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

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

1 Introduction **2**
1.1 Hypothesis . . . 2
1.2 General Goal . . . 3
1.3 Success Criteria . . . 3
1.4 Methodologies and Technologies . . . 3
1.5 Accomplishment . . . 4

2 Data Understanding **4**
2.1 Data Collection . . . 4
2.2 Data Characterisation . . . 5
2.3 Exploratory Data Analysis . . . 6

3 Data Preparation **6**
3.1 Data Cleaning . . . 7
3.2 Feature Engineering . . . 8
3.3 Train/Test Splitting . . . 10
3.4 Handling Imbalanced Data . . . 10
3.5 Data Scaling . . . 10
3.6 Dimensionality Reduction . . . 10

4 Modeling **12**
4.1 Baseline: DummyClassifier . . . 12
4.2 LinearSVC . . . 12
4.2.1 Hyperparameter optimisation . . . 13
4.2.2 Model training . . . 13
4.3 k-Nearest Neighbors Classifier . . . 14
4.3.1 Hyperparameter optimisation . . . 14
4.3.2 Model training . . . 14
4.4 Artificial Neural Network . . . 15
4.4.1 Data Preparation for Keras . . . 15
4.4.2 Model definition . . . 16
4.4.3 Handling The Model Bias . . . 18
4.4.4 Model Training . . . 19
4.5 Decision Tree Classifier . . . 22
4.5.1 Hyperparameter optimisation . . . 22
4.5.2 Model training . . . 22
4.6 Random Forest Classifier . . . 23
4.6.1 Hyperparameter optimisation . . . 23
4.6.2 Model training . . . 23

5 Evaluation **24**
5.1 Classification Report . . . 24
5.1.1 LinearSVC . . . 24
5.1.2 k-Neighbors Classifier . . . 24
5.1.3 Neural Networks . . . 24
5.1.4 Decision Tree . . . 24
5.1.5 Random Forest . . . 25
5.2 Confusion Matrices . . . 25
5.3 10-Fold Cross Validation . . . 25

6 Conclusion **30**

Detecting and Predicting Severe Slugging in Petrobras 3W Data Set

Giulio Calef, Kevin Byrne, Victor Ferreira Silva

May 6, 2023

1 Introduction

Operational safety, productivity, quality are general key objectives in any industry, and in the oil and gas industry, environmental concerns are not only crucial, but they also pose as an immense daily challenge.

Generally, the oil industry has been increasingly adopting automated controls and monitoring processes (Venkatasubramanian et al. 2003) to comply with the increasingly higher standards in their operations. Such standards not only require more productive operations, but safer processes and more energy-efficient methods to achieve greater quality (Jämsä-Jounela 2007).

Thus, any event 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.

One of the main challenges in oil industry is predicting undesirable events such as *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. According Vargas et al. (2019) depending on the frequency it occurs and intensity, this event may even damage the equipment in the well, although specific operational actions can be taken to mitigate this issue since it is detected.

According Vargas et al. (ibid.), a simplified description of a typical offshore well can be seen in Figure 1 and its structure is basically composed by:

- The Christmas Tree, a structure lying on the seabed, at the well head, with pressure and temperature sensors and safety valves
- An Electro-Hydraylic Umbilical, which is how The Christmas Tree is remotely controlled.
- The Permanent Downhole Gauge (PDG), installed at the Christmas Tree;
- The Temperature and Pressure Transducer (TPT), also a part of the Christmas Tree;
- The Production Choke (PCK), installed on the drilling vessel/rig at the top;
- The Downhole Safety Valve (DHSV), a safety valve installed in the production tubing of wells

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 oil well operational anomalies, such as Severe Slugging.

1.1 Hypothesis

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

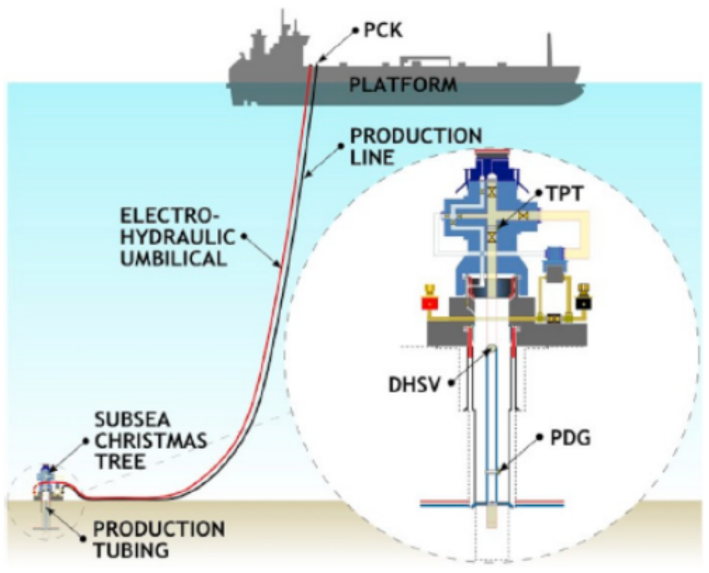

Figure 1: Schematic of a typical offshore well

1.2 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.

1.3 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.

1.4 Methodologies and Technologies

The methodology that guided this project was *CRISP-DM*, which is most widely adopted methodology for data mining, analytics, and data science projects (*IBM SPSS Modeler CRISP-DM Guide* 2017). Given this, a number of software libraries and modules were used throughout this process to support some 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.

1.5 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.

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

2.2 Data Characterisation

The selected data set consists of 13,952,911 observations, with 14 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:

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

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.

The labels for the possibel undesirable events found in the data set 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

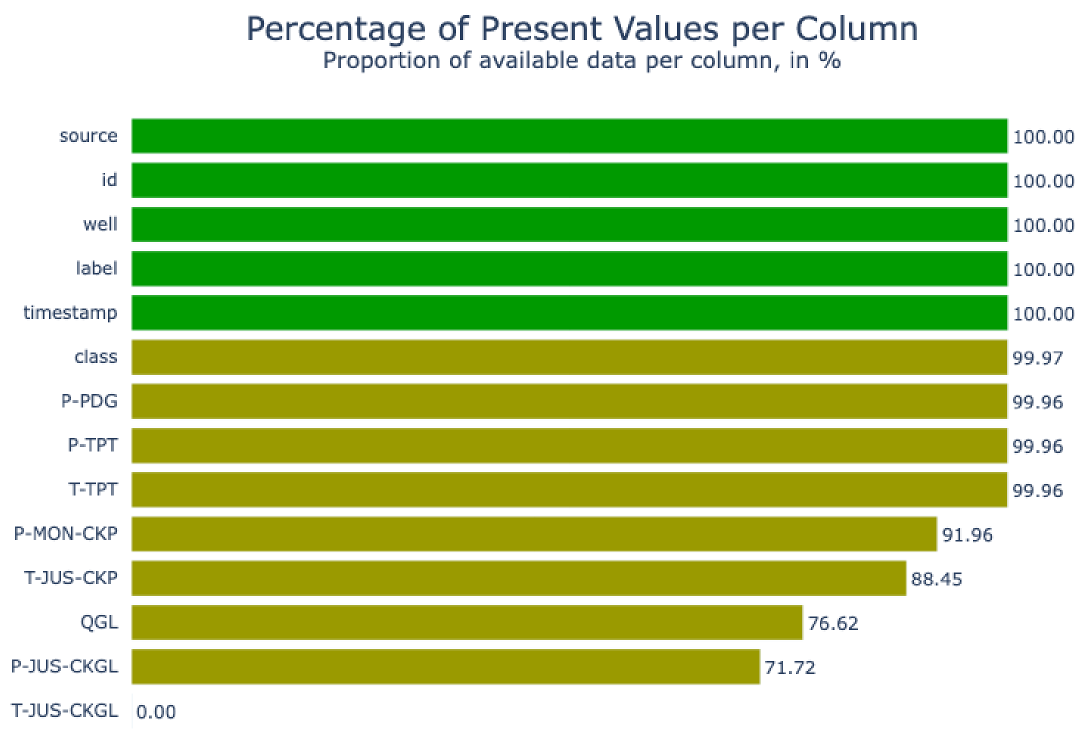

Figure 2: Proportion of available data per column, in %.

- 5 - Rapid Productivity Loss = RProdLoss
- 6 - Quick Restriction in PCK = QuiRestrPCK
- 7 - Scaling in PCK = ScalingPCK
- 8 - Hydrate in Production Line = HydrProdLine

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.

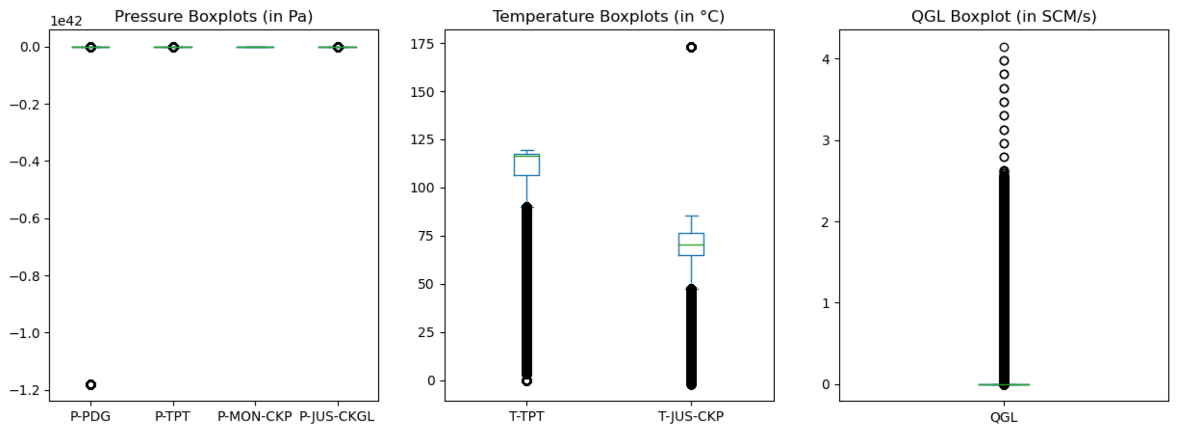

Figure 3: Box plots showing the distribution of pressure, temperature, and QGL (SCM/s) data for a set of oil wells.

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.

```python
# 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:

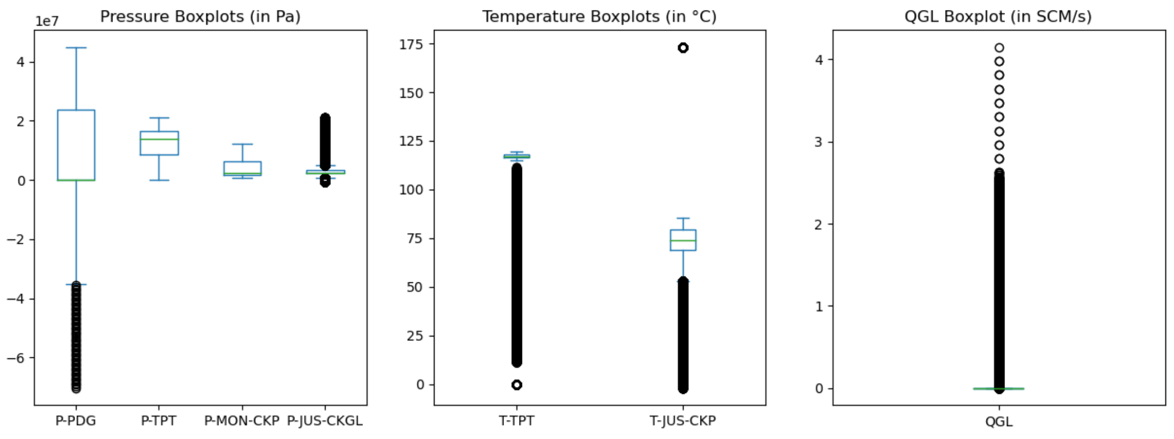

Figure 4: Box plots showing the distribution of pressure, temperature, and QGL (SCM/s) data without extreme outliers.

```
# 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.

```python
# 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.

```python
# 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.

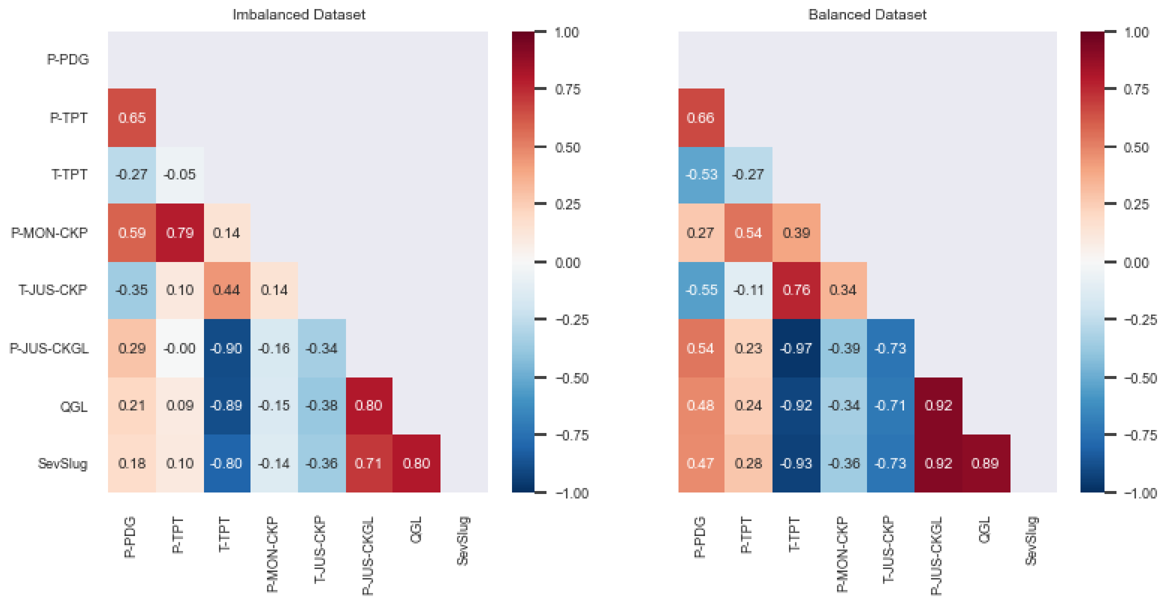

Figure 5: Correlations between variables before and after data balancing

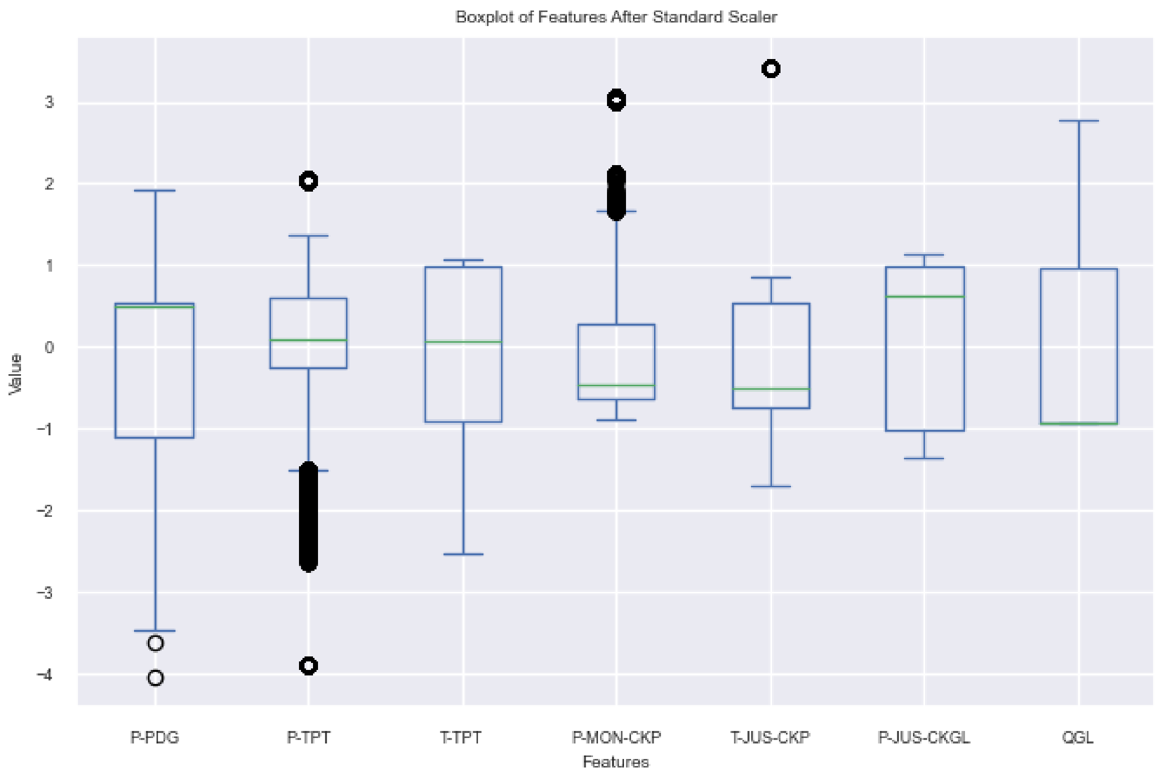

Figure 6: Box plot showing the distribution of the features in the training set after applying the StandardScaler transformation

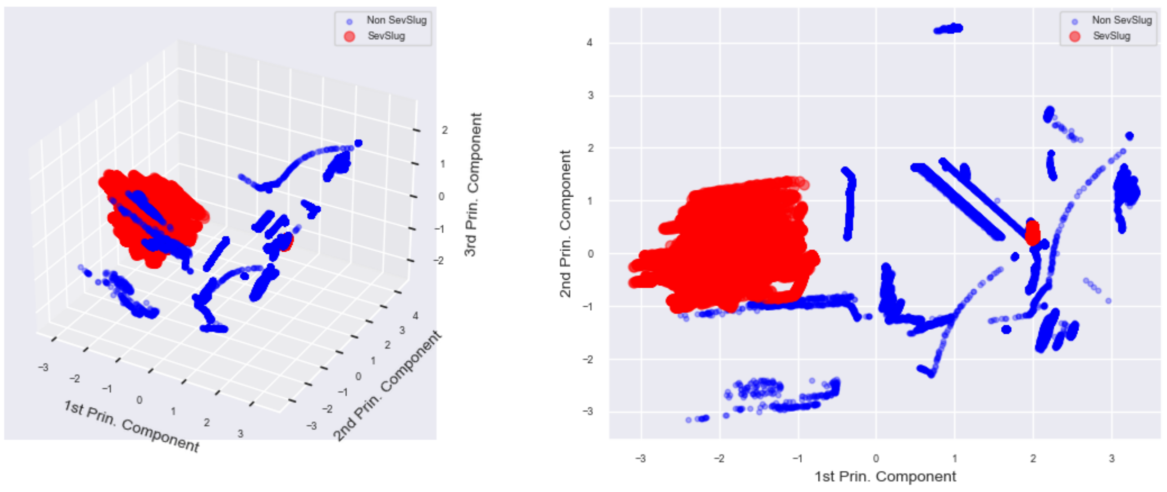

Figure 7: Visualisation of PCA applied to the data set showing a scatter plot for two (2D) and three (3D) principal components.

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%.

```python
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. The code below retrieved a previously trained LinearSVC model which was persisted as a Pickle file, then it predicted the class labels for the features in *X_test*, and finally generated the respective Classification Report in comparison with the labels in *y_test*.

```python
with open(os.path.join('pickle', 'linear_svc_pipeline.pkl'), 'rb') as f:
linear_svc_pipeline = pickle.load(f)

y_predicted_lin_clf = linear_svc_pipeline.predict(X_test)

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

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 f1 metric with the default cross-validation, that is, a 5-fold cross-validation.

```python
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='f1',
    verbose=1
)

grid_search_lsvc.fit(X_train, y_train)
```

After this, the pipeline was defined with the best parameters found:

1. StandardScaler()
2. PCA(n_components=3)
3. LinearSVC(C=0.01, class_weight='balanced', dual=False, penalty='l1')

4.2.2 Model training

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

```python
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. The code below retrieved a previously trained KNeighborsClassifier model which was persisted as a Pickle file, then it predicted the class labels for the features in *X_test*, and finally generated the respective Classification Report in comparison with the labels in *y_test*.

```
with open(os.path.join('pickle', 'knn_pipeline.pkl'), 'rb') as f:
    knn_pipeline = pickle.load(f)

y_predicted_knn = knn_pipeline.predict(X_test)

cr_knn = metrics.classification_report(y_test, y_predicted_knn, digits=5)
```

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='f1',
    verbose=1
)

grid_search_knn.fit(X_train, y_train)
```

Then, the pipeline was redefined with the best parameters found considering the proposed scenario:

1. StandardScaler()
2. PCA(n_components=4)
3. KNeighborsClassifier(n_neighbors=3)

4.3.2 Model training

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

```python
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

In this subsection, *Keras* and *Tensorflow* libraries were used to define a baseline model using Artificial Neural Network (ANN). Similar to previous models, the code below could retrieve a previously trained ANN model which was persisted as a Pickle file, then it predicted the class labels for the features in *X_test* after normalisation and adjusting the data type using *Numpy* function. After this, the corresponding Classification Report in comparison with the labels in *y_test* was generated.

```python
import tensorflow as tf
from tensorflow import keras
from keras import backend as K

BATCH_SIZE = 2048

with open(os.path.join('pickle', 'ann_model.pkl'), 'rb') as f:
    ann_model_ = pickle.load(f)

scaler = StandardScaler()

X_test_scaled = scaler.fit_transform(X_test)

y_predicted_ann = ann_model_.predict(np.array(X_test_scaled), batch_size=BATCH_SIZE)
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=5)
```

```
1433/1433 [==============================] - 1s 777us/step
```

Also the parameter *batch_size* was set as 2048, a relatively high value for this parameter, but capable to increase the chance of having some positive samples in each batch.

4.4.1 Data Preparation for Keras

For this model, a distinct approach to prepare the data set was adopted. Firstly, the original imbalanced clean data set was copied and split with the same parameter *random_state=42* used to generate the training data set used to fit and test the other models in this project. Secondly, the resulting train data set from this process was split again into train set and validation set.

Then, the validation set was only used during the model fitting to evaluate the loss, precision, recall and accuracy - however the model was not fit with this data. As a consequence, the test set was completely isolated during the training stage and was only used to evaluate how well the model generalizes to new data.

This approach is highly recommended to avoid over-fitting, which is a constant concern due to a possible lack of training data in imbalanced data sets (Martín Abadi et al. 2015) like the one used in this project, where the number of examples not presenting Severe Slugging class greatly outnumbered the examples presenting it.

Given this, the processes adopted for the data set in this section for data cleaning and scaling were the same processes adopted for the other models in this project, however the data is not balanced at this points. These adjustments were necessary to fulfill the requirements of a similar model described in TensorFlow Tutorial provided by Martín Abadi et al. (ibid.). The code that had performed these tasks can be see below.

```python
ann_df = df_feat.copy() # copying the imbalanced but clean data set

non_sev_slug, sev_slug = np.bincount(ann_df['SevSlug'])

total = non_sev_slug + sev_slug
print('Examples:\n    Total: {}\n    SevSlug: {} ({:.2f}% of total)\n'.format(
    total, sev_slug, 100 * sev_slug / total))
```

```
Examples:
    Total: 9780901
    SevSlug: 568352 (5.81% of total)
```

```python
# Dropping redundant columns for this project
ann_df.pop('timestamp')
ann_df.pop('well')
ann_df.columns
```

```
Index(['P-PDG', 'P-TPT', 'T-TPT', 'P-MON-CKP', 'T-JUS-CKP', 'P-JUS-CKGL',
       'QGL', 'SevSlug'],
      dtype='object')
```

```python
# Splitting data
ann_train_df, ann_test_df = train_test_split(ann_df, test_size=0.3, random_state=42)
ann_train_df, ann_val_df = train_test_split(ann_train_df, test_size=0.3)

# Form np arrays of labels and features
train_labels = np.array(ann_train_df.pop('SevSlug'))
bool_train_labels = train_labels != 0
val_labels = np.array(ann_val_df.pop('SevSlug'))
test_labels = np.array(ann_test_df.pop('SevSlug'))

train_features = np.array(ann_train_df)
val_features = np.array(ann_val_df)
test_features = np.array(ann_test_df)
```

```python
# Normalising data with a previously instantiated StandardScaler
train_features = scaler.fit_transform(train_features)
val_features = scaler.transform(val_features)
test_features = scaler.transform(test_features)

print('ANN Training labels shape:', train_labels.shape)
print('ANN Validation labels shape:', val_labels.shape)
print('Test labels shape:', test_labels.shape)

print('ANN Training features shape:', train_features.shape)
print('ANN Validation features shape:', val_features.shape)
print('Test features shape:', test_features.shape)
```

```
ANN Training labels shape: (4792641,)
ANN Validation labels shape: (2053989,)
Test labels shape: (2934271,)
ANN Training features shape: (4792641, 7)
ANN Validation features shape: (2053989, 7)
Test features shape: (2934271, 7)
```

4.4.2 Model definition

Instead of using GridSearchCV to tune hyper-parameters like in other models in this project, an initial model was defined and then after analysing its results the initial bias was fixed. Then the model was trained again and had its resulting metrics evaluated considering the validation data set. In the following code the implementation of this initial model is described after listing some metrics available in Keras library that were used to evaluate the model.

The architecture of this model is visible on Figure 8 and it has one input layer representing all 7 features from the data set, one densely connected hidden layer with 16 units, a dropout layer to mitigate over-fitting and an one-unit output layer that returns the probability of Severe Slugging.

```
ANN_METRICS = [
    keras.metrics.TruePositives(name='tp'),
    keras.metrics.FalsePositives(name='fp'),
    keras.metrics.TrueNegatives(name='tn'),
    keras.metrics.FalseNegatives(name='fn'),
    keras.metrics.BinaryAccuracy(name='accuracy'),
    keras.metrics.Precision(name='precision'),
    keras.metrics.Recall(name='recall'),
    keras.metrics.AUC(name='auc'),
    keras.metrics.AUC(name='prc', curve='PR'), # precision-recall curve
]

def make_model(metrics=ANN_METRICS, output_bias=None):
    if output_bias is not None:
        output_bias = tf.keras.initializers.Constant(output_bias)
    model = keras.Sequential([
        keras.layers.Dense(16, activation='relu',input_shape=(train_features.shape
    [-1],)),
        keras.layers.Dropout(0.5),
        keras.layers.Dense(1, activation='sigmoid',bias_initializer=output_bias),
    ])

    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=1e-3),
        loss=keras.losses.BinaryCrossentropy(),
        metrics=metrics)

    return model
```

The number of epochs used as a reference for this model in this project was 100, however it was not reached, because a resource called *EarlyStopping* was implemented to interrupt the training if a monitored metric has stopped improving. In this resource, the parameter *patience* can be set to make the training more tolerant to possible oscillations in the monitored metric throughout the epochs and the monitored metric here is the *validation precision-recall curve*, which evidences the trade-off between precision and recall for different threshold.

```
EPOCHS = 100

early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_prc',
    verbose=1,
    patience=10,
    mode='max',
    restore_best_weights=True)

ann_model = make_model()
ann_model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_10 (Dense)            (None, 16)                128

 dropout_5 (Dropout)         (None, 16)                0

 dense_11 (Dense)            (None, 1)                 17

=================================================================
Total params: 145 (580.00 Byte)
Trainable params: 145 (580.00 Byte)
Non-trainable params: 0 (0.00 Byte)
```

Finally, the newly-created model could be briefly tested:

```
ann_model.predict(train_features[:10])
```

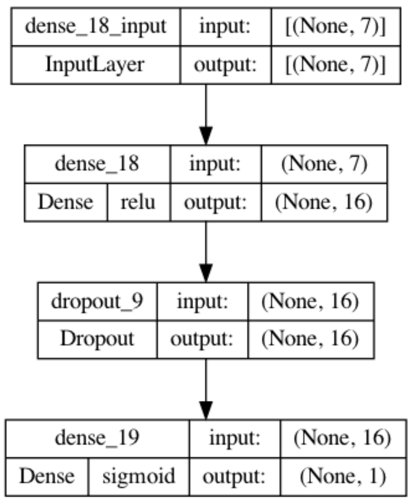

Figure 8: The architecture of a neural network model built using Keras

```
1/1 [==============================] - 0s 99ms/step
array([[0.21003543],
       [0.61643285],
       [0.6023897 ],
       [0.62401986],
       [0.62409544],
       [0.21703976],
       [0.2112424 ],
       [0.62370116],
       [0.6251195 ],
       [0.6153809 ]], dtype=float32)
```

4.4.3 Handling The Model Bias

Since the data set is imbalanced, there was an inherent bias in the first results. Given this default bias, the loss can be computed as in the following code:

```
results = ann_model.evaluate(train_features, train_labels, batch_size=BATCH_SIZE,
    verbose=0)
print("Loss: {:0.4f}".format(results[0]))
```

```
Loss: 0.9903
```

The correct bias can be computed as in the following code:

```
initial_bias = np.log([sev_slug / non_sev_slug])
initial_bias
```

```
array([-2.78558091])
```

Once the initial bias was computed, the model presented more reasonable initial guesses.

```
ann_model = make_model(output_bias=initial_bias)
ann_model.predict(train_features[:10])
```

```
1/1 [==============================] - 0s 55ms/step
array([[0.03573601],
       [0.03519705],
       [0.03306021],
       [0.04497113],
       [0.045381  ],
       [0.03489733],
       [0.03468524],
       [0.04532901],
       [0.04532651],
       [0.03519929]], dtype=float32)
```

As a result, the loss was significantly reduced in comparison to the loss found with the naive initialisation, that is, it was reduced from 0.9903 to 0.1167 - see the code below. Additionally, the model did not have to spend the first epochs registering that Severe Slugging only occurred in the minority of the records.

```python
results = ann_model.evaluate(train_features, train_labels, batch_size=BATCH_SIZE,
    verbose=0)
print("Loss: {:0.4f}".format(results[0]))
```

```
Loss: 0.1167
```

The bias adjustment could be confirmed after training the model for 20 epochs with the default initialization and then training it again with this careful initialisation and then the comparison between the losses could be visualised in Figure 9 :

```python
initial_weights = os.path.join(tempfile.mkdtemp(), 'initial_weights')
ann_model.save_weights(initial_weights)

# Confirming that the bias fix helps
ann_model = make_model()
ann_model.load_weights(initial_weights)
ann_model.layers[-1].bias.assign([0.0])

zero_bias_history = ann_model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=20,
    validation_data=(val_features, val_labels),
    verbose=0)
```

```python
ann_model = make_model()
ann_model.load_weights(initial_weights) # adjusted bias

careful_bias_history = ann_model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=20,
    validation_data=(val_features, val_labels),
    verbose=0)
```

Then a line chart (see Figure 9) comparing both scenarios was plotted with the following function:

```python
mpl.rcParams['figure.figsize'] = (12, 10)
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']

def plot_loss(history, label, n):
    # Using a log scale on y-axis to show the wide range of values.
    plt.semilogy(history.epoch, history.history['loss'], color=colors[n], label='Train
     ' + label)
    plt.semilogy(history.epoch, history.history['val_loss'], color=colors[n], label='
    Val ' + label, linestyle="--")

    plt.xlabel('Epoch')
    plt.ylabel('Loss')
```

4.4.4 Model Training

After fixing the default bias, the model training stage could be fully started with 100 epochs while the precision-recall curve (PRC) of validation data set was being monitored.

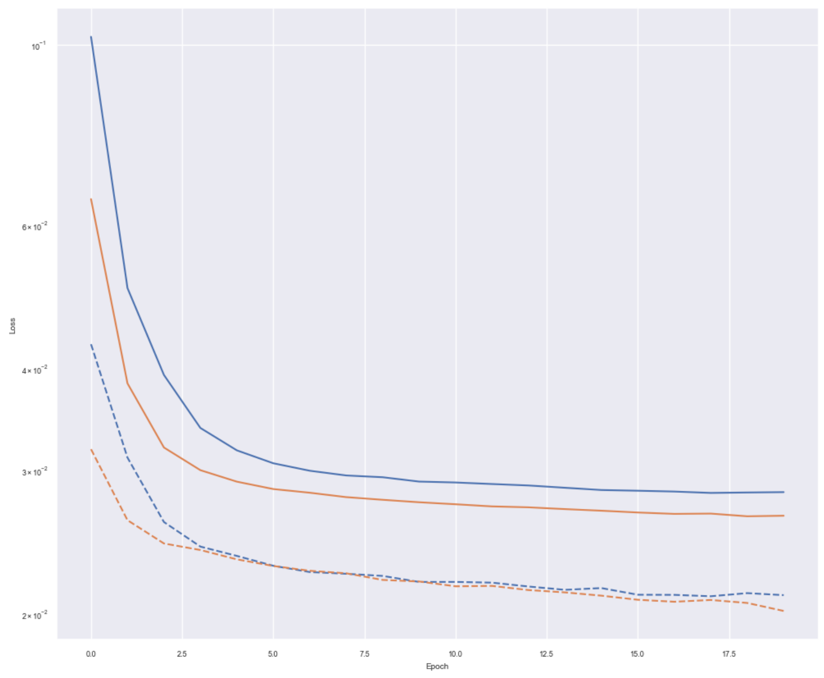

Figure 9: Comparison between validation loss before and after bias adjustment

```python
ann_model = make_model()
ann_model.load_weights(initial_weights)

baseline_history = ann_model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    callbacks=[early_stopping],
    validation_data=(val_features, val_labels))
```

Also, the training history was recorded to verify over-fitting and to collect data for visualise some relevant metrics in this project, as they can be seen on Figure 10. The code used to plot the metrics is below:

```python
def plot_metrics(history):
    metrics = ['loss', 'prc', 'precision', 'recall']

    for n, metric in enumerate(metrics):
        name = metric.replace("_"," ").capitalize()
        plt.subplot(2,2,n+1)
        plt.plot(history.epoch, history.history[metric], color=colors[0], label='Train
    ')
        plt.plot(history.epoch, history.history['val_' + metric],
                 color=colors[0], linestyle="--", label='Val')
        plt.xlabel('Epoch')
        plt.ylabel(name)

        if metric == 'loss':
            plt.ylim([0, plt.ylim()[1]])
        else:
            plt.ylim([0,1])

        plt.legend()

plot_metrics(baseline_history)
```

Ultimately, the training process was interrupted by the resource *EarlyStopping* after 58 epochs after not registering a better PRC for 10 epochs. More details about the metrics were presented at Evaluation section of this project.

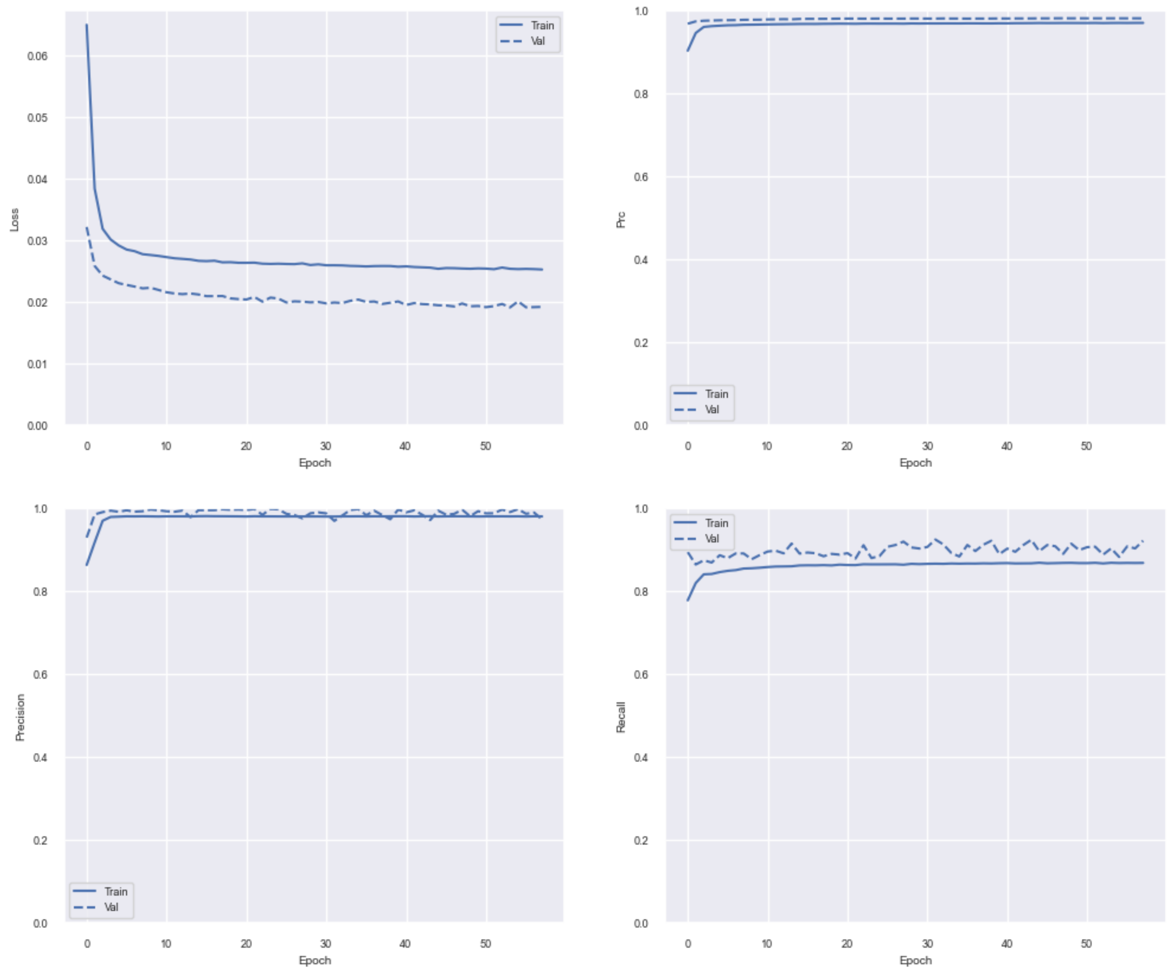

Figure 10: Metrics History After 58 Epochs

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. The code below retrieved a previously trained Decision Tree model which was persisted as a Pickle file, then it predicted the class labels for the features in *X_test*, and finally generated the respective Classification Report in comparison with the labels in *y_test*.

```python
with open(os.path.join('pickle', 'tree_pipeline.pkl'), 'rb') as f:
    tree_pipeline = pickle.load(f)

y_predicted_tree = tree_pipeline.predict(X_test)

cr_tree = metrics.classification_report(y_test, y_predicted_tree, digits=5)
```

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.

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.

```python
from sklearn.tree import DecisionTreeClassifier

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='f1',
    verbose=1
)

grid_search_tree.fit(X_train, y_train)
```

After this, the one-step pipeline was redefined with the best parameters found considering the proposed scenario for consistency with other models:

1. DecisionTreeClassifier(max_features='sqrt')

4.5.2 Model training

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

```python
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. The following code loaded a previously trained Random Forest Classifier model which was saved as a Pickle file, then it predicted the class labels for the features in *X_test*, and finally generated the respective Classification Report in comparison with the labels in *y_test*.

```
with open(os.path.join('pickle', 'rf_pipeline.pkl'), 'rb') as f:
rf_pipeline = pickle.load(f)

y_predicted_rf = rf_pipeline.predict(X_test)

cr_rf = metrics.classification_report(y_test, y_predicted_rf, digits=5)
```

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='f1',
    verbose=1
)

grid_search_rf.fit(X_train, y_train)
```

Then, the one-step pipeline was redefined with the best parameter found considering the proposed scenario:

1. RandomForestClassifier(class_weight='balanced')

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

As it could be seen in Modeling section, the classification reports for all models were generated immediately after they were instantiated and they were all detailed in this section.

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    0.99987   0.99487   0.99736   2763432
           1    0.92328   0.99787   0.95913    170839

    accuracy                        0.99505   2934271
   macro avg    0.96157   0.99637   0.97825   2934271
weighted avg    0.99541   0.99505   0.99514   2934271
```

5.1.3 Neural Networks

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

```
              precision    recall  f1-score   support

           0    0.99321   0.99987   0.99653   2763432
           1    0.99760   0.88938   0.94039    170839

    accuracy                        0.99343   2934271
   macro avg    0.99541   0.94462   0.96846   2934271
weighted avg    0.99346   0.99343   0.99326   2934271
```

5.1.4 Decision Tree

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

```
              precision    recall  f1-score   support

           0    0.99999   0.99972   0.99986   2763432
           1    0.99553   0.99980   0.99766    170839

    accuracy                        0.99973   2934271
   macro avg    0.99776   0.99976   0.99876   2934271
weighted avg    0.99973   0.99973   0.99973   2934271
```

5.1.5 Random Forest

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

```
              precision    recall  f1-score   support

           0    1.00000   0.99991   0.99995   2763432
           1    0.99852   1.00000   0.99926    170839

    accuracy                        0.99991   2934271
   macro avg    0.99926   0.99995   0.99961   2934271
weighted avg    0.99991   0.99991   0.99991   2934271
```

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 11.

5.3 10-Fold Cross Validation

Lastly, 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. Computing this report with TensorFlow 2.0 and Keras demanded a particular implementation, given the inputs requested by the function which instantiates this model in this project. Moreover, this cross-validation report had to be compatible and conceptually comparable to the reports ordinarily generated by the method *cross_val_score)*, that is, it should have the same information and data structure.

Consequently, a customised 10-fold cross-validation implementation for ANN was built in the following code, which was adapted from a solution proposed by Versloot (2022)

```
# implementing 10-fold cross-validation for NN

number_folds = 10
kfold = KFold(n_splits=number_folds, shuffle=True, random_state=42)

acc_per_fold = prec_per_fold = rec_per_fold = loss_per_fold = []

# Merge inputs and targets.
# At this point features and labels must be already scaled
inputs = np.concatenate((train_features, test_features), axis=0)
targets = np.concatenate((train_labels, test_labels), axis=0)

number_epochs = 58

fold_no = 1

print('#' * 72)
for train, test in kfold.split(inputs, targets):
    model = make_model()

    # Generate a print
    print(f'Fold #{fold_no}...')
```

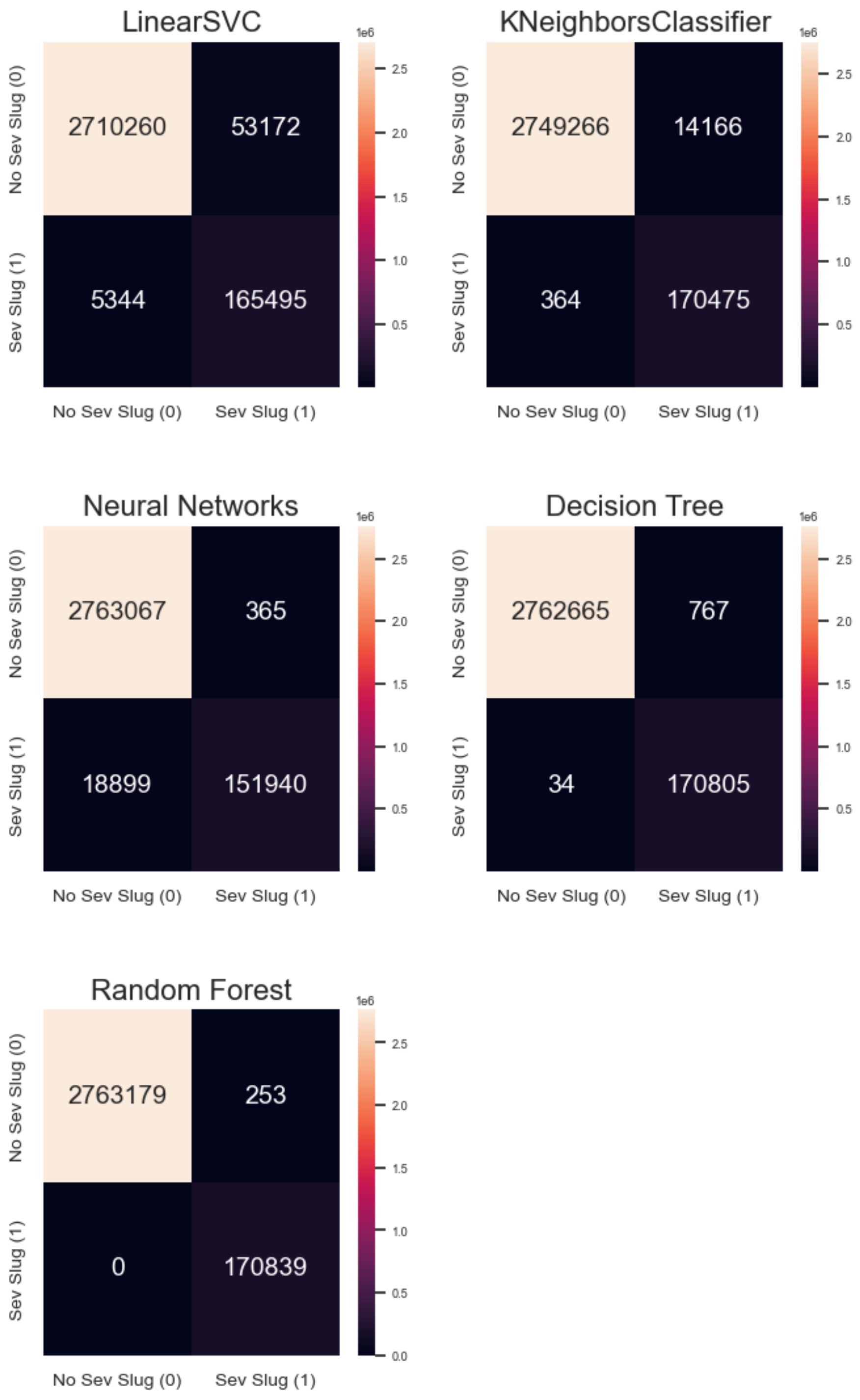

Figure 11: Confusion matrices showing the performance of five machine learning models in predicting severe slugging occurrences

```

    # Fit data to model
    history = model.fit(
      inputs[train], targets[train],
      batch_size=BATCH_SIZE,
      epochs=number_epochs,
      verbose=1
    )

    # Generate generalization metrics
    scores = model.evaluate(inputs[test], targets[test], batch_size=BATCH_SIZE,
    verbose=0)

    print(f'\nScores for Fold #{fold_no}:')

    fold_loss = (model.metrics_names[0], scores[0])
    fold_accu = (model.metrics_names[5], scores[5])
    fold_prec = (model.metrics_names[6], scores[6])
    fold_recl = (model.metrics_names[7], scores[7])

    print(f'{fold_loss[0]} = {fold_loss[1]};')
    print(f'{fold_accu[0]} = {fold_accu[1]};')
    print(f'{fold_prec[0]} = {fold_prec[1]};')
    print(f'{fold_recl[0]} = {fold_recl[1]};')

    loss_per_fold.append(fold_loss[1])
    acc_per_fold.append(fold_accu[1])
    prec_per_fold.append(fold_prec[1])
    rec_per_fold.append(fold_recl[1])

    fold_no = fold_no + 1
```

The other models were still put on the same pipelines used on modeling section.

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

After an empty dictionary to host the results from the cross-validation was initialised, a 10-fold cross-validation for each model (except ANN) was computed for 3 metrics, namely: accuracy, precision and recall.

```
cross_validation_dict = dict()

cv_metrics = ['accuracy','precision','recall']

cross_validation_dict = {name: {} for name, _ in models}

for cv_metric in cv_metrics:
    print(f'\nStarting a {number_folds}-fold CV for {cv_metric}...')
    for name, model in models:
        cv_results = cross_val_score(model, X_train, y_train, cv=kfold, scoring=
    cv_metric)
        cross_validation_dict[name][cv_metric] = cv_results
        msg = "-> %s: %f (%f)" % (name, np.nanmean(cv_results), np.nanstd(cv_results))
        print(msg)
```

```
Starting a 10-fold CV for accuracy...
-> Decision Tree: 0.999781 (0.000063)
-> Linear SVC: 0.974574 (0.000481)
-> KNeighborsClassifier: 0.996025 (0.000240)
-> Random Forest: 0.999948 (0.000022)

Starting a 10-fold CV for precision...
-> Decision Tree: 0.999718 (0.000098)
-> Linear SVC: 0.980681 (0.000688)
```

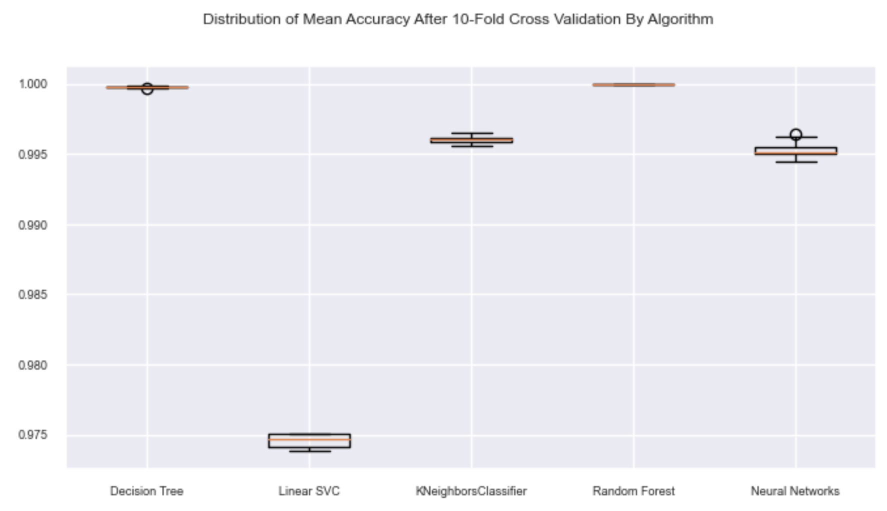

Figure 12: Box plots showing the distribution of Mean Accuracy After 10-fold Cross Validation By Algorithm

```
-> KNeighborsClassifier: 0.994433 (0.000448)
-> Random Forest: 0.999902 (0.000047)

Starting a 10-fold CV for recall...
-> Decision Tree: 0.999844 (0.000089)
-> Linear SVC: 0.968222 (0.000686)
-> KNeighborsClassifier: 0.997635 (0.000242)
-> Random Forest: 0.999995 (0.000010)
```

After this, the cross-validation results computed for ANN were added to the cross-validation dictionary.

```python
cv_val_dict = cross_validation_dict

cv_val_dict['Neural Networks'] = dict()
cv_val_dict['Neural Networks']['accuracy'] = np.array(acc_per_fold)
cv_val_dict['Neural Networks']['precision'] = np.array(prec_per_fold)
cv_val_dict['Neural Networks']['recall'] = np.array(rec_per_fold)

models.append(('Neural Networks', False))
```

The following code was implemented to plot how each metric averaged in each model. These corresponding charts can be visualised on Figures 12, 13 and 14.

```python
def plot_metric_boxplot(metric, cv_val_dict, number_folds):
    data = [cv_val_dict[model][metric] for model in cv_val_dict.keys()]
    fig = plt.figure()
    fig.suptitle(f'Distribution of Mean {metric.capitalize()} After {number_folds}-
    Fold Cross Validation By Algorithm')
    ax = fig.add_subplot(111)
    plt.boxplot(data)
    ax.set_xticklabels(cv_val_dict.keys())
    fig.set_size_inches(8,4)
    plt.show()

plot_metric_boxplot('accuracy', cv_val_dict, number_folds)
plot_metric_boxplot('precision', cv_val_dict, number_folds)
plot_metric_boxplot('recall', cv_val_dict, number_folds)
```

.
.
.

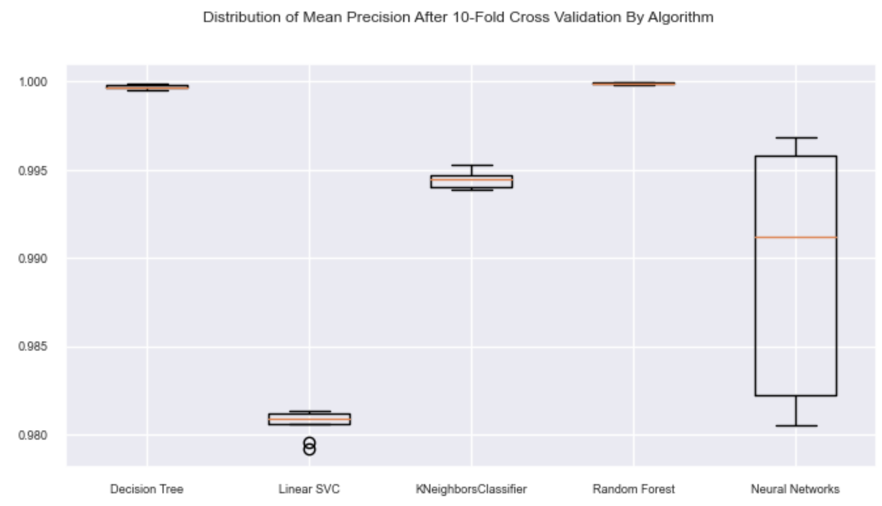

Figure 13: Box plots showing the distribution of Mean Accuracy After 10-fold Cross Validation By Algorithm

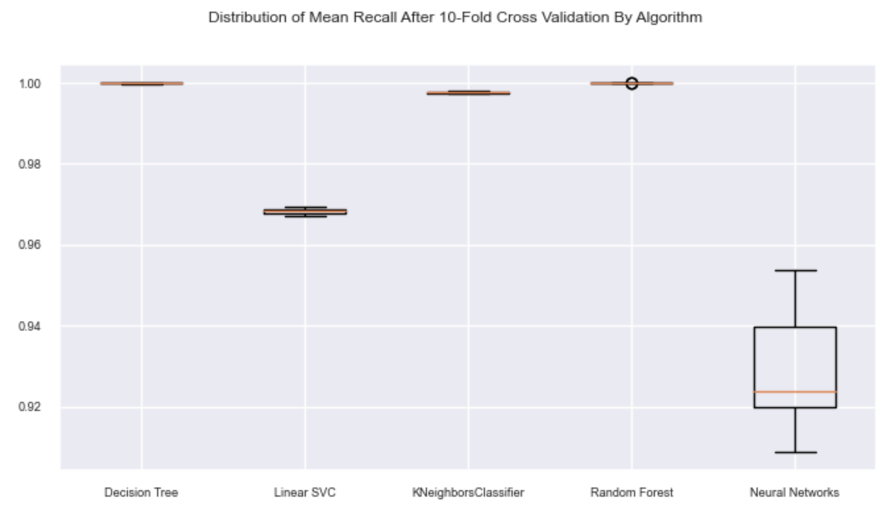

Figure 14: Box plots showing the distribution of Mean Accuracy After 10-fold Cross Validation By Algorithm

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*.

IBM SPSS Modeler CRISP-DM Guide (2017). Accessed: 2023-05-06. URL: https://www.ibm.com/docs/en/spss-modeler/18.1.1?topic=spss-modeler-crisp-dm-guide.

Jämsä-Jounela, Sirkka-Liisa (2007). "Future trends in Process Automation". In: *Annual Reviews in Control* 31.2, pp. 211–220. DOI: 10.1016/j.arcontrol.2007.08.003.

Martín Abadi et al. (2015). *Classification on imbalanced data: Tensorflow Core.* Software available from tensorflow.org. URL: https://www.tensorflow.org/tutorials/structured_data/imbalanced_data.

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_/.

— (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.

Venkatasubramanian, Venkat et al. (2003). "A review of process fault detection and diagnosis". In: *Computers &amp; Chemical Engineering* 27.3, pp. 293–311. DOI: 10.1016/s0098-1354(02)00160-6.

Versloot, Christian (Feb. 2022). *Machine-learning-articles/how-to-use-k-fold-cross-validation-with-keras.MD at main · Christianversloot/machine-learning-articles.* Accessed: 2023-05-06. URL: https://github.com/christianversloot/machine-learning-articles/blob/main/how-to-use-k-fold-cross-validation-with-keras.md.

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