I started by loading the training and test datasets from the folder.

I found missing values in *Arrival Delay in Minutes*, and replaced them with the **median** value to avoid distortion from outliers.
- Why use median value?
- I used the **median** to fill missing values because it is **robust to outliers**.
- Unlike the mean, the median is not affected by extremely large or small delay values

---

Feature Engineering

I created three additional features to capture passenger experience more effectively.
After exploring the dataset, I realized that **delay** and **service quality** were two major factors affecting passenger satisfaction.
So, I decided to engineer three new features to capture these patterns more clearly.

Total_Delay, which just sums the departure and arrival delays. I did this because passengers usually care about the **total time lost**, not necessarily whether it was at departure or arrival.

Avg_Service_Score, which averages all twelve service-related ratings. Instead of using each rating separately, this gives us a single, clear measure of overall service experience.

On_Time, a simple binary flag that shows whether the flight had **zero delay**.

Overall, they make it easier for the models to predict passenger satisfaction more accurately.

Categorical Encoding

There are four main categorical features: **Gender, Customer Type, Type of Travel,** and **Class**.
I applied **one-hot encoding** to convert these categories into numeric columns.
Each category gets its own binary column, which allows the model to use this information effectively without assuming any inherent order.

I combined the training and test sets first, so that the encoding is consistent across both sets.
Then, I applied get_dummies and split them back into the processed training and test sets.
I restored the original **IDs** for both sets, so we can always trace predictions back to the original data.

One-Hot Encoding

One-hot encoding is a method to convert categorical variables into numerical form.
For each category in a feature, it creates a new binary column: 1 if the sample belongs to that category, 0 otherwise.

pd.get_dummies

get_dummies is a convenient Pandas function to automatically apply one-hot encoding to multiple categorical columns at once.

drop_first=True

drop the first category so that our features are not redundant.

Encode Target Variable

The target variable, satisfaction, is categorical with labels like “satisfied” and “neutral or dissatisfied”.
Models need numerical values, so I used LabelEncoder to convert these categories into integers.
After encoding, “satisfied” might become 1 and “neutral or dissatisfied” becomes 0, for example.

Scale Continuous Features

The dataset has continuous numeric features like Age, Flight Distance, and the delay times.
These features have different scales
To make them comparable and help models converge faster, I used StandardScaler to normalize them.
This transforms each feature to have a mean of 0 and a standard deviation of 1.

Split datasets

I split the dataset into three parts: training, validation, and hold-out test sets.

First, I separated 20% of the data as a hold-out test set. This set is never used during model training, so it gives an unbiased estimate of how the final model will perform on unseen data.

Then, I split the remaining 80% into 60% for training and 20% for validation. The validation set is used to tune hyperparameters and compare different models before final evaluation.

XGBoost: scale_pos_weight

the number of negative samples divided by the number of positive samples.

By setting this, XGBoost gives more weight to the minority class, helping the model learn better when the classes are imbalanced.

Models

For each model, I used **GridSearchCV** to search over key hyperparameters and find the best combination. I set **scoring='f1_weighted'** because the dataset is imbalanced, so F1-score accounts for both precision and recall across classes.

I set random_state=42 to make our results reproducible.

cv=3, # sets 3-fold cross-validation to evaluate reliably.
n_jobs=-1, # uses all CPU cores to speed up training.
verbose=0 # keeps the output clean.

For this relatively small dataset, I selected these ranges through experiment. I tried larger values, but found that did not improve performance, sometimes even causing overfitting, while also increasing training time. This ranges balance between model stability, accuracy, and efficiency, and are also commonly used in practice.

1. Random Forest

Random Forest is an ensemble method that builds multiple decision trees and combines their predictions, usually by majority vote. It reduces overfitting compared to a single tree.

It handles high-dimensional tabular data well, is robust to outliers.
captures complex feature interactions, less overfitting than a single tree

'n_estimators': [100, 200], # Number of trees. More trees → more stability, higher accuracy, but slower.
'max_depth': [10, 20, None], # Max depth of each tree. Prevents overfitting; shallow trees may underfit, deep trees may overfit.
'min_samples_split': [2, 5], # Minimum samples to split an internal node. Larger → more

conservative splits.
'min_samples_leaf': [1, 2], # Minimum samples in a leaf. Prevents leaves with very few samples → reduces overfitting.
'max_features': ['sqrt', 'log2'] # Number of features to consider at each split. ‘sqrt’ and ‘log2’ increase randomness, reduce correlation between trees.

LogisticRegression

A linear model for classification that predicts the probability of a class using a logistic function.

Simple, interpretable, works well when features have linear relationships with the target.

'C': [0.01, 0.1, 1, 10, 100], # Inverse of regularization strength. Smaller → stronger regularization, prevents overfitting.
'penalty': ['l2'], # Regularization type. ‘l2’ → penalizes large coefficients.
'solver': ['lbfgs'] # Optimization algorithm. ‘lbfgs’ → efficient for small/medium datasets.

XGBoost

An optimized gradient boosting algorithm that builds trees sequentially, correcting previous errors.
it can handle missing values, and robust, high accuracy, supports regularization.

'n_estimators': [200, 300], # Number of trees. More trees → more stability, higher accuracy, but slower.
'max_depth': [4, 6, 8], # Max depth of each tree. Prevents overfitting; shallow trees may underfit, deep trees may overfit.
'learning_rate': [0.05, 0.1], # Step size shrinkage. Smaller → slower learning, more accurate.
'subsample': [0.8, 1.0], # Fraction of samples per tree. Prevents overfitting.
'colsample_bytree': [0.8, 1.0], # Fraction of features per tree. Reduces correlation.
'reg_lambda': [1, 10], # L2 regularization. Prevents overfitting.
'reg_alpha': [0, 1] # L1 regularization. Sparsity, feature selection.

DecisionTree

A model that splits data recursively based on feature thresholds to classify samples.

Simple, good baseline but prone to overfitting

'max_depth': [5, 10, None], # Max depth of each tree. Prevents overfitting; shallow trees may underfit, deep trees may overfit.
'min_samples_split': [2, 5, 10], # Minimum samples to split an internal node. Larger → more conservative splits.
'criterion': ['gini', 'entropy'] # Split quality metric. ‘gini’ or ‘entropy’.

KNN

A non-parametric method that classifies samples based on the majority class of their k nearest neighbors.

'n_neighbors': [3, 5, 7, 9], # Number of nearest neighbors.
'weights': ['uniform', 'distance'], # ‘uniform’ → all neighbors equal, ‘distance’ → closer neighbors weigh more.
'metric': ['euclidean', 'manhattan'] # Distance metric. Euclidean / Manhattan.

LinearSVM

A linear classifier that finds the hyperplane maximizing the margin between classes.

'C': [0.01, 0.1, 1, 10, 100], # Inverse of regularization. Smaller → stronger regularization.
'penalty': ['l2'] # Type of regularization

MLP

A feedforward neural network with one or more hidden layers, capable of learning non-linear mappings.

Can capture complex non-linear relationships in data

'hidden_layer_sizes': [(64,), (64, 32), (100,)], # Number of neurons per layer. More layers → more complex model.
'alpha': [0.0001, 0.001, 0.01], # L2 regularization term. Prevents overfitting.
'learning_rate': ['constant', 'adaptive'] # ‘constant’ → fixed, ‘adaptive’ → changes if loss stagnates.

Step 9: Train and Evaluate Models on Validation Set

For each model, I recorded both the **training time** and **prediction time**, and then evaluated its performance on the validation set.

The metrics I used include **accuracy, precision, recall, weighted F1, and macro F1** — these give a comprehensive view of how well the model performs, especially under class imbalance.

I stored all results in a list called validation_results,

Accuracy	**Ratio of correct predictions.**	**正确预测的比例。**
Precision (weighted)	Measures how many predicted positives are correct, weighted by class frequency.	衡量预测为正样本的准确性，对各类别加权。
Recall (weighted)	Measures how many actual positives were correctly identified.	衡量正样本被识别出来的比例，对各类别加权。
F1 (weighted)	Harmonic mean of precision and recall, weighted by class size.	精确率与召回率的加权调和平均。
F1 (macro)	Average F1 across all classes equally (good for imbalance).	所有类别的 F1 平均值，不考虑类别比例。

时间记录部分：
We use time.time() to measure how long the model takes to train and predict. This helps us compare computational efficiency between models.

Step 10: Select Best Model on Validation Set

In this step, I compared all models based on their **weighted F1-scores** on the validation set.

I selected the one with the highest F1 score as the **best model** — because F1 balances **precision and recall**, which is more reliable than just accuracy when the dataset is imbalanced.

Then, I saved the validation metrics to a CSV file for record keeping.

Step 11: Evaluate Every Model on Hold-out Test Set

After selecting the best model from validation,
I evaluated **all models again** on the hold-out test set.
This ensures the selected model doesn't just perform well on validation by chance.
I computed the same metrics — accuracy, precision, recall, weighted and macro F1 and compared them again in a DataFrame to confirm the final winner.

Step 11.5: Generate Confusion Matrix Visualization

I generated and visualized the confusion matrix for my best-performing model.
The confusion matrix helps me evaluate how well the model performs on each class — for example, how many samples were correctly or incorrectly classified.
I used *Seaborn* and *Matplotlib* to create a heatmap, where the rows represent the true labels and the columns represent the predicted labels.

Finally, I retrained the selected model on the full training dataset. This step ensures the model learns from all available data before being used for final deployment.

Step 12: Predict on Kaggle Test Set & Prepare Submission

I use the final model to predict the labels for the test set.
I first drop the ‘id’ column because it is not a feature.
Then I convert the numeric predictions back to the original class labels using the label encoder to make sure the submission format is correct.