CCT College Dublin

Assessment Cover Page

Module Title:	Strategy Thinking
Assessment Title:	Project Report
Lecturer Name:	James Garza
Student Full Name:	Giulio Calef, Kevin Byrne and Victor Ferreira Silva
Student Number:	sba22314, sba22264, 2021324
Assessment Due Date:	7th May 2023
Date of Submission:	7th May 2023

Declaration

By submitting this assessment, I confirm that I have read the CCT policy on Academic Misconduct and understand the implications of submitting work that is not my own or does not appropriately reference material taken from a third party or other source. I declare it to be my own work and that all material from third parties has been appropriately referenced. I further confirm that this work has not previously been submitted for assessment by myself or someone else in CCT College Dublin or any other higher education institution.

Detecting and Predicting Severe Slugging in Petrobras 3W Data Set

Strategic Thinking Capstone Project

Giulio Calef
Kevin Byrne
Victor Ferreira Silva

Strategic Thinking Capstone Project

Higher Diploma in Science in Artificial Intelligence Applications
CCT College Dublin
Ireland
May 2023

Contents

Detecting and Predicting Severe Slugging in Petrobras 3W Data Set

Giulio Calef, Kevin Byrne, Victor Ferreira Silva

May 4, 2023

1 Introduction

1.1 Business Understanding

Operational safety, productivity, quality are general key objectives in any industry, and in the oil and gas industry, environmental concerns not only are crucial importance, but poses as an immense daily challenge. Any events which can cause a production loss in an oil field are certainly very costly, in pure economic terms, but can also have dire costs in terms of human lives and of damage to our environment that can be hard to address or almost permanent in some cases. Therefore, any technique that can help early detection and/or prevention of technical accidents in the oil industry is therefore very welcome and "worth as gold". Offshore oil wells provide some of the most challenging operating conditions in the industry, with additional complexity due to the peculiarities of operating at sea and the limited amount of instrumentation that can be deployed to monitor and control the well operational status.

A simplified description of a typical offshore well can be seen below in Figure 1.

Basically, a key structure in the well is the so-called *Christmas Tree*, a structure lying on the seabed, at the well head, with pressure and temperature sensors (PDG – Permanent Downhole Gauge and TPT – Temperature and Pressure Transducer), and safety valves (DHSV – Downhole Safety Valve). In turn, a PCK (Production Choke valve) is installed on the drilling vessel/rig at the top.

Given this, Petrobras, the Brazilian oil company, has developed a data set (3W) that contains data for the most common monitored variables in offshore oil wells. This project aims to understand the relations between these variables and a specific oil well operational anomaly named *Severe Slugging*.

Severe Slugging is an critical flow assurance issue, commonly observed in offshore pipeline-riser systems, documented for the first time by Yocum (1973). Some of the consequences of this issue include flooding of downstream production facilities and an overall decrease in productivity.

1.2 Hypothesis

The data present in 3W Data Set allows classifier models to predict Severe Slugging with high accuracy.

1.3 General Goal

The business objective of our project is to apply machine learning techniques modeled in this project to the 3W data set to accurately predict *Severe Slugging* in an offshore well production line. Achieving a significant accuracy in these predictions should be possible by identifying the correlations between the variables presented in the records monitored in an offshore oil well operation.

1.4 Success Criteria

Success will be defined as the ability to accurately predict one or more of the conditions leading to an undesirable event. Given the potentially catastrophic impacts (ethical, environmental, economic) of accidents in the oil industry, the ability to predict a potential risk so that a quick reaction/fix can avoid unrecoverable conditions has a clear business value.

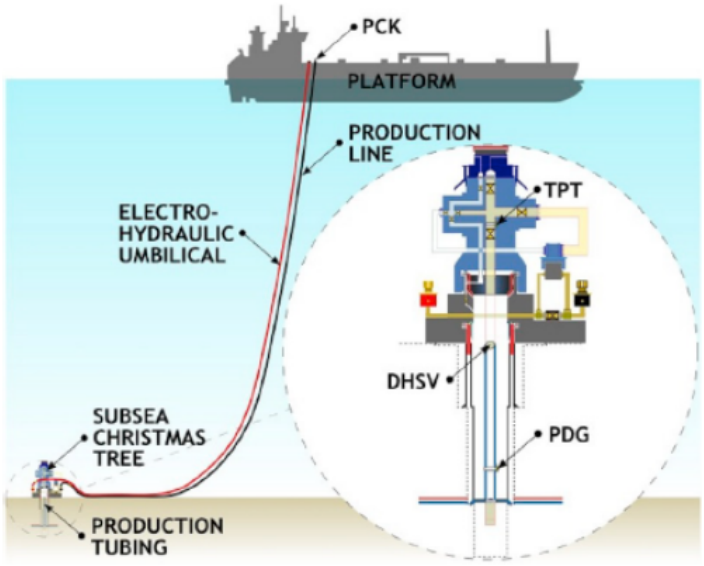

Figure 1: Schematic of a typical offshore well - source: Petrobras

1.5 Methodologies and Technologies

The methodology that guided the iterations this project was CRISP-DM. Given this, a number of libraries were used throughout this process to support most of its distinct stages, that is, Data Understanding, Data Preparation, Modeling, and Evaluation.

Once the methodology was defined, Petrobras' *3W Tool Kit* was studied and used to extract the data from the instances interesting to the project, that is, from the real instances that effectively presented the Severe Slugging event.

Then, *Pandas* and *NumPy* libraries were selected for data manipulation and analysis. Then, the base machine learning library for the majority of models in this project was *Scikit-learn*, as it offers various preprocessing, classification models and clustering algorithms. Another library imported in this project was *Keras*, which is is an open-source solution that provides an interface for artificial neural networks.

Besides that, *StandardScaler* from the *Scikit-learn* library was selected for scaling and normalisation of the data. Given the high data imbalance presented by the data set, a *RandomUnderSampler* was also from *Imbalanced-learn* library was imported. Regarding feature reduction, the decomposition algorithm *PCA*, also from *Scikit-learn*, was adopted here in this study.

Also, *Seaborn*, *Matplotlib* and *Plotly* libraries were widely used to analyse data throughout the project and provide data visualisation using at the Evaluation stage. Another tool from *Scikit-learn* selected as a baseline model was *DummyClassifier*.

Additionally, the following tools from module *model_selection* in *Scikit-learn* library were chose:

- *GridSearchCV*, for hyperparameter tuning
- *cross_val_score* for cross-validation,
- *train_test_split* for train/test splitting,
- *KFold* for cross-validation during Evaluation

Ultimately, *Scikit-learn* library also provided the following classifiers for this project:

- LinearSVC, from *svm* module,
- KNeighborsClassifier, from *neighbors* module,
- DecisionTreeClassifier, from *tree* module,

- RandomForestClassifier, from *ensemble* module

Lastly, the modules *make_pipeline* and *Pipeline* from *Scikit-learn* were used to chain all steps of the workflow together, and the library *Pickle* was selected to persist the resulting models in file.

1.6 Accomplishment

After extracting using 3W Tool Kit and some CRISP-DM iterations, this project presented 2 models with a high accuracy.

2 Data Understanding

Pre-processing a data set through data characterisation involves summarising the features and characteristics present in the data using statistical measures and visualisations techniques such as bar charts and scatter plots. After this stage, it should be possible to identify biases, patterns, trends, and any missing or irrelevant data in the data set that may need to be addressed.

This data set is composed by instances of eight types of undesirable events characterized by eight process variables from three different sources: real instances, simulated instances and hand-drawn instances. All real instances were taken from the plant information system that is used to monitor the industrial processes at an operational unit in Brazilian state of Espírito Santo. The simulated instances were all generated using OLGA, a dynamic multi-phase flow simulator that is widely used by oil companies worldwide (Andreolli 2016). Finally, the hand-drawn instances were generated by a specific tool developed by Petrobras researchers for this data set to incorporate undesirable events classified as rare.

Ultimately, only the data from the real instances was selected for this project, as simulated instances and hand-drawn instances did not present any record for two features relevant to Severe Slugging, namely Gas Lift Flow Rate and Pressure Variable Upstream Of the Gas Lift Choke.

2.1 Data Collection

The data used in this study was extracted after following the documentation from 3W took kit (Petrobras 2019b), which is a Python software package with resources to experiment machine learning-based approaches and algorithms for issues related to undesirable events. The specific data used in this study was from the *real instances*, and it was also available at Petrobras (2019a) at the time of publication of this study.

2.2 Data Characterisation

The data consists of over 50 million observations, with 13 columns of data for each observation. The first column, label, indicates the event type for each observation. The second column, well, contains the name of the well the observation was taken from. Hand-drawn and simulated instances have fixed names for in this column, while real instances have names masked with incremental id. The third column, id, is an identifier for the observation and it is incremental for hand-drawn and simulated instances, while each real instance has an id generated from its first timestamp. The columns representing the process variables are:

- P-PDG: pressure variable at the Permanent Downhole Gauge (PDG) - installed on Christmas Tree;
- P-TPT: pressure variable at the Temperature and Pressure Transducer (TPT) - installed on Christmas Tree;
- T-TPT: temperature variable at the Temperature and Pressure Transducer (TPT);
- P-MON-CKP: pressure variable upstream of the production choke (CKP) - located on platform;
- T-JUS-CKP: temperature variable downstream of the production choke (CKP);
- P-JUS-CKGL: pressure variable upstream of the gas lift choke (CKGL);

- T-JUS-CKGL: temperature variable upstream of the gas lift choke (CKGL);
- QGL: gas lift flow rate;

The pressure features are measured in Pascal (Pa), the volumetric flow rate features are measured in standard cubic meters per second (SCM/s), and the temperature features are measured in degrees Celsius (°C).

Other information are also loaded into each pandas Dataframe:

- label: instance label (event type) - target variable;
- well: well name. Hand-drawn and simulated instances have fixed names (respectively, drawn and simulated. Real instances have names masked with incremental id;
- id: instance identifier. Hand-drawn and simulated instances have incremental id. Each real instance has an id generated from its first timestamp;
- class: Although it can be used to identify periods of normal operation, fault transients, and faulty steady states, which can help with diagnosis and maintenance, it is a category which results from label, which is our target here

The labels are:

- 0 - Normal Operation = Normal
- 1 - Abrupt Increase of BSW = AbrIncrBSW
- 2 - Spurious Closure of DHSV = SpurClosDHSW
- 3 - Severe Slugging = SevSlug
- 4 - Flow Instability = FlowInst
- 5 - Rapid Productivity Loss = RProdLoss
- 6 - Quick Restriction in PCK = QuiRestrPCK
- 7 - Scaling in PCK = ScalingPCK
- 8 - Hydrate in Production Line = HydrProdLine

In order to maintain the realistic aspects of the data, the data set was built without pre-processing, including the presence of NaN values, frozen variables due to sensor or communication issues, instances with varying sizes, and outliers (Vargas et al. 2019).

A concise summary of this data set generated by *pandas.DataFrame.info* method can be seen on Table 1.

2.3 Exploratory Data Analysis

A bar chart was generated displaying the percentage of present values in each column of the data frame - see Figure 2. The data set contained missing values in several columns, thus some columns and row were deleted in order to obtain accurate and reliable results.

Three boxplots were plotted to show how the data was distributed before any data cleaning - see Figure 3. They were divided according the feature measurement unit: the pressure features were measured in Pascal (Pa), the temperature features are measured in degrees Celsius (°C) and one feature about volumetric flow rate which was measured in standard cubic meters per second (SCM/s).

3 Data Preparation

Data preparation included Data Cleaning, Feature Engineering, Train/Test Splitting and Handling Imbalanced Data, Data Scaling, and an analysis of the chosen approach regarding dimensionality reduction for some models.

Column	pandas.Dtype	Description
timestamp	datetime64[ns]	timestamp
label	int64	label
well	object	well
id	int64	id
P-PDG	float64	pressure variable at the PDG, in Pa
P-TPT	float64	pressure variable at the TPT, in Pa
T-TPT	float64	temperature variable at the TPT, in °C
P-MON-CKP	float64	pressure variable upstream of CKP, in Pa
T-JUS-CKP	float64	temperature variable downstream of CKP, in °C
P-JUS-CKGL	float64	pressure variable upstream of CKGL, in °C
T-JUS-CKGL	float64	temperature variable upstream of CKGL, in °C
QGL	float64	gas life flow rate, SCM/s
class	float64	operation state: normal, fault, faulty steady
source	object	type of instance: real, simulated or hand-drawn

Table 1: Summary of the data set compiled from real instances

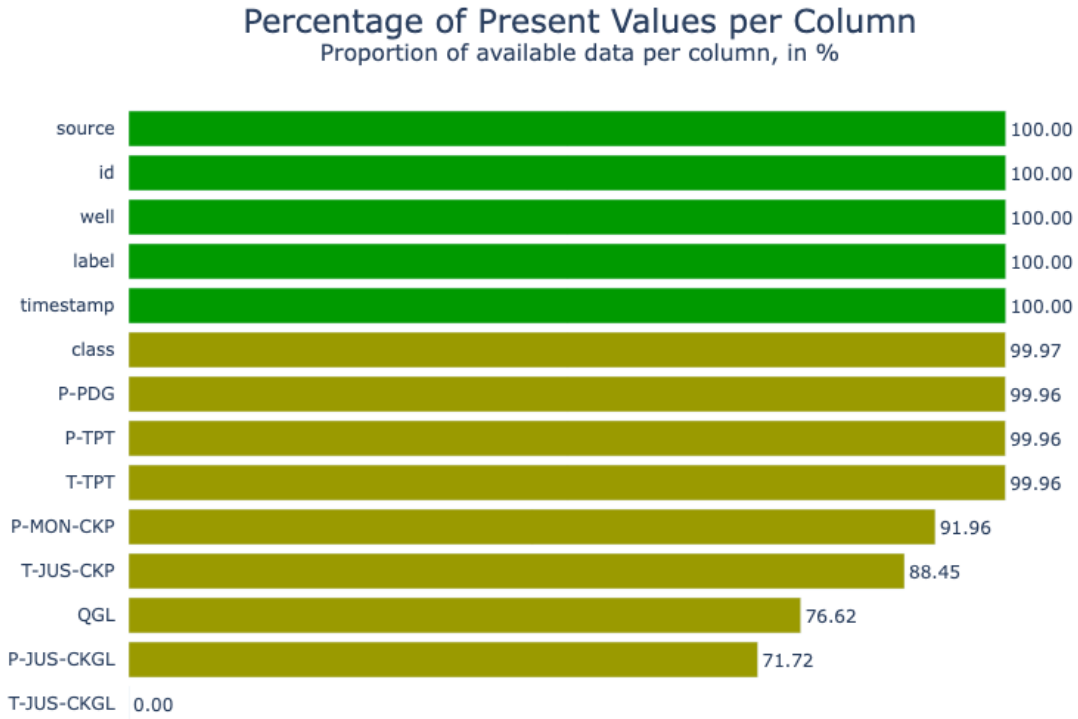

Figure 2: Proportion of available data per column, in %.

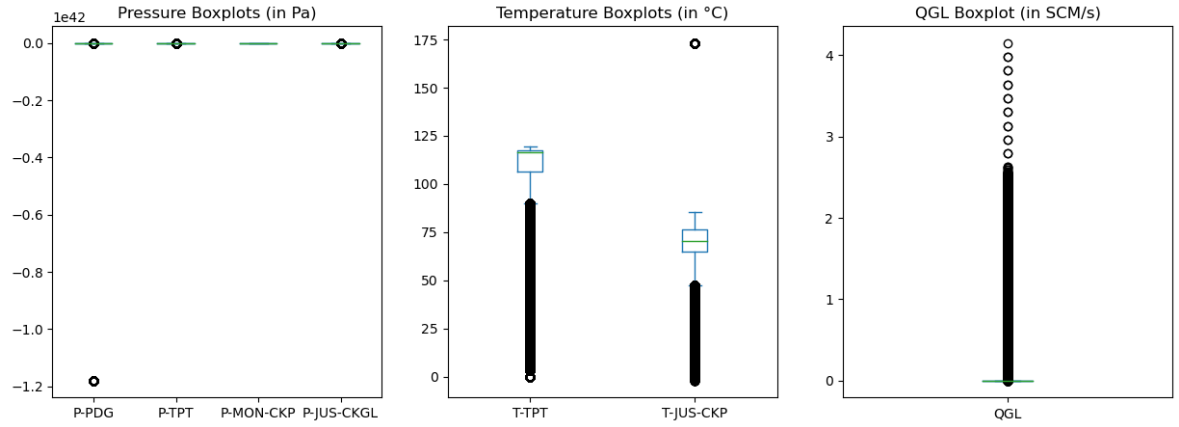

Figure 3: Box plots showing the distribution of pressure, temperature, and QGL (SCM/s) data for a set of oil wells.

3.1 Data Cleaning

The missing data from the following columns were removed: class, P-PDG, P-TPT, T-JUS-CKP, P-MON-CKP,T-TPT, P-MON-CKP, QGL and P-JUS-CKGL. After this, the columns class, T-JUS-CKGL (an empty column), id, source were dropped. Column class is a column which brings more details about label. Consider that columns timestamp, label were kept at this stage. Finally all duplicates were removed.

```
# dropping rows with missing or null class column
df_clean = df.dropna(subset=[
    'class','P-PDG','P-TPT','T-JUS-CKP','P-MON-CKP','T-TPT',
    'P-MON-CKP','QGL','P-JUS-CKGL'
])

# removing redundant columns
df_clean = df_clean.drop(['class','T-JUS-CKGL','id','source'], axis=1)

# checking duplicated rows after removing ids
df_clean = df_clean.drop_duplicates()

df_clean.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10003580 entries, 0 to 13952910
Data columns (total 10 columns):
 #   Column      Dtype
---  ------      -----
 0   timestamp   datetime64[ns]
 1   label       int64
 2   well        object
 3   P-PDG       float64
 4   P-TPT       float64
 5   T-TPT       float64
 6   P-MON-CKP   float64
 7   T-JUS-CKP   float64
 8   P-JUS-CKGL  float64
 9   QGL         float64
dtypes: datetime64[ns](1), float64(7), int64(1), object(1)
memory usage: 839.5+ MB
```

Also, as it can be seen on Figure 3, features P-PDG and P-TPT had the presence of extreme outliers. These outliers were also removed with the following code:

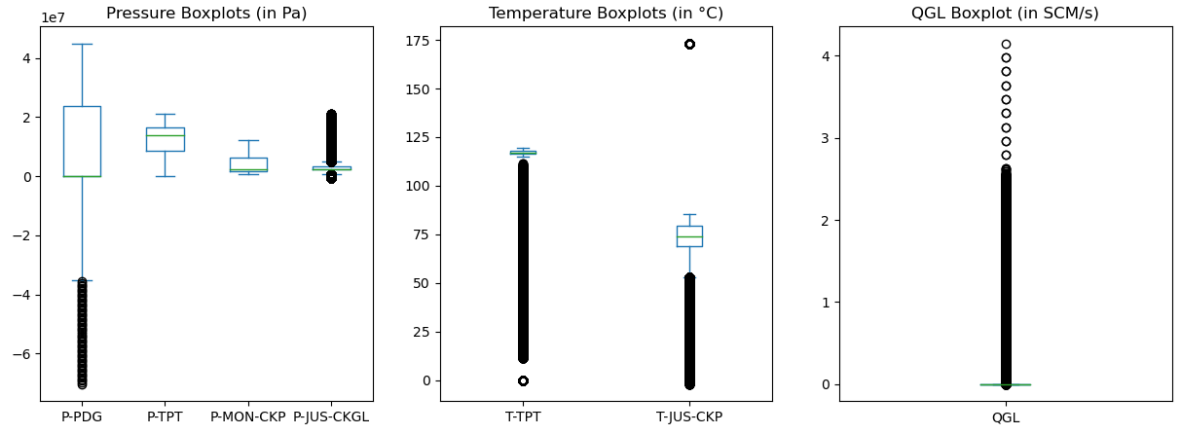

Figure 4: Box plots showing the distribution of pressure, temperature, and QGL (SCM/s) data without extreme outliers.

```
# removing extreme outliers from P-PDG
Q1 = df_clean['P-PDG'].quantile(0.25)
Q3 = df_clean['P-PDG'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - (3 * IQR)
df_no_outliers = df_clean[(df_clean['P-PDG'] >= lower_bound)]

# removing extreme outliers from P-TPT
Q1 = df_no_outliers['P-TPT'].quantile(0.25)
Q3 = df_no_outliers['P-TPT'].quantile(0.75)
IQR = Q3 - Q1
upper_bound = Q3 + (3 * IQR)
df_no_outliers = df_no_outliers[(df_no_outliers['P-TPT'] <= upper_bound)]

df_no_outliers.shape
```

```
(9780901, 10)
```

These rows with presence of extreme outliers represented 2.26% of the resulting rows so far. As a result the distribution of values in P-PDG and P-TPT were modified, as Figure 4 shows.

3.2 Feature Engineering

Given the label feature contains 8 possible numeric labels for each undesirable event and 1 label value 0 for normal observations, 8 new boolean columns were created for each one undesirable event, including for Severe Slugging, which is this project's target.

```
dt_feat = df_no_outliers

# Changing 'label' column to object dtype
dt_feat['label'] = dt_feat['label'].astype('object')

# Creating uint8 columns for each label
label_dummies = pd.get_dummies(dt_feat['label'], prefix='label')
dt_feat = pd.concat([dt_feat, label_dummies], axis=1)

# Renaming uint8 columns
column_names = {
    'label_0': 'Normal',
    'label_1': 'AbrIncrBSW',
    'label_2': 'SpurClosDHSW',
    'label_3': 'SevSlug', # target
    'label_4': 'FlowInst',
    'label_5': 'RProdLoss',
    'label_6': 'QuiRestrPCK',
    'label_7': 'ScalingPCK',
```

```
    'label_8': 'HydrProdLine'
}
dt_feat = dt_feat.rename(columns=column_names)

# Dropping the original 'label' column and Normal column,
# since all other events must be 0
dt_feat = dt_feat.drop(['label','Normal'], axis=1)
dt_feat.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 9780901 entries, 0 to 13952910
Data columns (total 16 columns):
 #   Column        Dtype
---  ------        -----
 0   timestamp     datetime64[ns]
 1   well          object
 2   P-PDG         float64
 3   P-TPT         float64
 4   T-TPT         float64
 5   P-MON-CKP     float64
 6   T-JUS-CKP     float64
 7   P-JUS-CKGL    float64
 8   QGL           float64
 9   AbrIncrBSW    uint8
 10  SpurClosDHSW  uint8
 11  SevSlug       uint8
 12  FlowInst      uint8
 13  RProdLoss     uint8
 14  QuiRestrPCK   uint8
 15  ScalingPCK    uint8
dtypes: datetime64[ns](1), float64(7), object(1), uint8(7)
memory usage: 811.5+ MB
```

Then all undesirable events columns were deleted but the column which denotes the observations presents Severe Slugging. The column *HydrProdLine* concerned to Hydrate in Production line, however this event was not found in the data set resulting from real instances.

```
dt_feat_target = dt_feat.drop([
#     , 'SevSlug', 'HydrProdLine',
    'AbrIncrBSW','SpurClosDHSW','FlowInst','RProdLoss','QuiRestrPCK','ScalingPCK'
], axis=1)

dt_feat_target.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 9780901 entries, 0 to 13952910
Data columns (total 10 columns):
 #   Column      Dtype
---  ------      -----
 0   timestamp   datetime64[ns]
 1   well        object
 2   P-PDG       float64
 3   P-TPT       float64
 4   T-TPT       float64
 5   P-MON-CKP   float64
 6   T-JUS-CKP   float64
 7   P-JUS-CKGL  float64
 8   QGL         float64
 9   SevSlug     uint8
dtypes: datetime64[ns](1), float64(7), object(1), uint8(1)
memory usage: 755.6+ MB
```

3.3 Train/Test Splitting

The following code defined how the data set was split in Train and Test data sets. Additionally, the columns *timestamp* and *well* were removed and at the end the distribution of the records according the presence or absence of Severe Slugging was computed.

```
# defining features (X) and label (y)
target = 'SevSlug'

X = dt_feat_target.drop([target,'timestamp','well'], axis=1)
y = dt_feat_target[target]

# splitting data into train and test sets
X_train_u, X_test, y_train_u, y_test = train_test_split(X, y, test_size=0.3,
    random_state=42)

class_names = {0:'Non Sev Slug', 1:'SEV SLUGGING'}
print(y_train_u.value_counts(normalize=True).rename(index=class_names))
```

```
Non Sev Slug    0.94194
SEV SLUGGING    0.05806
Name: SevSlug, dtype: float64
```

After the splitting process, the training data set had 6,846,630 rows and the test data set had 2,934,271 rows.

3.4 Handling Imbalanced Data

A *RandomUnderSampler* was chosen to balance training data. As a result 50% of observations presented Severe Slugging while the other 50% were normal or presented other undesirable event. Test data set was not balanced since this project aim to best represent your deployment scenarios in real life.

```
# balancing data
balancing = RandomUnderSampler(random_state=42)

X_train, y_train = balancing.fit_resample(X_train_u, y_train_u)

class_names = {0:'Non Sev Slug', 1:'SEV SLUGGING'}
print(y_train.value_counts(normalize=True).rename(index=class_names))
print([X_train.shape, y_train.shape])
```

```
Non Sev Slug    0.5
SEV SLUGGING    0.5
Name: SevSlug, dtype: float64
[(795026, 7), (795026,)]
```

Handling data imbalance is also important because it affects correlations - see as Figure 5 shows.

3.5 Data Scaling

Although there are features presenting non-normal distributions, *StandardScaler* was chosen as data scaler. It was chose because there are some features with strong correlation with Severe Slugging and lognormal distributions such as *QGL* and *P-JUS-CKGL* and as it is a method sensitive to the presence of outliers. The results of this transformation can be seen on Figure 6.

3.6 Dimensionality Reduction

The unsupervised learning technique Principal Component Analysis (PCA) was chosen not only to prepare the data for some of the models studied here, but also to evidence any possible linear separability in this model. In Figure 7 the results of this dimensionality reduction can be seen in two ways, with 2 and 3 components, although this process was unnecessary for the most successful models, that is, the non-linear classifiers.

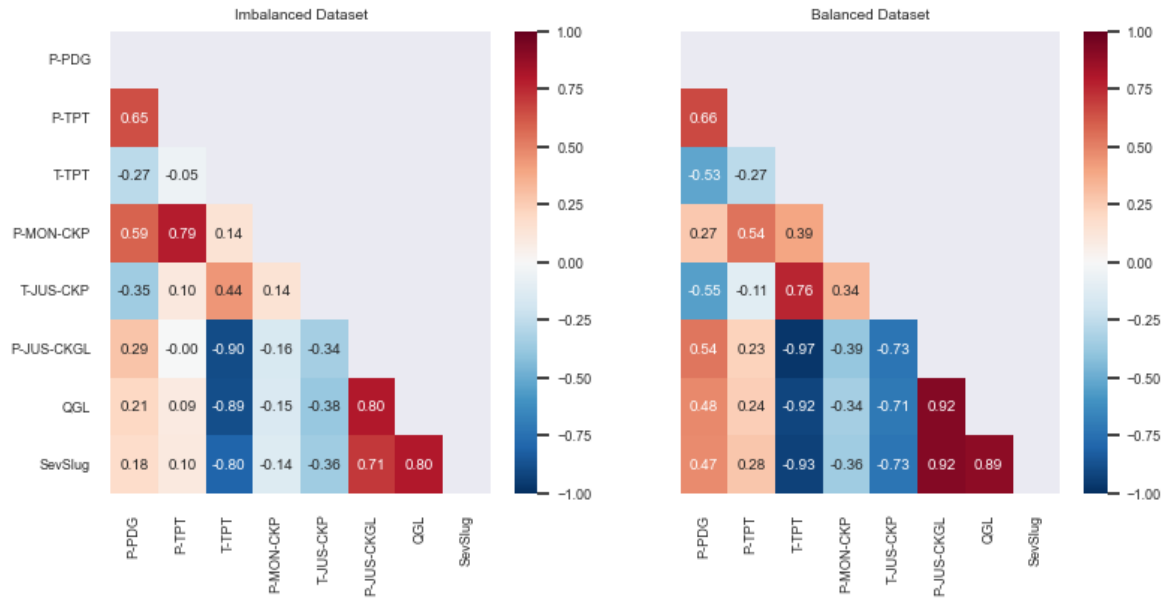

Figure 5: Correlations between variables before and after data balancing

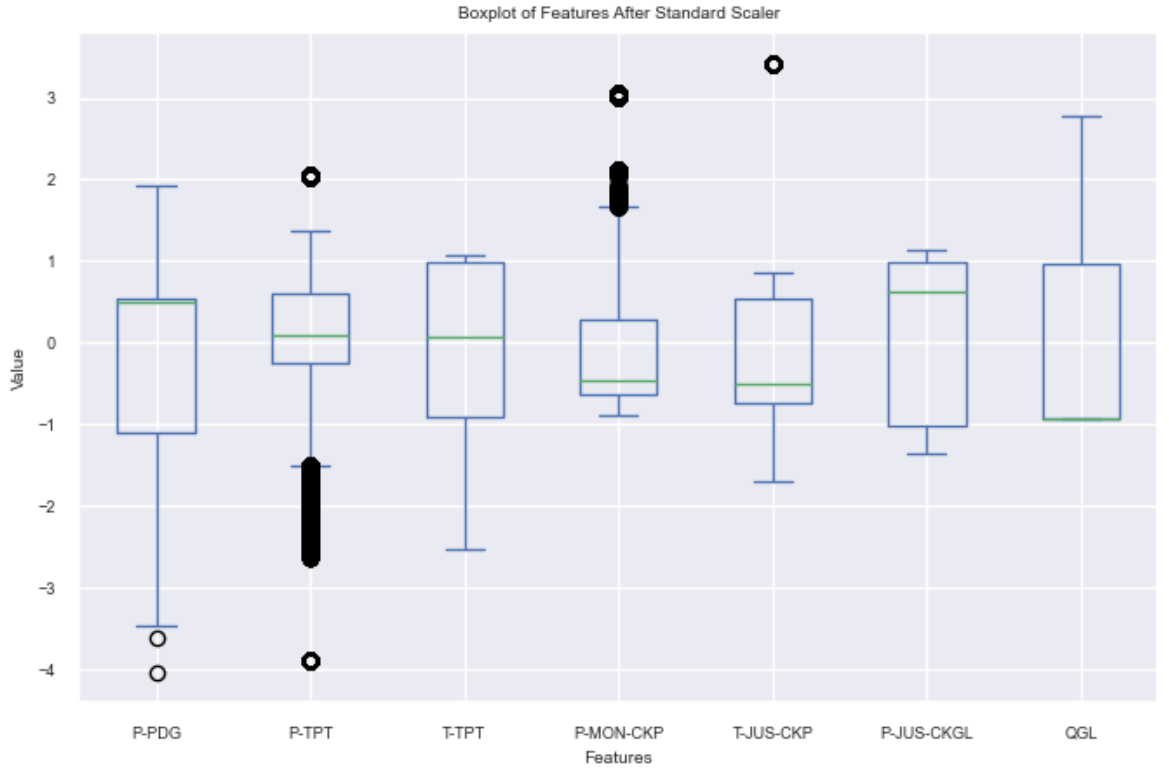

Figure 6: Box plot showing the distribution of the features in the training set after applying the StandardScaler transformation

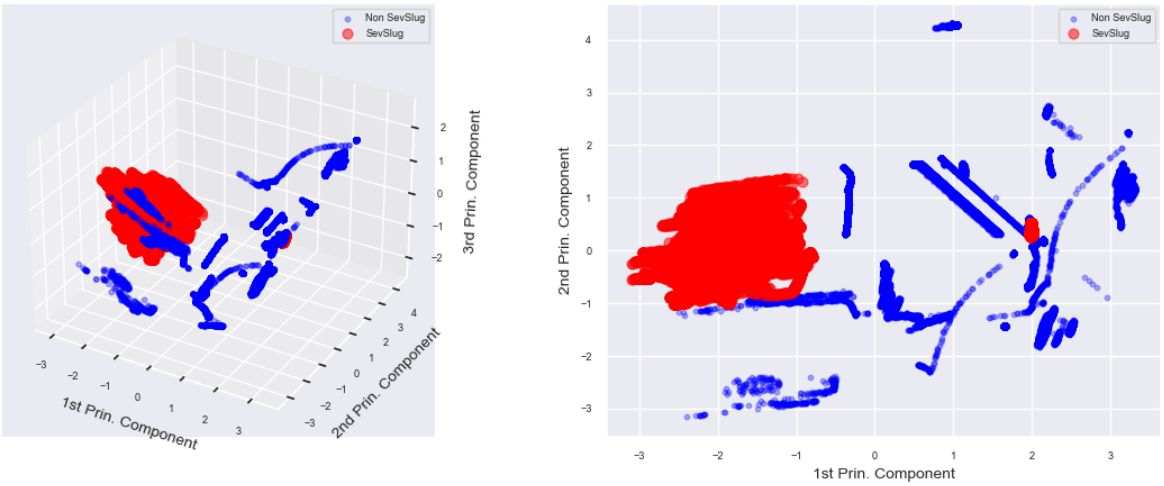

Figure 7: Visualisation of PCA applied to the data set showing a scatter plot for two (2D) and three (3D) principal components.

4 Modeling

For this project, five models were chosen: LinearSVC, k-Nearest Neighbors Classifier, Artificial Neural Network, Decision Tree Classifier and Random Forest Classifier.

4.1 Baseline: DummyClassifier

Before proceeding with other models, a DummyClassifier was adopted to find a baseline for validation accuracy using the test data set. Given the distribution of the target variable in test data set, the baseline for any model was set in 94.17%.

```
dummy_pipeline = make_pipeline(StandardScaler(), DummyClassifier())
dummy_pipeline.fit(X_train, y_train)

# confirming score for Dummy classifier results from a balanced dataset
score = dummy_pipeline.score(X_train, y_train)

# predicting
y_predicted = dummy_pipeline.predict(X_test)
baseline = metrics.accuracy_score(y_test, y_predicted)

print("Score: ", score)
print("Accuracy: ",baseline)
```

```
Score:  0.5
Accuracy:  0.9417780429960286
```

4.2 LinearSVC

Although the data is not linearly separable and it's not possible to find a condition of 100% correctly classified by a hyperplane, a linear support vector classifier (LinearSVC) was implemented as part of this benchmark.

4.2.1 Hyperparameter optimisation

For finding the best parameters for this model, the hyperparameters to tune PCA and the model were specified. Then a pipeline was created with 3 steps:

1. *scaler* which uses StandardScaler method to scale the data
2. *dimred* which applies PCA dimensionality reduction

3. *linearsvc* which applies the LinearSVC model.

Finally, a grid search was performed using the above-mentioned pipeline and the hyperparameter grid to find the combination with the best accuracy metric with the default cross-validation, that is, a 5-fold cross-validation.

```
from sklearn.svm import LinearSVC

param_grid = {
    'dimred__n_components': [3, 4],
    'linearsvc__C': [1e-2, 1e-1, 1, 10, 100],
    'linearsvc__penalty':['l1', 'l2'],
    'linearsvc__dual': [False, True],
    'linearsvc__class_weight': ['balanced', None]
}

linear_svc_pipeline = Pipeline([
    ('scaler', scaler_pipeline),
    ('dimred', PCA()),
    ('linearsvc', LinearSVC())
])

grid_search_lsvc = GridSearchCV(
    linear_svc_pipeline,
    param_grid=param_grid,
    n_jobs=-1,
    scoring='accuracy',
    verbose=1
)

grid_search_lsvc.fit(X_train, y_train)
```

4.2.2 Model training

Then the model was trained using a similar pipeline, but this time with the optimal combination of parameters.

```
linear_svc_pipeline = Pipeline([
    ('scaler', scaler_pipeline),
    ('dimred', PCA(
        n_components=grid_search_lsvc.best_params_['dimred__n_components']
    )),
    ('linearsvc', LinearSVC(
        dual=grid_search_lsvc.best_params_['linearsvc__dual'],
        C=grid_search_lsvc.best_params_['linearsvc__C'],
        penalty=grid_search_lsvc.best_params_['linearsvc__penalty'],
        class_weight=grid_search_lsvc.best_params_['linearsvc__class_weight']
    ))
])

linear_svc_pipeline.fit(X_train, y_train)
```

4.3 k-Nearest Neighbors Classifier

A k-Nearest Neighbors Classifier (KNeighborsClassifier) was implemented as part of this benchmark.

4.3.1 Hyperparameter optimisation

For finding the best parameters for this model, the hyperparameters to tune PCA and the model were specified. Then a pipeline was created with 3 steps:

1. *scaler* which uses StandardScaler method to scale the data
2. *dimred* which applies PCA dimensionality reduction
3. *kneighborsclassifier* which applies the KNeighborsClassifier model.

Finally, a grid search was performed using the above-mentioned pipeline and the hyperparameter grid to find the combination with the best accuracy metric with the default cross-validation, that is, a 5-fold cross-validation.

```
from sklearn.neighbors import KNeighborsClassifier

knn_pipeline = Pipeline([
    ('scaler', scaler_pipeline),
    ('dimred', PCA()),
    ('kneighborsclassifier', KNeighborsClassifier())
])

param_grid = {
    'dimred__n_components': [3, 4],
    'kneighborsclassifier__n_neighbors': range(3, 102, 3),
}

grid_search_knn = GridSearchCV(
    knn_pipeline,
    param_grid=param_grid,
    n_jobs=-1,
    scoring='accuracy',
    verbose=1
)

grid_search_knn.fit(X_train, y_train)
```

4.3.2 Model training

Then the model was trained using a similar pipeline, but this time with the optimal combination of parameters.

```
knn_pipeline = Pipeline([
    ('scaler', scaler_pipeline),
    ('dimred', PCA(
        n_components=grid_search_knn.best_params_['dimred__n_components']
    )),
    ('kneighborsclassifier', KNeighborsClassifier(
        n_neighbors=grid_search_knn.best_params_['kneighborsclassifier__n_neighbors']
    ))
])


knn_pipeline.fit(X_train, y_train)
```

4.4 Artificial Neural Network

A Neural Network was also implemented as part of this benchmark. For this model, before optimising hyperparameters, it was necessary to define a function that created the proper neural network using Keras library.

4.4.1 Model definition

The amount of units and hidden layers in this model was specified during hyperparameter optimisation, as this function was defined with two arguments, namely *num_units* and *num_hidden_layers*, respectively defined with default values 10 and 1.

```
import tensorflow as tf
from keras.wrappers.scikit_learn import KerasClassifier
from keras.models import Sequential
from keras.layers import Dense

def build_clf(num_units=10, num_hidden_layers=1):
    # initialising Sequential model and adding layers to it
    ann_clf = tf.keras.models.Sequential()

    # adding hidden layers
```

```
    for i in range(num_hidden_layers):
        ann_clf.add(tf.keras.layers.Dense(units=num_units, activation='relu'))

    # adding output layer
    ann_clf.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))

    # compiling model with chosen optimizer, loss function, and evaluation metrics
    ann_clf.compile(
        optimizer='adam',
        loss='binary_crossentropy',
        metrics=['accuracy']
    )

    return ann_clf
```

Also, the training data was split again to monitor the loss value and to find the best number of epochs on a further step, and this way this validation data was not considered by the model.

```
X_train_, X_val, y_train_, y_val = train_test_split(
    X_train, y_train,
    test_size=0.3, random_state=42
)

[X_train_.shape, X_val.shape, y_train_.shape, y_val.shape]
```

```
[(556518, 7), (238508, 7), (556518,), (238508,)]
```

4.4.2 Hyperparameter optimisation

For finding the best parameters for this model, the hyperparameters to tune the model were specified. Then a pipeline was created with 2 steps:

1. *scaler* which uses StandardScaler method to scale the data
2. *annmodel* which applies the Sequential model.

Finally, a grid search was performed using the above-mentioned pipeline and the hyperparameter grid to find the combination with the best accuracy metric with the default cross-validation, that is, a 5-fold cross-validation.

```
param_grid = {
    'ann_model__num_units': [3, 4, 5, 6, 7],
    'ann_model__num_hidden_layers': [2, 3, 4]
}

# creating an instance of the KerasClassifier using the defined function
ann_model = KerasClassifier(build_fn=build_clf, verbose=0)

ann_pipeline = Pipeline([
    ('scaler',scaler_pipeline),
    ('ann_model',ann_model)
])

grid_search_ann = GridSearchCV(
    ann_pipeline,
    param_grid=param_grid,
    n_jobs=-1,
    scoring='accuracy',
    verbose=0
)

grid_search_ann.fit(X_train_, y_train_)
```

In order to monitor the history of the loss evolution according the number of epochs, the *callback* parameter in the model *fit* method should be included, then the history could be assign to a variable. However, as the history object was not accessible by the pipeline, the model was scaled and trained separately to generate this history.

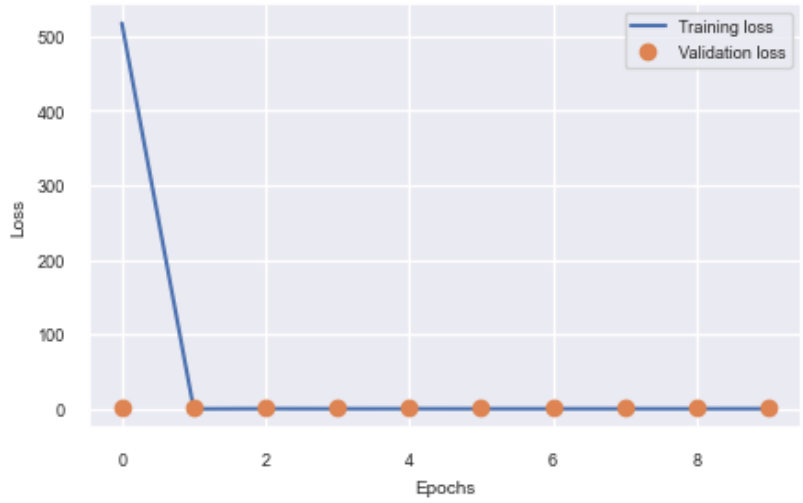

Figure 8: The training and validation loss for a neural network model trained over 15 epochs with early stopping

```
from keras import callbacks

earlystopping = callbacks.EarlyStopping(
    monitor="val_loss", mode="min", patience=5, restore_best_weights=True,
    verbose=-1
)

ann_model_ = KerasClassifier(
    build_fn=build_clf,
    num_hidden_layers=grid_search_ann.best_params_['ann_model__num_hidden_layers'],
    num_units=grid_search_ann.best_params_['ann_model__num_units'],
    verbose=1
)

scaler_pipeline.fit(X_val)
X_val = scaler_pipeline.transform(X_val)

history = ann_model_.fit(X_train_, y_train_,
    epochs=15,
    verbose=0,
    validation_data=(X_val, y_val),
    callbacks=[earlystopping]
)
```

The loss according the number of epochs on training and validation data can be seen on Figure 8 and a chart was created by the following code.

```
# plotting the training and validation loss
fig, (ax1) = plt.subplots(1, figsize=(5, 3))
ax1.plot(history.history['loss'])
ax1.plot(history.history['val_loss'],'o')
ax1.set_xlabel('Epochs')
ax1.set_ylabel('Loss')
ax1.legend(['Training loss', 'Validation loss'])

plt.show()
```

4.4.3 Model training

Finally, the model was trained using a similar pipeline, with the optimal combination of parameters and the proper amount of epochs. The final configuration of this model is visible on Figure 9.

```
ann_model = KerasClassifier(
    build_fn=build_clf,
```

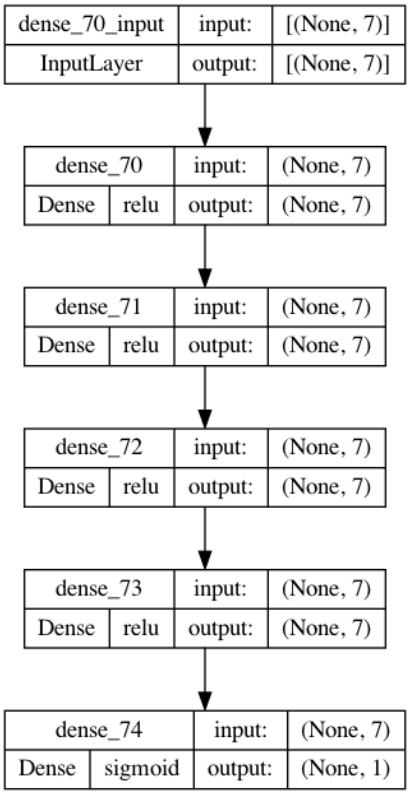

Figure 9: The architecture of a neural network model built using Keras

```
    num_units=grid_search_ann.best_params_['ann_model__num_units'],
    num_hidden_layers=grid_search_ann.best_params_['ann_model__num_hidden_layers'],
    verbose=0
)

ann_pipeline = Pipeline([
    ('scaler', scaler_pipeline),
    ('ann_model', ann_model)
])

ann_pipeline.fit(
    X_train,
    y_train,
    ann_model__epochs=5,
    ann_model__verbose=0
)
```

4.5 Decision Tree Classifier

As the data is not linearly separable and it's not possible to find a condition of 100% correctly classified by a hyperplane, a non-linear classifier is recommended, thus a Decision Tree Classifier was implemented as part of this benchmark.

4.5.1 Hyperparameter optimisation

For finding the best parameters for this model, the hyperparameters to tune the model were specified. Then a pipeline was created with 1 step:

1. *decisiontreeclassifier* which applies the Decision Tree model.

Finally, a grid search was performed using the above-mentioned pipeline and the hyperparameter grid to find the combination with the best accuracy metric with the default cross-validation, that is,

a 5-fold cross-validation.

```
tree_pipeline = Pipeline([
    ('decisiontreeclassifier', DecisionTreeClassifier())
])

param_grid = {
    'decisiontreeclassifier__min_samples_split': [2, 5, 10],
    'decisiontreeclassifier__min_samples_leaf': [1, 2, 4],
    'decisiontreeclassifier__max_features': ['sqrt', 'log2']
}

grid_search_tree = GridSearchCV(
    tree_pipeline,
    param_grid=param_grid,
    n_jobs=-1,
    scoring='accuracy',
    verbose=1
)

grid_search_tree.fit(X_train, y_train)
```

4.5.2 Model training

Then the model was trained using a similar pipeline, but this time with the optimal combination of parameters.

```
tree_pipeline = Pipeline([
    ('decisiontreeclassifier', DecisionTreeClassifier(
        min_samples_split=grid_search_tree.best_params_['decisiontreeclassifier__min_samples_split'],
        min_samples_leaf=grid_search_tree.best_params_['decisiontreeclassifier__min_samples_leaf'],
        max_features=grid_search_tree.best_params_['decisiontreeclassifier__max_features']
    ))
])

tree_pipeline.fit(X_train, y_train)
```

4.6 Random Forest Classifier

As the data is not linearly separable and it's not possible to find a condition of 100% correctly classified by a hyperplane, a non-linear classifier is recommended, thus a Random Forest Classifier was implemented as part of this benchmark.

4.6.1 Hyperparameter optimisation

For finding the best parameters for this model, the hyperparameters to tune the model were specified. Then a pipeline was created with 1 step:

1. *randomforestclassifier* which applies the Random Forest model.

Then, a grid search was performed using the above-mentioned pipeline and the hyperparameter grid to find the combination with the best accuracy metric with the default cross-validation, that is, a 5-fold cross-validation.

```
from sklearn.ensemble import RandomForestClassifier

rf_pipeline = Pipeline([
    ('randomforestclassifier', RandomForestClassifier())
])

param_grid = {
    'randomforestclassifier__class_weight': ['balanced', 'balanced_subsample']
}

```

```
grid_search_rf = GridSearchCV(
    rf_pipeline,
    param_grid=param_grid,
    n_jobs=-1,
    scoring='accuracy',
    verbose=1
)

grid_search_rf.fit(X_train, y_train)
```

4.6.2 Model training

Then the model was trained using a similar pipeline, but this time with the optimal combination of parameters.

```
rf_pipeline = Pipeline([
    ('randomforestclassifier', RandomForestClassifier(
        class_weight=grid_search_rf.best_params_['randomforestclassifier__class_weight
    '])
    ))
])

rf_pipeline.fit(X_train, y_train)
```

5 Evaluation

5.1 Classification Report

The classification reports for all 5 models were generated by the code below.

```
# LinearSVC
cr_linearsvc = metrics.classification_report(y_test, y_predicted_lin_clf, digits=4)

# kNN
cr_knn = metrics.classification_report(y_test, y_predicted_knn, digits=4)

# Neural Networks
y_predicted_ann = y_predicted_ann.flatten()
y_predicted_ann = np.where(y_predicted_ann.round(2) > 0.5, 1, 0)
cr_ann = metrics.classification_report(y_test, y_predicted_ann, digits=4)

# Decision Tree
cr_tree = metrics.classification_report(y_test, y_predicted_tree, digits=4)

# Random Forest
cr_rf = metrics.classification_report(y_test, y_predicted_rf, digits=4)
```

5.1.1 LinearSVC

```
# printing classification report for LinearSVC
print(cr_linearsvc)
```

```
              precision    recall  f1-score   support

           0     0.9980    0.9808    0.9893   2763432
           1     0.7568    0.9687    0.8498    170839

    accuracy                         0.9801   2934271
   macro avg     0.8774    0.9747    0.9195   2934271
weighted avg     0.9840    0.9801    0.9812   2934271
```

5.1.2 k-Neighbors Classifier

```
# printing classification report for kNN classifier
print(cr_knn)
```

```
              precision    recall  f1-score   support

           0     0.9999    0.9949    0.9974   2763432
           1     0.9233    0.9979    0.9591    170839

    accuracy                         0.9950   2934271
   macro avg     0.9616    0.9964    0.9782   2934271
weighted avg     0.9954    0.9950    0.9951   2934271
```

5.1.3 Neural Networks

```
# printing classification report for ANN
print(cr_ann)
```

```
              precision    recall  f1-score   support

           0     0.9980    0.9831    0.9905   2763432
           1     0.7795    0.9688    0.8639    170839

    accuracy                         0.9822   2934271
   macro avg     0.8888    0.9759    0.9272   2934271
weighted avg     0.9853    0.9822    0.9831   2934271
```

5.1.4 Decision Tree

```
# printing classification report for Decision Tree
print(cr_tree)
```

```
              precision    recall  f1-score   support

           0     1.0000    0.9998    0.9999   2763432
           1     0.9960    0.9998    0.9979    170839

    accuracy                         0.9998   2934271
   macro avg     0.9980    0.9998    0.9989   2934271
weighted avg     0.9998    0.9998    0.9998   2934271
```

5.1.5 Random Forest

```
# printing classification report for Random Forest
print(cr_rf)
```

```
              precision    recall  f1-score   support

           0     1.0000    0.9999    1.0000   2763432
           1     0.9987    1.0000    0.9993    170839

    accuracy                         0.9999   2934271
   macro avg     0.9993    1.0000    0.9996   2934271
weighted avg     0.9999    0.9999    0.9999   2934271
```

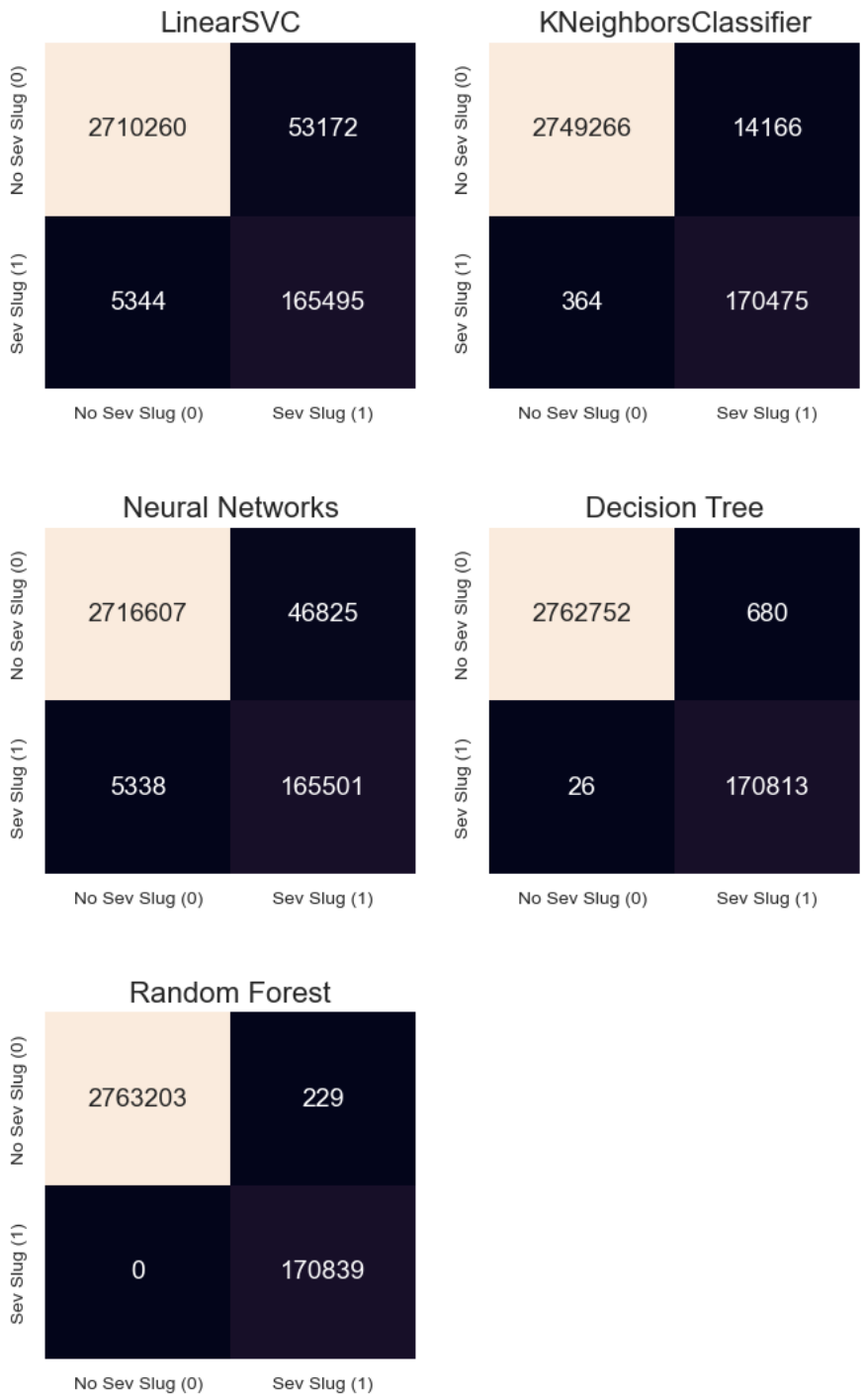

Figure 10: Confusion matrices showing the performance of five machine learning models in predicting severe slugging occurrences

5.2 Confusion Matrices

In this section the confusion matrices for every model for test labels were computed and plotted side by side on Figure 10.

5.3 10-Fold Cross Validation

Finally, a K-Folds cross-validator was selected to split training data set into 10 consecutive folds and compute the mean accuracy and the respective standard deviation for each model. The models were still put on the same pipelines used on modeling section.

```
models = [
    ('Random Forest', rf_pipeline),
    ('Decision Tree', tree_pipeline),
    ('KNeighborsClassifier', knn_pipeline),
    ('Neural Networks', ann_pipeline),
    ('Linear SVC', linear_svc_pipeline)
]
```

```
for name, model in models:
    kfold = KFold(n_splits=10, shuffle=True, random_state=42)

    cv_results = cross_val_score(model, X_train, y_train, cv=kfold, scoring='accuracy'
    )
    cross_validation_dict[name] = cv_results

    msg = "%s: %f (%f)" % (name, np.nanmean(cv_results), np.nanstd(cv_results))
    print(msg)
```

```
Random Forest: 0.999940 (0.000018)
Decision Tree: 0.999826 (0.000069)
KNeighborsClassifier: 0.998610 (0.000138)
Neural Networks: 0.978187 (0.001739)
Linear SVC: 0.974343 (0.000403)
```

The following code was implemented to plot how the mean score after cross validation was distributed in each model. This chart can be visualised on Figure 11.

```
# comparing algorithms
fig = plt.figure()
fig.suptitle('Distribution of Mean Score After 10-fold Cross Validation By Algorithm')
ax = fig.add_subplot(111)
plt.boxplot(cross_validation_dict.values())
ax.set_xticklabels(cross_validation_dict.keys())
fig.set_size_inches(8,4)
plt.show()
```

.

6 Conclusion

In this study, Random Forest and Decision Tree Classifiers had the best performance overall with 99.994% and 99.982% of accuracy respectively. These non-linear classifiers could find better results because the 3W data set does not present a clear linear separability between records related to Severe Slugging and other normal records or other undesirable events.

References

Andreolli, Ivanilto (2016). "Introdução à elevação e escoamento monofásico e multifásico de petróleo". In: *Rio de Janeiro: Interciência*.

Petrobras, Petróleo Brasileiro S.A (2019a). *3W Dataset - Real Instances*. Accessed on: 2023-05-04. Data retrieved from 3W tool kit, only real instances, size: 1.5 GB, accessible only for CCT. URL: https://drive.google.com/file/d/1loOkCdH-3hu5-llI-_ZImVbEVUmoaZI_/.

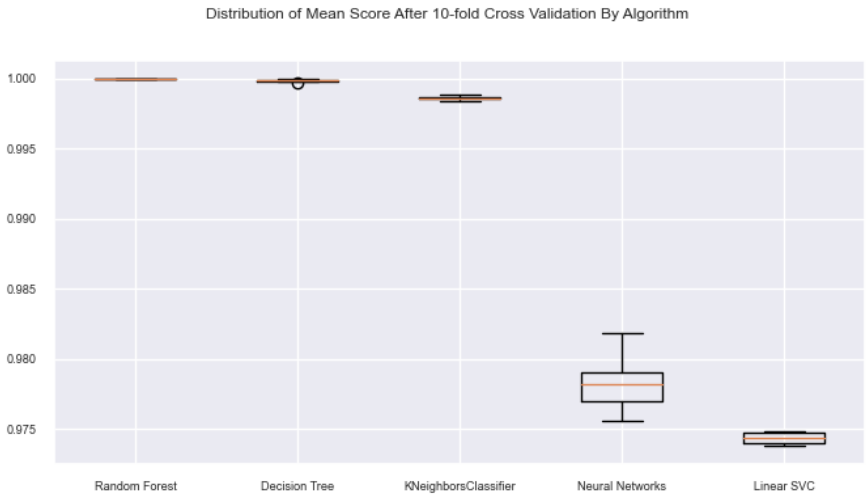

Figure 11: Box plots showing the distribution of Mean Score After 10-fold Cross Validation By Algorithm

Petrobras, Petróleo Brasileiro S.A (July 2019b). *Petrobras/3W: The first repository published by Petrobras on Github*. Accessed: 2023-05-04. URL: https://github.com/petrobras/3W#3w-toolkit.

Vargas, Ricardo Emanuel Vaz et al. (July 2019). *A realistic and public dataset with rare undesirable real events in oil wells*. URL: https://www.sciencedirect.com/science/article/pii/S0920410519306357.

Yocum, B.T. (1973). "Offshore riser slug flow avoidance: Mathematical models for design and optimization". In: *All Days*. DOI: 10.2118/4312-ms.