Flight Delay Predictor

Machine Learning Project

Milestone 4

Mariam ElGhobary
CSCE Department
The American University In Cairo
m.elghobary@aucegypt.edu
SID: 900211608

Youssef Elhagg
CSCE Department
The American University In Cairo
yousseframi@aucegypt.edu
SID: 900211812

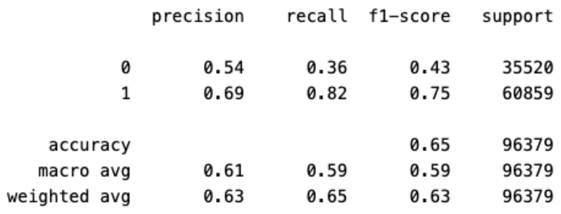
I. INTRODUCTION

In our previous milestone we trained our dataset using various supervised learning models and analyzed the performance, pros and cons of each followed in order to help us determine which model will be chosen for our application. Building onto that, this milestone is split into two main parts, the first of which is dedicated to our final model choice with its justifications based on our pilot study as well as explaining elements of our model design. The second part of this milestone is focused on our utility application structure including: application architecture, human interaction flow and data flow.

II. MODEL CHOICE AND DESIGN

A. Chosen Model

After careful consideration and analysis of various machine learning models for our project, we have decided to opt for **Random Forests**. Our pilot study result run on our binary label (<= 30 minutes is class 0 and >30 is class 1) using default parameters with only class weight adjustment boosting weights of the minority class (0) with 5-fold cross validation is shown below:

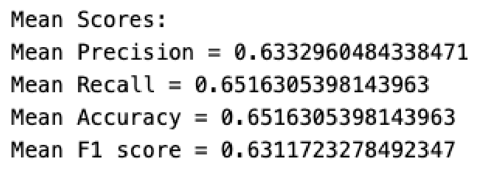


```
              precision    recall  f1-score   support

           0       0.54      0.36      0.43     35520
           1       0.69      0.82      0.75     60859

    accuracy                           0.65     96379
   macro avg       0.61      0.59      0.59     96379
weighted avg       0.63      0.65      0.63     96379
```

Our decision was influenced by several factors that make Random Forests particularly suitable for our dataset, which is imbalanced and has a high dimensionality of just over 4,000 columns. Those reasons include:

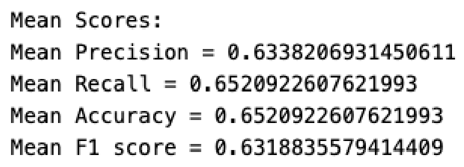
1. **Ensemble Learning**: Since Random Forests is an ensemble classifier it essentially combines predictions of multiple decision trees hence model performance is enhanced. Such an approach often results in more robust predictions when dealing with complex datasets with numerous features like ours.
2. **Feature Importance and Selection**: Random Forests can also measure feature importance which can help identify the most relevant columns influencing the label. This could give us a boost when it comes to our dataset's high dimensionality. In addition, Random Forests can automatically handle feature selection and only consider subsets of features per decision tree in the ensemble, thereby focusing on the most informative ones. This feature subsampling may also reduce overfitting.
3. **Robustness to Overfitting**: Compared to single decision trees, Random Forests are

less prone to overfitting, especially when dealing with high-dimensional data. Each tree in the forest is trained on a random subset of the data and a random subset of the features, which helps reduce overfitting and improve the model's generalization performance. For instance, when we trained our model using 5-fold cross validation, using default parameters and adding class weights to handle the class imbalance (class zero nearly half the points of class 1), we got the following scores:



```
Mean Scores:
Mean Precision = 0.6332960484338471
Mean Recall = 0.6516305398143963
Mean Accuracy = 0.6516305398143963
Mean F1 score = 0.6311723278492347
```

Weighted Average scores for the 5 folds



```
Mean Scores:
Mean Precision = 0.6338206931450611
Mean Recall = 0.6520922607621993
Mean Accuracy = 0.6520922607621993
Mean F1 score = 0.6318835579414409
```

Weighted Average scores for the test set

We notice that the scores are almost identical, which suggests that the model did not overfit.

4. **Handling Imbalanced Data**: As our data is imbalanced, Random Forests can effectively handle imbalanced datasets, especially when class weights are assigned. By adjusting the class weights, the algorithm can give more importance to minority class samples during training, thus mitigating the impact of class imbalance on the model's performance.

5. **Non-linearity**: Random Forests are capable of capturing complex nonlinear relationships between features and the label which is the case for our data hence giving it a cutting edge over other models.

Hence, considering the above factors, we believe that Random Forests would be the most suitable choice for our project.

B. Model Design

Given the high dimensionality and class imbalance within our data, we need to implement a model that manages to maintain robustness whilst tackling overfitting. This balance requires very careful selection of parameters and overall approaches within our implementation. Parameters and points of focus include:

- min_samples_split:

  The min_samples_split hyperparameter in a Random Forest classifier determines the minimum number of samples required to split an internal node.

  In the context of our dataset, min_samples_split is important for:

  - Preventing overfitting by making the model more general.
  - Reducing computational complexity by limiting the number of splits.
  - Handling high dimensionality by ensuring meaningful splits in sparse data.
  - Improving model interpretability by simplifying the decision tree structure.

  We will use **GridSearchCV** to determine the best value for this parameter.

- max_features:

  The max_features hyperparameter determines the number of features to consider when looking for the best split during the tree building process.

  In the context of our dataset, max_features is important for:

  - Reducing overfitting by limiting the number of features considered at each split.
  - Improving computational efficiency by reducing the computational cost of finding the best feature and split point at each node.

    - Adding randomness to the model by considering different subsets of features at each node, making the model more robust.
    - Dealing with high dimensionality (like our case) by helping the model focus on the most important features.

- n_estimators:

  The n_estimators parameter in the Random Forest algorithm from Sklearn refers to the **number of trees** in the forest. Each tree in the forest is constructed independently, and the final prediction is made by averaging the predictions of each individual tree (for regression) or by majority voting (for classification).

  In the context of our dataset, n_estimators is important for:

    - **Model Performance**: More trees can lead to better model performance as it reduces overfitting.
    - **Computational Complexity**: However, more trees also mean more computational complexity, which can significantly increase training time, especially for large datasets.
    - **Diminishing Returns**: Beyond a certain point, adding more trees doesn't significantly improve performance but continues to use computational resources.

- criterion:

  This parameter is used to measure the quality of each split. We opted for the Gini index. One of the reasons we chose Gini is because Gini gives higher scores to nodes that are more impure and then the split with the lowest Gini is selected. In other words, priority is given to nodes that are more pure in the sense that data points within that nodes are highly homogeneous with respect to the label.

  Gini index often yields trees with shorter depths and simpler trees relative to other criterions such as entropy. This can be an effective form of tackling overfitting.

- Pruning: Pruning is a key perk of using CART and Random Forest and so implementing it as part of our model is essential especially to handle overfitting and generalize our model. Pruning can take one of two forms. The first form is pre-pruning which is essentially setting a maximum depth which each tree cannot grow beyond to begin with. The second type is post-pruning where we allow the tree to grow to its maximum size and then replace nodes bottom up with their most dominant value and recompute scores and repeat this till scores stop improving. We decided to implement the pre pruning method for the following reasons:

    - Effective in tackling overfitting and creating much simpler trees than post-pruning
    - Much more computationally efficient given our very high dimensionality.

  However, for pre-pruning to be effective, the maximum depth value must be chosen very carefully to not lead to underfitting.

  We will use **GridSearchCV** to determine the best value for this parameter.

- Class and Weight imbalance:

  As mentioned previously, our no delay class (class 0) has significantly less data points than our delayed class (class 1) and has relatively lower scores. This could be tackled within our implementation using several methods including;

    - **Adjusting Class Weights:**

      This is whereby we adjust our impurity calculation (gini in our case) to include class weights where higher priority is given to the minority class by penalizing minority class misclassification more heavily. Weight combination can be selected by testing several options using **GridSearchCV**

    - **Weighted Sampling:**

      Data point sampling could be adjusted so that it samples more of the underrepresented class.

III. UTILITY APPLICATION

A. Motivation

The main purpose of our application is to provide a user-friendly service that not only predicts whether or not flights will experience departure delay, but also integrates and builds upon user feedback to improve its predictions. This gives users a head-up when needed so that they can account for such delays in layover flight reservations and even be on the look-out for potentially feasible compensations.

B. Architecture

The utility application's architecture is by far one of its fundamental cornerstones. Hence, after careful consideration, we decided that we will implement the client server architecture with the client's side being implemented as a web-based application.

Some of the primary advantages of choosing the client-server model are:

- Scalable and secure
- Centralization of the fundamental logic of the application in the server ensures consistency when it comes to prediction for all clients.
- Separation of concerns also enables eased model debugging , modification and maintenance.
- Centralizing historical data storage also eases data management, consistency and integrity.
- Compatibility with various platforms is enabled as our web-application becomes accessible from various devices and platforms without any restrictions to abide by certain platforms which increases usability.

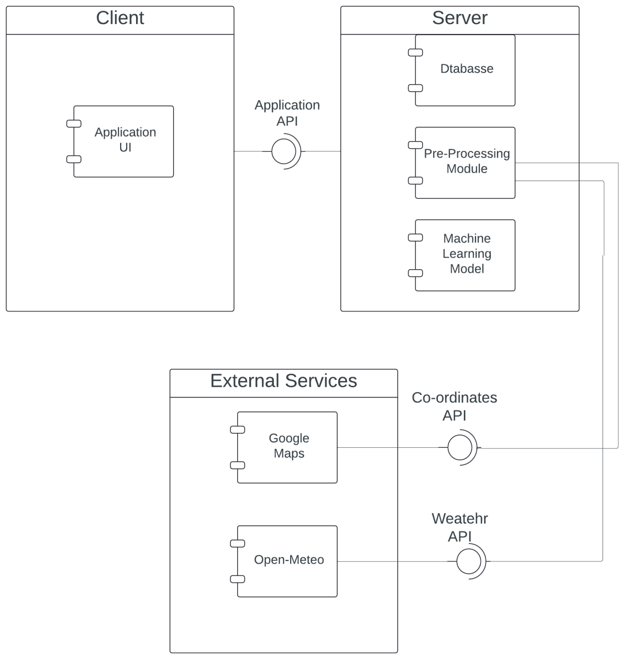
The first component of the application is the client application containing the application's UI with all of its visual and interactive elements such as pages, buttons and so on. Here data presentation and visual formatting are handled and the input fields are provided for the user to input their flight details. So its what essentially offers the user-friendly interface that users interact with.

The client application also communicates with the server to either send the flight details to the server for prediction generation/ feedback or receive predictions both done via an API.

The server side, on the other hand, conducts required preprocessing and hosts our chosen machine learning model. The server also contains and handles the storage and management of our training points database as well as incorporating new data points as user feedback is utilized for the model retraining which it also runs (more on this to be explained later).

The external services of Google Maps and Open Meteo APIs respectively give us the airport coordinates and weather information at departure and arrival locations for the entered data point. Both of these sets of data obtained are part of our features required for prediction and so can be incorporated amongst other forms of preprocessing before the model is run.

Our component diagram is shown below:

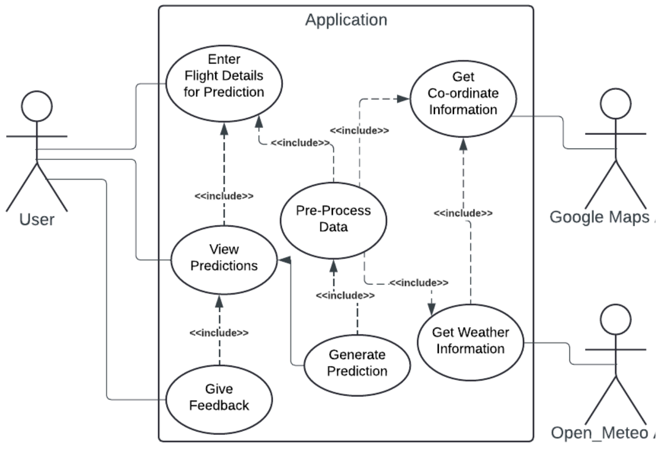


C. User and External Systems Interaction

We will now assess the forms of user interactions with the system, as well as the system's use cases and any other external systems involved. As shown in the use case diagram below, we have 3 main actors interacting with our system. The first being the user, and the other 2 are our coordinates and weather APIs.

The user can perform 3 main actions. The first action is entering flight details to predict its delay. The entered data undergoes necessary preprocessing by carrying out necessary one-hot-encoding and normalization. As explained in previous milestones, not only do we clean, normalize and one hot encode our raw data points, but we also obtain extra weather information relating to both the flight source and destination as part of our preprocessing procedure. This is enabled via external APIs. The Google Maps API returns the coordinates of the source and destination airports. These coordinates are then passed to the Open-Meteo API to return the sought weather information. Now that preprocessing is complete our machine learning model can generate the new data point's prediction and once it's generated it is displayed to be viewed by the user.The final action done by the user is providing feedback on some previously made predictions to enable model improvement.

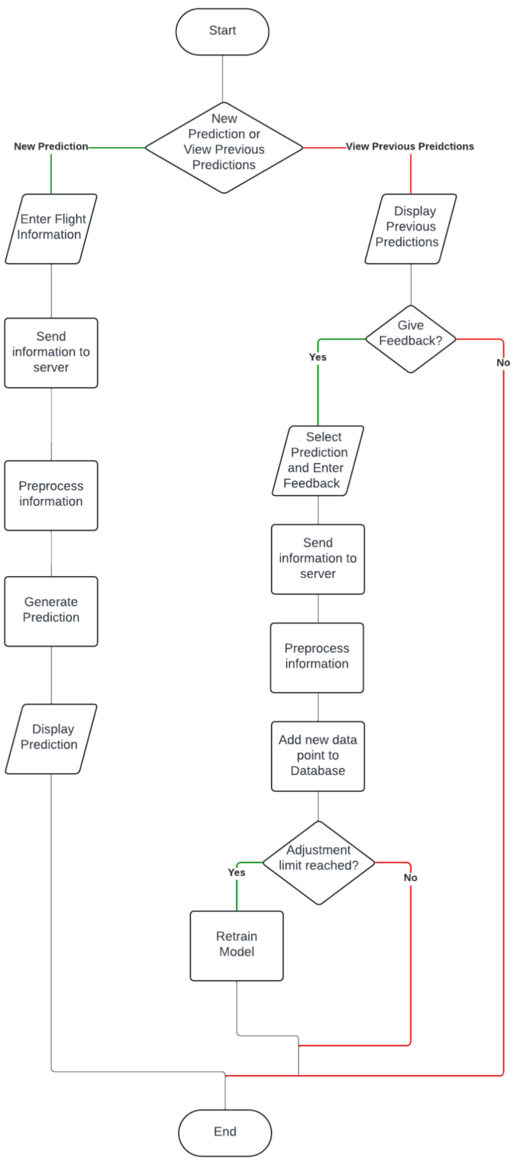
All of the procedures, user interactions and system interactions mentioned above are demonstrated in the use case diagram below:



D. DataFlow

After the application is started, the user gets to decide whether they would like to predict departure delay for a flight (new predictions) or view their previous predictions. If they choose to make a prediction, they enter the flight information which is then sent to the server where preprocessing and later prediction generation via our machine learning model take place. The prediction is then displayed to the user. If , on the other hand, they opt to view their previous predictions, they are taken to a page where they are displayed and are given an option to provide feedback. If they choose to provide feedback, they can select which one to provide feedback for. They then enter the correct level of delay and only then would this data point be added to our database. This is because adding all data points post model-prediction immediately without confirmation of the correctness of the prediction form the user would insert data points with incorrect labels which would be nothing short of catastrophic for our model. After adding the new data point to the database the system checks how many insertions to the database have been made since the model was last trained. If the count exceeds our selected limit (yet to be determined) then the model is retrained. If the limit was not reached yet the counter is simply incremented.

The breakdown of the data flow as explained above is illustrated by the flowchart below:


Start
New Prediction or View Previous Predictions
New Prediction
View Previous Preidctions
Enter Flight Information
Send information to server
Preprocess information
Generate Prediction
Display Prediction
Display Previous Predictions
Give Feedback?
Yes
No
Select Prediction and Enter Feedback
Send information to server
Preprocess information
Add new data point to Database
Adjustment limit reached?
Yes
No
Retrain Model
End