Predicting Crash Survivability of Planes

DE1 Data Science

Bohan Wang[a], Harry Han[a], Devansh Goel[a], Lara Merican[a]

[a]*Dyson School of Design Engineering, Imperial College London*

June 2024

Abstract

Our analysis focused on historical plane crashes data spanning from 1918 to 2022, aiming to uncover the relationship between the causes of aviation accidents and their occurrence rates over time. Specifically, we investigated how four key elements affected the likelihood of surviving a plane crash using different machine learning algorithms. The chosen factors were phase of flight (Decision Trees), cause of crash (Logistic Regression), aircraft type (Support Vector Machines) and crash sites (Random Forest). The Decision Tree model had a validation set accuracy of 0.640 and recall of 0.953. The Logistic Regression model had a validation set accuracy of 0.897, recall of 0.810 and a notable precision of 0.997. The Support Vector Machines model had a validation set accuracy of 0.684 and a recall of 0.690. The Random Forest model had a test set accuracy of 0.730 and a recall of 0.825. Overall, the Logistic Regression model appeared to have the best values for accuracy, precision and recall. However, each model was used for a different area of the dataset, so the models cannot be directly compared.

Table of Contents

1. Introduction

Millions of people rely on air travel as a primary means of transport, highlighting the importance of survivability of passengers and crew in emergencies. While the number of aviation accidents is low compared to other transport modes (0.2 per million), their consequences can be catastrophic.

Understanding the factors that influence survivability can lead to advancements in aircraft design, technology, safety protocols and aviation trends. This study contributes to enhancing aviation safety, mitigating the risks associated with air travel.

Analysing data from past incidents using machine learning may uncover correlations that would not seem apparent. Machine learning algorithms can process vast amounts of data to identify factors that significantly affect survivability.

By leveraging modern analytical tools and building upon existing research such as reports from the Federal Aviation Administration (FAA) and studies by the Royal Aeronautics Society (RAeS), we can refine our understanding of what contributes to survival in plane crashes.

2. Methodology

This report refers to the dataset from the Bureau of Aircraft Accident Archives with 28536 data points of crashes ranging from May 1918 to June 2022. It contains 24 features regarding aircraft, temporal, locational and flight information.

2.1 Data pre-processing methods

- Filtering: Removed irrelevant and/or missing data points.
- Splitting: Dividing into training, validation and test sets (some models combined validation and test sets together) trained the model to adapt to unseen data.
- Balancing: SMOTE or undersampling techniques made the number of positive and negative instances equal.
- Feature selection: Selected intuitively to simplify the dataset, ensuring relevance to the key element chosen by each member.

2.2 Machine learning algorithms

- Decision Trees: The hyper-parameters max_depth and min_samples_split was tuned to get the best possible fit.
- Logistic Regression: Forward selection regression was used to identify the most significant predictors.
- Support Vector Machines (SVM): The hyper-parameters kernel type, C and gamma were tuned using an iterative approach of plotting graphs and analysing.
- Random Forest: The hyper-parameters max_depth, min_samples_split and n_estimators were tuned using a similar iterative approach.

3. Predicting crash survivability for Landing – Bohan Wang

3.1 Methodology

Decision trees are a versatile tool in machine learning, particularly useful for predicting crash survivability during landings. They operate by partitioning data based on features that maximize the information gain at each node. As the tree splits, it aims to decrease the Gini impurity, thereby improving the reliability of classification results. This method is non-parametric and supervised, making it effective for both classification and regression tasks in various domains.

3.1.1 Filtering the Data

This analysis aimed to determine which factors affect crash survivability during the landing phase. Therefore, the data was first filtered to include only records where the 'Flight phase' column (column F) was marked as 'Landing'. The landing phase involves various critical factors such as time, operator, Country and weather conditions, making this data particularly important.

3.1.2 Splitting the Data

After filtering the data, I split it into a training set and a secondary set that included both the test and validation sets, with 70% and 30% of the data points respectively. Then, I divided the secondary set in half, with one half used as the training set and the other half as the validation set.

3.1.3 Finding the Hyperparameters

To prevent overfitting, several graphs were plotted showing the accuracy based on the maximum depth and the minimum impurity decrease—these are known as hyperparameters.

Maximum Depth

To optimize the decision tree model for predicting plane survivability upon landing, the hyperparameter of maximum depth was analyzed. The graph of maximum depth versus accuracy for both training and validation data is shown.

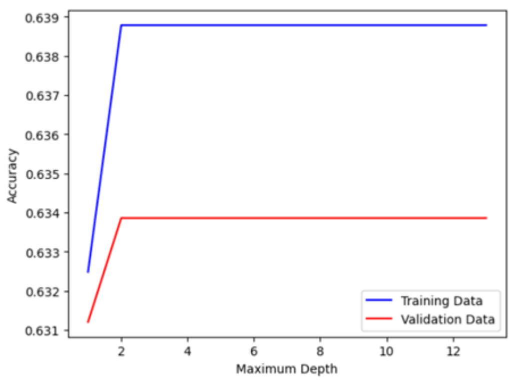

Figure 1: Maximum Depth vs Accuracy

From the graph, the training accuracy increases sharply and plateaus at a maximum depth of 3, reaching an accuracy of about 0.639. This suggests that the model captures the patterns in the training data well up to this point. Beyond a depth of 3, further increasing the depth does not yield significant improvements in accuracy and may lead to overfitting.

The validation accuracy shows a slight improvement up to a maximum depth of 3, stabilizing at around 0.634. This indicates that the model generalizes well to unseen data up to this depth but does not benefit from additional complexity.

Based on these observations, the optimal maximum depth for the decision tree model is determined to be 3. This depth provides a good balance between model complexity and generalization, ensuring that the model is neither underfitting nor overfitting the data.

Minimum Impurity Decrease

To optimize the decision tree model for predicting plane survivability upon landing, the hyperparameter of minimum impurity decrease (scaled by 0.01) was analyzed. The graph of minimum impurity decrease versus accuracy for both training and validation data is shown.

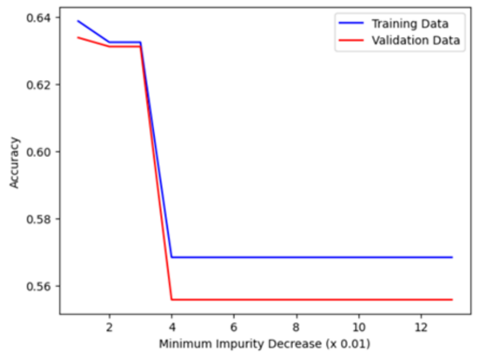

Figure 2: Minimum Impurity Decrease vs Accuracy

From the graph, the training accuracy increases steadily as the minimum impurity decrease is lowered, reaching about 0.57 when the minimum impurity decrease is close to 0.05. This suggests that the model effectively captures patterns in the training data as the impurity decrease threshold becomes more lenient, allowing more splits to occur.

The validation accuracy shows a slight improvement as the minimum impurity decrease is reduced, stabilizing at around 0.55 when the decrease is approximately 0.05. This indicates that the model generalizes well to unseen data with this threshold but does not benefit from further reductions in impurity decrease.

Based on these observations, the optimal minimum impurity decrease for the decision tree model is determined to be 0.05. This threshold provides a good balance between model complexity and generalization, ensuring that the model is neither underfitting nor overfitting the data.

3.2 Results

3.2.1 Making the Decision Tree

The decision tree was then built based on the identified hyperparameters. The Gini coefficients of the leaf nodes ranged from 0.34 to 0.5. The model also showed a recall of 95.3% on the validation set, and given that this is the most important metric for this dataset, it is a satisfactory result.

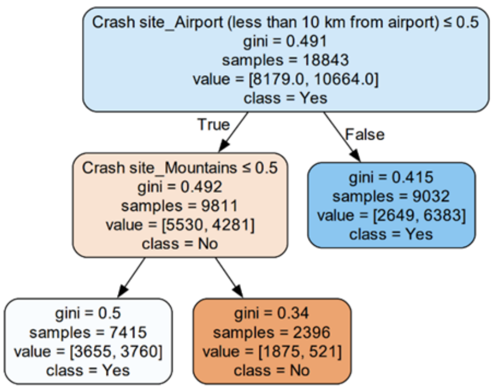

Figure 3: Decision Tree

Each condition statement in the nodes of the decision tree is based on whether a condition is less than or equal to 0.5. This is because the data has been binarized.

Table 1: Model results

	Accuracy	**Precision**	**Recall**
Training	0.638	0.617	0.951
Validation	0.640	0.620	0.953

3.2.2 Final Results

The optimized decision tree model achieved an accuracy of 0.638, precision of 0.617, and recall of 0.951 on the training data. On the testing data, it achieved an accuracy of 0.640, precision of 0.620, and recall of 0.953. These results indicate that the model generalizes well and effectively identifies survivable cases. However, considering that plane crashes are caused by a variety of factors, only some of which have been selected for analysis in this report, further discussion is required.

3.3 Discussion

The decision tree reveals that the factors that most affect crash survivability during the landing phase are crash sites (Mountains and Airport). If there are no critical factors such as operator error or adverse weather conditions, the model shows a higher survival rate. The Gini impurity of this node is less than 0.5, reveals a relatively high accuracy.

The decision tree analysis reveals key insights into the classification of crash sites. At the root node, which splits data based on whether the crash site is within 10 km of an airport, we see a moderate Gini impurity of 0.491. This node's samples are predominantly "No" (10,664) compared to "Yes" (8,179). Further splitting on whether the site is in mountainous regions shows a slightly higher impurity of 0.492 for non-mountainous areas within 10 km of an airport, indicating a balanced classification. Notably, for non-mountainous sites within this proximity, the Gini impurity drops to 0.415, with a majority classified as "Yes" (6,383 out of 9,032). Conversely, mountainous sites show a perfect Gini impurity of 0.5, indicating an equal distribution of "Yes" and "No" classifications. For sites more than 10 km from an airport, the impurity is lower at 0.34, with a majority classified as "No" (1,875 out of 2,396). These patterns suggest that proximity to an airport and the presence of mountains significantly influence the classification of crash sites.

4. Predicting crash survivability caused by technical failure – Harry Han

4.1 Methodology

Logistic regression was selected as the statistical model for our analysis because it is adept at establishing the relationship between one or more independent variables and a binary dependent variable. In this study, the binary dependent variable is the survival outcome of crew members in plane crashes, which is dichotomized into survived or not survived. The purpose of using logistic regression is to identify the features that correlate with survival rates in crashes caused by technical failures and to construct a model with high sensitivity.

To accurately identify features that positively correlate with survival rates, we employed the forward selection method. This method helps in systematically assessing the contribution of each variable to the model, selecting those that are statistically significant and substantively improve the prediction of the outcome. The features considered include, but are not limited to, the phase of flight, number of crew members, number of passengers, and the type of flight.

4.1.1 Filtering the Data

This aspect of the study aimed to identify factors that influence survival rates specifically in crashes caused by technical failures. Considering the focus on technical failure, it was pertinent to isolate incidents primarily attributed to this cause. Therefore, the dataset was filtered to exclude all crashes that did not cite 'Technical failure' as the primary crash cause. This exclusion criterion was vital to ensure the analysis remained targeted and relevant, enhancing the accuracy of the logistic regression model in predicting survival outcomes based solely on crashes resulting from technical failures.

4.1.2 Balancing the Data

The initial assessment of the dataset revealed a significant imbalance in the number of survivors compared to non-survivors in crashes attributed to technical failures. This imbalance could potentially bias the predictive model towards the more frequent class (survivors). To address this and enhance the accuracy and fairness of the model in real-world applications, we implemented a downsampling technique. This approach reduced the number of survivor cases to equal the number of non-survivor cases, thus balancing the dataset.

By downsampling, we created a new, balanced dataset where each class—survivors and non-survivors—comprised 1,678 cases. Although this method reduced the overall size of the dataset, which might increase the risk of overfitting, we plan to mitigate this risk by employing cross-validation during model training. This will involve comparing results from the training set with those from a validation set to ensure the model's generalizability.

4.1.3 Splitting the Data

The balanced dataset underwent a strategic segmentation to facilitate robust model training and evaluation. Initially, 60% of the data was allocated to the training set, with the remaining 40% reserved for both testing and validation purposes. This remaining portion was then equally divided, yielding two smaller sets: one for testing and the other for validation.

The rationale behind this segmentation approach is to provide a comprehensive foundation for the model's learning while ensuring adequate and separate resources for its thorough evaluation. Experimenting with different segmentation ratios, such as 80/20 and 70/30, allowed us to assess the optimal distribution of data that balances training depth with evaluation accuracy.

4.2 Results

4.2.1 Combination of Features that Correlate with the Predictor

To determine the optimal set of features that influence survival outcomes in crashes due to technical failures, we utilized an Automated Forward Selection technique. This iterative method begins with an empty model, progressively adding features that show the highest correlation with the target variable, in this case, the survival outcome. Each feature's addition is contingent upon it improving the model's validation accuracy by at least 0.005. This threshold ensures that only significantly contributive features are retained, enhancing model efficiency and relevance.

4.2.2 Validation

I employed three distinct data split ratios during our experiments: 60/40, 70/30, and 80/20. Each split ratio was evaluated to determine how the model's complexity—measured by the number of features included—affects its accuracy and potential for overfitting.

60/40 Split: This ratio provided a balanced approach, allowing enough data for training while retaining sufficient data for validation. However, as the model complexity increased, signs of overfitting began to appear, particularly after including more than three features.

70/30 Split: With a greater proportion of the data allocated to training, this split initially showed promise. Yet, it quickly led to overfitting, with a significant accuracy discrepancy between the training and test datasets, indicating a lack of generalizability.

80/20 Split: This was the most effective split for our study. The model maintained reasonable accuracy up to four features but started showing overfitting tendencies beyond this point. This split was optimal for achieving the best balance between training depth and model validation.

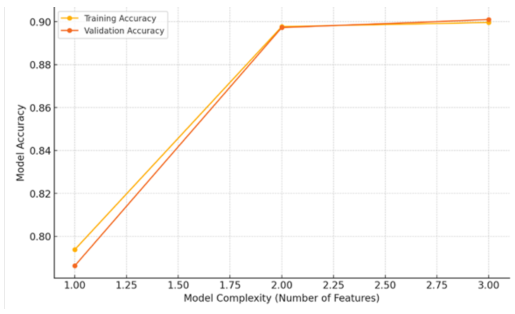

Figure 4: 60/40 split Model Complexity vs Model Accuracy

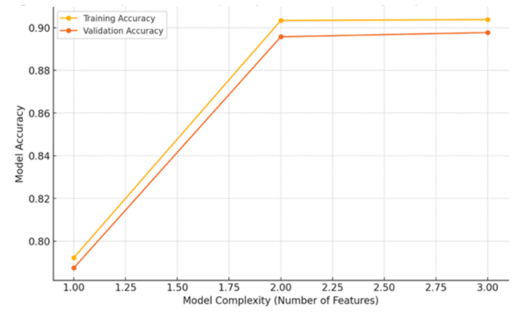

Figure 5: 70/30 Model Complexity vs Model Accuracy

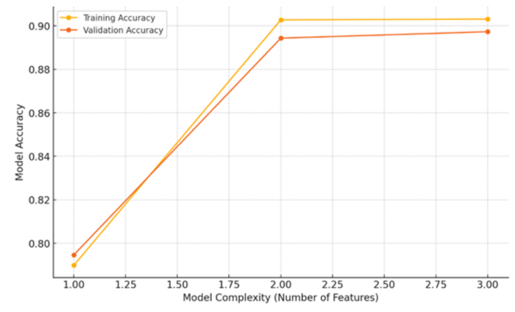

Figure 6: 80/20 Split Model Complexity vs Model Accuracy

4.2.3 Final Results

Table 2: Final results

Metrics	3 Train	4 Train	3 Validation	4 Validation
Accuracy	0.903	0.903	0.897	0.897
Precision	0.991	0.991	0.997	0.997
Recall	0.811	0.811	0.810	0.810

4.3 Discussion

The logistic regression forward selection method illuminated the primary factors affecting aircraft crash survivability, such as the number of crew onboard, the flight phase, and the type of flight. The model began to exhibit overfitting upon adding the fourth variable, aligning with the validation accuracy graph, which showed a decline in model accuracy as the model complexity reached four.

Given that this dataset involves critical safety data, the model should possess high sensitivity to effectively predict survivability in aircraft crashes. This is crucial not only for the safety of passengers and crew but also could impact the operational costs and reputation of airlines. As previously mentioned, the turning point for overfitting is at a model complexity of four, thus, we compared the validation accuracy, precision, and particularly the recall for model complexities of three and four.

High sensitivity is paramount in this context because predicting survivability accurately can significantly aid in emergency response planning and improving safety protocols. To mitigate overfitting as much as possible, the model selected had a model complexity of three, which still yielded relatively high recall, ensuring the model's high sensitivity.

To further enhance the model's performance, conducting a backward selection might also be beneficial as forward selection can sometimes produce suppressor effects. These occur when predictors are only significant in the presence of certain other variables, potentially reducing accuracy. This approach could help refine the model by eliminating less impactful features, thus potentially reducing the risk of overfitting that was observed with higher model complexities.

5. Predicting crash survivability for Boeing aircrafts – Devansh Goel

Boeing planes generally exhibit high crash survivability due to their robust design and advanced safety features. While each crash's specifics, such as flight phase and weather, influence survivability, ongoing technological advancements and safety regulations continuously attempt to enhance passenger survival rates in Boeing aircraft accidents. However, recently, several incidents of fatal crashes of Boeing Aircrafts have been reported.

5.1 Methodology

A Support Vector Machine (SVM) is a supervised machine learning algorithm that classifies data by finding an optimal hyperplane (plane in an n-dimensional plane) that maximises the distance between each class. The aim of the algorithm is to maximise the margin between points enabling it to find the best decision boundary between classes. In this section, the SVM will be used to predict the crash survivability of Boeing Aircrafts. SVM is used as the response variable is categorical – survivable or not-survivable crashes. Further, SVM's can manage imbalance datasets which is often the case with crash data as the number of non-survivable incidents might greatly exceed the survivable ones.

Hyperparameters of an SVM algorithm:

- C: Regularization. It is the penalty for misclassification. The larger the width of the margins, the lesser the value of C.
- Kernel: Chooses what shape is used to separate the classes.
- Gamma: A measure of how flexible the separating of classes is. A high gamma would fit a class around each data point.

5.2 Data pre-processing

New columns of relevance were created using the existing columns in the dataset. Crew and passenger numbers were combined to determine total number of people on board and the number of fatalities to assess the survivability of each crash. The dataset was then filtered to keep only those entries in which the aircraft was a Boeing.

The issue of class imbalance was tackled using Synthetic Minority Over-sampling Technique (SMOTE) to further increase the accuracy of the results.

The data was then split into training, validation and test sets to train and evaluate the SVM model efficiently.

5.3 Hyper-parameter Tuning

An iterative approach was employed to determine the optimal values for the SVM parameters. This process involved training the model with different parameter values and comparing the performance using accuracy, precision and recall metrics.

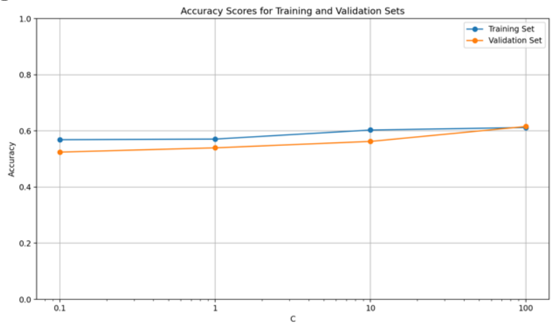

Figure 7: Accuracy Scores for different C values

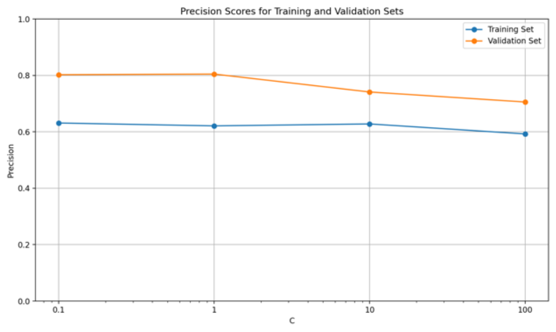

Figure 8: Precision Scores for different C values

From the above graphs, it is seen that accuracy and precision is highest with high values of C (kernel shape is set to RBF as default).

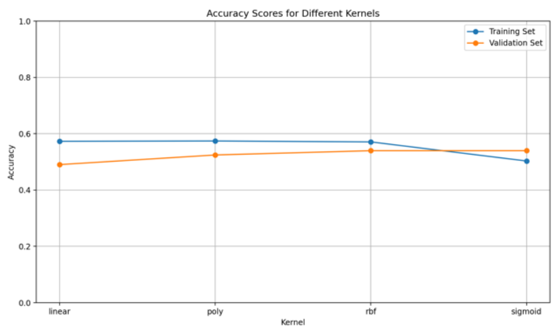

Figure 9: Accuracy Scores for Different Kernels

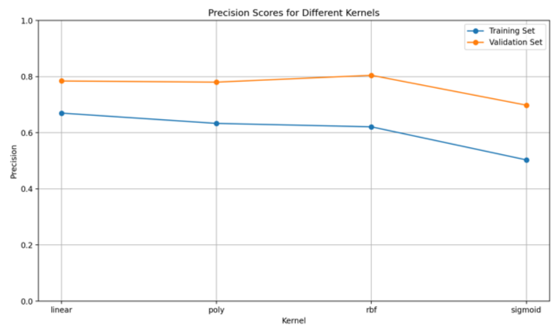

Figure 10: Precision Scores for different Kernels

Sigmoid Kernels give least accuracy and precision, while RBF gives highest (Value of C set to 1 as default).

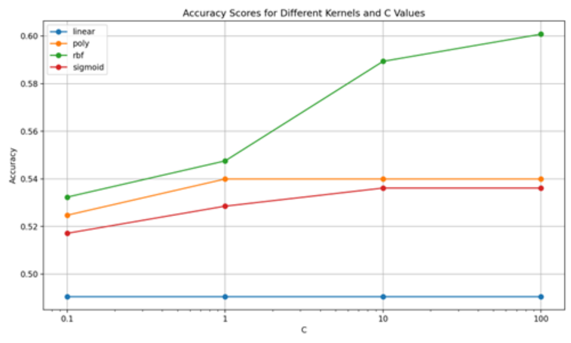

Figure 11: Accuracy for Kernel and C

This graph shows how the accuracy of the model changes for different C values across different kernels. The 'rbf' kernel shows a significant increase in accuracy at higher C values. This indicates that the rbf kernel benefits more from less regularization.

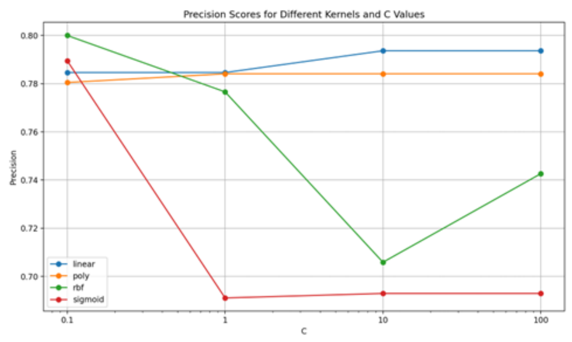

Figure 12: Precision Scores

This graph shows how precision changes with different values of C for each kernel. Precision reflects how many of the positive predictions made by the model are actually correct. Different kernels respond differently to changes in C. Ideally, the best-performing kernel will show higher precision scores across a range of C values.

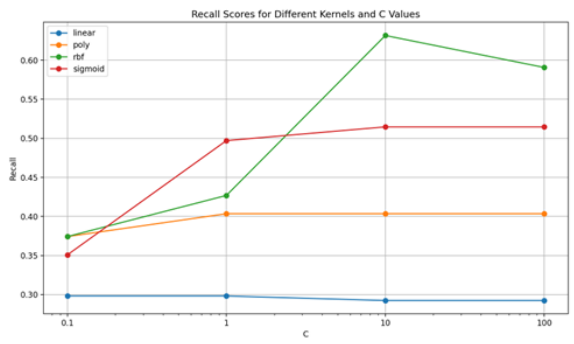

Figure 13: Recall Scores

This graph plots recall against C values for different kernels which shows the ability of the model to find all relevant positive cases. At higher values of C, the rbf kernel shows the greatest recall.

These graphs visually represent the hyperparameter tuning and the trade-offs between precision and recall.

5.4 Results

The best parameters based on the validation set's accuracy were found to be:

Kernel = rbf, C = 100, Gamma = 1

Table 3: Test set Classification report

	Precision	Recall
0 (Non-survivor)	0.61	0.71
1 (Survivor)	0.75	0.67

Accuracy = 68.44%

5.5 Discussion

This model has moderate performance with an overall accuracy of 68%. This indicates that while the model is reasonably good at predicting the survival outcome of plane crashes, especially given that plane crashes are attributed to several factors, there is still some room for improvement.

The model is more reliable when it predicts that the crash is survivable. In other words, there are fewer false positives. The use of RBF kernel with a high C value suggests that the model benefits from less regularisation, fitting the training data more closely. However, this can sometimes lead to overfitting wherein the model performs well on training data but less on unseen data. There is generally a trade-off between precision and recall – as one increases, the other decreases.

Table 4: Training set vs Validation set classification report

	Accuracy	Precision	Recall
Training	0.6522	0.65	0.65
Validation	0.6844	0.68	0.69

The model's performance reveals a difference in recall between the training and test sets for survivors. The jump from 0.65 recall for the training set to 0.69 in the test set suggests that the model is better at identifying survivors when applied to unseen data. Similarly, precision jump also indicates the same.

While the model's recall and precision rate are acceptable, they are not very high. The misclassifications can have serious real-life consequences as they could lead to inadequate safety measures.

Several enhancements could be considered to improve the model's performance. Incorporating additional relevant features that affect the plane crash such as weather conditions, crash site and flight operator, could provide more context and improve the accuracy of prediction. Further, fine-tuning hyperparameters using more sophisticated techniques like grid search or random search with cross-validation could also optimize the model further.

6. Predicting crash survivability in water bodies – Lara Merican

6.1 Methodology

Random forest is an ensemble learning algorithm that runs several decision tree models simultaneously to reach a single result, mitigating errors made by a single decision tree. A classification model was used as the predictor variable was categorical.

6.1.1 Feature removal

The following attributes were removed to simplify the dataset:

- *Registration, Flight no, MSN*: Identification attributes did not provide insight for predictive analysis.
- *Crew, PAX , Other, Total fatalities*: Had a direct causal relationship with the predictor variable.
- *Circumstances*: Did not add any further information for predicting survivability.
- *Date, Time, YOM*: Temporal features were not considered for this analysis.
- *Schedule, Aircraft, Operator, Region, Crash location, Flight type*: Too many value possibilities made the dataset too large to process after dummy values were created.

Remaining features: *Flight phase, Survivors, Crash site, Country, Crew on board, Pax on board* and *Crash cause.*

6.1.2 Feature engineering

The predictor variable *'Survivors'* was recorded as 'Yes' or 'No' – this was binarized into 1 and 0 respectively, as most algorithms operate on numerical data. The features *'Flight phase'*, *'Country'* and *'Crash cause'* were recorded as strings – one-hot encoding was used to create dummy variables with Boolean values. *'Crew on board'* and *'Pax on board'* remained as integers.

6.1.3 Filtering the data

The aim of this analysis was to predict the probability of whether there would be survivors if a plane crashed in a water body. The data was filtered to keep rows where *'Crash site'* was 'Lake, Sea, Ocean, River'. Due to the large size of the dataset, rows with missing values were removed, leaving 3708 data points.

6.1.2 Balancing the data

The data was slightly imbalanced, with 1914 positive and 1794 negative instances. The majority class of positive instances was undersampled to equal the number of negative instances, leaving 3588 data points.

6.1.3 Splitting the data

To ensure the model adapts to unseen data and avoids overfitting, a splitting process was essential. The remaining data was split randomly into training (60%), validation (30%) and test (10%) sets. This split was suitable for a dataset of this size (a 10% test split may not be suitable for a small dataset).

6.1.4 Hyper-parameter Tuning

Three hyper-parameters were set: max_depth, min_samples_split and n_estimators. They were tuned to get the best possible results from the model by plotting each hyper-parameter against recall to choose the best values. Parameters were plotted against recall to maximise the number of true positives. The model was tested after each hyper-parameter value was chosen – checking for improvements in recall.

An initial model was set to hyper-parameter values of max_depth = 2, min_samples_split = 4 and n_estimators = 1.

Maximum depth

'Max_depth' describes the maximum depth of each decision tree in the forest.

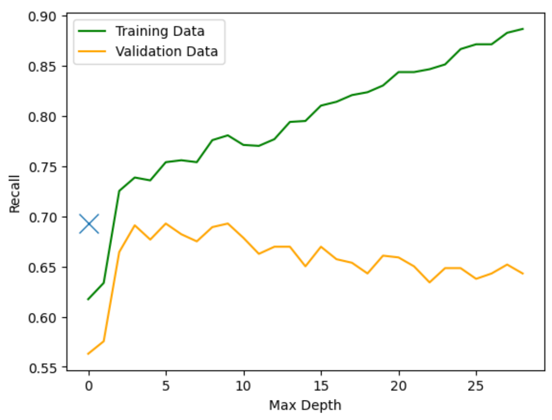

Figure 14: Max Depth vs Recall

The validation data had a significantly lower recall compared to the training data, especially after a maximum depth of ~7. A max_depth of 5 was chosen for Model 2 as it had the highest recall in the validation set. However, the graph indicates overfitting as the recall for validation data decreased while the training data increased.

Minimum samples split

'Min_samples_split' describes the minimum number of samples required for a node to be a split.

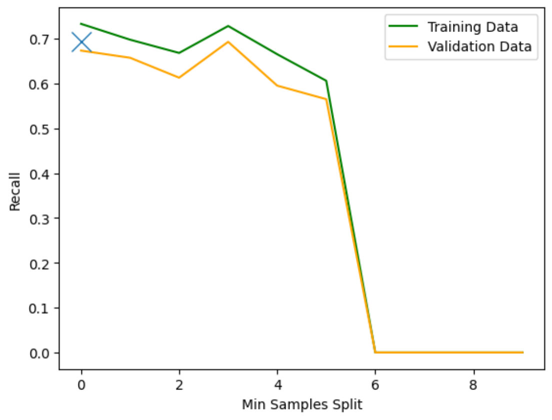

Figure 15: Min Samples Split vs Recall

The validation data and training data had a similar recall; however, it drops significantly after minimum samples split of 5. A min_samples_split of 3 was chosen for Model 3.

N estimators

'N_estimators' describes the number of decision trees in the random forest.

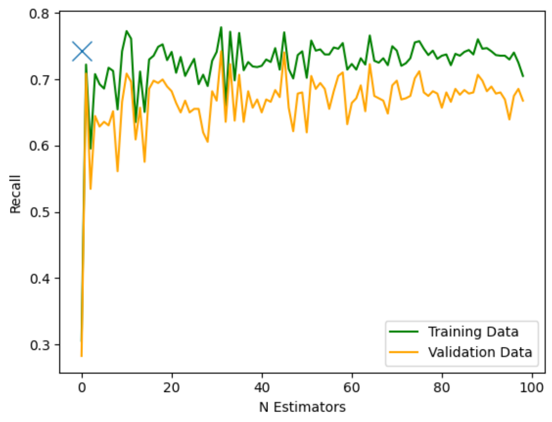

Figure 16: N Estimators vs Recall

A value of 35 was chosen to maximise recall. Hence, Model 4 had a max_depth of 5, min_samples_split of 3 and n_estimators of 35.

6.2 Results

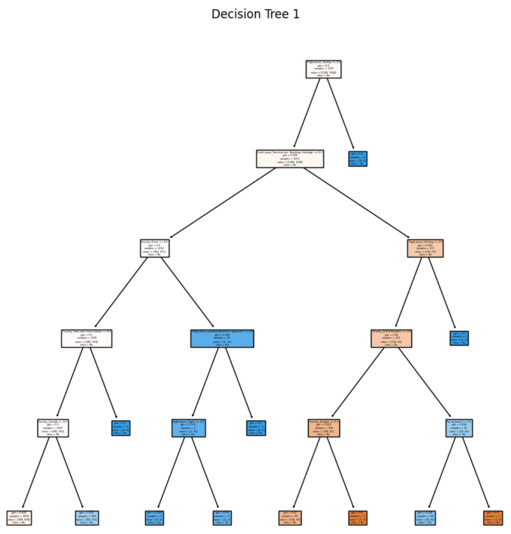

Figure 17: First decision tree of the random forest

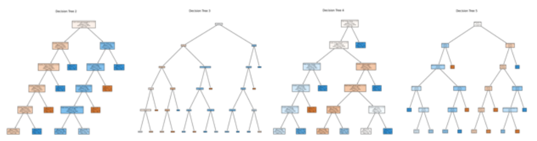

Figure 18: 4 out of 35 decision trees in the Random Forest Model

Table 4: Comparing recall of different models during different stages of hyper-parameter tuning

Model	Training	Validation	Test
1	0.350	0.366	0.377
2	0.741	0.659	0.770
3	0.752	0.702	0.776
4	0.758	0.725	0.825

The recall of the training, validation and test sets improved significantly during the hyper-parameter tuning process.

- Model 1: Showed signs of underfitting – this was expected as random hyper-parameter values were used.
- Model 2: Recall values improved, however a significant difference between the training and validation sets suggested overfitting.
- Model 3: Recall values improved further.
- Model 4: Had the best recall values out of all models and no significant difference between the training and validation sets.

6.2.1 Final results

Table 5: Model 4 Results

	Accuracy	Precision	Recall
Training	0.717	0.690	0.758
Validation	0.664	0.663	0.725
Test	0.730	0.699	0.825

There was a small drop in values from the training to the validation set, possibly suggesting some minor overfitting. The test set showed very good results, supporting the effectiveness of Model 4, however, it was unusual that it had higher values compared to the training and validation sets. Perhaps this test set was slightly easier to predict compared to other sets.

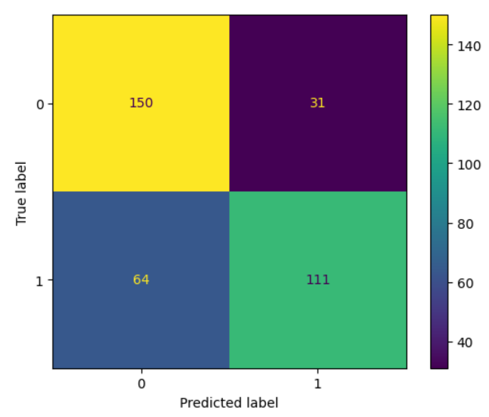

Figure 19: Confusion matrix of Model 4 test set

The confusion matrix showed a promising number of true positives and true negatives, reinforcing the effectiveness of Model 4. There were more false negatives than false positives, which would be important in the real world as it avoided false assurances of survivability.

6.3 Discussion

The final random forest model showed promising results for predicting crash survivability in water bodies as shown by the test results and confusion matrix. However, there is substantial room for improvement.

Attribute removal was an important step to simplify the dataset, but using a LASSO regression model and dimensionality reduction may have been more ideal to choose the best features. This was attempted, however after making dummy variables for all the features, the dataset was too large for the model to process.

While the hyper-parameter tuning process significantly improved the recall values, it could be further improved by using a cross-validation method such as GridSearch CV.

7. Conclusion

A key aspect of our methodology was the SVM, which identified the most influential predictors of survivability. This confirmed the significance of certain features – flight phase and aircraft type emerging as critical determinants.

However, the dataset's scope, primarily historical and limited by geographical and temporal factors, may restrict our results. Predicting survivability in crashes also presents challenges due to the multitude of factors influencing outcomes. Moreover, the imbalance in data, where incidents resulting in no survivors outnumbered those with survivors, added another layer of difficulty. This made it challenging to train accurate models that accounted for both scenarios effectively. The current models work to a certain degree of accuracy for the historical data but may require adjustments and broader data inclusion to enhance their predictive power for future and more diverse scenarios.

Overall, our findings underscore the value of data-driven methodologies in enhancing accuracy, contributing to safer aviation practices. As data availability and modeling technologies evolve, ongoing refinement of these tools will be crucial in sustaining their effectiveness in improving aircraft safety.

Appendix A - Decision Tree Code (Bohan Wang)

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score
import graphviz
from sklearn.tree import export_graphviz
import matplotlib.pyplot as plt

# import file
file_path = 'C:/Users/hp/Desktop/xian/Plane Crashes.csv'
data = pd.read_csv(file_path, encoding='ISO-8859-1')

# Delete columns with too many irrelevant or missing values
threshold = 0.5  # Threshold ratio, greater than which missing value columns will be deleted
data = data.dropna(thresh=len(data) * threshold, axis=1)

# Handling of missing values
data = data.dropna(subset=['Date', 'Aircraft', 'Operator', 'Flight phase', 'Flight type', 'Survivors',
'Crash site'])

# Coding Target Variables
data['Survivors'] = data['Survivors'].apply(lambda x: 1 if x == 'Yes' else 0)

# Select Features
features = ['Date', 'Aircraft', 'Operator', 'Flight phase', 'Flight type', 'Crash site']
X = data[features]
y = data['Survivors']

# Coded categorical variables
X = pd.get_dummies(X)

# Divide the data setv
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)

# Finding the optimal maximum depth
acc_train_depth = []
acc_val_depth = []

X_train_sub,   X_val,   y_train_sub,   y_val   =   train_test_split(X_train,   y_train,   test_size=0.2,
random_state=0)

for i in range(1, 14):
    clf = DecisionTreeClassifier(max_depth=i, min_impurity_decrease=0.01)
    clf.fit(X_train_sub, y_train_sub)
    y_pred_train_sub = clf.predict(X_train_sub)
    y_pred_val = clf.predict(X_val)
    acc_train_depth.append(accuracy_score(y_train_sub, y_pred_train_sub))
    acc_val_depth.append(accuracy_score(y_val, y_pred_val))

plt.figure()
plt.plot(range(1, 14), acc_train_depth, "b", label="Training Data")
plt.plot(range(1, 14), acc_val_depth, "r", label="Validation Data")
plt.xlabel("Maximum Depth")
plt.ylabel("Accuracy")
plt.legend()
plt.show()

# Selection of optimum maximum depth
best_max_depth = acc_val_depth.index(max(acc_val_depth)) + 1
```

```
# Finding the optimal minimum purity reduction
acc_train_impurity = []
acc_val_impurity = []

for i in range(1, 14):
    clf = DecisionTreeClassifier(max_depth=best_max_depth, min_impurity_decrease=i/100)
    clf.fit(X_train_sub, y_train_sub)
    y_pred_train_sub = clf.predict(X_train_sub)
    y_pred_val = clf.predict(X_val)
    acc_train_impurity.append(accuracy_score(y_train_sub, y_pred_train_sub))
    acc_val_impurity.append(accuracy_score(y_val, y_pred_val))

plt.figure()
plt.plot(range(1, 14), acc_train_impurity, "b", label="Training Data")
plt.plot(range(1, 14), acc_val_impurity, "r", label="Validation Data")
plt.xlabel("Minimum Impurity Decrease (x 0.01)")
plt.ylabel("Accuracy")
plt.legend()
plt.show()

# Select the best minimum purity reduction value and train the final model
best_min_impurity_decrease = (acc_val_impurity.index(max(acc_val_impurity)) + 1) / 100
clf = DecisionTreeClassifier(max_depth=best_max_depth, min_impurity_decrease=best_min_impurity_decrease)
clf.fit(X_train, y_train)

# Forecasting and assessment
y_pred_train = clf.predict(X_train)
y_pred_test = clf.predict(X_test)

acc_train = accuracy_score(y_train, y_pred_train)
prec_train = precision_score(y_train, y_pred_train)
rec_train = recall_score(y_train, y_pred_train)

acc_test = accuracy_score(y_test, y_pred_test)
prec_test = precision_score(y_test, y_pred_test)
rec_test = recall_score(y_test, y_pred_test)

# export
print('训练集上的准确率: {:.3f}'.format(acc_train))
print('训练集上的精确率: {:.3f}'.format(prec_train))
print('训练集上的召回率: {:.3f}'.format(rec_train))

print('测试集上的准确率: {:.3f}'.format(acc_test))
print('测试集上的精确率: {:.3f}'.format(prec_test))
print('测试集上的召回率: {:.3f}'.format(rec_test))

# Visual Decision Tree
dot_data = export_graphviz(clf, out_file=None, feature_names=X.columns, class_names=['No', 'Yes'],
filled=True, rounded=True, special_characters=True)
graph = graphviz.Source(dot_data)
graph.render("plane_crash_decision_tree")
graph.view()
```

Appendix B – Logistic Regression Code (Harry Han)

Balancing data:

```
from sklearn.utils import resample

# Separate the majority and minority classes
```

```
df_majority = technical_failure_data[technical_failure_data['Survivors'] == 'Yes']
df_minority = technical_failure_data[technical_failure_data['Survivors'] == 'No']

# Downsample the majority class
df_majority_downsampled = resample(df_majority,
                                   replace=False,    # sample without replacement
                                   n_samples=len(df_minority),  # to match minority class
                                   random_state=123) # reproducible results

# Combine minority class with downsampled majority class
df_balanced = pd.concat([df_majority_downsampled, df_minority])

# Display new class counts
balanced_counts = df_balanced['Survivors'].value_counts()
balanced_counts, df_balanced.head()
```

Splitting data:

```
from sklearn.model_selection import train_test_split

# First, split the data into training (60%) and secondary set (40%)
train_set, secondary_set = train_test_split(df_balanced, test_size=0.4, random_state=42)

# Split the secondary set equally into test and validation sets
test_set, validation_set = train_test_split(secondary_set, test_size=0.5, random_state=42)

# Display the sizes of each set to confirm the split
len_train = len(train_set)
len_test = len(test_set)
len_validation = len(validation_set)

(len_train, len_test, len_validation)
```

Combination of Features:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

# Assuming 'df_balanced' is the pre-processed dataset ready for modeling
# and 'Survivors' is the target variable
X = df_balanced.drop('Survivors', axis=1)
y = df_balanced['Survivors'].apply(lambda x: 1 if x == 'Yes' else 0)  # Convert to binary

# Split data into training and validation sets
X_train, X_validation, y_train, y_validation = train_test_split(X, y, test_size=0.4, random_state=42)

def forward_feature_selection(X_train, y_train, X_validation, y_validation):
    initial_features = X_train.columns.tolist()
    selected_features = []
    best_acc = 0

    while initial_features:
        acc_dict = {}
        for feature in initial_features:
            model = LogisticRegression(solver='liblinear')
            X_train_selected = X_train[selected_features + [feature]]
            model.fit(X_train_selected, y_train)
            y_pred = model.predict(X_validation[selected_features + [feature]])
            acc = accuracy_score(y_validation, y_pred)
            acc_dict[feature] = acc

        best_feature, best_feature_acc = max(acc_dict.items(), key=lambda x: x[1])
        if best_feature_acc - best_acc > 0.005:
            selected_features.append(best_feature)
            best_acc = best_feature_acc
            initial_features.remove(best_feature)
        else:
            break

    return selected_features, best_acc
```

```
selected_features,   model_accuracy   =   forward_feature_selection(X_train,   y_train,   X_validation,
y_validation)
print("Selected Features:", selected_features)
print("Validation Accuracy with selected features:", model_accuracy)
```

Figure : 60/40 split Model Complexity vs Model Accuracy

```
# Check the actual columns that are causing issues by filtering out any completely NaN columns before
imputation
non_nan_columns = X_train_numeric.columns[~X_train_numeric.isna().all()].tolist()

# Applying imputation only on non-NaN columns to avoid dropping any columns
X_train_non_nan = X_train_numeric[non_nan_columns]
X_test_non_nan = X_test_numeric[non_nan_columns]

# Applying imputation
imputer = SimpleImputer(strategy='mean')
X_train_imputed = imputer.fit_transform(X_train_non_nan)
X_test_imputed = imputer.transform(X_test_non_nan)

# Convert the numpy arrays back to pandas DataFrames with the correct columns
X_train_imputed_df = pd.DataFrame(X_train_imputed, columns=non_nan_columns)
X_test_imputed_df = pd.DataFrame(X_test_imputed, columns=non_nan_columns)

# Re-run forward selection with the correct imputed data
features_imputed, train_accs_imputed, test_accs_imputed = forward_selection_numeric(
    X_train_imputed_df, y_train, X_test_imputed_df, y_test)

# Plotting the results after proper imputation and selection
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(train_accs_imputed)+1), train_accs_imputed, label='Training Accuracy', marker='o')
plt.plot(range(1, len(test_accs_imputed)+1), test_accs_imputed, label='Validation Accuracy', marker='o')
plt.title('Figure 7: 60/40 Split Model Complexity vs Model Accuracy (Imputed Numeric Data)')
plt.xlabel('Model Complexity (Number of Features)')
plt.ylabel('Model Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```

Figure : 70/30 Model Complexity vs Model Accuracy

```
# Reapply the imputation and make sure to handle any column discrepancies in the 70/30 split data

# Filter the training and test data to include only the non-NaN columns to avoid the previous error
X_train_70_non_nan = X_train_70[non_nan_columns]
X_test_30_non_nan = X_test_30[non_nan_columns]

# Reapply the imputer
X_train_70_imputed = pd.DataFrame(imputer.fit_transform(X_train_70_non_nan), columns=non_nan_columns)
X_test_30_imputed = pd.DataFrame(imputer.transform(X_test_30_non_nan), columns=non_nan_columns)

# Perform forward selection on imputed numeric data for 70/30 split again
features_70, train_accs_70, test_accs_70 = forward_selection_numeric(
    X_train_70_imputed, y_train_70, X_test_30_imputed, y_test_30)

# Plotting the results for 70/30 split
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(train_accs_70)+1), train_accs_70, label='Training Accuracy', marker='o')
plt.plot(range(1, len(test_accs_70)+1), test_accs_70, label='Validation Accuracy', marker='o')
plt.title('Figure 8: 70/30 Split Model Complexity vs Model Accuracy (Imputed Numeric Data)')
plt.xlabel('Model Complexity (Number of Features)')
plt.ylabel('Model Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```

Figure : 80/20 Split Model Complexity vs Model Accuracy

```
# Check for any all-NaN columns in the 80/20 split data and handle them before imputation
X_train_80_non_nan = X_train_80[non_nan_columns]
X_test_20_non_nan = X_test_20[non_nan_columns]

# Reapply the imputer on the non-NaN column data
X_train_80_imputed = pd.DataFrame(imputer.fit_transform(X_train_80_non_nan), columns=non_nan_columns)
```

```
X_test_20_imputed = pd.DataFrame(imputer.transform(X_test_20_non_nan), columns=non_nan_columns)

# Perform forward selection on the imputed numeric data for the 80/20 split
features_80, train_accs_80, test_accs_80 = forward_selection_numeric(
    X_train_80_imputed, y_train_80, X_test_20_imputed, y_test_20)

# Plotting the results for the 80/20 split
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(train_accs_80)+1), train_accs_80, label='Training Accuracy', marker='o')
plt.plot(range(1, len(test_accs_80)+1), test_accs_80, label='Validation Accuracy', marker='o')
plt.title('Figure 9: 80/20 Split Model Complexity vs Model Accuracy (Imputed Numeric Data)')
plt.xlabel('Model Complexity (Number of Features)')
plt.ylabel('Model Accuracy')
plt.legend()
plt.grid(True)
plt.show()

Final result

from sklearn.metrics import precision_score, recall_score

# Re-calculate the metrics using the correct imports
results = {
    "Metrics": ["Accuracy", "Precision", "Recall"],
    "3 Train": [accuracy_score(y_train_80, y_train_pred_3), precision_score(y_train_80, y_train_pred_3), recall_score(y_train_80, y_train_pred_3)],
    "4 Train": [accuracy_score(y_train_80, y_train_pred_4), precision_score(y_train_80, y_train_pred_4), recall_score(y_train_80, y_train_pred_4)],
    "3 Validation": [accuracy_score(y_test_20, y_test_pred_3), precision_score(y_test_20, y_test_pred_3), recall_score(y_test_20, y_test_pred_3)],
    "4 Validation": [accuracy_score(y_test_20, y_test_pred_4), precision_score(y_test_20, y_test_pred_4), recall_score(y_test_20, y_test_pred_4)]
}

# Convert results to DataFrame
final_results_df = pd.DataFrame(results)
final_results_df
```

Appendix C – SVM Code (Devansh Goel)

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score, recall_score, classification_report
from imblearn.over_sampling import SMOTE

# Load the dataset
data = pd.read_csv('Plane_Crashes.csv')

# Data Preprocessing
data['Aboard'] = data['Crew on board'].fillna(0) + data['Pax on board'].fillna(0)
data['Fatalities'] = data['Crew fatalities'].fillna(0) + data['PAX fatalities'].fillna(0)
data['Survival'] = np.where(data['Fatalities'] < data['Aboard'], 1, 0)

# Filter the dataset to include only Boeing aircraft
data = data[data['Aircraft'].str.contains('Boeing', na=False)]
data = data[['Aircraft', 'Aboard', 'Fatalities', 'Survival']]
data = data.dropna()

# Encode categorical variables
label_encoder = LabelEncoder()
data['Aircraft'] = label_encoder.fit_transform(data['Aircraft'])

# Define features and target variable
X = data[['Aircraft', 'Aboard']]
y = data['Survival']
```

```
# Split the dataset into training and test sets
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Further split training and validation sets
X_train,   X_val,   y_train,   y_val   =   train_test_split(X_train_val,   y_train_val,   test_size=0.25,
random_state=42)

# Address class imbalance using SMOTE on training set
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)

# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_resampled)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)

# Define hyperparameters and kernels to iterate over
Cs = [0.1, 1, 10, 100]
gammas = [1, 0.1, 0.01, 0.001]
kernels = ['linear', 'poly', 'rbf', 'sigmoid']

# Containers for storing results
accuracy_scores = np.zeros((len(kernels), len(Cs), len(gammas)))
precision_scores = np.zeros((len(kernels), len(Cs), len(gammas)))
recall_scores = np.zeros((len(kernels), len(Cs), len(gammas)))
train_accuracies = []
val_accuracies = []
train_precisions = []
val_precisions = []

# Iterate over each value of C
for C in Cs:
    # Train SVM model
    model = SVC(kernel='rbf', C=C, gamma='scale', class_weight='balanced', random_state=42)
    model.fit(X_train_scaled, y_train_resampled)

    # Evaluate on training set
    y_train_pred = model.predict(X_train_scaled)
    train_accuracy = accuracy_score(y_train_resampled, y_train_pred)
    train_accuracies.append(train_accuracy)
    train_precision = precision_score(y_train_resampled, y_train_pred)
    train_precisions.append(train_precision)

    # Evaluate on validation set
    y_val_pred = model.predict(X_val_scaled)
    val_accuracy = accuracy_score(y_val, y_val_pred)
    val_accuracies.append(val_accuracy)
    val_precision = precision_score(y_val, y_val_pred)
    val_precisions.append(val_precision)

# Plotting accuracy scores for training and validation sets
plt.figure(figsize=(10, 6))
plt.plot(Cs, train_accuracies, marker='o', label='Training Set')
plt.plot(Cs, val_accuracies, marker='o', label='Validation Set')
plt.xscale('log')
plt.xlabel('C')
plt.ylabel('Accuracy')
plt.title('Accuracy Scores for Training and Validation Sets')
plt.xticks(Cs, Cs)
plt.ylim(0, 1)  # Set y-axis limits
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Plotting precision scores for training and validation sets
plt.figure(figsize=(10, 6))
plt.plot(Cs, train_precisions, marker='o', label='Training Set')
plt.plot(Cs, val_precisions, marker='o', label='Validation Set')
plt.xscale('log')
plt.xlabel('C')
plt.ylabel('Precision')
plt.title('Precision Scores for Training and Validation Sets')
plt.xticks(Cs, Cs)
plt.ylim(0, 1)  # Set y-axis limits
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Iterate over each combination of parameters
for i, kernel in enumerate(kernels):
    for j, C in enumerate(Cs):
        for k, gamma in enumerate(gammas):
            # Train SVM model
            model = SVC(kernel=kernel, C=C, gamma=gamma, class_weight='balanced', random_state=42)
            model.fit(X_train_scaled, y_train_resampled)

            # Evaluate on validation set
            y_pred = model.predict(X_val_scaled)
            accuracy_scores[i, j, k] = accuracy_score(y_val, y_pred)
            precision_scores[i, j, k] = precision_score(y_val, y_pred)
            recall_scores[i, j, k] = recall_score(y_val, y_pred)

# Plotting accuracy scores for each kernel
plt.figure(figsize=(10, 6))
for i, kernel in enumerate(kernels):
    plt.plot(Cs, accuracy_scores[i, :, 0], marker='o', label=f'{kernel}')
plt.xscale('log')
plt.xlabel('C')
plt.ylabel('Accuracy')
plt.title('Accuracy Scores for Different Kernels and C Values')
plt.xticks(Cs, Cs)
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Plotting precision scores for each kernel
plt.figure(figsize=(10, 6))
for i, kernel in enumerate(kernels):
    plt.plot(Cs, precision_scores[i, :, 0], marker='o', label=f'{kernel}')
plt.xscale('log')
plt.xlabel('C')
plt.ylabel('Precision')
plt.title('Precision Scores for Different Kernels and C Values')
plt.xticks(Cs, Cs)
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Plotting recall scores for each kernel
plt.figure(figsize=(10, 6))
for i, kernel in enumerate(kernels):
    plt.plot(Cs, recall_scores[i, :, 0], marker='o', label=f'{kernel}')
plt.xscale('log')
plt.xlabel('C')
plt.ylabel('Recall')
plt.title('Recall Scores for Different Kernels and C Values')
plt.xticks(Cs, Cs)
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Find the best parameters
```

```
best_indices = np.unravel_index(np.argmax(accuracy_scores, axis=None), accuracy_scores.shape)
best_kernel = kernels[best_indices[0]]
best_C = Cs[best_indices[1]]
best_gamma = gammas[best_indices[2]]
print(f'Best Parameters - Kernel: {best_kernel}, C: {best_C}, Gamma: {best_gamma}')

# Train final model on best parameters using full training set
final_model    =    SVC(kernel=best_kernel,    C=best_C,    gamma=best_gamma,    class_weight='balanced',
random_state=42)
final_model.fit(X_train_scaled, y_train_resampled)

# Predict on training set
y_pred_train = final_model.predict(X_train_scaled)

# Predict on test set
y_pred_test = final_model.predict(X_test_scaled)

# Print classification report and accuracy score on training set
print("Training Set Classification Report:")
print(classification_report(y_train_resampled, y_pred_train))
print("Training Set Accuracy:", accuracy_score(y_train_resampled, y_pred_train))

# Print classification report and accuracy score on test set
print("Test Set Classification Report:")
print(classification_report(y_test, y_pred_test))
print("Test Set Accuracy:", accuracy_score(y_test, y_pred_test))
```

Appendix D – Random Forest Code (Lara Merican)

https://colab.research.google.com/drive/1lsljXjOzJMmtwtGqitYE74KKpOp7ustq?usp=sharing

```
# Connecting to Google Drive
from google.colab import drive
drive.mount('/content/drive')

# Importing packages
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import warnings
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression as logreg
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.tree import plot_tree
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
warnings.filterwarnings('ignore') # prevents future version warnings

# Filtering data
planes = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/Plane Crashes.csv')
planes_water = planes[planes['Crash site'] == 'Lake, Sea, Ocean, River']
planes_water = planes_water.drop(columns=['Registration','Flight no.','MSN','Circumstances','Crew
fatalities','PAX fatalities','Other fatalities','Total
fatalities','Date','Time','Schedule','Aircraft','Operator','YOM','Region','Crash location','Flight
type'])
planes_water.dropna(inplace=True)
num_rows = planes_water.shape[0]
print(num_rows)
planes_water.head()
planes_water.info(verbose=True)

# Feature engineering
planes_water.replace(to_replace=['Yes', 'No'], value=[1, 0], inplace=True)
planes_water = pd.get_dummies(planes_water)
planes_water.head()
planes_water['Survivors'].describe()

# Balancing data
total = len(planes_water)
```

```
nb_pos = planes_water['Survivors'].sum()
nb_neg = total - nb_pos
print(nb_pos, nb_neg)
survivors_pos = planes_water[planes_water['Survivors']==1].sample(nb_neg)
survivors_neg = planes_water[planes_water['Survivors']==0]
resampled_planes_water = pd.concat((survivors_pos,survivors_neg))
total = len(resampled_planes_water)
nb_pos = resampled_planes_water['Survivors'].sum()
nb_neg = total - nb_pos
print(nb_pos, nb_neg)

# Splitting data
X = resampled_planes_water.drop(columns=['Survivors'])
y = resampled_planes_water['Survivors']
print(len(X))
print(len(y))
#60-30-10 (0.4,0.25)
X_train, X_test_validation, y_train, y_test_validation = train_test_split(X, y, test_size=0.4,
random_state=0)
X_validation, X_test, y_validation, y_test = train_test_split(X_test_validation, y_test_validation,
test_size=0.25, random_state=0)
print('X_train: {}, y_train: {}'.format(len(X_train), len(y_train)))
print('X_validation: {}, y_validation: {}'.format(len(X_validation), len(y_validation)))
print('X_test: {}, y_test: {}'.format(len(X_test), len(y_test)))

# Defining testing the model
def test_model():
  ypred_train = model.predict(X_train)
  ypred_val = model.predict(X_validation)
  ypred_test = model.predict(X_test)
  acc_train = accuracy_score(y_train, ypred_train)
  prec_train = precision_score(y_train,ypred_train)
  rec_train = recall_score(y_train,ypred_train)
  acc_val = accuracy_score(y_validation, ypred_val)
  prec_val = precision_score(y_validation,ypred_val)
  rec_val = recall_score(y_validation,ypred_val)
  acc_test = accuracy_score(y_test, ypred_test)
  prec_test = precision_score(y_test,ypred_test)
  rec_test = recall_score(y_test,ypred_test)
  print('TRAIN')
  print('Accuracy :{}'.format(acc_train))
  print('Precision :{}'.format(prec_train))
  print('Recall :{}'.format(rec_train))
  print('')
  print('VALIDATION')
  print('Accuracy Validation:{}'.format(acc_val))
  print('Precision Validation:{}'.format(prec_val))
  print('Recall Validation:{}'.format(rec_val))
  print('')
  print('TEST')
  print('Accuracy Test:{}'.format(acc_test))
  print('Precision Test:{}'.format(prec_test))
  print('Recall Test:{}'.format(rec_test))


# Model 1 - Random parameters
model = RandomForestClassifier(max_depth = 2, min_samples_split = 4, n_estimators=1)
model = model.fit(X_train,y_train)
test_model()


# Tuning max_depth
rec_train = []
rec_val = []

for i in range (1,30):
  dt = RandomForestClassifier(max_depth = i)
  dt.fit(X_train, y_train)
  ypred_train = model.predict(X_train)
  ypred_validation = model.predict(X_validation)
  ypred_test = model.predict(X_test)
  rec_train.append(recall_score(y_train, dt.predict(X_train)))
  rec_val.append(recall_score(y_validation, dt.predict(X_validation)))
```

```
plt.figure()
plt.plot(rec_train,'g')
plt.plot(rec_val, 'orange')
plt.xlabel('Max Depth')
plt.ylabel('Recall')
plt.legend(['Training Data','Validation Data'])

max_recall = np.max(rec_val)
plt.plot(max_recall, marker='x', markersize=15)
plt.show()
print(max_recall)

# Model 2 – With tuned max_depth
model = RandomForestClassifier(max_depth = 5)
model = model.fit(X_train,y_train)
test_model()

# Tuning min_samples_split
rec_train = []
rec_val = []

samples = np.linspace(0.01,1,10, endpoint = True)

for i in samples:
  dt = RandomForestClassifier(max_depth = 5, min_samples_split = i)
  dt.fit(X_train, y_train)
  ypred_train = model.predict(X_train)
  ypred_val = model.predict(X_validation)
  ypred_test = model.predict(X_test)
  rec_train.append(recall_score(y_train, dt.predict(X_train)))
  rec_val.append(recall_score(y_validation, dt.predict(X_validation)))

plt.figure()
plt.plot(rec_train,'g')
plt.plot(rec_val, 'orange')
plt.xlabel('Min Samples Split')
plt.ylabel('Recall')
plt.legend(['Training Data','Validation Data'])
max_recall = np.max(rec_val)
print(max_recall)
plt.plot(max_recall, marker='x', markersize=15)
plt.show()

# Model 3 – With tuned max_depth and min_samples_split
model = RandomForestClassifier(max_depth = 5, min_samples_split = 3)
model = model.fit(X_train,y_train)
test_model()

# Tuning n_estimators
rec_train = []
rec_val = []

for i in range (1,100):
  dt = RandomForestClassifier(max_depth = 5, min_samples_split = 3,n_estimators=i)
  dt.fit(X_train, y_train)
  ypred_train = model.predict(X_train)
  ypred_validation = model.predict(X_validation)
  ypred_test = model.predict(X_test)
  rec_train.append(recall_score(y_train, dt.predict(X_train)))
  rec_val.append(recall_score(y_validation, dt.predict(X_validation)))

plt.figure()
plt.plot(rec_train,'g')
plt.plot(rec_val, 'orange')
plt.xlabel('N Estimators')
plt.ylabel('Recall')
plt.legend(['Training Data','Validation Data'])

max_recall = np.max(rec_val)
print(max_recall)
plt.plot(max_recall, marker='x', markersize=15)
plt.show()

# Model 4 – With tuned max_depth, min_samples_split and n_estimators
```

```
model = RandomForestClassifier(max_depth = 5, min_samples_split = 3, n_estimators=35)
model = model.fit(X_train,y_train)
test_model()

# Random Forest
for i, tree_in_forest in enumerate(model.estimators_):
    plt.figure(figsize=(10, 10))
    plot_tree(tree_in_forest, feature_names=X_train.columns, class_names=['No', 'Yes'], filled=True)
    plt.title(f'Decision Tree {i+1}')
    plt.show()


# Confusion Matrices
ypred_train = model.predict(X_train)
ypred_validation = model.predict(X_validation)
ypred_test = model.predict(X_test)

cm_train = confusion_matrix(y_train, ypred_train)
cm_val = confusion_matrix(y_validation, ypred_val)
cm_test = confusion_matrix(y_test, ypred_test)

fig, ax = plt.subplots(1, 3, figsize=(18, 6))

# Training Confusion Matrix
ConfusionMatrixDisplay(cm_train, display_labels=model.classes_).plot(ax=ax[0])
ax[0].set_title('Confusion Matrix - Training Set')

# Validation Confusion Matrix
ConfusionMatrixDisplay(cm_val, display_labels=model.classes_).plot(ax=ax[1])
ax[1].set_title('Confusion Matrix - Validation Set')

# Test Confusion Matrix
ConfusionMatrixDisplay(cm_test, display_labels=model.classes_).plot(ax=ax[2])
ax[2].set_title('Confusion Matrix - Test Set')

plt.show()
```