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

ML Assignment 1

Ticket Cancellation Dataset (Classification)

Overview

This report contains an in-detail record of how we approached the dataset. This includes:

1. Preprocessing Data
2. Exploratory Data Analysis (EDA)
3. Feature Engineering
4. Data Encoding
5. Training Models
6. Final Results and Accuracy

1. 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
```

5...

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

```
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
```

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. Exploratory Data Analysis (EDA)

Percentage Cancellations:

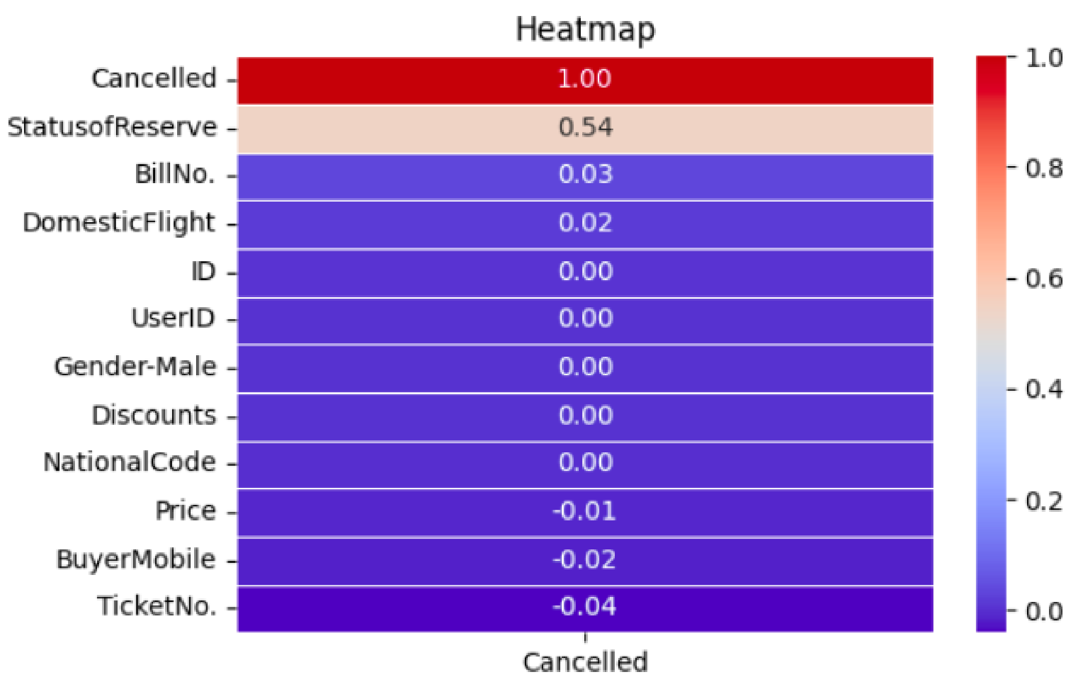
```
Percentage cancelation=  0.15204140798461344
StatusofReserve
3    42.095289
2    30.911739
5    19.883752
4     7.109219
Name: proportion, dtype: float64
```

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
```

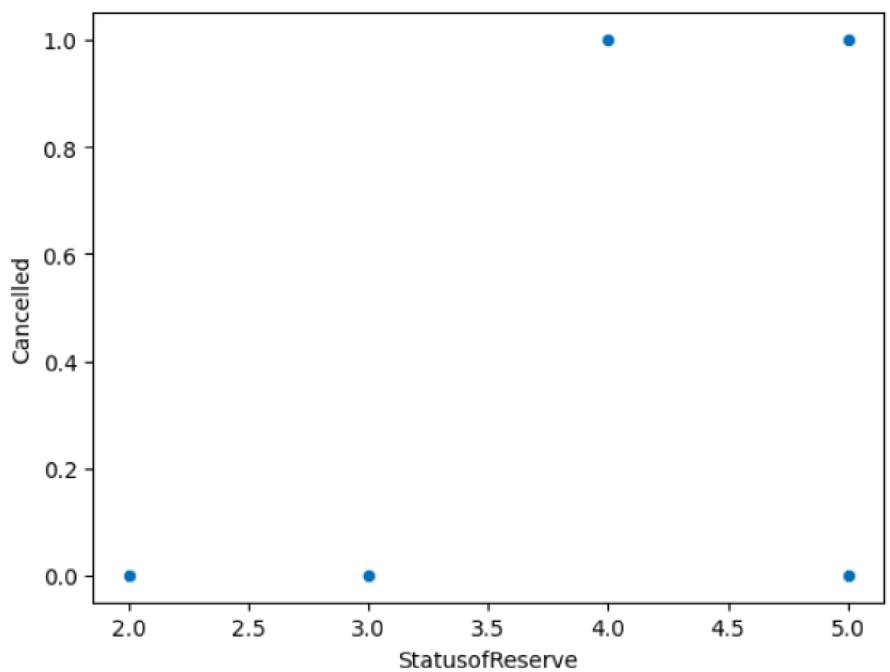
Status of Reserve

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"])
```

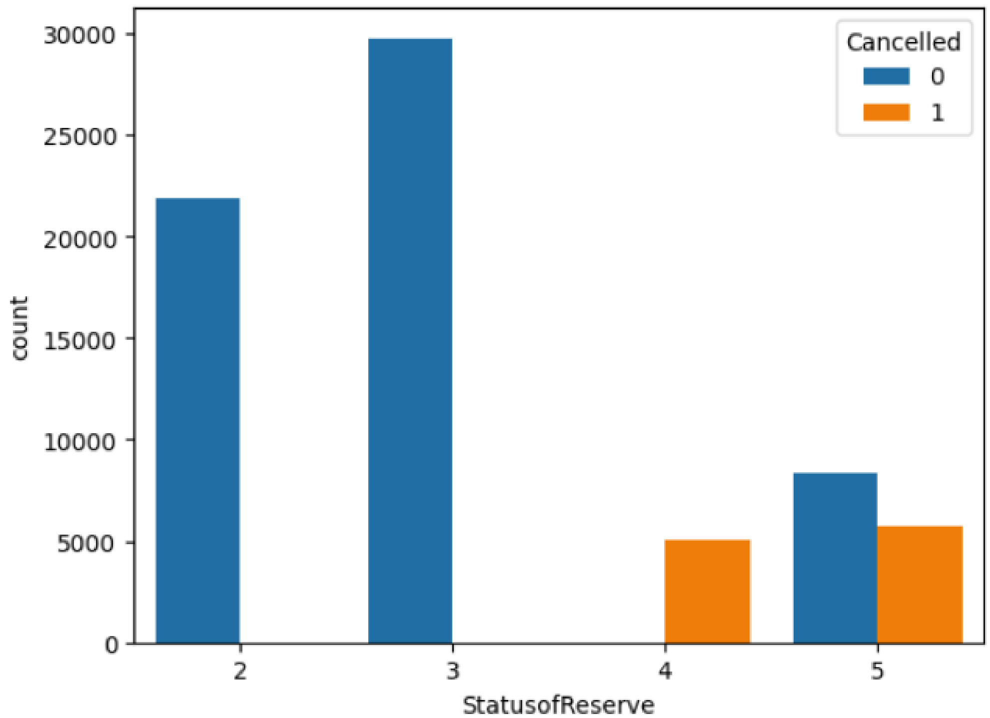
```
<Axes: xlabel='StatusofReserve', ylabel='Cancelled'>
```



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. 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

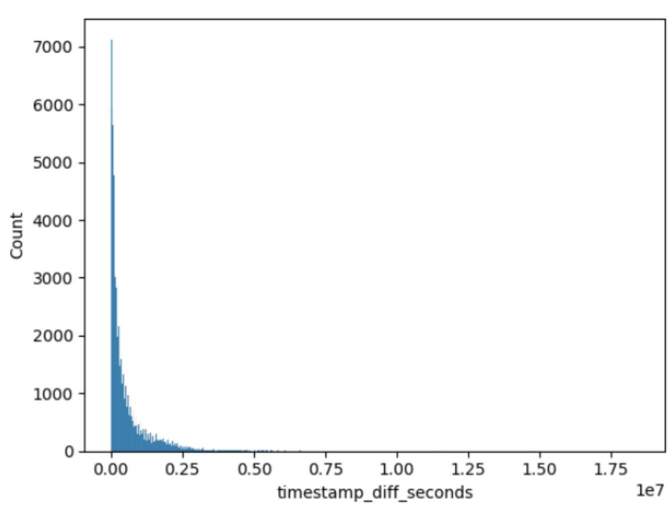
```
train_df['TimeOfCreation'] = (pd.to_datetime(train_df['TimeOfCreation']) - pd.Timestamp("1970-01-01")) // pd.Timedelta('1s')
train_df['TimeOfDeparture'] = (pd.to_datetime(train_df['TimeOfDeparture']) - pd.Timestamp("1970-01-01")) // pd.Timedelta('1s')
```

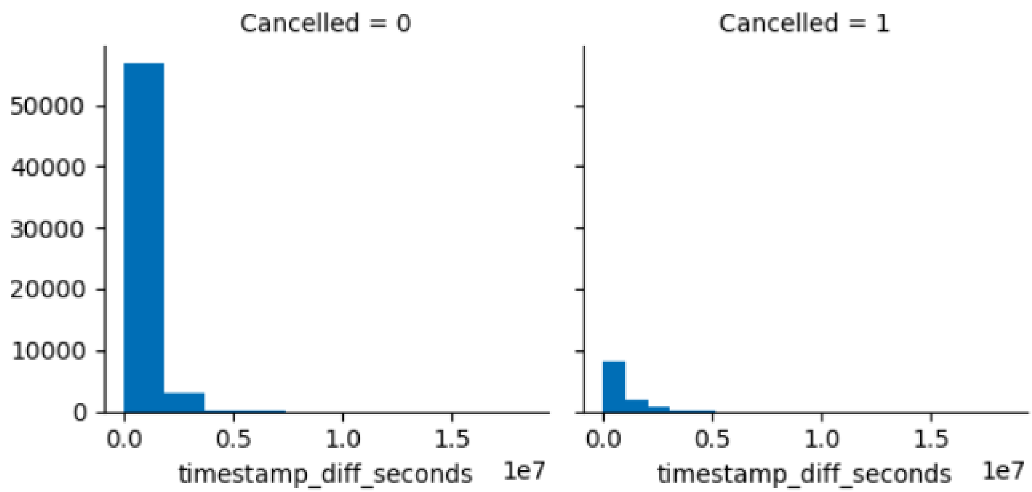
Adding a timestamp difference column

```
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. 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	False
1	34241	17779	0.891411	-0.098388	0	176031	False	False	False	False	...	False	False	False
2	24277	15625	-0.311670	-0.098388	1	2008411	False	False	False	False	...	True	False	False
3	39553	19718	0.046264	-0.098388	0	35311	False	False	False	False	...	False	False	False
4	41021	20355	-0.210482	-0.098388	0	70284	False	False	False	False	...	False	False	False
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
70706	6358	3895	-0.343386	-0.098388	1	97198	False	False	False	False	...	False	True	False
70707	15249	8793	0.333215	-0.098388	0	32509	False	False	False	False	...	False	False	False
70708	26618	14596	-0.199910	-0.098388	0	143721	False	False	False	False	...	False	False	False
70709	9633	5667	0.245619	-0.098388	0	109283	False	False	False	False	...	False	False	False
70710	37733	21370	-0.592127	-0.098388	0	1470068	False	False	False	False	...	False	False	False

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. Training Models

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

We mainly used 2 models while working on this dataset - Gradient Boosting and Random Forest. We chose these as they are known to be best for categorical data and since we had training with very few categories in each feature we thought decision trees would be a good fit.

Some advantages of the above decision tree-based models that motivated us to use them are:

- No Data Preprocessing: Decision trees can handle both categorical and numerical data **without requiring extensive preprocessing like one-hot encoding or standardization**, **scaling, and normalization. Hence we did not standardize the data at any point.**
- Non-Parametric: Decision trees are non-parametric models, which means they make no assumptions about the data's underlying distribution. Since the data doesn't follow any particular distribution, a decision trees based model is perfect.
- Ensemble Models: Random forests and gradient boosting are ensemble techniques built on decision trees. They combine the strength of multiple decision trees to improve predictive performance and reduce overfitting.

We tried the **logistic classifier and KNN** too but they did not give good results (ranging from **0.92 to 0.93**) within the first few predictions so we discontinued training them.

We started out with the Gradient Boosting Classifier.

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

```python
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
```

Because we train them to correct each other's errors, they're capable of capturing 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 overfit and hence we were unable to increase accuracy using this model.

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 parallely, hence this may have not overfit compared to the previous model.

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

Random Forest Classifier

While training we got,

F1 score: 0.962

Accuracy: 0.988

Test data: 0.955

```
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
```

On the test data, we got a final score of **0.955**. This was clearly a significant improvement compared to GBC.

6. 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: 0.955

F1 score: 0.96

Accuracy: 0.989

End