An Analysis of Canadian Marine Accident Fatality Factors

Initial Results and Code

Name: Rares Finatan
Student Number: 501140875
Supervisor: Uzair Ahmad, Ph.D
Week of November 21, 2022

Table of Contents

Section 1: Initial Analysis **3**

1.1 - Fundamental Data Analysis 3

Section 2: Data Cleaning **7**

2.1 - Basic Data Cleaning, Primitive Dimensionality Reduction 7

2.2 - Contextual Data Cleaning 9

2.2.1 - Removing Duplicate Attributes and Keeping Integer Encoding 11

2.2.2 - Removing Attributes Unfit for Encoding 12

2.2.3 - Removing Attributes with Low Variance 12

2.2.4 - Removing Attributes Irrelevant to Hypothesis 13

Section 3: Target Attribute Creation, and Building Train, Validation, Testing Sets **13**

3.1 - Target Variable Creation 13

3.2 - Data Splitting 15

Section 4: Formal Dimensionality Reduction **16**

4.1 - Removing Attributes with High Correlations 16

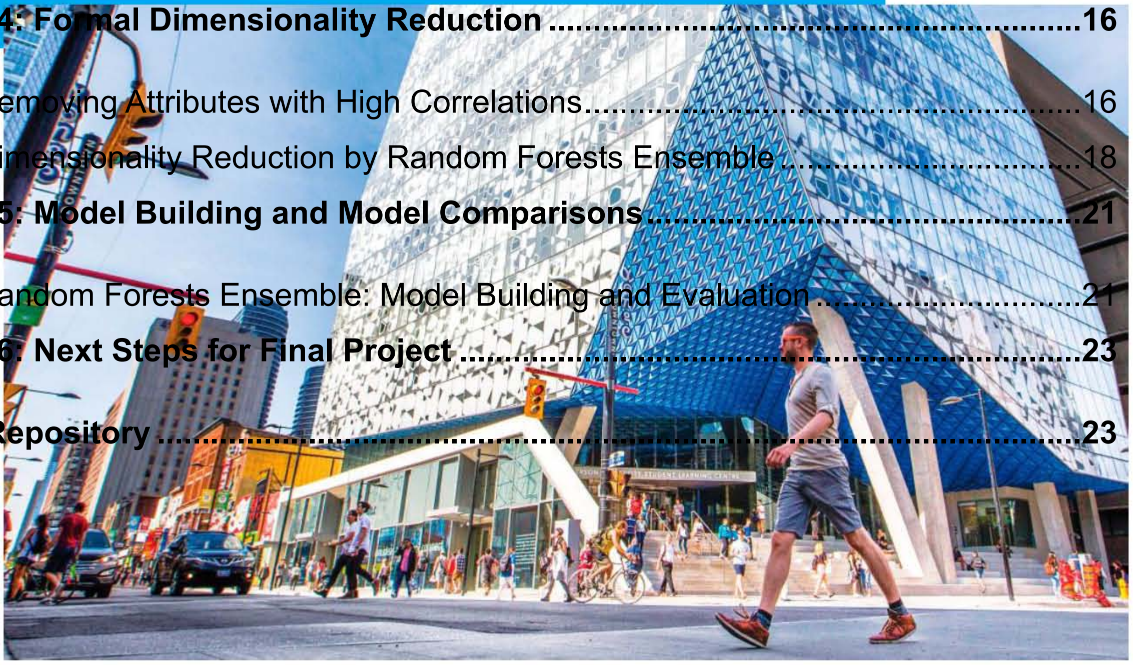
4.2 - Dimensionality Reduction by Random Forests Ensemble 18

Section 5: Model Building and Model Comparisons **21**

5.1 - Random Forests Ensemble: Model Building and Evaluation 21

Section 6: Next Steps for Final Project **23**

Github Repository **23**



Ryerson University

Section 1: Initial Analysis

1.1 - Fundamental Data Analysis

Before beginning any elementary data analysis, recall the datasets understudy:

- MARSISdb_MDOTW_VW_OCCURRENCE_PUBLIC - Data of marine occurrences in Canadian waters from January 1995 to present
- MARSISdb_MDOTW_VW_OCCURRENCE_VESSEL_PUBLIC - Vessel-specific data involving marine occurrences in Canadian waters from January 1995 to present

The two datasets are saved locally in the project's working directory from the provided URLs, in .CSV format. Analysis begins with initial data imports of the CSVs, stored as pandas data frames in two respective variables.

```
#initial data import
occurence_raw = 
pd.read_csv('MARSISdb_MDOTW_VW_OCCURRENCE_PUBLIC.csv')
vessel_raw = 
pd.read_csv('MARSISdb_MDOTW_VW_OCCURRENCE_VESSEL_PUBLIC.csv')
```

For the purposes of this study, both occurrence-specific attributes and vessel-specific attributes are taken under consideration in order to analyze their respective impact on fatality occurrences. The two datasets share the same occurrence primary key, `OccID`. In order to have both datasets' attributes in a single data frame, they are merged using the following command:

```
#merge the raw data sets
marsis_raw = pd.merge(
    occurence_raw,
    vessel_raw,
    how = 'inner',
    left_on = 'OccID',
    right_on = 'OccID')
```

With the merged data frame, one can begin inspecting the structure of the data and its respective attributes. This step is crucial in understanding the amount of rows, columns, and types of attributes comprising each record.

```
#view structure of the dataset
marsis_raw.info()
```

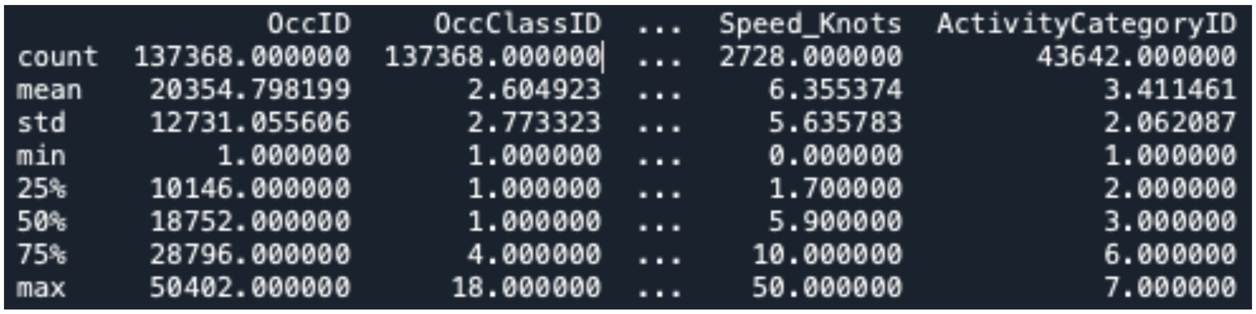
```
Out[]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 137368 entries, 0 to 137367
Columns: 194 entries, OccID to ActivityCategoryDisplayFre
dtypes: float64(68), int64(12), object(114)
```

```
#view attribute data types
marsis_raw.dtypes
```

```
Out[]:
OccID                              int64
OccNo                             object
OccClassID                         int64
OccClassDisplayEng                object
OccClassDisplayFre                object
VesselPhaseDisplayFre             object
Speed_Knots                      float64
ActivityCategoryID               float64
ActivityCategoryDisplayEng        object
ActivityCategoryDisplayFre        object
Length: 194, dtype: object
```

Glimpsing at the combined data frame using pandas also provides a view of some basic descriptive statistics for each attribute.

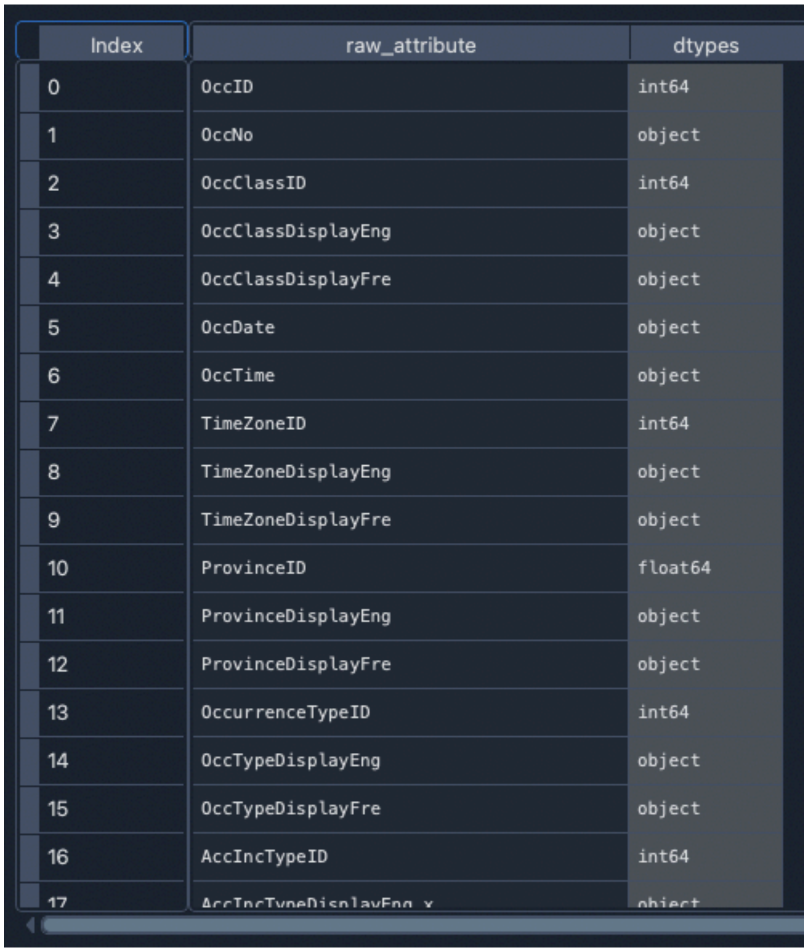
```
#view structure of combined data frame using pandas
pd.DataFrame.describe(marsis_raw)
```



```
               OccID      OccClassID  ...  Speed_Knots  ActivityCategoryID
count  137368.000000   137368.000000  ...  2728.000000        43642.000000
mean    20354.798199        2.604923  ...     6.355374            3.411461
std     12731.055606        2.773323  ...     5.635783            2.062087
min         1.000000        1.000000  ...     0.000000            1.000000
25%     10146.000000        1.000000  ...     1.700000            2.000000
50%     18752.000000        1.000000  ...     5.900000            3.000000
75%     28796.000000        4.000000  ...    10.000000            6.000000
max     50402.000000       18.000000  ...    50.000000            7.000000
```

For the extracted data types, one can store them into a data frame for further analysis or manipulation. The data types of the columns are stored in a data frame, pointing to a variable named `marsis_data_types`.

```
#create a dataframe of the datatypes in the dataset
marsis_data_types = marsis_raw.dtypes.to_frame('dtypes').reset_index()
marsis_data_types.rename(columns = {'index':'raw_attribute'}, inplace
= True)
```

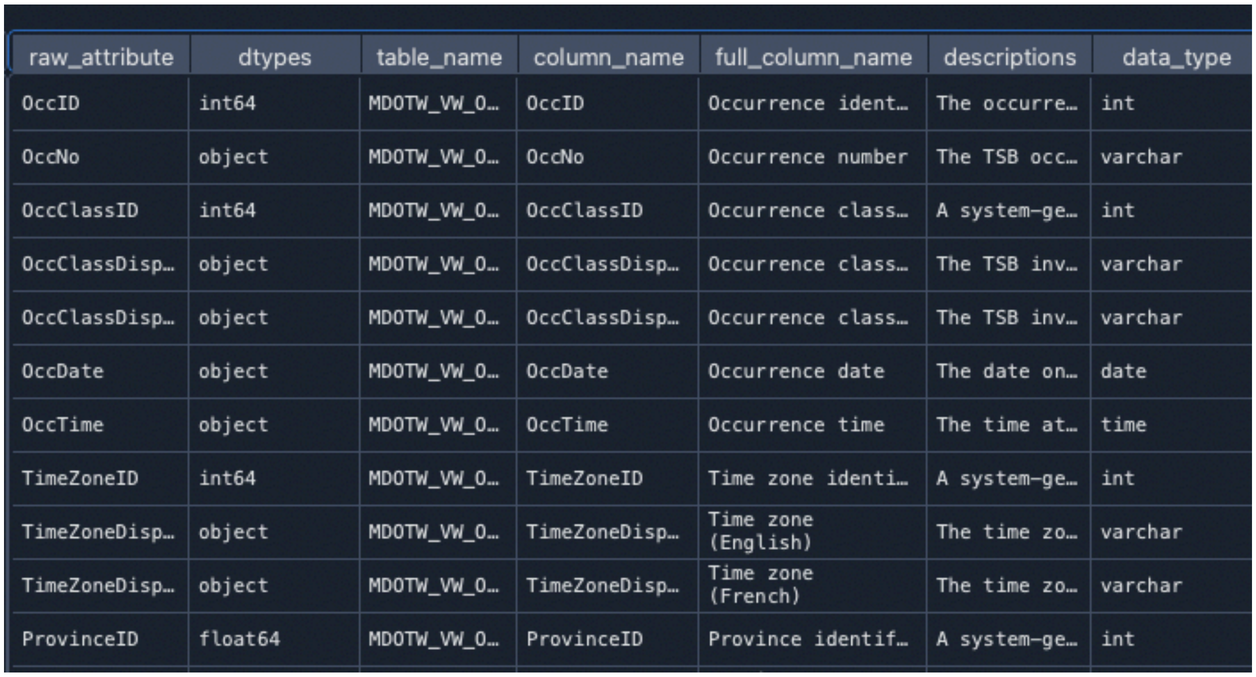


Index	raw_attribute	dtypes
0	OccID	int64
1	OccNo	object
2	OccClassID	int64
3	OccClassDisplayEng	object
4	OccClassDisplayFre	object
5	OccDate	object
6	OccTime	object
7	TimeZoneID	int64
8	TimeZoneDisplayEng	object
9	TimeZoneDisplayFre	object
10	ProvinceID	float64
11	ProvinceDisplayEng	object
12	ProvinceDisplayFre	object
13	OccurrenceTypeID	int64
14	OccTypeDisplayEng	object
15	OccTypeDisplayFre	object
16	AccIncTypeID	int64
17	AccIncTypeDisplayEng_x	object

The output data frame containing pandas-identified data types.

With the pandas-identified data types in the merged dataset, one can compare the attributes' data types with the ones provided by the author of the dataset via a data dictionary. The data dictionary for this study is found here, and imported into the IDE as `MARSISdb-dd-processed.csv`, and consequently stored in a new variable `marsis_dd`. The data dictionary is then merged with the previously created `marsis_data_types` data frame in order to view the pandas-identified data types and the author-identified data types side-by-side. This side-by-side comparison data frame is stored in a new variable, `data_type_comparison`.

```
#compare identified data types to provided data dictionary
marsis_dd = pd.read_csv("MARSISdb-dd-processed.csv")
marsis_dd = marsis_dd[marsis_dd.table_name == 
'MDOTW_VW_OCCURRENCE_PUBLIC']
data_type_comparison = pd.merge(
    marsis_data_types,
    marsis_dd,
    how = "inner",
    left_on = "raw_attribute",
    right_on = "column_name"
)
```



raw_attribute	dtypes	table_name	column_name	full_column_name	descriptions	data_type
OccID	int64	MDOTW_VW_O…	OccID	Occurrence ident…	The occurre…	int
OccNo	object	MDOTW_VW_O…	OccNo	Occurrence number	The TSB occ…	varchar
OccClassID	int64	MDOTW_VW_O…	OccClassID	Occurrence class…	A system-ge…	int
OccClassDisp…	object	MDOTW_VW_O…	OccClassDisp…	Occurrence class…	The TSB inv…	varchar
OccClassDisp…	object	MDOTW_VW_O…	OccClassDisp…	Occurrence class…	The TSB inv…	varchar
OccDate	object	MDOTW_VW_O…	OccDate	Occurrence date	The date on…	date
OccTime	object	MDOTW_VW_O…	OccTime	Occurrence time	The time at…	time
TimeZoneID	int64	MDOTW_VW_O…	TimeZoneID	Time zone identi…	A system-ge…	int
TimeZoneDisp…	object	MDOTW_VW_O…	TimeZoneDisp…	Time zone (English)	The time zo…	varchar
TimeZoneDisp…	object	MDOTW_VW_O…	TimeZoneDisp…	Time zone (French)	The time zo…	varchar
ProvinceID	float64	MDOTW_VW_O…	ProvinceID	Province identif…	A system-ge…	int

A glimpse of the first rows of the pandas-identified data types and the authored data dictionary data types.

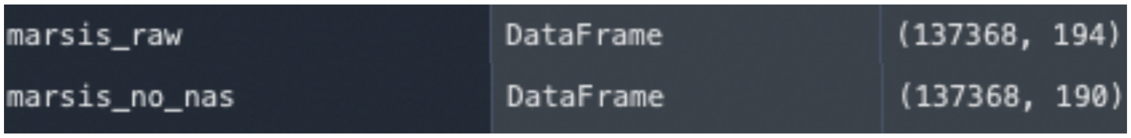
Section 2: Data Cleaning

2.1 - Basic Data Cleaning, Primitive Dimensionality Reduction

As part of elementary data cleaning, some inconsistencies in the data set need to be addressed before formal dimensionality reduction. Given that many data analysis tools are unable to accept missing data when training models, elimination of missing data records or columns may be necessary, despite the loss of information incurred. Removing records and columns that are completely empty will not impact the inferential capacity of a model given that no data is present for an attribute.

```
#remove attributes with no values
marsis_no_nas = marsis_raw.copy(deep=True)
marsis_no_nas.dropna(how='all', axis='columns', inplace=True)
```

With this command, the data set under study is reduced from 194 dimensions, to 190. Four attributes had no data records at all.



```
marsis_raw        DataFrame    (137368, 194)
marsis_no_nas     DataFrame    (137368, 190)
```

Given that the data set is published by the Government of Canada and its Transformation Safety Board (TSB), the attributes that are non-numerical are duplicated in both English and French. For the purposes of analyzing the attributes and their record row values, only one language is necessary. Continue cleaning the data set by dropping all French attributes, denoted by a text string ending in 'DisplayFre'.

```
#some columns are duplicated in English and French
#only require the English attribute variants, remove the French
marsis_eng_only = marsis_no_nas.copy(deep=True)
marsis_eng_only.drop(marsis_eng_only.filter(regex='DisplayFre').column
s, axis=1, inplace=True)
```

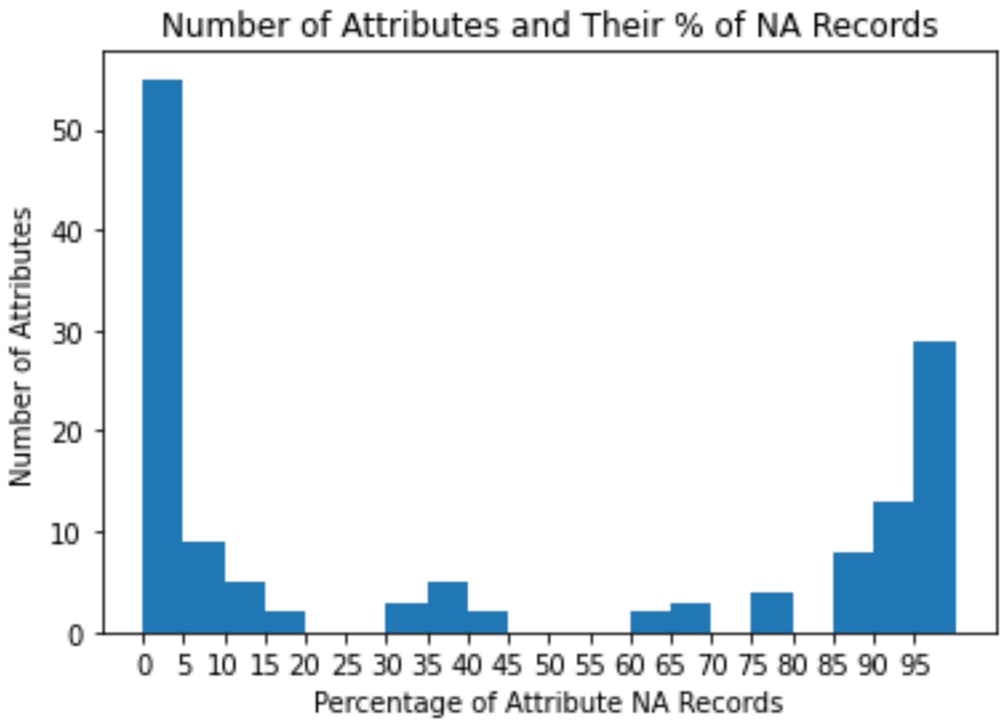
By removing French duplicated attributes, the data set under study is further reduced from 190 dimensions to 140.

In addition to removing attributes which have 100% NA records, one ought to also reduce the amount of dimensions that have NAs that exceed a particular proportionality relative to the overall attribute's records. Determine which attributes have more than 80% NA records, add them to a list, and remove all attributes which comprise of more than 80% NA records.

```
#determine each column's percentage of NA records
nas_percentage = (marsis_eng_only.isna().mean().round(2) * 
100).to_frame(name = 'percentage')
#determine the columns with high percentage of NAs, more specifically 
greater than 80% NAs
nas_80_percent = nas_percentage.loc[nas_percentage['percentage'] > 80]
```

Next, a visual check is conducted for the anticipated inflection point for NA percentages. One ought to verify if 80% NA records is an appropriate threshold for attribute elimination.

```
#visualize distribution of NA percentages
import matplotlib.pyplot as plt
import numpy as np
plt.hist(nas_percentage['percentage'], bins = 20)
plt.xticks(np.arange(0, max(nas_percentage['percentage']), 5))
plt.title('Number of Attributes and Their % of NA Records')
plt.xlabel('Percentage of Attribute NA Records')
plt.ylabel('Number of Attributes')
plt.show()
```



Judging from the visualization, 80% NAs seems to be a good threshold to remove attributes. There are too few attributes in between 85% and 20% to account for significant dimensionality reduction. Next, a list of attributes is created that matches the 80% NAs condition, and after, the attributes contained in the list are dropped from the overall data set.

```
#create a list of the attributes that have higher than 80% NA records
nas_80_percent.reset_index(inplace=True)
nas_80_percent_list = list(nas_80_percent['index'])

#remove the attributes with more than 80% NA records
marsis_no_nas_80 = marsis_eng_only.drop(columns = nas_80_percent_list)
```

By removing the attributes with more than 80% NA records, the data sets dimensions have been further reduced from 140 to 90.

Continuing with basic dimensionality reduction, a separate source of NAs persists in the processed data set so far, whereby NAs are masked as NULL text strings. All columns which are completely containing NAs masked as NULL text strings are removed.

```
#there are also columns that have NA values masked as NULL strings
#identify the columns with all NULL string records
marsis_nulls = marsis_no_nas_80.loc[: , ((marsis_no_nas_80 == 
'Null').any())]
marsis_nulls = list(marsis_nulls.columns)

#remove the columns with all NULL string records
marsis_basic_cleaned = marsis_no_nas_80.drop(columns = marsis_nulls)
```

With this command, only a single attribute was dropped from the data set, to a new running total of 89 attributes.

2.2 - Contextual Data Cleaning

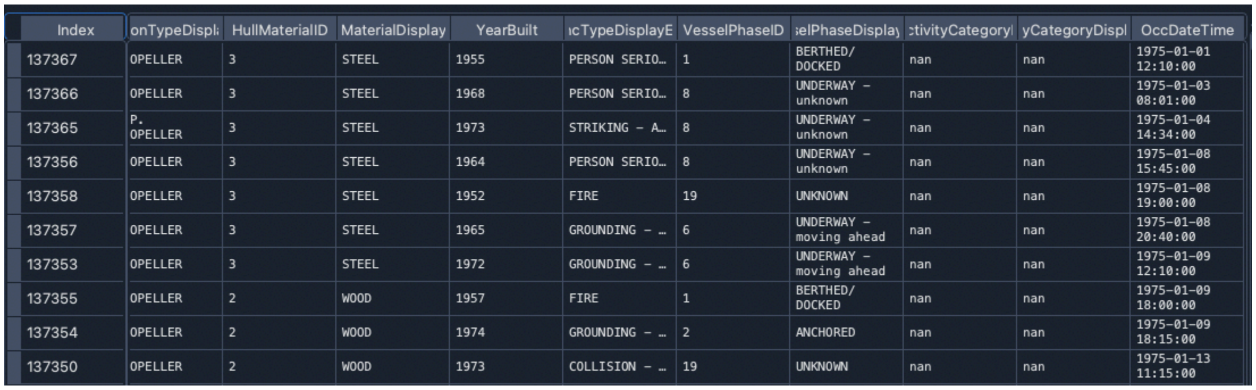
Up until this point, the raw data and its respective iterations of processing had been saved to new variables to account for any retroactive errors or overzealous data cleansing. In addition, the data cleansing techniques employed in section 2.1 were wide-sweeping and general in nature; they are techniques that likely could be deployed to any dataset. The next step is to contextualize data munging relative to the study's hypothesis - identifying deterministic factors of marine occurrences resulting in fatalities in Canadian waters from 1975 to the present day. From herein, cleansing of the dataset will occur iteratively and have its results stored in a data frame variable named `marsis_processed`.

```
#given the context of the study, the area of focus is occurrences 
resulting in deaths
marsis_processed = marsis_basic_cleaned.copy(deep=True)
```

The processed dataset contains many records which are seemingly duplicates. Particularly when analyzing the attributes `OccID` (the unique ID assigned to an occurrence) and `OccNo` (the unique case file number assigned to an occurrence), one can see that multiple `OccID` and `OccNo` records refer to the same event. This is because the Government of Canada's marine authorities are required to log individual phases of an occurrence, while maintaining a record of which occurrences had multiple phases. The recommended cleaning technique is to keep records with the most recent date-time stamps as they are rows containing the final outcome of each marine occurrence.

```
#to identify the unique cases of deadly occurrences, duplicate OccIDs 
must be dropped, and only the most recent is kept
#the most recent OccID is kept because it houses the final 
investigative data leading to the occurrence's result

#all OccDate instances have a 12:00:00 timestamp applied to them, 
whether actual or not
#strip timestamp from OccDate attribute records
marsis_processed['OccDate'] = 
marsis_processed['OccDate'].str.replace(' 12:00:00 AM', '')

#some OccTime instances have NA values; assume 00:00:00 timestamp 
given lack of information
marsis_processed['OccTime'] = 
marsis_processed['OccTime'].fillna('00:00:00')

#concatenate date and time attributes
marsis_processed['OccDateTime'] = marsis_processed['OccDate'] + ' ' + 
marsis_processed['OccTime']

#ensure date attributes are of dtype 'datetime'
marsis_processed['OccDateTime'] = 
pd.to_datetime(marsis_processed['OccDateTime'])
```

```
#drop duplicate OccId instances, but keep latest OccId instance
marsis_processed_unique =
marsis_processed.sort_values('OccDateTime').drop_duplicates('OccID',
keep = 'last')
```



Index	onTypeDispl	HullMaterialID	MaterialDisplay	YearBuilt	icTypeDisplayE	VesselPhaseID	selPhaseDisplay	ctivityCategoryI	yCategoryDispl	OccDateTime
137367	OPELLER	3	STEEL	1955	PERSON SERIO…	1	BERTHED/ DOCKED	nan	nan	1975-01-01 12:10:00
137366	OPELLER	3	STEEL	1968	PERSON SERIO…	8	UNDERWAY – unknown	nan	nan	1975-01-03 08:01:00
137365	P. OPELLER	3	STEEL	1973	STRIKING – A…	8	UNDERWAY – unknown	nan	nan	1975-01-04 14:34:00
137356	OPELLER	3	STEEL	1964	PERSON SERIO…	8	UNDERWAY – unknown	nan	nan	1975-01-08 15:45:00
137358	OPELLER	3	STEEL	1952	FIRE	19	UNKNOWN	nan	nan	1975-01-08 19:00:00
137357	OPELLER	3	STEEL	1965	GROUNDING – …	6	UNDERWAY – moving ahead	nan	nan	1975-01-08 20:40:00
137353	OPELLER	3	STEEL	1972	GROUNDING – …	6	UNDERWAY – moving ahead	nan	nan	1975-01-09 12:10:00
137355	OPELLER	2	WOOD	1957	FIRE	1	BERTHED/ DOCKED	nan	nan	1975-01-09 18:00:00
137354	OPELLER	2	WOOD	1974	GROUNDING – …	2	ANCHORED	nan	nan	1975-01-09 18:15:00
137350	OPELLER	2	WOOD	1973	COLLISION – …	19	UNKNOWN	nan	nan	1975-01-13 11:15:00

A glimpse into a few rows of the dataset showing unique occurrences, with only the most recent date-time stamp intact for each occurrence.

2.2.1 - Removing Duplicate Attributes and Keeping Integer Encoding

Record rows within the dataset are now unique based off of the attribute `OccID`. The processed dataset is now comprised of 45,059 records and 90 attributes, which is still containing many attributes undesirable for model building and evaluation. Within the `marsis_processed` data frame, there are many duplicate attributes. These duplicate attributes are shown in the dataset firstly as an encoded value (using TSB nomenclature), followed by a textual representation of the encoded value. For model creation, one ought to only keep encoded values - any other attribute classes would need to be encoded separately using one-hot encoding, or another preferred method.

```
#remove attributes that have an equivalent attribute, but keep the
attribute that has integer encoding

#OccID and OccNo represent the same occurrence, but only one is
necessary for classification
marsis_processed = marsis_processed_unique.drop(columns = 'OccNo')
```

```
#OccClassID and OccClassDisplayEng are identical, but only OccClassID 
is integer encoded
marsis_processed = marsis_processed.drop(columns = 
'OccClassDisplayEng')
```

```
#the same duplicate attribute (integer encoding vs non-encoded) 
scenario applies to all attributes ending in '[...]DisplayEng'
marsis_processed.drop(marsis_processed.filter(regex='DisplayEng').colu
mns, axis=1, inplace=True)
```

2.2.2 - Removing Attributes Unfit for Encoding

Within the processed data frame, the reduced dimensions also include several attributes that cannot be encoded efficiently due to the contents of their records. Some variables that are unfit for encoding are the textual summaries of the occurrences, the reporting officers' names filing the occurrence, the textual descriptions of the nearest identifiable location to the occurrence location, and more. These attributes are subsequently dropped from the `marsis_processed` data frame.

```
#the dataset contains a set of attributes too varied or complex in 
their contents to be encoded
#remove attributes unfit for encoding 
unfit_for_encoding = ['IICName', 'Summary', 
'NearestLocationDescription', 'OccDate', 'OccTime', 'OccDateTime', 
'WindDirection']

marsis_processed = marsis_processed.drop(columns = unfit_for_encoding)
```

2.2.3 - Removing Attributes with Low Variance

In order to distill the dataset to its most relevant attributes, a low variance filter should also be applied to the `marsis_processed` data frame. Any attributes with variance beyond a certain threshold, or approximating zero variance, should be dropped. In the case of the study at hand, the numeric attributes' variance is calculated, stored in a list if they are approaching zero variance, and are then removed. Before the attributes are added to the removal list, they are manually contextualized with respect to the hypothesis to ensure no critical data is lost.

```
#determine variance of records per column in dataset
attribute_variance = marsis_processed.var(numeric_only = 
True).to_frame()
```

```
#remove low variance attributes from dataset
low_variance_attributes = ['IncludedInDailyEnum', 
'MajorChangesIncludedInDaily', 'LatEnum', 'LongEnum']
```

```
marsis_processed = marsis_processed.drop(columns = 
low_variance_attributes)
```

2.2.4 - Removing Attributes Irrelevant to Hypothesis

The final step of data this project's cleansing is removing any attributes which may be irrelevant to the hypothesis under study. These are primarily attributes which serve an administrative purpose on when records were filed, closed, released, and published by the Transportation Safety Board. They are attributes which only exist for internal reporting or record identification purposes, and are to be dropped from the processed dataset.

```
#remove administrative attributes
administrative_attributes = ['ReleasedDate', 'OccClosedDate', 
'EntryDate', 'ReportedDate', 'ReportedByID', 'OccID', 'TimeZoneID']
```

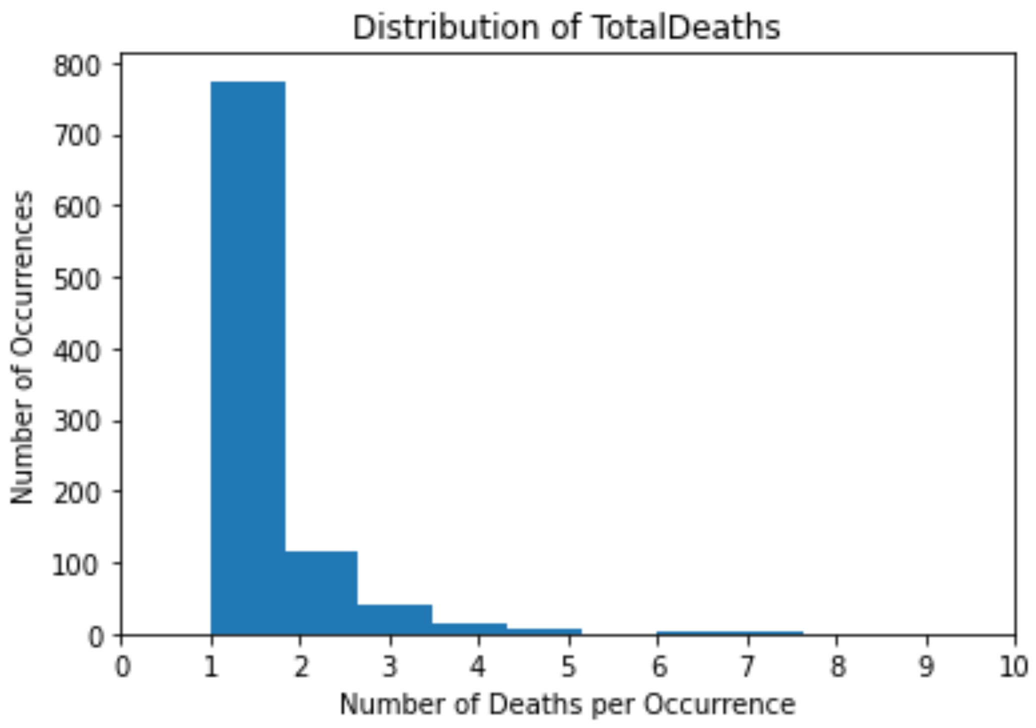
```
marsis_processed = marsis_processed.drop(columns = 
administrative_attributes)
```

Section 3: Target Attribute Creation, and Building Train, Validation, Testing Sets

3.1 - Target Variable Creation

From the final `marsis_processed` dataset, one can infer the target variable to be `TotalDeaths`, but this assumption is not optimal for the hypothesis under study. The hypothesis aims to identify the deterministic factors in occurrences resulting in a marine fatality, and not to quantify the amount of deaths caused by marine occurrence attributes. This frames the study as one which identifies the classification of a marine fatality occurrence. Firstly, calculating the number of class levels required for fatality classification is required.

```
#store distribution of TotalDeaths
marsis_deadly = marsis_processed[marsis_processed['TotalDeaths'] > 0]

#the range of deaths is between 0, and 84
#most deaths are distributed between 0 and 5
plt.hist(marsis_deadly['TotalDeaths'], bins = 100)
plt.xticks(np.arange(0, max(marsis_deadly['TotalDeaths']), 1))
plt.title('Distribution of TotalDeaths')
plt.xlabel('Number of Deaths per Occurrence')
plt.ylabel('Number of Occurrences')
plt.xlim(0, 10)
plt.show()
```



```
#the average death count for fatal occurrences is 1.45 deaths
import statistics
statistics.mean(marsis_deadly['TotalDeaths'])

#since the number of average deaths is less than 2, the target 
variable death classifier will be a binary Yes or No
#add the target variable OccDeathClassID with the classification 1 = 
yes, 0 = no
marsis_processed['OccDeathClassID'] = 
np.where(marsis_processed['TotalDeaths']!= 0, 1, 0)
```

3.2 - Data Splitting

With the target variable OccDeathClassID appended as a binary classification attribute to the marsis_processed data frame, the next step is to create input and output arrays for further splitting into training and test sets.

The input array will be a data frame composed of all the attributes under study, excluding the target variable and any of its direct attribute dependents. These attribute dependents are attributes which simulate or have an equivalent data value that strongly correlates to the target variable's classification.

```
#create input array (excluding target variable and its direct 
attribute dependents)
x = marsis_processed.loc[:, 
~marsis_processed.columns.isin(['OccDeathClassID', 'TotalDeaths', 
'OccClassID', 'ImoClassLevelID'])]
```

The corresponding output array is a 1 dimensional data frame consisting only of the target variable, 'OccDeathClassID'.

```
#create output array (including target variable)
y = marsis_processed['OccDeathClassID'].to_frame()
```

With input and output arrays separating the target variables from the input variables, one can use sklearn's model_selection module to invoke a train_test_split() function to create training and testing data sets. Training sets are defined as x_train (with the input array attributes), and y_train (with the output array attribute). Similarly, the test sets are defined as x_test (with the input array attributes), and y_test (with the output array attribute). The split between training and testing sets is set at an 80/20 proportionality, with a stratified sampling technique whereby an approximately equal quantity of target attribute records are distributed amongst both training and testing sets.

```
import numpy as np
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(
    x, y,
    train_size = 0.8,
    test_size = 0.2,
    # random_state = 4,
    stratify = y)
```

One can verify with a subsequent command what the stratified sampling proportion approximates to. In this case, `y_train` has approximately 2.136% observations with a positive `OccDeathClassID`, and y_test has approximately 2.141% observations with a positive `OccDeathClassID`.

```
#verify proportionality of stratification in output array
y_train.dtypes
stratified_y_train = len(y_train[y_train['OccDeathClassID'] == 1]) / 
len(y_train)
print(stratified_y_train)
```

```
Out[]: 0.021361000915471468
```

```
stratified_y_test = len(y_test[y_test['OccDeathClassID'] == 1]) / 
len(y_test)
print(stratified_y_test)
```

```
Out[]: 0.021415889924545052
```

Section 4: Formal Dimensionality Reduction

Begin by loading Section 4 dependencies into the IDE:

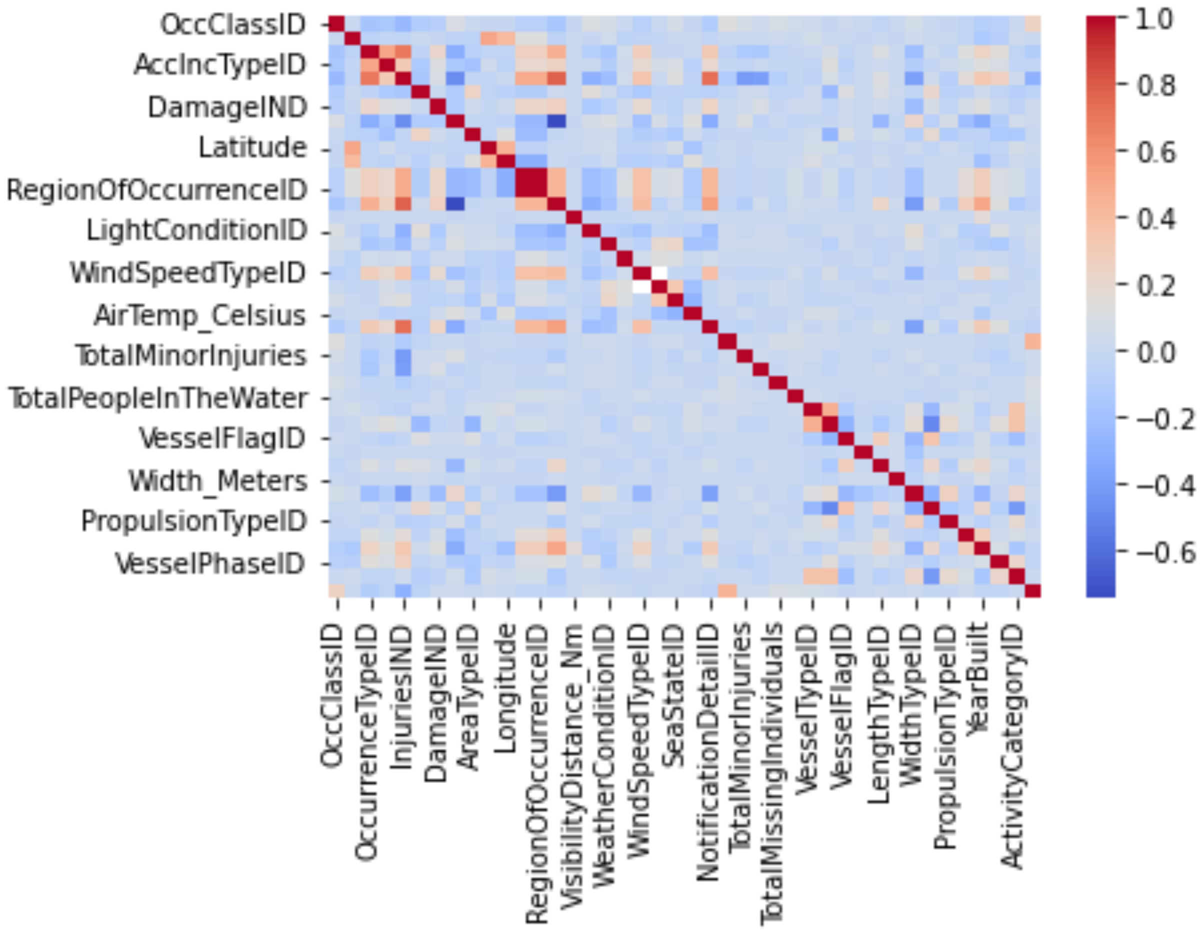
```
import xgboost
import numpy as np
import seaborn as sns
import sklearn.model_selection
from matplotlib import pyplot
from math import sqrt
```

4.1 - Removing Attributes with High Correlations

One of the selected methods of formal dimensionality reduction for this study is the identification and removal of highly correlated attributes in the already reduced feature set.

From the remaining attributes in the `marsis_processed` data frame, one can store the results of a correlation matrix to identify any directional relationships between the attributes. These directionalities are afterwards visualized as to easily identify them using a heatmap.

```
#verify if there are attributes that are too closely correlated and
are dependents of the target variable
attribute_correlation = marsis_processed.corr()
sns.heatmap(marsis_processed.corr(), fmt='.2g',cmap= 'coolwarm')
```



The strongest relationships identified include ‘`OccClassID`’ and ‘`TotalDeaths`’, which have already been removed from the training and testing split data. No further dimensionality reduction using this technique is appropriate given the results.

4.2 - Dimensionality Reduction by Random Forests Ensemble

In the processed dataset, many records contain NAs, and for some attributes' records, imputing them with a value may not make the most sense. To enable any further dimensionality reduction, the simplest way to identify the most important features is to use a modelling algorithm that supports missing values. Given that scikit-learn models are not capable of dealing with missing values, one can opt for XGBoost models to accomplish this task with potentially better computational efficiency.

Given the previously calculated proportionality of stratified sampling with respecting to `y_train` and `y_test` class levels, it is evident that the MARSIS dataset is very imbalanced. Out of all the records from 1975 to 2022, only 2% on average have an incident attached to them resulting in a maritime fatality. To compensate for this imbalance, the XGBRFClassifier will use the hyperparameter `scale_pos_weight` to give greater weight to the target variable's positive class. Ideally, the value of `scale_pos_weight` should correspond to the same proportionality of total negative samples to total positive samples. With this technique, one can mimic oversampling the data wherein the positive class is more prevalent. The hyperparameter `scale_pos_weight` will also be tweaked manually through the use of a floating point factor stored in a variable named `tweaking_param`.

```
#with XGBRFClassifier(), pass scale_pos_weight = x, where x is the 
total_negative_examples / total_positive_examples

#use tweaking param to adjust scale_pos_weight impact on False 
Positive and False Negative results
tweaking_param = 0.45
estimate = ((len(y_train[y_train['OccDeathClassID'] == 0])) / 
(len(y_train[y_train['OccDeathClassID'] == 1]))) * tweaking_param
```

An additional consideration that must be taken with the XGBRFClassifier is that it cannot support bootstrapping at each individual tree level in the random forest. To simulate bootstrapping, the subsample parameter in the model definition will serve to sample 80% of the data specified - a sufficient amount to train the model on, while also introducing some variability into the ensemble.

Lastly, a consideration needs to be made for how the model will designate node split points within the random forest. The optimal node splitting heuristic parameter can be calculated by the following formula:

$$h = \frac{\sqrt{|features|}}{|features|}$$

```
#define the random forest ensembles model
heuristic_parameter = sqrt(len(marsis_processed.columns)) / 
len(marsis_processed.columns)
model = xgboost.XGBRFClassifier(n_estimators = 100, subsample = 0.8, 
colsample_bynode = heuristic_parameter, scale_pos_weight = estimate)
```

The performance of the model on its validation data will be performed via repeated k-fold cross validation, with three repeats across 10 folds.

```
#evaluate the model using repeated k-fold cross validation, with three 
repeats and 10 folds
cv = sklearn.model_selection.RepeatedStratifiedKFold(n_splits=10, 
n_repeats=3, random_state=1)

#evaluate the model and collect the scores
n_scores = sklearn.model_selection.cross_val_score(model, x_train, 
y_train, scoring='accuracy', cv=cv, n_jobs=-1)

#evaluate model accuracy by taking the mean of the cross validation 
scores, recording its standard deviation
print('Mean Accuracy: %.2f (%.2f)' % (np.mean(n_scores), 
np.std(n_scores)))
```

```
Out[]: 0.94 (0.00)
```

Prior to applying the `scale_pos_weight` hyperparameter, the model's accuracy was closer to 99%, indicative of overfitting. After applying an appropriate value for `scale_pos_weight`, the model's accuracy is at 94% on the validation set, but its accuracy score alone cannot determine its ability to predict skillfully.

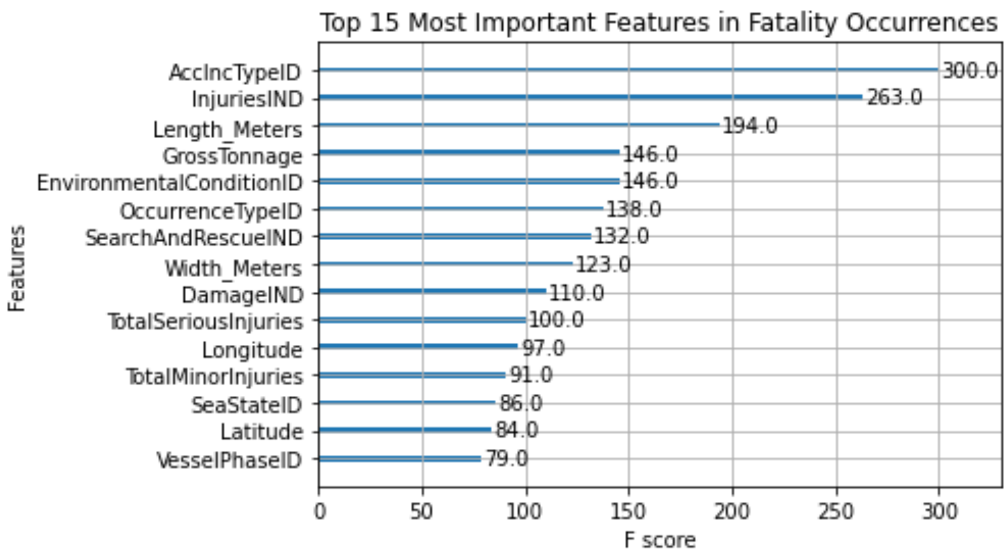
The model is ready to be fit to the training datasets using the following command:

```
#fit the random forests model to the training data
model.fit(x_train, y_train)
```

Having trained the model on the created training data sets, one is now able to extract the most important features contributing to an `OccDeathClassID` level. This is done by creating an importance matrix where the metric under consideration is the information gain of splits using each respective feature. This importance matrix is then visualized, showing the top 15 most importance features contributing to marine fatalities. The full importance matrix is also made available in a data frame named `feature_importance_df`.

```
#store feature importance
feature_important = 
model.get_booster().get_score(importance_type='gain')
keys = list(feature_important.keys())
values = list(feature_important.values())
feature_importance_df = pd.DataFrame(data=values, index=keys, 
columns=["score"])
```

```
#visualize feature importance
xgboost.plot_importance(model, title = 'Top 15 Most Important Features 
in Fatality Occurrences', max_num_features = 15)
pyplot.show()
```



Section 5: Model Building and Model Comparisons

5.1 - Random Forests Ensemble: Model Building and Evaluation

With the model built out and its most important features identified, one can test it on the previously created testing data sets. The next step is to pass the model the test data set and report on its accuracy.

```
#make predictions for test data and evaluate
from sklearn.metrics import accuracy_score
```

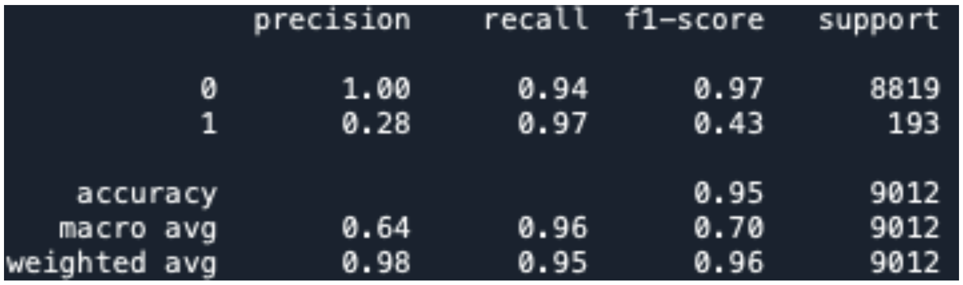
```
y_pred = model.predict(x_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy: %.2f%%" % (accuracy * 100.0))
```

```
Out[]: Accuracy: 94.53%
```

For this run, the model maintains its same degree of accuracy as per the training data set. In addition to measuring accuracy, one can also create a classification report of the model's performance using sklearn.metrics.

```
#test and pred results
from sklearn.metrics import classification_report
```

```
print(classification_report(y_test, y_pred))
results_comparison = x_test.assign(target = y_test.values, prediction
= y_pred)
```

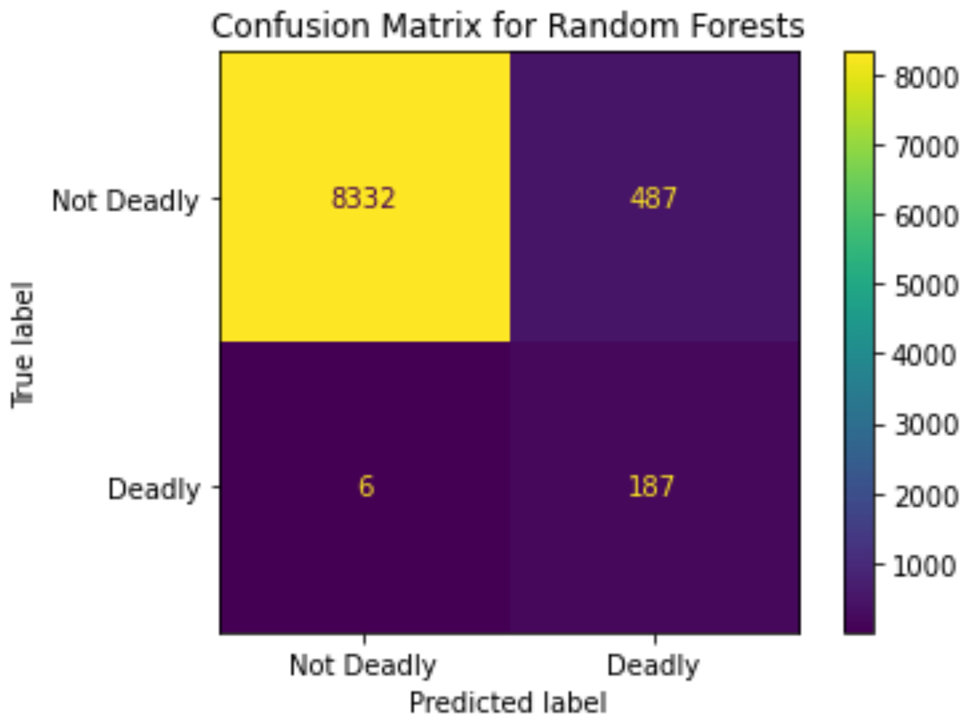


```
              precision    recall  f1-score   support

           0       1.00      0.94      0.97      8819
           1       0.28      0.97      0.43       193

    accuracy                           0.95      9012
   macro avg       0.64      0.96      0.70      9012
weighted avg       0.98      0.95      0.96      9012
```

Classification report created for random forests ensemble.

The overall performance of the model is very favourable, with a high recall percentage, indicating that positive classifications are identified approximately 97% of the time. The model's precision score is also favourable given its likelihood to minimize false positives. One can visualize the impact of the precision and recall metrics in a confusion matrix using pyplot.

```
#visualize test and pred results
disp = sklearn.metrics.ConfusionMatrixDisplay.from_predictions(
    y_test,
    y_pred,
    display_labels= ['Not Deadly', 'Deadly'])
disp.ax_.set_title('Confusion Matrix for Random Forests')
plt.show()
```



In the case of predicting marine occurrences that are likely to occur in deaths, having false positives may be indicative of a more overzealous model - something not to be seen as a negative, as it may prevent fatal marine accidents from occurring. Given the circumstances under study, false negatives are much more costly, potentially costing lives if disproportionate to true positives. Overall, the model's performance is admirable, and is able to identify deadly and not deadly scenarios effectively, while identifying false positives 5.4% of the time.

Section 6: Next Steps for Final Project

- Code complete for section 5.2, 5.3, and 5.4, but required to include model performance analysis for 5.2 - Logistic Regression: Model Building and Evaluation, 5.3 - Naive Bayes Classifier: Model Building and Evaluation, and 5.4 - Consolidated Evaluation of Models
- Calculate feature importance for logistic regression and naive bayes classifiers
- Tweak logistic regression and naive bayes classifiers for greater accuracy with reduction of dimensions identified as having low f-scores by the random forests ensemble

Github Repository

All relevant project files can be found at https://github.com/rfinatan/CIND-820-Big-Data-Analytics-Project.