Team: Birgerkings
Yukta Rajapur - IMT2021066
Brij Desai - IMT2021067
Varshith Vattikuti - IMT2021078

# ML Assignment: Part 2

### Ticket Cancellation Dataset (Classification)

**Following Part 1 of this assignment, we have used the same pre-processed data.

## Overview

1. Non-Neural Ensembles:
    a. Preprocessing Data
    b. Exploratory Data Analysis (EDA)
    c. Feature Engineering
    d. Data Encoding
    e. Balancing the dataset
    f. Training Models
    g. Final Results and Accuracy
2. SVM:
    a. Encoding and Standardisation
    b. Balancing the dataset: Sampling
    c. Testing different Kernels
    d. Final Results
3. Neural Networks
    a. Encoding and Standardization
    b. Balancing the dataset
    c. Creating Tensors and Dataloaders
    d. Defining the model
    e. Training the model
    f. Final Results

# 1. Ensemble: Preprocessing Data

We applied a few techniques to clean the data before moving further. We observed the original dataset has **70711 rows × 22 columns.**

```
train_df
```

| | ID | TimeOfCreation | TimeOfDeparture | BillNo. | TicketNo. | StatusofReserve | UserID | Gender-Male | Pr |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 100505 | 2022-07-06 06:02:12.407 | 2022-07-06 10:30:00 | 38131030 | 7359427.0 | 3 | NaN | True | 85000 |
| 1 | 100506 | 2022-09-11 13:51:08.797 | 2022-09-13 14:45:00 | 39115817 | 3002688.0 | 2 | 891421.0 | True | 533800 |
| 2 | 100507 | 2022-08-01 14:45:28.883 | 2022-08-24 20:39:00 | 38510118 | 2927990.0 | 4 | NaN | False | 135500 |
| 3 | 100508 | 2022-09-29 10:41:28.120 | 2022-09-29 20:30:00 | 39403118 | 7663791.0 | 3 | 264716.0 | True | 254000 |
| 4 | 100509 | 2022-10-03 16:43:35.277 | 2022-10-04 12:15:00 | 39470084 | 7681449.0 | 3 | 76842.0 | True | 169000 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | |
| 70706 | 171211 | 2022-04-07 20:00:01.463 | 2022-04-08 23:00:00 | 36839872 | 7018030.0 | 5 | NaN | True | 125000 |
| 70707 | 171212 | 2022-06-09 07:48:10.583 | 2022-06-09 16:50:00 | 37704940 | 2825554.0 | 2 | NaN | True | 349000 |
| 70708 | 171213 | 2022-08-13 07:19:38.040 | 2022-08-14 23:15:00 | 38660767 | 7510813.0 | 3 | NaN | True | 172500 |
| 70709 | 171214 | 2022-05-02 12:38:36.460 | 2022-05-03 19:00:00 | 37152781 | 7096569.0 | 3 | NaN | False | 320000 |
| 70710 | 171215 | 2022-09-24 21:48:51.137 | 2022-10-11 22:10:00 | 39326282 | 3028631.0 | 2 | 795382.0 | False | 42650 |

70711 rows × 22 columns

## Removing Duplicates

We dropped any duplicate values in place. We normally drop duplicates from the dataset to avoid unnecessary biases in the ML Model. These might occur if the data points are repeated unnecessarily during data collection, etc.

After this, we had **70711 rows × 22 columns. Hence there were no duplicates.**

[36]:
```
train_df.drop_duplicates(inplace=True)
```

```
train_df.shape
```

[37…
```
(70711, 22)
```

## Dealing with null values

We next searched for the presence of null values and based on that dropped rows and columns accordingly.

First, we checked the **null value percentage across different columns** as those with a higher percentage of null values are less likely to contribute to the learning algorithm.

```
null_value_percentages=(train_df.isna().sum()/train_df.shape[0])*100
null_value_percentages.sort_values(ascending=False)
```

```
PassportNumberHashed    99.130263
UserID                  58.026333
EmailHashed             57.446508
VehicleClass            38.020959
TypeOfVehicle            7.479741
ID                       0.000000
DomesticFlight           0.000000
NationalCode             0.000000
BuyerMobile              0.000000
ModeOfTravel             0.000000
ReasonForTrip            0.000000
CityTo                   0.000000
TimeOfCreation           0.000000
CityFrom                 0.000000
Discounts                0.000000
Price                    0.000000
Gender-Male              0.000000
StatusofReserve          0.000000
TicketNo.                0.000000
BillNo.                  0.000000
TimeOfDeparture          0.000000
Cancelled                0.000000
dtype: float64
```

We dropped columns with **> 30% null values**.

```
columns_to_drop = ['UserID', 'PassportNumberHashed', 'EmailHashed', 'VehicleClass']
```

## Unique values in each column

We analyzed the number of unique values in each column to get an idea of which columns could possibly contribute more to the overall model predictions.

**Hypothesis 1:** Columns that are more unique contribute less to overall predictions and model training.

***Result:*** *We trained our model based on this assumption and dropped columns with a high number of unique values but got an*

```
train_df.nunique().sort_values(ascending=False)
```

```
ID                      70711
TicketNo.               70668
BillNo.                 54448
TimeOfCreation          54448
NationalCode            51001
BuyerMobile             35773
TimeOfDeparture         26891
EmailHashed             13826
UserID                  12773
Price                    3735
TypeOfVehicle            2844
Discounts                1624
PassportNumberHashed      543
CityTo                    287
CityFrom                  219
ModeOfTravel                4
StatusofReserve             4
VehicleClass                2
ReasonForTrip               2
DomesticFlight              2
Gender-Male                 2
Cancelled                   2
dtype: int64
```

*accuracy of only 0.93. We had to retrace back and discard this assumption.*

**Hypothesis 2:** There may be some hidden correlation so treat all features equally.

***Result:*** *We didn't drop any columns and in fact got a higher score due to some correlation with features like 'TicketNo.'*

## 2. Ensemble: Exploratory Data Analysis (EDA)

### Percentage Cancellations:

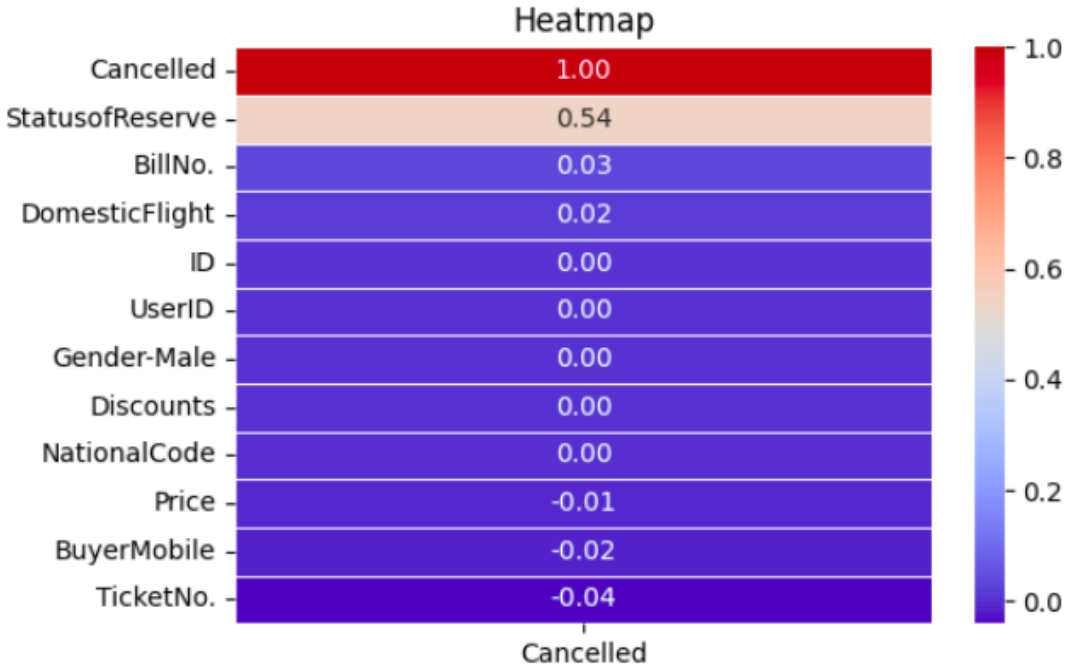
```
Percentage cancelation=  0.15204140798461344
StatusofReserve
3    42.095289
2    30.911739
5    19.883752
4     7.109219
Name: proportion, dtype: float64
```

Out of all the samples, 15% of them were cancellations.

### Analyzing correlations

**Hypothesis 1:** The 'Status of reserve' feature looked very closely related to the cancellation.

We used a heatmap to analyze the correlation between various features with respect to the cancellation.




Clearly, our assumption was right, and 'StatusofReserve' had the most correlation. We decided to drop those features with a 0.00 correlation factor.

Since the city names were strings they are not shown in this map so we made a few assumptions and tested accordingly.

**Assumption 1:** Cities of arrival and departure ['CityTo', 'CityFrom'] had some correlation.

**Result:** We performed one-hot-encoding on the cities and got an **accuracy of 0.937.**

**Assumption 2:** Cities of arrival and departure ['CityTo', 'CityFrom'] had less or no correlation.

**Result:** We dropped them along with other less correlated columns and observed **0.94 accuracy.**

We dropped the following features based on inference from the heatmap.

### Identifying columns with low correlation

```
columns_to_drop.extend(['ID', 'NationalCode', 'EmailHashed', 'UserID', 'Gender-Male', 'Discounts', 'TypeOfVehicle', 'CityTo', 'CityFrom'])
train_df = train_df.drop(columns_to_drop, axis=1)
train_df.shape
```

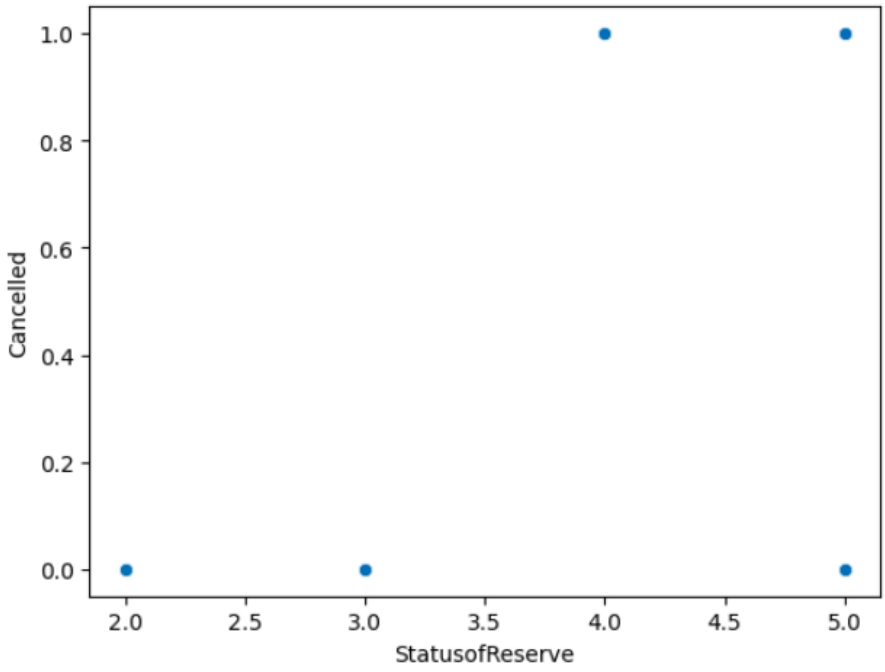
## Status of Reserve

We performed more analysis on this feature as it was highly correlated.

**Scatterplot against 'Cancelled'**

```
sns.scatterplot(x=train_df["StatusofReserve"], y=train_df["Cancelled"])
```

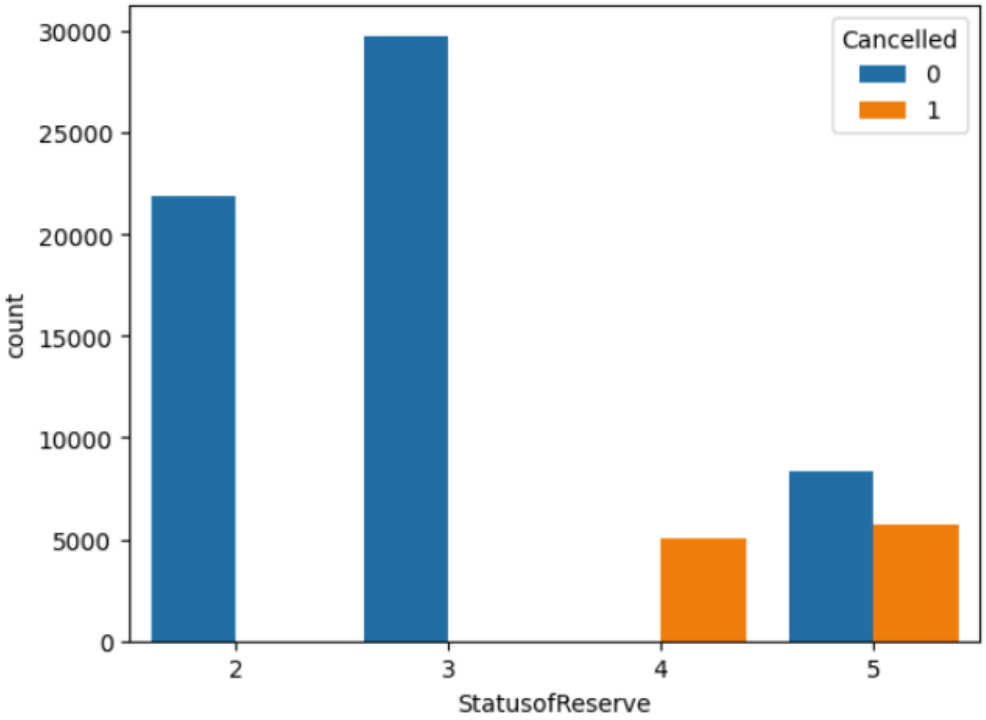
```
<Axes: xlabel='StatusofReserve', ylabel='Cancelled'>
```




**Countplot**

```
sns.countplot(data=train_df, x='StatusofReserve', hue='Cancelled')
```

```
<Axes: xlabel='StatusofReserve', ylabel='count'>
```




From this, we inferred the following:

**Status 2: Not cancelled**

**Status 3: Not cancelled**

**Status 4: Cancelled**

**Status 5: Equally likely**

Since this feature alone isn't enough we decided to do some feature engineering and extract more meaningful interpretations out of the given features.

## 3. Ensemble: Feature Engineering

Feature engineering is the process of transforming raw data into features that are suitable for machine learning models. It is the process of selecting, extracting, and transforming the most relevant features from the available data to build more accurate and efficient machine-learning models.

## Timestamp difference

**Idea:** There could be some correlation based on the difference in the time of booking the ticket and the time of departure. It could be either way, for eg if you book a ticket last minute you're highly unlikely to cancel it whereas if you book well in advance there are more chances of cancellation.

We introduced a new column **'timestamp_diff_seconds'** to capture this. We did this by first converting 'TimeOfCreation', 'TimeOfDeparture' **timestamp to datetime** format and subtracted it to find the difference in seconds.

Converting timestamp to datetime

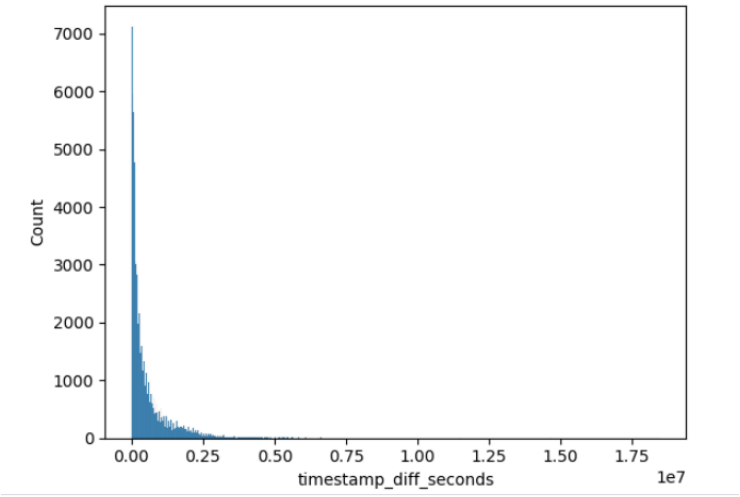
```
train_df['TimeOfCreation'] = (pd.to_datetime(train_df['TimeOfCreation']) - pd.Timestamp("1970-01-01")) // pd.Timedelta('1s')
train_df['TimeOfDeparture'] = (pd.to_datetime(train_df['TimeOfDeparture']) - pd.Timestamp("1970-01-01")) // pd.Timedelta('1s')
```

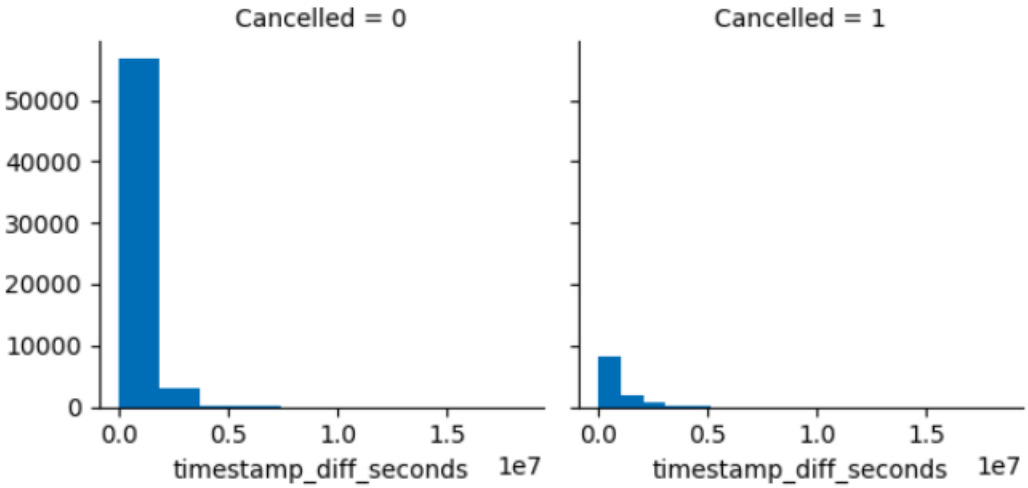
Adding a timestamp difference column

```
train_df['timestamp_diff_seconds'] = (train_df['TimeOfDeparture'] - train_df['TimeOfCreation'])
```

**In order to analyze any potential correlation, we performed some EDA on this new feature**

Histogram plot






The above graphs show that for low values of **'timestamp_diff_seconds',** the number of **'Cancelled' = 0** is significantly higher than the number of **'Cancelled' = 1.** However, as **'timestamp_diff_seconds'** increases, the difference in the number goes down. This confirms our hypothesis that people are more likely to cancel if they have made the booking well in advance. Keeping in mind that the percentage of cancellation in the entire dataset is only 15% the above graphs seem to indicate that **'timestamp_diff_seconds'** has a positive correlation with **'Cancelled'.**

## 4. Ensemble: Data Encoding

Since some features like ['ReasonForTrip', 'ModeOfTravel', 'VehicleClass', 'CityTo', 'CityFrom'] were categorical string data we had to encode them to numeric data to use them to train our model. We mainly tried 2 approaches.

### Approach 1: One hot encoding

One hot encoding is a common approach for transforming categorical variables into numeric values. This is converting categorical data into binary data, where all categories are stated by a boolean value.

We first tried to one hot encode the data using all the string-type features.

```
ohc_df = pd.get_dummies(train_df, columns = ['CityTo', 'CityFrom','TypeOfVehicle','StatusofReserve', 'DomesticFlight', "ReasonForTrip", "ModeOfTravel"])
ohc_df
```

| | TimeOfCreation | TimeOfDeparture | Price | Discounts | Cancelled | timestamp_diff_seconds | CityTo_آبادان | CityTo_آباده | CityTo_آبیک | CityTo_آزادور | ... | StatusofReserve_4 | StatusofReserve_5 | DomesticFlight_ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 18835 | 10768 | -0.464207 | -0.098388 | 0 | 16067 | False | False | False | False | ... | False | False | Fals |
| 1 | 34241 | 17779 | 0.891411 | -0.098388 | 0 | 176031 | False | False | False | False | ... | False | False | Fals |
| 2 | 24277 | 15625 | -0.311670 | -0.098388 | 1 | 2008411 | False | False | False | False | ... | True | False | Fals |
| 3 | 39553 | 19718 | 0.046264 | -0.098388 | 0 | 35311 | False | False | False | False | ... | False | False | Fals |
| 4 | 41021 | 20355 | -0.210482 | -0.098388 | 0 | 70284 | False | False | False | False | ... | False | False | Fals |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | .. |
| 70706 | 6358 | 3895 | -0.343386 | -0.098388 | 1 | 97198 | False | False | False | False | ... | False | True | Fals |
| 70707 | 15249 | 8793 | 0.333215 | -0.098388 | 0 | 32509 | False | False | False | False | ... | False | False | Fals |
| 70708 | 26618 | 14596 | -0.199910 | -0.098388 | 0 | 143721 | False | False | False | False | ... | False | False | Fals |
| 70709 | 9633 | 5667 | 0.245619 | -0.098388 | 0 | 109283 | False | False | False | False | ... | False | False | Fals |
| 70710 | 37733 | 21370 | -0.592127 | -0.098388 | 0 | 1470068 | False | False | False | False | ... | False | False | Fals |

65422 rows × 3308 columns

**This resulted in having a training set of 65422 x 3306.**

Due to the huge number of columns, the models took a few minutes to train. We got an accuracy of only **0.936** using this method. This led us to believe that either the fields “CityTo”, “CityFrom” and “TypeOfVehicle” have no correlation with the label and just contribute to the model complexity in vain, or that the model was performing poorly due to a large number of columns.

Hence we decided to try Label Encoding next.

## Approach 2: Label Encoding

Label encoding operates by assigning a number value to each category to transform it to ordinal data. Each category is allocated a unique integer value using this technique.

We encoded [ 'ReasonForTrip', 'ModeOfTravel'] using this. We had decided to drop the rest of the categorical string data by the time we came to this approach.

```
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
columns_to_encode = [ 'ReasonForTrip', 'ModeOfTravel']
for column in columns_to_encode:
    train_df[column] = label_encoder.fit_transform(train_df[column])
train_df.columns
```

We also tried one hot encoding with only the above two features, however, this resulted in a lower accuracy. This might be due to some underlying correlation between the above two features and the label that prioritizes some values of ReasonForTrip and ModeOfTravel over others.

## 5. Ensemble: Balancing dataset

Our dataset was heavily one-sided with around 85% of the records having class 1 and 15% class 0.

We tried mainly 2 methods to increase the sampling of the minority class:

- **SMOTE** (Synthetic Minority Oversampling Technique). SMOTE is designed to tackle imbalanced datasets by generating synthetic samples for the minority class.
- **ADASYN** (Adaptive Synthetic Algorithm). This method is similar to SMOTE but generates different samples depending on an estimate of the local distribution of the class to be oversampled.

We saw a slight improvement in the submission score after doing so. We got the highest score of **0.956** with the Random Forest Classifier.

## 6. Ensemble: Training Ensemble Models

We **train, test, and split using an 80:20** ratio to train our models.

### Boosting: Gradient Boosting Classifier

This model builds decision trees one at a time, where each new tree helps to correct errors made by previously trained trees.

By training this model on various sets of differently processed data, the highest score we got was **0.937 on the test data.**

Gradient Boosting Classifier

While training we got,

**F1 score: 0.943**

**Accuracy: 0.983**

**Test data: 0.937**

Because we train them to correct each other's errors, they're capable of capturing

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import f1_score
from sklearn.metrics import accuracy_score
clf = GradientBoostingClassifier().fit(x_train, y_train)
y_pred1 = clf.predict(x_test)
f1 = f1_score(y_test, y_pred1)
accuracy = accuracy_score(y_test, y_pred1)
print('F1:', f1)
print('Accuracy:', accuracy)
```

```
F1: 0.9433962264150945
Accuracy: 0.9830508474576272
```

complex patterns in the data. However, if the data are noisy, the boosted trees may overfit and start modeling the noise.

Since we were feeding a huge number of features into it, the model may have always overfitted and hence we were unable to increase accuracy using this model.

## Boosting: Histogram Gradient Boosting Classifier

Histogram gradient boosting is a  variant of gradient boosting that uses histograms to represent the distribution of feature values during the training process. It aims to speed up the training process by discretizing continuous features into bins or histograms. Instead of considering all possible split points, the algorithm works with these bins, which reduces the number of candidate split points and accelerates the training process. This can be particularly advantageous for large datasets.

```
from sklearn.ensemble import HistGradientBoostingClassifier
hgbc = HistGradientBoostingClassifier().fit(x_train_sm, y_train_sm)
y_pred4 = hgbc.predict(x_test)
f1 = f1_score(y_test, y_pred4)
accuracy = accuracy_score(y_test, y_pred4)
print('F1:', f1)
print('Accuracy:', accuracy)
```

While training we got,

**F1 score: 0.944**

**Accuracy: 0.983**

**Test data: 0.937**

## Bagging: Random Forest Classifier

We switched to this model later, when GBC failed to give us better results.

A random forest is a collection of trees, all of which are trained independently and on different subsets of instances and features. The rationale is that although a single tree may be inaccurate, the collective decisions of a bunch of trees are likely to be right most of the time. The main difference compared to GBC is that trees are trained parallelly, hence this may have not overfit compared to the previous model.

We limited the **trees (hyperparameter) to 100**.

We also tried to do hyperparameter tuning using **RandomisedSearchCV** but it took too much time to train so we could not infer anything useful**.**

While training we got,

**F1 score: 0.962**

**Accuracy: 0.988**

**Test data: 0.955**

On the test data, we got a final score of **0.956**. This was clearly a significant improvement compared to GBC.

Random Forest Classifier

```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier(n_estimators=100, random_state=42)
rfc.fit(x_train, y_train)
y_pred3 = rfc.predict(x_test)
f1 = f1_score(y_test, y_pred3)
accuracy = accuracy_score(y_test, y_pred3)
print('F1:', f1)
print('Accuracy:', accuracy)
```

```
F1: 0.962962962962963
Accuracy: 0.9887005649717514
```

## Stacking Classifier

Stacking is an ensemble learning technique that involves training a model to combine the predictions of multiple base models. In the context of a stacking classifier, the idea is to use multiple diverse base classifiers and then train a meta-classifier on top of them to make the final predictions.

```
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
estimators = [('rf', rfc),('gbc', clf), ('hgbc', hgbc)]
sc = StackingClassifier(estimators=estimators, final_estimator=LogisticRegression()).fit(x_train_sm, y_train_sm)
y_pred5 = sc.predict(x_test)
f1 = f1_score(y_test, y_pred5)
accuracy = accuracy_score(y_test, y_pred5)
print('F1:', f1)
print('Accuracy:', accuracy)
```

While training we got,

**F1 score: 0.954**

**Accuracy: 0.985**

**Test data: 0.952**

## Voting Classifier

A voting classifier is another type of ensemble learning method, but unlike stacking, it involves combining the predictions of multiple base classifiers through a "voting" mechanism. The idea is to train several diverse base classifiers independently and let them "vote" on the predicted class for a given input. The class that receives the majority of votes is then chosen as the final prediction.

```
from sklearn.ensemble import VotingClassifier
vc = VotingClassifier(estimators=[('rf', rfc),('gbc', clf), ('hgbc', hgbc)], voting='hard').fit(x_train_sm, y_train_sm)
y_pred6 = vc.predict(x_test)
f1 = f1_score(y_test, y_pred6)
accuracy = accuracy_score(y_test, y_pred6)
print('F1:', f1)
print('Accuracy:', accuracy)
```

While training we got,

**F1 score: 0.948**

**Accuracy: 0.984**

**Test data: 0.941**

## 7. Ensemble: Final Results and Accuracy

The **random forest classifier** gave us the best scores with the preprocessed data being as follows:

```
train_df
```

| | TimeOfCreation | TimeOfDeparture | BillNo. | TicketNo. | StatusofReserve | Price | DomesticFlight | ReasonForTrip | ModeOfTravel | BuyerMobile | Cancelled | timestamp_diff_seconds |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1657087332 | 1657103400 | 38131030 | 7359427.0 | 3 | 850000.0 | 1 | 0 | 0 | 965396967731 | 0 | 16068 |
| **1** | 1662904268 | 1663080300 | 39115817 | 3002688.0 | 2 | 5338000.0 | 1 | 0 | 3 | 452719996887 | 0 | 176032 |
| **2** | 1659365128 | 1661373540 | 38510118 | 2927990.0 | 4 | 1355000.0 | 1 | 1 | 3 | 116690640411 | 1 | 2008412 |
| **3** | 1664448088 | 1664483400 | 39403118 | 7663791.0 | 3 | 2540000.0 | 1 | 1 | 0 | 642337257287 | 0 | 35312 |
| **4** | 1664815415 | 1664885700 | 39470084 | 7681449.0 | 3 | 1690000.0 | 1 | 0 | 0 | 138128253547 | 0 | 70285 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **70706** | 1649361601 | 1649458800 | 36839872 | 7018030.0 | 5 | 1250000.0 | 1 | 1 | 0 | 331267793363 | 1 | 97199 |
| **70707** | 1654760890 | 1654793400 | 37704940 | 2825554.0 | 2 | 3490000.0 | 1 | 1 | 3 | 409302394890 | 0 | 32510 |
| **70708** | 1660375178 | 1660518900 | 38660767 | 7510813.0 | 3 | 1725000.0 | 1 | 1 | 0 | 666188659988 | 0 | 143722 |
| **70709** | 1651495116 | 1651604400 | 37152781 | 7096569.0 | 3 | 3200000.0 | 1 | 0 | 0 | 832973699414 | 0 | 109284 |
| **70710** | 1664056131 | 1665526200 | 39326282 | 3028631.0 | 2 | 426500.0 | 1 | 1 | 3 | 504607789241 | 0 | 1470069 |

70711 rows × 12 columns

Test data score**: 0.956**

F1 score**: 0.96**

Accuracy: **0.98**

## 1. SVM: Encoding and Standardization

We tried 2 approaches from our previous assignment

1. Label Encoding
2. One hot encoding

With label encoding, we got the best f1 score of 0.94.

One hot encoding of the data we got a slight improvement of 0.942.

We encoded the following columns:

```
train_df = pd.get_dummies(train_df, columns = ["ReasonForTrip", "ModeOfTravel"])
train_df
```

We also standardised the data before starting training as SVMs respond better to standardised data.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

x_train = scaler.fit_transform(x_train)
x_train_sm1 = scaler.fit_transform(x_train_sm)

x_test = scaler.transform(x_test)

x_train_sm1
```

## 2. SVM: Balancing the dataset

Our dataset was heavily one-sided with around 85% of the records having class 1 and 15% class 0.

We tried mainly 2 methods to increase the sampling of the minority class:

- **SMOTE** (Synthetic Minority Oversampling Technique). SMOTE is designed to tackle imbalanced datasets by generating synthetic samples for the minority class.
- **ADASYN** (Adaptive Synthetic Algorithm). This method is similar to SMOTE but generates different samples depending on an estimate of the local distribution of the class to be oversampled.

```
]: len(y_train_sm[y_train_sm==0]),len(y_train_sm[y_train_sm==1])

... (48035, 48035)
```

Here we can see that both the classes are now balanced.

The results for both of these across different kernels were very similar for both so we just stuck with ADASYN afterwards due to a 0.0003 better f1 score in the rbf kernel.

**SMOTE**

```
Model accuracy score with default hyperparameters: 0.9827
F1 score: 0.941652774470112

Confusion matrix
```

```
Model accuracy score with polynomial kernel and C=1.0 : 0.9825
F1 score: 0.9411203814064363
Confusion matrix
```

```
Model accuracy score with linear kernel and C=0.1.0 : 0.9109
F1 score: 0.773462783171521
```

```
print( F1 score: , f1)

Model accuracy score with rbf kernel, c = 1, gamma = 10: 0.9668
F1 score: 0.8988570196247573
```

**ADASYN**

```
print("F1 score:", f1)
```

```
Model accuracy score with sigmoid kernel and C=1.0 : 0.7669
F1 score: 0.5215498476273401
```

```
Model accuracy score with default hyperparameters: 0.9827
F1 score: 0.9418216499761565
```

```
print('Model accuracy score with linear kernel and C=10
print("F1 score:", f1)
```

```
Model accuracy score with linear kernel and C=100.0 : 0.9084
F1 score: 0.7689839572192513
```

## 3. SVM: Training different kernels

We did a train-test split with a ratio of 80:20.

We tried 4 different kernels with various hyperparameters.

1. RBF Kernel (Default)
2. Linear Kernel
3. Polynomial Kernel
4. Sigmoid Kernel

For hyperparameter tuning, we tried to run **GridSearchCV** over various combinations of kernels, C, and gamma along with the cross-validation value of 5. But this takes a long time to run so we weren't able to conclude any information.

```
# import metrics to compute accuracy
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score

# instantiate classifier with default hyperparameters
svc=SVC()
svc.get_params()

from sklearn.model_selection import GridSearchCV

parameters_dictionary = {'kernel':['rbf', 'linear', 'poly'],
                         'C':[0.0001, 1, 10],
                         'gamma':[1, 10, 100]}
svc = SVC()

grid_search = GridSearchCV(svc,
                           parameters_dictionary,
                           scoring = 'f1',
                           return_train_score=True,
                           cv = 5,
                           verbose = 1) # Displays how many combinati
grid_search.fit(x_train, y_train)

best_model = grid_search.best_estimator_
best_parameters = grid_search.best_params_
best_f1 = grid_search.best_score_

print('The best model was:', best_model)
print('The best parameter values were:', best_parameters)
print('The best f1-score was:', best_f1)
```

1. **RBF kernel**

   The default kernel settings of kernel = ‘rbf’ and C = 1.0 gave us good results combined with one hot encoding and ADASYN balancing.

```
Model accuracy score with default hyperparameters: 0.9827
F1 score: 0.9418216499761565

Confusion matrix

 [[11924     1]
 [  243  1975]]

True Positives(TP) =  11924

True Negatives(TN) =  1975

False Positives(FP) =  1

False Negatives(FN) =  243
Total Errors =  244
```

On submission with the given test data, we obtained a score of **0.937** on the leaderboard.


2. **Linear Kernel**

We tried this with default hyperparameters but got low f1 score.

```
Model accuracy score with linear kernel and C=1 : 0.9109
F1 score: 0.773462783171521
```


So we decided to discard this model

3. **Polynomial kernel**

We tried this model with various degrees. The default degree is 3. We tried with degree = 4, 5 too.

With degree = 3,

```
Model accuracy score with polynomial kernel and C=1.0 : 0.9827
F1 score: 0.9415692821368948
Confusion matrix

 [[11924     1]
 [  244  1974]]

True Positives(TP) =  11924

True Negatives(TN) =  1974

False Positives(FP) =  1

False Negatives(FN) =  244
Total Errors =  245
```

But on submission, we got a score of **0.787 despite a good f1 score**.

**poly (5).csv** **0.787**

Complete · yuktaX · 15s ago

With degree 4 we got,

```
print('\nFalse Negatives(FN) = ', cm[1,0])
print('Total Errors = ', cm[0, 1] + cm[1, 0])
```

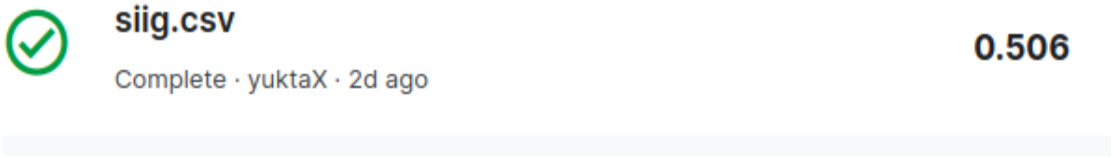
```
Model accuracy score with polynomial kernel and C=1.0 : 0.9822
F1 score: 0.9401140684410646
Confusion matrix

 [[11913    12]
 [  240  1978]]

True Positives(TP) =  11913

True Negatives(TN) =  1978

False Positives(FP) =  12

False Negatives(FN) =  240
Total Errors =  252
```

But on submission, we got 0.006 so it looks like the model overfit. The same happened with degree = 5.


4. **Sigmoid Kernel**

   This kernel performed the poorest compared to all the others.

```
Model accuracy score with sigmoid kernel and C=1.0 : 0.7677
F1 score: 0.5227371785558622
```




   So we discarded this too.

## 4. SVM: Final Results

Best kernel: RBF

Hyperparameters: C = 1, default parameters.

Best F1 score: **0.942** from RBF kernel

Best submission: **0.937** corresponding to RBF kernel

## 1. Neural Networks: Encoding and Standardization

We tried 2 approaches from our previous assignment

3. Label Encoding
4. One hot encoding

With label encoding, we got the best f1 score of 0.94.

One hot encoding, it took significantly longer to train and we got an f1 score 0.938

Hence we decided to go with label encoding.

We encoded the following columns:

```
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
columns_to_encode = [ 'ReasonForTrip', 'ModeOfTravel']
for column in columns_to_encode:
    train_df[column] = label_encoder.fit_transform(train_df[column])
train_df.info()
```

Next we standardize the data using StandardScaler()

## 2. Neural Networks: Balancing the dataset

Our dataset was heavily one-sided with around 85% of the records having class 1 and 15% class 0.

We tried mainly 2 methods to increase the sampling of the minority class:

- **SMOTE** (Synthetic Minority Oversampling Technique). SMOTE is designed to tackle imbalanced datasets by generating synthetic samples for the minority class.
- **ADASYN** (Adaptive Synthetic Algorithm). This method is similar to SMOTE but generates different samples depending on an estimate of the local distribution of the class to be oversampled.

```
]: len(y_train_sm[y_train_sm==0]),len(y_train_sm[y_train_sm==1])

... (48035, 48035)
```

Here we can see that both the classes are now balanced.

However when we tested the model we ended up getting slightly poor results as compared to without balancing.

```
Confusion Matrix of the Test Set
-----------
[[14926    38]
 [  295  2419]]
Precision of the MLP :  0.9845339845339846
Recall of the MLP    :  0.8913043478260869
F1 Score of the Model : 0.9356023979887836
```

Therefore we decided to proceed without balancing the data.

## 3. Neural Networks: Creating Tensors and DataLoaders

In PyTorch, tensors are a fundamental data structure used for numerical computations. A tensor is a multi-dimensional array that can be used to represent both scalar values and complex mathematical entities, such as vectors and matrices. Tensors are the basic building blocks for creating and working with neural networks in PyTorch.

All the training and testing data must be converted into tensor form before being used.

```
x_tensor =  torch.from_numpy(x_train).float()
y_tensor =  torch.from_numpy(train_y.values.ravel()).float()
xtest_tensor =  torch.from_numpy(x_test).float()
ytest_tensor =  torch.from_numpy(test_y.values.ravel()).float()
```

DataLoaders in PyTorch serve as a convenient and efficient mechanism for handling data during the training and evaluation of machine learning models. DataLoaders help us to perform batch gradient descent by separating the data into batches.

```
#Define a batch size ,
bs = 128
#Both x_train and y_train can be combined in a single TensorDataset, which will be easier to iterate over and slice
y_tensor = y_tensor.unsqueeze(1)
train_ds = TensorDataset(x_tensor, y_tensor)
#Pytorch's DataLoader is responsible for managing batches.
#You can create a DataLoader from any Dataset. DataLoader makes it easier to iterate over batches
train_dl = DataLoader(train_ds, batch_size=bs)


#For the validation/test dataset
ytest_tensor = ytest_tensor.unsqueeze(1)
test_ds = TensorDataset(xtest_tensor, ytest_tensor)
test_loader = DataLoader(test_ds, batch_size=32)
```

## 4. Neural Networks: Defining the model

The main components that define a neural network and its training are:

1. Number of layers
2. Number of neurons per layer
3. Activation function between each layer
4. Loss function
5. Learning rate
6. Optimizer
7. Epochs

We tried out a lot of permutations and combinations of the above 7 factors.

1. Number of layers and number of neurons per layer:

   Layer 1 with 300 neurons and Layer 2 with 100 neurons:

```
Confusion Matrix of the Test Set
-----------
[[14926    38]
 [  295  2419]]
Precision of the MLP :  0.9845339845339846
Recall of the MLP    :  0.8913043478260869
F1 Score of the Model : 0.9356023979887836
```

Layer 1 with 60 neurons and Layer 2 with 60 neurons:

```
Confusion Matrix of the Test Set
-----------
[[14935    29]
 [  292  2422]]
Precision of the MLP :  0.9881680946552428
Recall of the MLP    :  0.89240972733972
F1 Score of the Model : 0.9378509196515005
```

Layer 1 with 30 neurons and Layer 2 with 30 neurons:

```
Confusion Matrix of the Test Set
-----------
[[14952    12]
 [  299  2415]]
Precision of the MLP :  0.9950556242274413
Recall of the MLP    :  0.8898305084745762
F1 Score of the Model : 0.9395059326979186
```

Layer 1 with 15 neurons and Layer 2 with 15 neurons:

```
Confusion Matrix of the Test Set
-----------
[[14964     0]
 [  304  2410]]
Precision of the MLP :  1.0
Recall of the MLP    :  0.887988209285188
F1 Score of the Model : 0.9406713505074161
```

Adding a 3rd layer with 15 neurons:

```
Confusion Matrix of the Test Set
-----------
[[14943    21]
 [  301  2413]]
Precision of the MLP :  0.9913722267871816
Recall of the MLP    :  0.8890935887988209
F1 Score of the Model : 0.9374514374514376
```

Layer 1 with 10 neurons and Layer 2 with 10 neurons:

```
Confusion Matrix of the Test Set
-----------
[[14964     0]
 [  304  2410]]
Precision of the MLP :  1.0
Recall of the MLP    :  0.887988209285188
F1 Score of the Model : 0.9406713505074161
```

Since one the best f1 scores was obtained by the configuration of 2 layers with 10 neurons each, and it had the shortest training time, we decided to go ahead with it.

2. Activation Function and Loss Function:

   For the activation function, we chose RELU for the hidden layers to introduce non-linearity and Sigmoid for the output layer to “squish” the output between 0 and 1.

```
n_hidden2 = 10
#n_hidden3 = 15
n_output =  1   # Number of output nodes = for binary classifier


class ChurnModel(nn.Module):
    def __init__(self):
        super(ChurnModel, self).__init__()
        self.layer_1 = nn.Linear(n_input_dim, n_hidden1)
        self.layer_2 = nn.Linear(n_hidden1, n_hidden2)
        #self.layer_3 = nn.Linear(n_hidden2, n_hidden3)
        self.layer_out = nn.Linear(n_hidden2, n_output)

        self.relu = nn.ReLU()
        self.sigmoid =  nn.Sigmoid()
        self.dropout = nn.Dropout(p=0.1)
        self.batchnorm1 = nn.BatchNorm1d(n_hidden1)
        self.batchnorm2 = nn.BatchNorm1d(n_hidden2)
        #self.batchnorm3 = nn.BatchNorm1d(n_hidden3)

    def forward(self, inputs):
        x = self.relu(self.layer_1(inputs))
        x = self.batchnorm1(x)
        x = self.relu(self.layer_2(x))
        x = self.batchnorm2(x)
#         x = self.relu(self.layer_3(x))
#         x = self.batchnorm3(x)
        x = self.dropout(x)
        x = self.sigmoid(self.layer_out(x))

        return x
```

We also utilized batch normalization in our model. Batch Normalisation is a technique used in neural networks to improve training stability and convergence speed. The key idea behind Batch Normalisation is to normalize the input of each layer in a mini-batch to have zero mean and unit variance. This normalization is applied after the activation function, and the normalized values are then scaled and shifted using learnable parameters.

Dropout was also utilized to help mitigate over-fitting.

The loss function we chose was Binary Cross Entropy Loss, which is the standard loss function used for Binary Classification problems.

3. Optimizer, Learning Rate and Epochs:

   We chose the Adam optimizer because it combines the Momentum and RMSprop methods and is the single most widely used optimization method.

   We set the learning rate to 0.001 and set the Epochs to 150

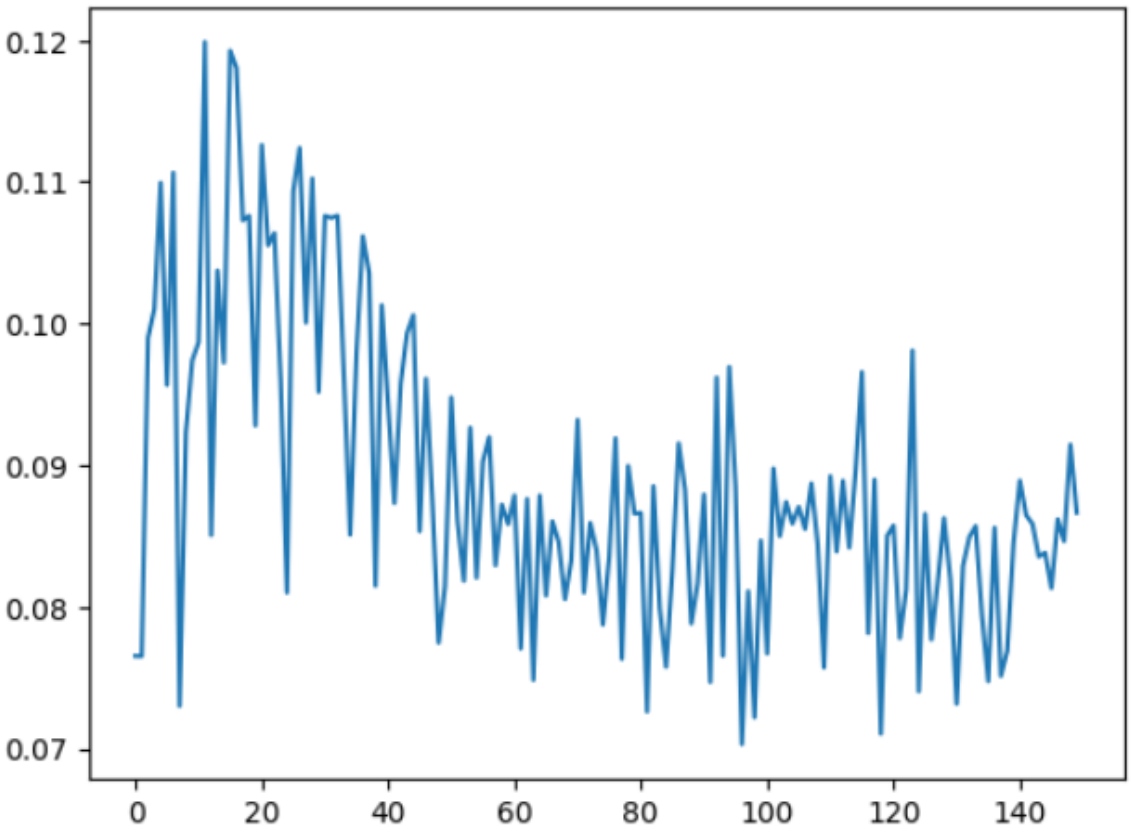
```
#Loss Computation
loss_func = nn.BCELoss()
#Optimizer
learning_rate = 0.001
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
epochs = 150
```

## 5. Neural Networks: Training the model

After defining the model, we proceeded to train it.

```
model.train()
train_loss = []
for epoch in range(epochs):
    #Within each epoch run the subsets of data = batch sizes.
    for xb, yb in train_dl:
        y_pred = model(xb)            # Forward Propagation
        loss = loss_func(y_pred, yb)  # Loss Computation
        optimizer.zero_grad()         # Clearing all previous gradients, setting to zero
        loss.backward()               # Back Propagation
        optimizer.step()              # Updating the parameters
    #print("Loss in iteration :"+str(epoch)+" is: "+str(loss.item()))
    train_loss.append(loss.item())
print('Last iteration loss value: '+str(loss.item()))
```

The below graph shows the Loss vs Epoch count




As we can see, the loss starts stabilizing after approximately 60 epochs.

## 6. Neural Networks: Results


```
Confusion Matrix of the Test Set
-----------
[[14964     0]
 [  304  2410]]
Precision of the MLP :  1.0
Recall of the MLP    :  0.887988209285188
F1 Score of the Model : 0.9406713505074161
```

*END*