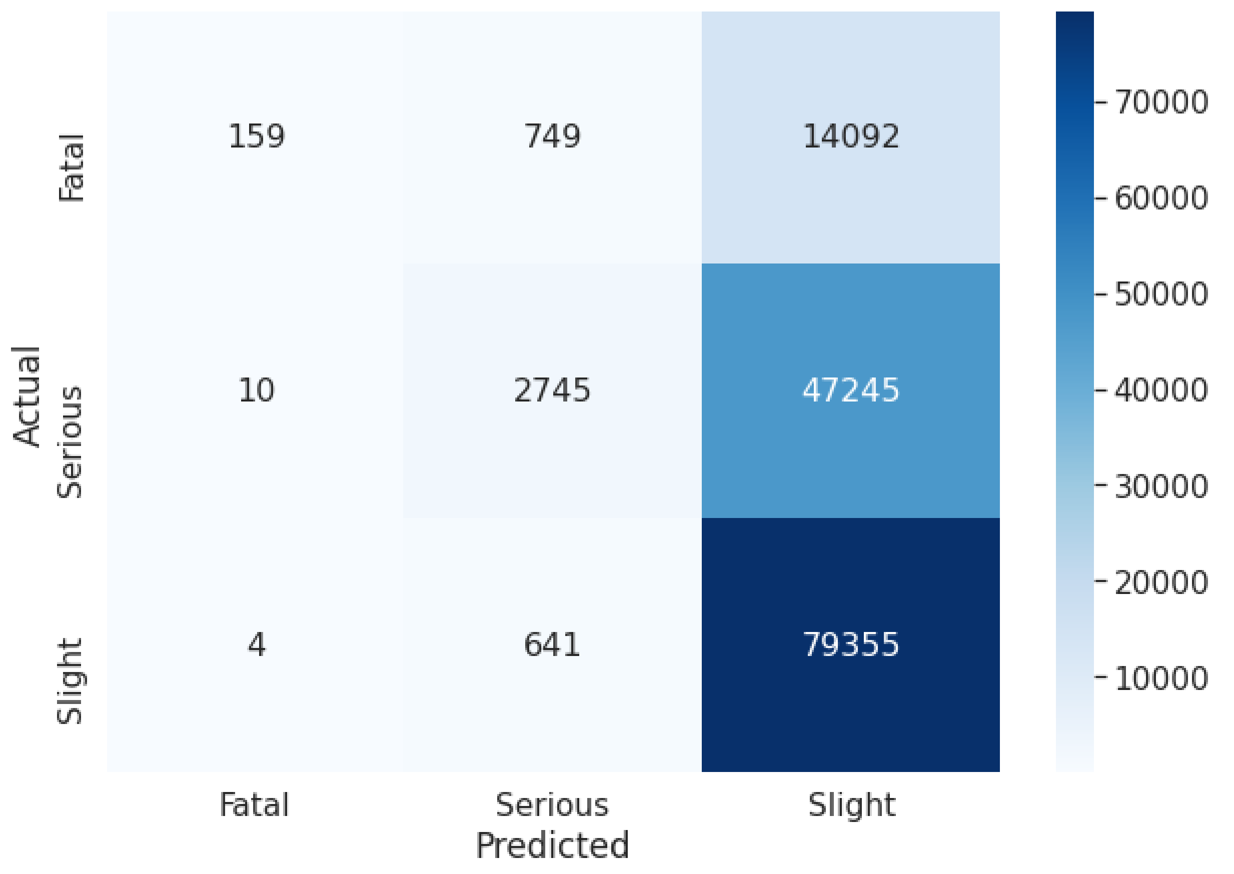
In [ ]:
```
# f1 score
f1_score(y_test, dtree_predictions,average="micro")
```

Out[87]:
```
0.5673034482758621
```

The simple regression tree has F-score of 0.57 which is better than The Random Forest model above which was having the F-score of 0.31.

In [ ]:
```
#confusion matrix
data = confusion_matrix(y_test, dtree_predictions)
df_cm = pd.DataFrame(data, columns=np.unique(y_test),index = np.unique(y_test))
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'
plt.figure(figsize = (10,7))
#sns.set(font_scale=1.4)#for label size
sns.heatmap(df_cm, cmap="Blues", annot=True,fmt='g', annot_kws={"size": 16})# font size
```

Out[88]: <matplotlib.axes._subplots.AxesSubplot at 0x7f288fc2b710>



Finding accuracy through different values of max_depth

Finding the optimal value for max_depth is one way way to tune the model. The code below outputs the accuracy for decision trees with different values for the max_depth.

In [ ]:
```
# List of values to try for max_depth:
max_depth_range = list(range(1, 15))

# List to store the average RMSE for each value of max_depth:
accuracy = []

for depth in max_depth_range:
    clf = DecisionTreeClassifier(max_depth = depth, random_state = 0)
    clf.fit(x_train, y_train)
    score = clf.score(x_test, y_test)
    accuracy.append(score)

accuracy
```

Out[139]:
```
[0.5517241379310345,
 0.5517241379310345,
 0.5517241379310345,
 0.5517241379310345,
 0.5526965517241379,
 0.5530689655172414,
 0.5546137931034483,
 0.5590689655172414,
 0.5620413793103448,
 0.5672413793103448,
 0.574951724137931,
 0.5816965517241379,
 0.5914551724137931,
 0.5999517241379311]
```

From the above accuracy score, it depicts that the accuracy score is improved by increasing the depth of the tree.

Moreover, it should be noted that max_depth is not the same thing as depth of a decision tree. max_depth is a way to preprune a decision tree. In other words, if a tree is already as pure as possible at its depth, it will not continue to split further.

Tuning - Decision Tree

In [ ]:
```
#making the instance
model= DecisionTreeClassifier(random_state=1234)
#Hyper Parameters Set
params = {'max_features': ['auto', 'sqrt', 'log2'],
          'min_samples_split': [2,3,4,5,6,7,8,9,10,11,12,13,14,15],
          'min_samples_leaf':[1,2,3,4,5,6,7,8,9,10,11],
          'max_depth': [3, 5, 15],
          'random_state':[123]}
#Making models with hyper parameters sets
model1 = GridSearchCV(model, param_grid=params, n_jobs=-1)
#Learning
model1.fit(x_train,y_train)

#Prediction
prediction_decision=model1.predict(x_test)
```

In [ ]:
```
#evaluation(Accuracy)
print("Accuracy:",metrics.accuracy_score(prediction_decision,y_test))
```

```
Accuracy: 0.5517241379310345
```

In [ ]:
```
# f1 score
f1_score(y_test, prediction_decision,average="micro")
```

Out[98]: 0.5517241379310345

In [ ]:
```
#The best hyper parameters set
print("Best Hyper Parameters:",model1.best_params_)
```

```
Best Hyper Parameters: {'max_depth': 3, 'max_features': 'auto', 'min_samples_leaf': 1, 'min_samples_split': 2, 'random_state': 123}
```

The best hyperparameters for Tuned Deciosn Tree has come out to be - Max depth=3, min_sample_leaf=1 and min_sample_split=2

Displaying feature importance for Decision Trees

In [ ]:
```
importances = pd.DataFrame({'feature':list(x_train.columns),'importance':np.round(model1.best_estimator_.feature_importances_,3)})
importances = importances.sort_values('importance',ascending=False)

importances
```

Out[100]:

	feature	importance
23	Urban	0.437
3	Latitude	0.409
14	Roundabout	0.131
25	Summer	0.010
16	Slip road	0.009
31	Other	0.003
0	Location_Easting_OSGR	0.000
21	None	0.000
22	Road Defect	0.000
24	Spring	0.000
26	Winter	0.000
19	Snow	0.000
27	Morning	0.000
28	Night	0.000
29	Road Hazard	0.000
30	Fog or mist	0.000
32	Raining	0.000
20	Water	0.000
17	Darkness - No lighting	0.000
18	Daylight	0.000
1	Location_Northing_OSGR	0.000
15	Single carriageway	0.000
13	One way street	0.000
12	Wednsday	0.000
11	Tuesday	0.000
10	Thursday	0.000
9	Sunday	0.000
8	Saturday	0.000
7	Monday	0.000
6	high_winds	0.000
5	Speed_limit	0.000
4	Number_of_Vehicles	0.000
2	Longitude	0.000
33	Snowing	0.000

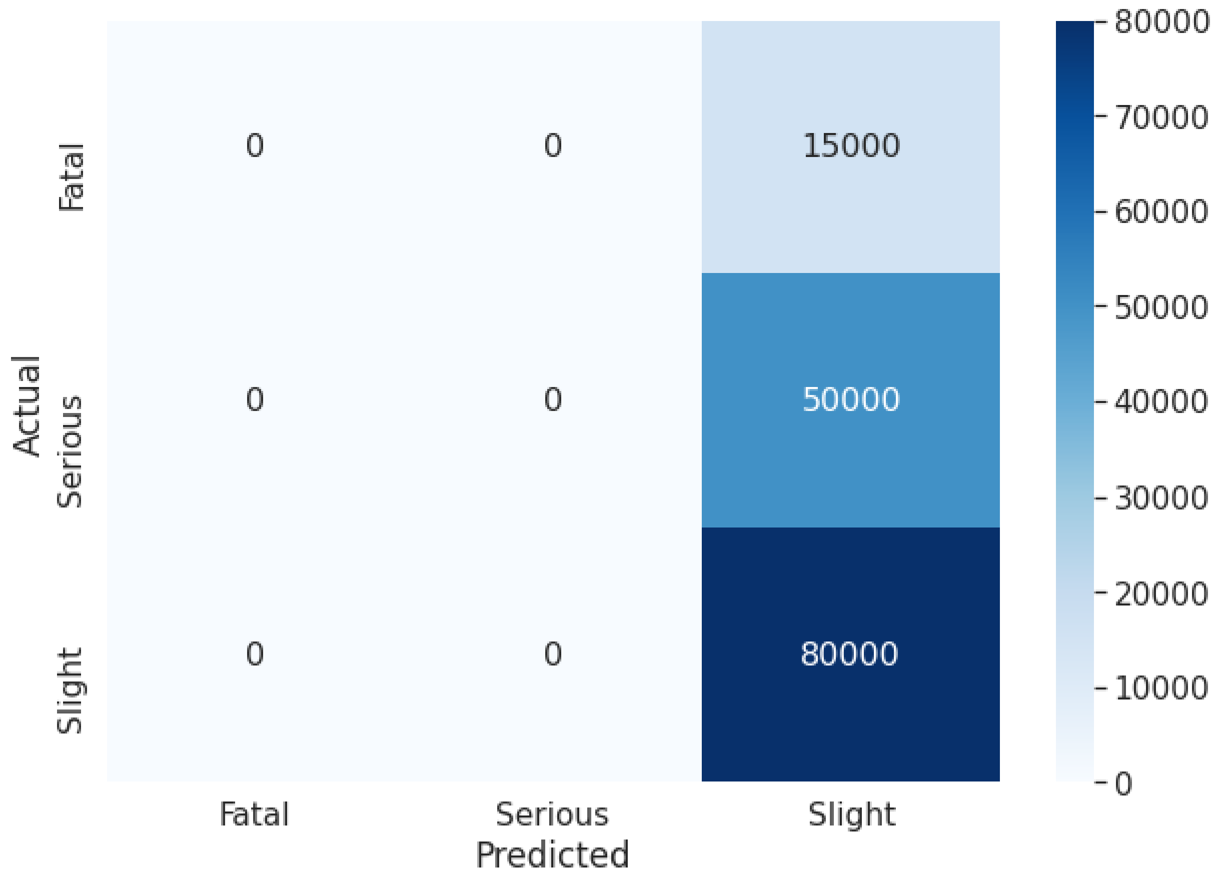
In [ ]:
```
# Creating a bar plot for feature importance visualization

plt.figure(figsize=(50,8))
sns.barplot(x=importances["feature"], y=importances["importance"])


# Add labels to your graph
plt.xlabel('Feature Importance Score')
plt.ylabel('Features')
plt.title("Visualizing Important Features")
plt.legend()
plt.show()
```

```
No handles with labels found to put in legend.
```

From the feature importance table, it is evident that Urban and Latitude holds high importance in prediction of the accident severity and, Roundabout holds moderate severity while many of the varaibles like Snowing, Longitude, Number of vehicles etc. holds leat importance in forecasting the Accident Severity.

In [ ]:
```
#confusion matrix
data = confusion_matrix(y_test, prediction_decision)
df_cm = pd.DataFrame(data, columns=np.unique(y_test),index = np.unique(y_test))
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'
plt.figure(figsize = (10,7))
#sns.set(font_scale=1.4)#for label size
sns.heatmap(df_cm, cmap="Blues", annot=True,fmt='g', annot_kws={"size": 16})# font size
```

Out[101]: <matplotlib.axes._subplots.AxesSubplot at 0x7f288fe18bd0>



Discussion of evaluation results

The accuracy and the F1-score the Decision Tree(initial and after hyper parameter tuning) is quite different from that of Random Forest.

Adding to this, the F1-score of the Decison Tree(initial) and after the Tuning are quite similar.

As per my analysis, on comparing the F1 scores between Baseline method(on basis of comparison of Majority Baseline, Random forest and SMOTE implementation for handling the variation of imbalanced proportion of Accident severity), Random Forest and Decision Trees, The baseline method is getting the better F1-score i.e 0.75 in tree based algorithm which is better as compared to Random forest i.e 0.31 and Decision Tree 0.55(hyper parameter tuned)

We are taking F1_score in account for this scenario as it holds the combine result of precision and recall.

The predictive models achieved performance below the baseline which signifies the model is not appropriate for my specific problem. the possible reasons could be noise in the data or Stochastic nature of the modeling algorithm.

When basline model with limited or no predictors outperform the other predictive models then one of the reasons can be **overfitting**

As a model's complexity grows, it becomes more capable of fitting the training data's peculiarities. If these peculiarities reflect something true about the environment, the more complicated fit may also produce better test data predictions.

A complex model, on the other hand, would eventually pick up random noise in the training data. In the training sample, this may minimise prediction error, but in the test sample, it will make predictions worse!

Possible Future improvements

The future improvements would include:

- Choosing a small number of predictors carefully based on theory as per the predictor importance.
- Using penalized regression approach such as LASSO or ridge regression.
- Using cross-validation to estimate the model's generalization error within the training set.

Possible scenarios to deploy the models in real-world business scenarios

As per the results of my predictive models, it is not adviceable to deploy them in real-world.

The above recommendations provided for future improvemnts might help to boost the accuarcy of the model and make it capable to deploy in real-world.