Spaceship Titanic: Using Machine Learning Techniques to Predict Passengers Transported to an Alternate Dimension

An **Ongoing** Kaggle Competition

Cai Xue'er
G2204858L

Leon Sun
G2204908A

Abstract

This project addresses the ongoing Kaggle challenge [1], 'Spaceship Titanic: Predict which passengers are transported to an alternate dimension.' The aim is to determine if passengers aboard the Spaceship Titanic were transported to an alternate dimension using the provided dataset. We experimented with various Machine Learning (ML) classification models from scikit-learn and TensorFlow, incorporating hyperparameter tuning to optimise baseline predictions. The most successful model, TensorFlow Decision Trees, achieved an impressive 80.71% accuracy on Kaggle's official test dataset, placing us within the top 10%.

1. Introduction

The Kaggle competition describes a scenario during the year 2912, where the Spaceship Titanic, an interstellar passenger liner collided with a spacetime anomaly. As a result, almost half of its 13,000 passengers were transported to an alternate dimension. Our task is to accurately identify these passengers to aid an urgent rescue mission.

1.1. Problem Statement

This binary classification problem involves a labelled training dataset of passenger records. Leveraging feature engineering and state-of-the-art Machine Learning (ML) techniques, our model classifies test dataset passengers as either 'Transported' (class label 1 | TRUE) or 'Not Transported' (class label 0 | FALSE).

1.2. Challenges of Problem

Data fragmentation and damage during the accident make our datasets potentially incomplete, posing a challenge for machine learning models. Proper handling of missing values is crucial to avoid bias and ensure accurate predictions. Feature engineering is another problem due to the limited original dataset features (only 13 features in total). We will need to transform informative features while considering the impact of noisy data on prediction accuracy.

There are a multitude of machine learning models available for binary classification. This will require rigorous experimentation and optimization to enhance our prediction accuracy.

In the following sections, we will detail how these challenges are tackled to meet the objective of predicting passengers transported to the alternate dimension.

2. Proposed Solution

Our Machine Learning Experimentation lifecycle is segmented into 5 key steps:

1. **Raw Data Exploration**
   We delve into the dataset, extracting vital statistics and meticulously examining class imbalances. Understanding the raw data is fundamental to the subsequent stages.

2. **Feature Engineering & Selection**
   Feature engineering is paramount before model selection. Our approach consists of feature cleaning, aggregation, construction, transformation, normalisation, and discretization. Engineering meaningful features lays the foundation for robust machine learning models.

   We then conducted a careful review to select the features for our machine learning model. Unnecessary features are pruned.

3. **Model Experimentation**
   We experimented with a variety of state-of-the-art (SOTA) and traditional Machine Learning models.

[1] https://www.kaggle.com/competitions/spaceship-titanic

Each model's suitability is evaluated based on our dataset's characteristics.

Experiments are rigorously trained and validated using 5-fold Cross Validation, ensuring reliability and generalizability by preventing overfitting of data. This technique divides the data into five subsets, holding one for testing and utilising the remaining four for training, ensuring robustness in predictions.

Basic hyperparameter tuning and iterative refinement are conducted to optimise the experimented model performance. Fine-tuning the models' settings is crucial for achieving optimal results.

4. **Model Selection**
The model with the best cross-validation score is chosen. Further iterations involve additional hyperparameter tuning, feature engineering, and refinement, optimising the model's settings for peak performance.

5. **Testing and Evaluation**
The finalised model undergoes rigorous testing on the designated test dataset. Performance metrics such as accuracy, precision and recall are employed for comprehensive evaluation.

The model's predictions are submitted to Kaggle, where final test accuracy scores are obtained, validating the model's effectiveness in predicting affected passengers.

By meticulously following these steps, we ensure a systematic and thorough approach, maximising the potential of our model for the given assignment.

2.1. Data Exploration

2.1.1 Key Statistics of Training[2] And Testing[3] Dataset

The training dataset contains 8693 data instances, with a total of 8 categorical features and 4 numerical features (see Appendix A; Figure A1)

The testing dataset contains 4277 data instances, with a total of 7 categorical features (excludes final prediction of transported value) and 4 numerical features (see Appendix A; Figure A2).

2.1.2 Missing Values

As mentioned in Section 1.2, a challenge we face is missing values. Approximately 2% of attribute values were missing in both the train and test datasets.

Heatmaps in Figure 1 and 2 visually depict missing values across different passenger records and attributes (see Appendix B; Figure B1 for detailed statistics).

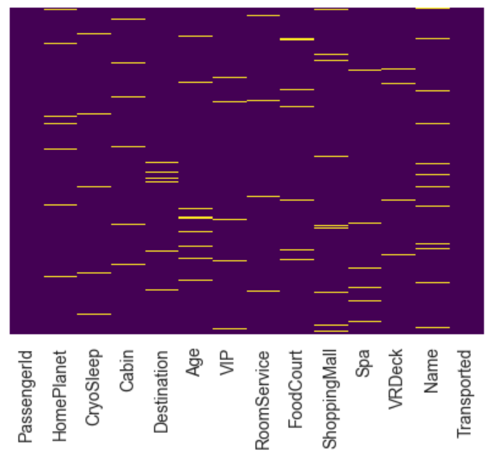

Figure 1. Heatmap of Missing Values in Training Dataset

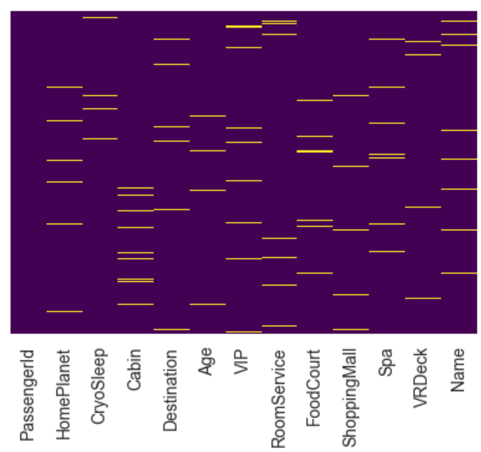

Figure 2. Heatmap of Missing Values in Testing Dataset

2.1.3 High Cardinality Features

Next, we explore categorical features with high cardinality.

The following categorical features in the training dataset has more than 2 unique values:

[2] Training Dataset = 'train.csv' provided in Kaggle Competition with PassengerID and Transported Results.

[3] Test Dataset = 'test.csv' provided in Kaggle Competition with PassengerID and without Transported Results (we are required to provide predicted results for Transported Field).

- HomePlanet: 3 unique values
- Destination: 3 unique values
- Cabin: 6560 unique values
- Name: 8473 unique values
- PassengerId: 8693 unique values

We retained ['HomePlanet, 'Destination'] due to manageable number of unique values. For ['Cabin'], engineering new variables is planned to minimise noise and overfitting. ['Name', 'PassengerId'] are dropped to avoid overfitting and computational load.

In summary, the general goal will be to separate or generalise high cardinality categories through feature engineering (detailed in Section 2.2).

2.1.4 Check for Class Imbalances

Finally, we check if there are potential class imbalances from our training dataset. In the event that there is any class imbalance, we may need to conduct resampling techniques to balance the class distribution or modify our machine learning algorithm to penalise misclassifying minority class more than majority class. We may also need to use ensemble techniques like Balanced Random Forest to handle imbalanced data.

However, we observed that approximately half of the passengers in the training dataset were transported to the alternate dimensions (Figure 3). Hence, additional handling of class imbalance will not be required.

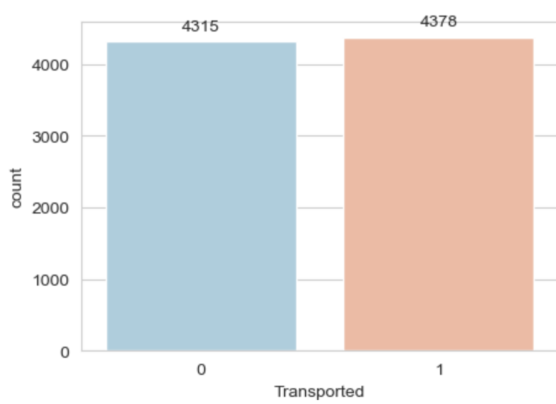

Figure 3. Proportion of Passengers Not Transported (0) and Transported (1)

2.2. Feature Engineering

We initiate the machine learning process by cleaning and preprocessing the data. By handling missing values and resolving high cardinality features, we can provide a reliable foundation for our subsequent steps.

2.2.1 Feature Cleaning

Instead of eliminating data instances with missing values, we fill in missing values in the following manner:

- Numerical features such as CryoSleep, RoomService, FoodCourt, ShoppingMall, Spa, VRDeck were filled with 0. We assume the passengers did not spend any money on these activities as no records of spending was shown.
- Only numerical feature ['Age'] was filled with mean value to facilitate feature discretisation in Section 2.2.3.
- Categorical features ['HomePlanet', 'Destination', 'VIP', 'CabinDeck', 'CabinNumber', 'CabinSide'] were filled with mode values.

2.2.2 Feature Construction

We created new features to capture more important information of the data instead of relying on high cardinality features which can lead to unnecessary noise and overfitting.

Cabin Deck | Cabin Number | Cabin Side
We split ['Cabin'] into three different columns - ['CabinDeck', 'CabinNumber' and 'CabinSide']. Subsequently, we dropped the high cardinality ['Cabin'] feature.

Cabin	**→**	**Cabin Deck**	**Cabin Number**	**CabinSide**
B/0/P		B	0	P
F/1/S		F	1	S

Table 1. Splitting of Column ['Cabin'] to 3 separate features

IsAlone
We also constructed a feature ['IsAlone'] where we identify if a passenger was alone based on their PassengerID (see Appendix B; Figure B2).

Example: PassengerID **0003**_01 and **0003**_02 were not alone as they have the same first 4 integers.

2.2.3 Feature Transformation: Feature Discretization (Binning)

Age Group

Next, [‘Age’] is discretized into bins, resulting in the 'AgeGroup' column with 8 intervals (Table 2, Figure 4).

Age Group	Age Range
0	0 <= Age <10
1	10 <= Age < 20
2	20 <= Age < 30
3	30 <= Age < 40
4	40 <= Age < 50
5	50 <= Age < 60
6	60 <= Age < 70
7	70 <= Age < 80

Table 2. Binning of Age into Age Groups (10 years)

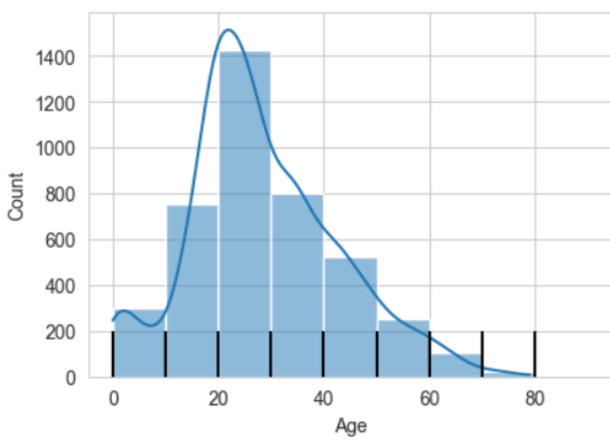

Figure 4. Histogram of Age feature

2.2.4 Feature Transformation: Convert Boolean Values to Integer Values

Some Machine Learning models are not able to accept boolean values and require numerical values. Hence, we converted boolean values of Columns [‘VIP’, ‘Transported’, ’CryoSleep’] in the training dataset to integer values where False is represented as 0 and True is represented as 1.

2.2.5 Feature Transformation: One-Hot Encoding

Finally, we conducted one-hot encoding for categorical features with more than 2 unique values - ['HomePlanet', 'Destination', 'CabinDeck', 'CabinSide', 'AgeGroup'].

One-hot encoding is essential in machine learning as it transforms categorical variables into a numerical format, preventing misinterpretation by algorithms. By assigning a unique binary code to each category, it maintains categorical distinctions. This technique enhances model performance, allowing for effective generalisation to new data.

2.3. Features Selection for Machine Learning Model

Following One-Hot Encoding, we finalised the features for our Machine Learning Models. We eliminated high cardinality features such as ['Name', 'Cabin', 'CabinNum', 'Age'].

By systematically converting boolean values, applying one-hot encoding, and carefully selecting features, we established a refined set of inputs for our machine learning models as depicted in Figure 5.

```
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   PassengerId                8693 non-null   object
 1   CryoSleep                  8693 non-null   int32
 2   VIP                        8693 non-null   int32
 3   RoomService                8693 non-null   float64
 4   FoodCourt                  8693 non-null   float64
 5   ShoppingMall               8693 non-null   float64
 6   Spa                        8693 non-null   float64
 7   VRDeck                     8693 non-null   float64
 8   Transported                8693 non-null   int32
 9   IsAlone                    8693 non-null   int32
 10  HomePlanet_Earth           8693 non-null   int32
 11  HomePlanet_Europa          8693 non-null   int32
 12  HomePlanet_Mars            8693 non-null   int32
 13  Destination_55 Cancri e    8693 non-null   int32
 14  Destination_PSO J318.5-22  8693 non-null   int32
 15  Destination_TRAPPIST-1e    8693 non-null   int32
 16  CabinDeck_A                8693 non-null   int32
 17  CabinDeck_B                8693 non-null   int32
 18  CabinDeck_C                8693 non-null   int32
 19  CabinDeck_D                8693 non-null   int32
 20  CabinDeck_E                8693 non-null   int32
 21  CabinDeck_F                8693 non-null   int32
 22  CabinDeck_G                8693 non-null   int32
 23  CabinDeck_T                8693 non-null   int32
 24  CabinSide_P                8693 non-null   int32
 25  CabinSide_S                8693 non-null   int32
 26  AgeGroup_0                 8693 non-null   int32
 27  AgeGroup_1                 8693 non-null   int32
 28  AgeGroup_2                 8693 non-null   int32
 29  AgeGroup_3                 8693 non-null   int32
 30  AgeGroup_4                 8693 non-null   int32
 31  AgeGroup_5                 8693 non-null   int32
 32  AgeGroup_6                 8693 non-null   int32
 33  AgeGroup_7                 8693 non-null   int32
```

Figure 5. Final Features **after** Feature Engineering for Training Dataset (Similar for Test Dataset excluding Transported Columns)

2.4. Performance Metrics

Submissions on Kaggle are evaluated based on classification accuracy score as seen in Figure 6.

Accuracy score refers to the percentage of correct predictions out of the total number of predictions.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

For binary classification, accuracy can also be calculated in terms of positives and negatives as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Where *TP* = True Positives, *TN* = True Negatives, *FP* = False Positives, and *FN* = False Negatives.

Figure 6. Snippet of evaluation criteria from Kaggle

3. Experiments and Performance

3.1. Machine Learning Algorithms

We train our training dataset of 8693 data instances by experimenting with the following machine learning algorithms from the scikit-learn framework. This high-level library allowed us to implement both traditional and SOTA machine learning algorithms quickly and derive baseline estimates of the models' performance:

- K-Nearest Neighbors
- Support Vector Machine
- Logistics Regression
- Random Forests
- AdaBoost
- Bagging
- Gradient Boosted Trees

In the following section, we explore and tune their hyperparameters to improve their cross-validation accuracy score.

The hyperparameters listed for each algorithm represent the best configurations obtained through initial tuning and manual experimentation, while those not specified utilise default values from the scikit-learn package.

3.1.1 K-Nearest Neighbors

Best Hyperparameters: KNeighborsClassifier ('metric': 'manhattan', 'n_neighbors': 10, 'weights': 'uniform')

The distance metric used for K-Nearest Neighbors is the Manhattan distance, which represents the sum of the absolute differences between corresponding coordinates. We consider the 10 closest neighbours based on their Manhattan distance, and all points within the neighbourhood are given equal weightage in the model.

3.1.2 Support Vector Machine

Best Hyperparameters: SVC ('C': 2, 'kernel': 'linear')

We used a moderate value of C = 2 to regularise the smoothness of the decision boundary. Smaller 'C' values (e.g., 0.1) yield a smoother boundary, while larger values (e.g., 100) can lead to a more intricate boundary by emphasising accurate data point classification. The 'linear' kernel is employed for a linear decision boundary, ideal when the feature-target relationship is roughly linear.

3.1.3 Logistics Regression

Best Hyperparameters:
LogisticRegression ('C': 3, 'solver': 'liblinear')

In Logistic Regression, the optimal hyperparameters were 'C = 3' and 'solver = liblinear'. The regularisation parameter 'C' influences the trade-off between a smooth decision boundary and accurate classification, with a smaller 'C' emphasising smoothness and a larger 'C' allowing complexity. The 'liblinear' solver, chosen for optimization, is effective for small to medium-sized datasets, employing a coordinate descent method to efficiently optimise the logistic regression objective function for both binary classification tasks.

3.1.4 Random Forests

Best Hyperparameters: RandomForestClassifier ('max_depth': 10, 'n_estimators': 400)

In Random Forest, 'max_depth' controls the depth of individual decision trees, influencing their complexity and potential for overfitting. With 'max_depth' set to 10 in this instance, a balance is struck between capturing intricate patterns and preventing overfitting. Meanwhile, 'n_estimators' dictates the total number of trees in the ensemble, with a higher value, such as 400 in this case, generally improving model performance at the expense of increased training time.

3.1.5 AdaBoost

Best Hyperparameters:
AdaBoostClassifier ('learning_rate': 1.0, 'n_estimators': 100)

AdaBoost is a boosting ensemble classifier that iteratively fits additional copies of a base classifier on the dataset while adjusting weights for misclassified instances. It iteratively fits additional copies of a base classifier on the dataset, adjusting weights for misclassified instances.

Key parameters include 'n_estimators = 100' defining the maximum boosting iterations, and the unspecified base estimator, defaulting to DecisionTreeClassifier with a maximum depth of 1. 'Learning_rate' reduces each classifier's contribution, creating a trade-off with 'n_estimators' that influences the boosting process.

3.1.6 Bagging

Best Hyperparameters: BaggingClassifier ('max_features': 0.6, 'max_samples': 0.7, 'n_estimators': 50)

This ensemble technique combines predictions from multiple base models trained on random subsets of the data. 'n_estimators' sets the number of models, while 'max_features' and 'max_samples' control feature and data subset randomness, enhancing model diversity. These hyperparameters optimise Bagging for improved generalisation and performance.

3.1.7 Gradient Boosted Trees

Best Hyperparameters: GradientBoostingClassifier ('max_depth': 2, 'n_estimators': 300)

The Gradient Boosted Trees model attained peak performance with 'max_depth' at 2 and 'n_estimators' at 300, denoting a shallow tree structure and 300 boosting iterations, respectively. While a higher 'n_estimators' often yields a more accurate model, it comes with increased computational complexity and training time. The choice of a shallow tree (low 'max_depth') mitigates overfitting risks but may limit the capture of complex patterns.

3.2. Cross-validation Scores

We employ five-fold cross-validation with GridSearchCV, selecting the model with the highest cross-validation accuracy score as our final choice for further enhancement in Section 3.4. From Table 3, Gradient Boosted Trees stand out, surpassing other models with an accuracy score of 80.54%.

Algorithm	Cross Validation Accuracy (%)	Training Time (seconds)
K-Nearest Neighbors (KNN)	75.37	8.04
Logistics Regression	79.64	6.62
AdaBoost	79.72	41.87
Support Vector Machine (SVM)	80.11	84.98
Bagging	80.20	189.23
Random Forests	80.23	130.74
Gradient Boosted Trees (GBT)	**80.54**	**223.23**

Table 3. Cross Validation Accuracy Score (in ascending order)

3.3. Advantages and Disadvantages

After experimenting and observing the cross-validation accuracy of the different models, we summarise their performance along with their advantages and disadvantages.

Worst Performer: K-Nearest Neighbors (KNN)

Despite KNN’s advantages of being a simple and flexible non-parametric, instance-based learning algorithm, it had the lowest cross validation accuracy score of 75.37%.

Using one-hot encoded features in high-dimensional and sparsely populated data poses a challenge for KNN. Our KNN relies on Manhattan distances to measure dissimilarity between data points, but in sparse one-hot encoding, instances with similar token counts can yield similar distances, even if they are dissimilar. As seen in Appendix B (Figure B3), our data has sparse one-hot encoding. Hence, this can mislead KNN, making the model unsuitable for our binary classification problem.

Mid-Tier Performer: Logistics Regression

Logistic Regression demonstrated a respectable cross-validation accuracy of 79.64%. This model is widely used for binary classification tasks due to its simplicity and interpretability. Logistic Regression calculates the probability of a sample belonging to a certain class, making it effective for decision-making processes.

One of the advantages of Logistic Regression is its interpretability. The model provides coefficients for each feature, allowing us to understand the impact of individual features on the prediction. Additionally, Logistic Regression is less prone to overfitting when the number of features is relatively small.

However, Logistic Regression has limitations. It assumes a linear relationship between features and the log-odds of the target variable, which might not capture complex, non-linear patterns in the data. This could limit its performance as compared to our better performers like ensemble methods, especially when dealing with intricate decision boundaries.

Mid-Tier Performer: AdaBoost

AdaBoost demonstrated a cross-validation accuracy of 79.72%, positioning it as a mid-tier performer in our experiment. AdaBoost, short for Adaptive Boosting, is an ensemble learning method that combines the predictions of several weak learners (usually decision trees) to create a strong learner.

One of the main advantages of AdaBoost is its ability to improve the accuracy of weak learners by assigning more weight to misclassified samples in subsequent iterations. This adaptability makes AdaBoost robust and capable of capturing complex patterns in the data. Additionally, AdaBoost's performance can be affected if weak learners are too complex, potentially causing overfitting.

However, it appears that AdaBoost did not perform as well compared to subsequent ensembles. This suggests that its sensitivity to sparse data might be a contributing factor to its slightly diminished effectiveness.

High Performer: Support Vector Machine (SVM)

SVM performed very well with 80.11% cross-validation accuracy. SVM is a powerful binary classification task, especially suitable for high-dimensional and sparse data. It efficiently finds the optimal hyperplane to separate different classes in these spaces, focusing on relevant support vectors for decision making. SVMs are robust to sparse data, providing a wider margin and improved accuracy, making them ideal for scenarios like one-hot encoded features which we implemented in Section 2.2.5.

A disadvantage of SVM is that it can be computationally intensive for large datasets and is sensitive to noisy data and outliers. This may affect the positioning of the optimal hyperplane and lead to suboptimal results. Hence, proper feature engineering as we have done in Section 2.2 is extremely crucial. The steps conducted during feature engineering have helped to improve our SVM model.

High Performers: Random Forests, Bagging

In general, ensemble methods like Random Forests, Bagging, Gradient Boosted Trees performed better than traditional machine learning models and AdaBoost with high validation accuracy scores of above 80.20%.

AdaBoost can be sensitive to noisy data and outliers, where outliers can receive more weight during the training process, leading to a skewed model. On the other hand, Random Forests and Bagging are less affected by outliers because they take a vote from multiple independent models, reducing the impact of individual misclassified samples.

Ensemble methods are powerful due to their ability to harness the collective power of multiple weak learners (individual models) to create a stronger, accurate predictive model. Furthermore, ensemble methods are robust to outliers and noisy data and are able to improve generalisation.

However, ensembles come with disadvantages as they can be computationally expensive (over 100 seconds of training time in our case), especially when dealing with a large number of models or extensive hyperparameter tuning. They are also often like "black box" models, making it challenging to interpret their decisions compared to simpler models such as logistic regression or support vector machines.

Best Performer: Gradient Boosted Trees (GBT)

Gradient Boosted Trees stand out among machine learning models due to their ability to combine multiple weak learners, typically shallow decision trees, in a sequential manner. During training, GBT corrects errors made by the previous models by calculating residuals and fitting new decision trees specifically to these errors. This iterative process enables GBT to capture complex, nonlinear relationships in the data, making it highly accurate.

GBT's success can be attributed to several factors. Firstly, its ensemble of decision trees focuses on samples that are challenging to classify, enhancing its accuracy. Secondly, GBT reveals feature importance for enhanced interpretability. Notably, GBT excels in capturing nuanced patterns, enduring noisy or less informative features, ensuring robust performance across diverse data types. This collective wisdom from sequentially integrated decision trees forms a potent and reliable model for our binary classification challenge.

These advantages culminate in GBT achieving the highest cross-validation accuracy of 80.54% in the given dataset, outperforming other methods like K-Nearest Neighbours, Logistic Regression, AdaBoost, Support Vector Machine, Bagging, and Random Forests.

While GBT's training time emerges as the lengthiest among the experimented models, the substantial predictive capabilities outweigh this drawback.

Consequently, in the following sections, we choose **Gradient Boosted Tree** as our baseline model to be further enhanced to increase our predictive performance.

3.4. Gradient Boosted Trees (GBT)

3.4.1 Baseline Results from scikit-learn GBT

Validation Dataset Classification Report - GBT

```
Classification Report for Validation Set:
              precision    recall  f1-score   support

           0       0.81      0.79      0.80       839
           1       0.80      0.81      0.81       852

    accuracy                           0.80      1691
   macro avg       0.80      0.80      0.80      1691
weighted avg       0.80      0.80      0.80      1691
```

Figure 7. Classification Report for Validation Set (scikit-learn GBT)

The baseline model exhibits strong performance, achieving an overall accuracy of 80% in Figure 7.

Actual Test Accuracy - GBT

Upon submission of the gradient boosted tree model results for our final testing set on Kaggle, we obtain a high test accuracy result of **80.102%.**

3.5. TensorFlow Decision Forests (TF-DF)

We leverage the TensorFlow framework, a low level ML library to further **improve the baseline result from our best performing model - Gradient Boosted Trees**.

Boosting is an ensemble learning method where the algorithm dynamically adjusts the weights of data instances in each boosting round. Instances that are currently misclassified are given higher weights, increasing their likelihood of being sampled in the next round.

TensorFlow Decision Forests implements boosting with gradient descent, using Gradient Boosted Trees as the base classifier. This library is historically successful in achieving leading results on benchmarks, to improve model accuracy and predictive power. It is powered by the Yggdrasil Decision Forest library.

3.5.1 Training Process with TensorFlow Decision Forests

We conduct an extensive training process comprising 126 iterations. Throughout this training, we monitored the model's performance using metrics such as accuracy and log loss. Our primary focus was on understanding how the number of trees in the ensemble impacts the model's accuracy and generalisation to unseen data.

Accuracy Analysis

The training logs can help us understand the quality of the model, using metrics such as accuracy evaluated on the out-of-bag dataset according to the number of trees in the model.

While the training logs captured information up to 126 iterations, the final model was determined at **iteration 80**. The reported progressive improvement in accuracy reached approximately 89% by iteration 126, demonstrating that the model achieved a satisfactory level of performance, and further iterations did not significantly enhance the accuracy beyond this point.

```
Training logs:
Number of iteration to final model: 100
        Iter:1 train-loss:1.309657 valid-loss:1.313810  train-accuracy:0.772655 valid-accuracy:0.760870
        Iter:2 train-loss:1.240528 valid-loss:1.246706  train-accuracy:0.797529 valid-accuracy:0.797101
        Iter:3 train-loss:1.186829 valid-loss:1.195112  train-accuracy:0.796736 valid-accuracy:0.794203
        Iter:4 train-loss:1.138942 valid-loss:1.149668  train-accuracy:0.796261 valid-accuracy:0.789855
        Iter:5 train-loss:1.097397 valid-loss:1.110631  train-accuracy:0.799905 valid-accuracy:0.791304
        Iter:6 train-loss:1.060898 valid-loss:1.077680  train-accuracy:0.800697 valid-accuracy:0.789855
        Iter:16 train-loss:0.872407 valid-loss:0.914657  train-accuracy:0.805608 valid-accuracy:0.791304
        Iter:26 train-loss:0.785789 valid-loss:0.843266  train-accuracy:0.816382 valid-accuracy:0.804348
        Iter:36 train-loss:0.737475 valid-loss:0.809644  train-accuracy:0.821451 valid-accuracy:0.818841
        Iter:46 train-loss:0.700657 valid-loss:0.787842  train-accuracy:0.828105 valid-accuracy:0.820290
        Iter:56 train-loss:0.676675 valid-loss:0.781992  train-accuracy:0.835710 valid-accuracy:0.826087
        Iter:66 train-loss:0.652581 valid-loss:0.775861  train-accuracy:0.845374 valid-accuracy:0.826087
        Iter:76 train-loss:0.631963 valid-loss:0.771820  train-accuracy:0.856939 valid-accuracy:0.828986
        Iter:86 train-loss:0.614764 valid-loss:0.771511  train-accuracy:0.863276 valid-accuracy:0.824638
        Iter:96 train-loss:0.595337 valid-loss:0.769255  train-accuracy:0.870247 valid-accuracy:0.823188
        Iter:106 train-loss:0.580408 valid-loss:0.772026  train-accuracy:0.876426 valid-accuracy:0.824638
        Iter:116 train-loss:0.567052 valid-loss:0.772747  train-accuracy:0.881971 valid-accuracy:0.818841
        Iter:126 train-loss:0.549859 valid-loss:0.772494  train-accuracy:0.891318 valid-accuracy:0.820290
```

Figure 8. Snippet of Python Training Log for TF-DF Model

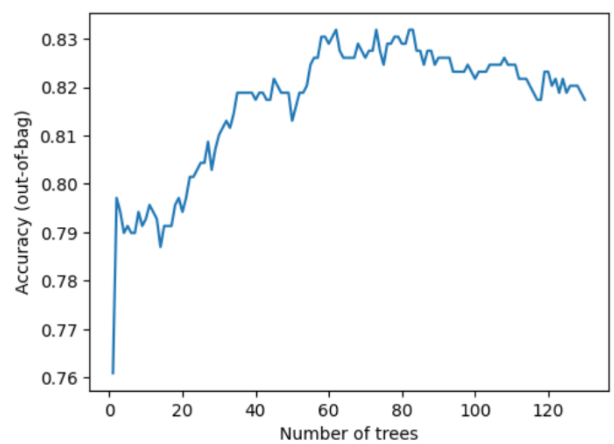

Figure 9. Accuracy Training Log Chart

Log Loss Analysis

Log loss, an important metric measuring alignment between predicted probabilities and actual labels, showed a similar trend.

Initially, log loss decreased steadily, indicating improved predictive accuracy. The model then begins converging almost immediately with 80 trees being the optimal number. Beyond that, log loss difference starts to stagnate, signifying a decline in model performance.

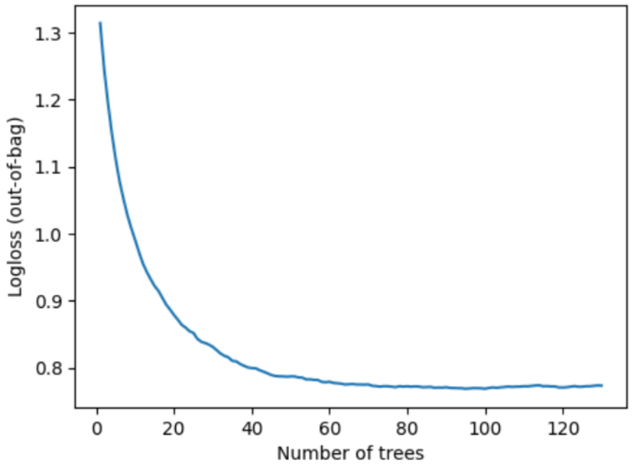

Figure 10. Logloss Training Log Chart

Conclusion

Based on our observations from the training log and visualisations, we deduced the following:

- The optimal number of trees for our ensemble model appears to be around 80, where accuracy peaks and log loss minimisation stagnates.
- Adding too many trees beyond this point may not yield significant improvements; instead, it may lead to overfitting and reduced accuracy on unseen data.

3.5.2 Hyperparameter Optimisation with TensorFlow Decision Forests

Hyperparameter Optimisation: The model underwent hyperparameter optimisation using various settings for parameters such as split axis, sparse oblique projection density factor, categorical algorithm.

Instead of manually selecting a range of hyperparameters, we leveraged the TensorFlow auto tuner which automatically experiment and derive the optimal hyperparameters for the model.

Best Parameters

The best parameters found included:

- split_axis: SPARSE_OBLIQUE
- Sparse_oblique_projection_density_factor: 2
- sparse_oblique_normalization: MIN_MAX
- sparse_oblique_weights: BINARY
- categorical_algorithm: RANDOM
- growing_strategy: BEST_FIRST_GLOBAL
- Max_num_nodes: 16
- sampling_method: RANDOM
- subsample: 1
- Shrinkage: 0.1
- Min_examples:5
- Use_hessian_gain:true
- num_candidate_attributes_ratio:0.5

Tree Structure

The final model consisted of 80 trees, with an average of 30.92 nodes per tree. The depth of the trees varied, with the majority having a depth of 6.

3.5.3 Final Results from TensorFlow Decision Forests

Validation Dataset Classification Report - TF-DF

	precision	recall	f1-score	support
0	0.81	0.80	0.81	849
1	0.80	0.81	0.81	842
accuracy			0.81	1691
macro avg	0.81	0.81	0.81	1691
weighted avg	0.81	0.81	0.81	1691

Figure 11. Classification Report for Validation Set (TF-DF)

We compare the performance of using TensorFlow and scikit-learn frameworks.

With TensorFlow framework, the GBT model achieved an overall accuracy of 81% (Figure 7), which is 1% higher than scikit-learn framework (Figure 11).

The key improvement lies in the increase in recall results of 1% for class 0 cases, while other recall and precision remains approximately the same.

In summary, with TensorFlow Decision Forests, we predict a total of 849 passengers who were not transported to the alternate dimension (class 0) and 842 passengers were transported to the alternate dimension (class 1) in our validation dataset.

Actual Test Dataset Accuracy - TF-DF

Finally, we submit our predictions for the testing set on Kaggle and our test accuracy result increased to **80.71%.** This was an improvement in the final testing accuracy results as compared to our baseline results using scikit-learn gradient boosted trees which was 80.102%.

This confirms we have successfully enhanced the performance of the Gradient Boosted Trees model using TensorFlow.

3.6. Kaggle Leaderboard Results (Top 10%)

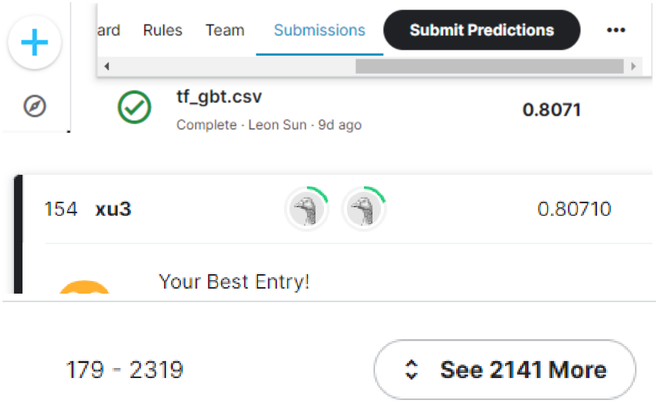

Figure 12. Screenshots from Kaggle leaderboard. Our team stands at 154 out of 2319 teams (top 6.64%).

Our final prediction with TensorFlow Decision Forests scored a high accuracy score of **80.71%** and was ranked **154 out of 2319 participants (in top 6.64% of leaderboard)**.

4. Conclusion (Learnings from Project)

TensorFlow is a Powerful Ensemble tool

In solving our binary classification problem, TensorFlow has proven to be an invaluable asset, showcasing high accuracy and excelling in various aspects. Its strength lies in (1) efficient handling of high-dimensional and sparse data, (2) employing an ensemble learning approach that enhances accuracy and generalisation, (3) demonstrating robustness to noisy data, and (4) effectively handling diverse data types. The ensemble learning strategy, combining predictions from multiple individual models (trees), has significantly improved accuracy and generalisation while mitigating overfitting and capturing intricate relationships within our data.

Balancing Accuracy and Interpretability

We learn that TensorFlow Decision Forests excel in accuracy but the trade-off is often a lack of interpretability compared to simpler models. In our project, prioritising accuracy is paramount, and despite the interpretability challenge, TensorFlow Decision Forests remain the optimal choice for achieving our key goal.

Iterative Model Tuning for Optimization

Hyperparameter tuning stands out as a critical element in optimising model performance. Our meticulous approach, detailed in Section 3.5.1, spanned 126 iterations, showcasing the depth of our exploration. In Section 3.5.2, we leveraged the TensorFlow auto tuner, demonstrating adaptability by utilising automated tools to derive optimal hyperparameters for the GBT model. This involved a comprehensive search across various hyperparameter combinations, leading us to identify the most effective configuration. The iterative nature of our tuning process played a pivotal role in achieving the best possible results, emphasising the significance of adaptability and precision in model optimization.

Feature Engineering and Data Quality

Feature engineering is essential for maximising ensemble method benefits. No "perfect" model exists without understanding the problem and conducting proper feature engineering. Meticulous preprocessing, including handling missing values and scaling features, contributes to overall success.

Embracing Continuous Learning and Exploration

Finally, a fundamental takeaway from our project is the dynamic nature of the machine learning field. We have learned to stay open to incorporating and experimenting with different tools and models available in the market to enhance our approach.

5. Appendix

5.1. A. Training and Testing Dataset Original Features

Table A1. Training and Testing Dataset Features (Before Feature Engineering)

Feature Name	Data Type	Category	Training Dataset	Testing Dataset
PassengerId	Object	Categorical (Ordinal)	✓	✓
HomePlanet	Object	Categorical (Nominal)	✓	✓
CryoSleep	Object	Categorical (Nominal)	✓	✓
Cabin	Object	Categorical (Ordinal)	✓	✓
Destination	Object	Categorical (Nominal)	✓	✓
Age	Float64	Numerical	✓	✓
VIP	Object	Categorical (Nominal)	✓	✓
RoomService	Float64	Numerical	✓	✓
FoodCourt	Float64	Numerical	✓	✓
ShoppingMall	Float64	Numerical	✓	✓
Spa	Float64	Numerical	✓	✓
VRDeck	Float64	Numerical	✓	✓
Name	Object	Categorical (Nominal)	✓	✓
Transported	Bool	Categorical (Nominal)	✓	To be predicted

Figure A1. Summary information of training dataset in Python Jupyter Notebook

```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8693 entries, 0 to 8692
Data columns (total 14 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   PassengerId   8693 non-null   object
 1   HomePlanet    8492 non-null   object
 2   CryoSleep     8476 non-null   object
 3   Cabin         8494 non-null   object
 4   Destination   8511 non-null   object
 5   Age           8514 non-null   float64
 6   VIP           8490 non-null   object
 7   RoomService   8512 non-null   float64
 8   FoodCourt     8510 non-null   float64
 9   ShoppingMall  8485 non-null   float64
 10  Spa           8510 non-null   float64
 11  VRDeck        8505 non-null   float64
 12  Name          8493 non-null   object
 13  Transported   8693 non-null   bool
dtypes: bool(1), float64(6), object(7)
```

```
train.head()
```

	PassengerId	HomePlanet	CryoSleep	Cabin	Destination	Age	VIP	RoomService	FoodCourt	ShoppingMall	Spa	VRDeck	Name	Transported
0	0001_01	Europa	False	B/0/P	TRAPPIST-1e	39.0	False	0.0	0.0	0.0	0.0	0.0	Maham Ofracculy	False
1	0002_01	Earth	False	F/0/S	TRAPPIST-1e	24.0	False	109.0	9.0	25.0	549.0	44.0	Juanna Vines	True
2	0003_01	Europa	False	A/0/S	TRAPPIST-1e	58.0	True	43.0	3576.0	0.0	6715.0	49.0	Altark Susent	False
3	0003_02	Europa	False	A/0/S	TRAPPIST-1e	33.0	False	0.0	1283.0	371.0	3329.0	193.0	Solam Susent	False
4	0004_01	Earth	False	F/1/S	TRAPPIST-1e	16.0	False	303.0	70.0	151.0	565.0	2.0	Willy Santantines	True

Figure A2. Summary information of testing dataset in Python Jupyter Notebook

```
test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4277 entries, 0 to 4276
Data columns (total 13 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   PassengerId   4277 non-null   object
 1   HomePlanet    4190 non-null   object
 2   CryoSleep     4184 non-null   object
 3   Cabin         4177 non-null   object
 4   Destination   4185 non-null   object
 5   Age           4186 non-null   float64
 6   VIP           4184 non-null   object
 7   RoomService   4195 non-null   float64
 8   FoodCourt     4171 non-null   float64
 9   ShoppingMall  4179 non-null   float64
 10  Spa           4176 non-null   float64
 11  VRDeck        4197 non-null   float64
 12  Name          4183 non-null   object
dtypes: float64(6), object(7)
```

```
test.head()
```

	PassengerId	HomePlanet	CryoSleep	Cabin	Destination	Age	VIP	RoomService	FoodCourt	ShoppingMall	Spa	VRDeck	Name
0	0013_01	Earth	True	G/3/S	TRAPPIST-1e	27.0	False	0.0	0.0	0.0	0.0	0.0	Nelly Carsoning
1	0018_01	Earth	False	F/4/S	TRAPPIST-1e	19.0	False	0.0	9.0	0.0	2823.0	0.0	Lerome Peckers
2	0019_01	Europa	True	C/0/S	55 Cancri e	31.0	False	0.0	0.0	0.0	0.0	0.0	Sabih Unhearfus
3	0021_01	Europa	False	C/1/S	TRAPPIST-1e	38.0	False	0.0	6652.0	0.0	181.0	585.0	Meratz Caltilter
4	0023_01	Earth	False	F/5/S	TRAPPIST-1e	20.0	False	10.0	0.0	635.0	0.0	0.0	Brence Harperez

5.2. B. Supplementary Information for Feature Engineering

Figure B1. Missing Value Count and Percentage of Training and Testing Dataset

	columns	train_missing	train_missing_percentage	test_missing	test_missing_percent
0	CryoSleep	217	2.496261	93	2.174
1	ShoppingMall	208	2.392730	98	2.291
2	VIP	203	2.335212	93	2.174
3	HomePlanet	201	2.312205	87	2.034
4	Name	200	2.300702	94	2.197
5	Cabin	199	2.289198	100	2.338
6	VRDeck	188	2.162660	80	1.870
7	FoodCourt	183	2.105142	106	2.478
8	Spa	183	2.105142	101	2.361
9	Destination	182	2.093639	92	2.151
10	RoomService	181	2.082135	82	1.917
11	Age	179	2.059128	91	2.127
12	PassengerId	0	0.000000	0	0.000

Figure B2: Feature engineering of IsAlone using PassengerId

We create a new column IsAlone with reference to PassengerNo Column.

```
train['isAlone'] = train.groupby('GroupNo')['PassengerNo'].transform('max').astype(int) < 2
```

```
train['isAlone'] = train['isAlone'].astype(int)
```

```
train.head(5)
```

	PassengerId	HomePlanet	CryoSleep	Cabin	Destination	Age	VIP	RoomService	FoodCourt	ShoppingMall	...	CabinDeck	CabinNum	CabinSide	FirstNan
0	0001_01	Europa	0.0	B/0/P	TRAPPIST-1e	39.0	0.0	0.0	0.0	0.0	...	B	0.0	P	Maha
1	0002_01	Earth	0.0	F/0/S	TRAPPIST-1e	24.0	0.0	109.0	9.0	25.0	...	F	0.0	S	Juan
2	0003_01	Europa	0.0	A/0/S	TRAPPIST-1e	58.0	1.0	43.0	3576.0	0.0	...	A	0.0	S	Alta
3	0003_02	Europa	0.0	A/0/S	TRAPPIST-1e	33.0	0.0	0.0	1283.0	371.0	...	A	0.0	S	Sola
4	0004_01	Earth	0.0	F/1/S	TRAPPIST-1e	16.0	0.0	303.0	70.0	151.0	...	F	1.0	S	Wi

Figure B3. Snippet of Python Script to check if one-hot encoding data is sparse

```
# Calculate density
total_elements = one_hot_matrix.size
non_zero_elements = np.count_nonzero(one_hot_matrix)
density = (non_zero_elements / total_elements) * 100
sparsity_threshold = 50

if density < sparsity_threshold:
    print("The one-hot encoding is sparse.")
else:
    print("The one-hot encoding is not sparse.")
```

```
The one-hot encoding is sparse.
```