What's My Flight Status?: Using Flight Data to Predict Flight Delays

Author: Andre Layton

Overview

One of the most common applications of supervised learning in the aviation industry is predicting flight delays. By analyzing historical data on flight delays, machine learning and deep learning algorithms can identify patterns and factors that correlate with delays. This information can then be used to predict if a particular flight is likely to be delayed. Carriers can use these predictions to take preemptive measures, such as adjusting schedules or re-routing passengers, in order to minimize the impact of delays and their sales.

Business Problem

I've been hired to create an algorithm that can predict flight delays, which will eventually be deployed as an app for consumers to be able to track their flights. This is beneficial to both the airline and potential passengers – for the airline, it will help with flight logistics and reduce fees due to delays (i.e. tarmac fees, reimbursements, etc.). For passengers, the app will allow them to make delay arrangements and take measures ahead of time, and possibly save on delay expenses. While delays are frustrating whether expected or not, United aims to use this strategy to display company honesty and gain more control over their flights.

> Note: The ultimate objective is to develop an app for consumer use, but within the constraints of this analysis, the model will be saved and stored in the repository here. (/best_model.h5)

Data Understanding

To start, I import all the necessary packages, and I set a seed for reproducibility purposes. Then, I load the data, which is split into two files - a text file (`.txt`) and a CSV file (`.csv`). The text file contains the metadata, which in this case contains various column names and a short description. The CSV file contains the flight data needed for analysis.

In [1]:
```
# Import relevant libraries
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import matplotlib.ticker as ticker
from matplotlib.ticker import AutoMinorLocator
import seaborn as sns
%matplotlib inline
plt.style.use('ggplot')

from sklearn.pipeline import make_pipeline, Pipeline
from sklearn.model_selection import train_test_split, GridSearchCV, cross_vali
from imblearn.over_sampling import SMOTE
from collections import Counter
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import precision_score, recall_score, f1_score, confusion
                            ConfusionMatrixDisplay, make_scorer
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifie

from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras import regularizers, optimizers

# Create a seed for reproducibility
seed=24
```

In [2]:
```
# Load the metadata (column descriptions)
metadata = open("data/raw_data_documentation.txt", "r")
print(metadata.read())
```

```
AIRPORT_COORDINATES
        ORIGIN_AIRPORT_ID:      Airport ID, matches to ORIGIN_AIRPORT_ID i
n other files
        DISPLAY_AIRPORT_NAME:   Display Airport, matches to DISPLAY_AIRPOR
T_NAME in other files
        LATITUDE:               Latitude for airport
        LONGITUDE:              Longitude for airport

B43_AIRCRAFT_INVENTORY
        MANUFACTURE_YEAR:       Manufacture year
        TAIL_NUM:               Unique tail number, matches to TAIL_NUM in
other files
        NUMBER_OF_SEATS:        Number of seats on aircraft

CARRIER_DECODE
        AIRLINE_ID:             Airport ID, matches to AIRLINE_ID in other
files
        OP_UNIQUE_CARRIER:      Carrier code, matches to OP_UNIQUE_CARRIER
in other files
```

In [3]:
```
# Load the flight data
data = pd.read_csv('data/full_data_flightdelay.csv')

# Preview the first 10 records
data.head(10)
```

Out[3]:

	MONTH	DAY_OF_WEEK	DEP_DEL15	DEP_TIME_BLK	DISTANCE_GROUP	SEGMENT_NUMBE
0	1	7	0	0800-0859	2	
1	1	7	0	0700-0759	7	
2	1	7	0	0600-0659	7	
3	1	7	0	0600-0659	9	
4	1	7	0	0001-0559	7	
5	1	7	0	0001-0559	3	
6	1	7	0	0700-0759	6	
7	1	7	1	0001-0559	7	
8	1	7	0	0001-0559	7	
9	1	7	0	0600-0659	8	

10 rows × 26 columns

The dataframe above gives me an initial look into the dataset, but I will apply a few more methods to gain a better understanding (i.e. `info()`, `isna()`, etc.). These techniques will help me learn more about my data, including the existence of any missing values and the data types of each column.

In [4]:
```
# Print column information
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6489062 entries, 0 to 6489061
Data columns (total 26 columns):
 #   Column                          Dtype
---  ------                          -----
 0   MONTH                           int64
 1   DAY_OF_WEEK                     int64
 2   DEP_DEL15                       int64
 3   DEP_TIME_BLK                    object
 4   DISTANCE_GROUP                  int64
 5   SEGMENT_NUMBER                  int64
 6   CONCURRENT_FLIGHTS              int64
 7   NUMBER_OF_SEATS                 int64
 8   CARRIER_NAME                    object
 9   AIRPORT_FLIGHTS_MONTH           int64
 10  AIRLINE_FLIGHTS_MONTH           int64
 11  AIRLINE_AIRPORT_FLIGHTS_MONTH   int64
 12  AVG_MONTHLY_PASS_AIRPORT        int64
 13  AVG_MONTHLY_PASS_AIRLINE        int64
 14  FLT_ATTENDANTS_PER_PASS         float64
 15  GROUND_SERV_PER_PASS            float64
 16  PLANE_AGE                       int64
 17  DEPARTING_AIRPORT               object
 18  LATITUDE                        float64
 19  LONGITUDE                       float64
 20  PREVIOUS_AIRPORT                object
 21  PRCP                            float64
 22  SNOW                            float64
 23  SNWD                            float64
 24  TMAX                            float64
 25  AWND                            float64
dtypes: float64(9), int64(13), object(4)
memory usage: 1.3+ GB
```

I can see the method above displays the data type for each feature. Another point is the amount of data present. There are over 6.4 million observations, which will need to be reduced considering I'm only interested in United Airlines flights. The method below filters out the records based on the airline, revealing there are over 600,000 observations that will be analyzed to build my algorithm.

In [5]:
```
# Print airline information
data['CARRIER_NAME'].value_counts()
```

Out[5]:
```
Southwest Airlines Co.          1296329
Delta Air Lines Inc.             938346
American Airlines Inc.           903640
United Air Lines Inc.            601044
SkyWest Airlines Inc.            584204
Midwest Airline, Inc.            300154
JetBlue Airways                  269596
Alaska Airlines Inc.             239337
American Eagle Airlines Inc.     228792
Comair Inc.                      219324
Endeavor Air Inc.                203827
Spirit Air Lines                 189419
Mesa Airlines Inc.               177600
Frontier Airlines Inc.           120872
Atlantic Southeast Airlines       99044
Hawaiian Airlines Inc.            74898
Allegiant Air                     42636
Name: CARRIER_NAME, dtype: int64
```

The first method also gives me an idea into whether there are any missing values in the dataset. To know for sure, I will apply some more methods that will take the sum of every missing value in each column and returns those values. I find that, fortunately, there are no missing values present in my data, which will make cleaning the data more straightforward.

In [6]:
```
# Find the amount of missing values in each column
data.isna().sum()
```

Out[6]:
```
MONTH                           0
DAY_OF_WEEK                     0
DEP_DEL15                       0
DEP_TIME_BLK                    0
DISTANCE_GROUP                  0
SEGMENT_NUMBER                  0
CONCURRENT_FLIGHTS              0
NUMBER_OF_SEATS                 0
CARRIER_NAME                    0
AIRPORT_FLIGHTS_MONTH           0
AIRLINE_FLIGHTS_MONTH           0
AIRLINE_AIRPORT_FLIGHTS_MONTH   0
AVG_MONTHLY_PASS_AIRPORT        0
AVG_MONTHLY_PASS_AIRLINE        0
FLT_ATTENDANTS_PER_PASS         0
GROUND_SERV_PER_PASS            0
PLANE_AGE                       0
DEPARTING_AIRPORT               0
LATITUDE                        0
LONGITUDE                       0
PREVIOUS_AIRPORT                0
PRCP                            0
SNOW                            0
SNWD                            0
TMAX                            0
AWND                            0
dtype: int64
```

Data Preparation

Now that I've gotten an initial look, it's time to begin preparing the data for modeling. I will start by making a copy of the original dataset, then filtering the data to keep only flights taken with United Airlines. From my earlier observations, I can see that will greatly reduce the data from 6 million to a little over 600,000 records.

In [7]:
```
# Make a copy of the dataset
data2 = data.copy()
```

In [8]:
```
# Filter United Airlines's records and list the first 10
data2 = data2.loc[data2['CARRIER_NAME'] == 'United Air Lines Inc.']
data2.head(10)
```

Out[8]:

	MONTH	DAY_OF_WEEK	DEP_DEL15	DEP_TIME_BLK	DISTANCE_GROUP	SEGMENT_NUMB
21	1	7	0	0800-0859	2	
22	1	7	0	0800-0859	3	
23	1	7	0	0900-0959	7	
24	1	7	1	1000-1059	3	
25	1	7	0	0600-0659	7	
26	1	7	0	0700-0759	1	
27	1	7	0	0600-0659	2	
28	1	7	0	0600-0659	9	
29	1	7	0	0001-0559	5	
30	1	7	0	0600-0659	3	

10 rows × 26 columns

Next step is to remove all the columns I believe are unnecessary or irrelevant to my model. This will reduce my dimensions from 26 to 12, which includes my target column.

In [9]:
```
# Drop unnecessary columns and display column information
cols_to_drop = ['SEGMENT_NUMBER', 'NUMBER_OF_SEATS', 'FLT_ATTENDANTS_PER_PASS'
                'GROUND_SERV_PER_PASS', 'AIRLINE_AIRPORT_FLIGHTS_MONTH', 'PREV
                'LATITUDE', 'LONGITUDE', 'CARRIER_NAME', 'CONCURRENT_FLIGHTS',
data2 = data2.drop(cols_to_drop, axis=1)
data2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 601044 entries, 21 to 6489030
Data columns (total 12 columns):
 #   Column             Non-Null Count   Dtype  
---  ------             --------------   -----  
 0   MONTH              601044 non-null  int64  
 1   DAY_OF_WEEK        601044 non-null  int64  
 2   DEP_DEL15          601044 non-null  int64  
 3   DEP_TIME_BLK       601044 non-null  object 
 4   DISTANCE_GROUP     601044 non-null  int64  
 5   PLANE_AGE          601044 non-null  int64  
 6   DEPARTING_AIRPORT  601044 non-null  object 
 7   PRCP               601044 non-null  float64
 8   SNOW               601044 non-null  float64
 9   SNWD               601044 non-null  float64
 10  TMAX               601044 non-null  float64
 11  AWND               601044 non-null  float64
dtypes: float64(5), int64(5), object(2)
memory usage: 59.6+ MB
```

Now that the columns have been reduced, I will look to do the same with the rows by dropping any duplicates that exist in my dataset. I will also rename some of the column names for easier comprehension, moving forward, and then shift the target column to the end of my dataframe.

In [10]:
```
# Find the number of duplicate records
data2.duplicated().sum()
```

Out[10]:
```
18105
```

In [11]:
```
# Drop duplicates
data2.drop_duplicates(inplace=True)
```

In [12]:
```
# Rename the columns for easier comprehension & list the first 5 records
new_col_names = {'DEP_DEL15': 'DELAYED',
                 'PRCP': 'PRECIPITATION',
                 'SNWD': 'SNOW_ON_GROUND',
                 'TMAX': 'MAX_TEMP_FOR_DAY',
                 'AWND': 'MAX_WIND_FOR_DAY'}
data2 = data2.rename(new_col_names, axis=1)
data2.head()
```

Out[12]:

	MONTH	DAY_OF_WEEK	DELAYED	DEP_TIME_BLK	DISTANCE_GROUP	PLANE_AGE	DEPA
21	1	7	0	0800-0859	2	6	McC
22	1	7	0	0800-0859	3	22	McC
23	1	7	0	0900-0959	7	3	McC
24	1	7	1	1000-1059	3	19	McC
25	1	7	0	0600-0659	7	4	McC

In [13]:
```
# Shift the target column to the end
cols_at_end = ['DELAYED']
data2 = data2[[col for col in data2 if col not in cols_at_end]
        + [col for col in cols_at_end if col in data2]]

# Preview the first 5 records to confirm the change
data2.head()
```

Out[13]:

	MONTH	DAY_OF_WEEK	DEP_TIME_BLK	DISTANCE_GROUP	PLANE_AGE	DEPARTING_AIR
21	1	7	0800-0859	2	6	McCarran Interna
22	1	7	0800-0859	3	22	McCarran Interna
23	1	7	0900-0959	7	3	McCarran Interna
24	1	7	1000-1059	3	19	McCarran Interna
25	1	7	0600-0659	7	4	McCarran Interna

Now that the dataset has been cleaned up, I want to take a look at the distributions of both the features and the target. This will give me an idea of whether a class imbalance exists (which I suspect there does), and how the feature data is distributed within that imbalance. I'll start by getting a count of the records, based on the `'MONTH'` and `'DAY_OF_WEEK'` features. I plot these value counts, but I divide them based on the target variable - labeling the bars as `'Not Delayed'` or `'Delayed'`.

In [14]:
```
# Break down the records by month
data2['MONTH'].value_counts().sort_index()
```

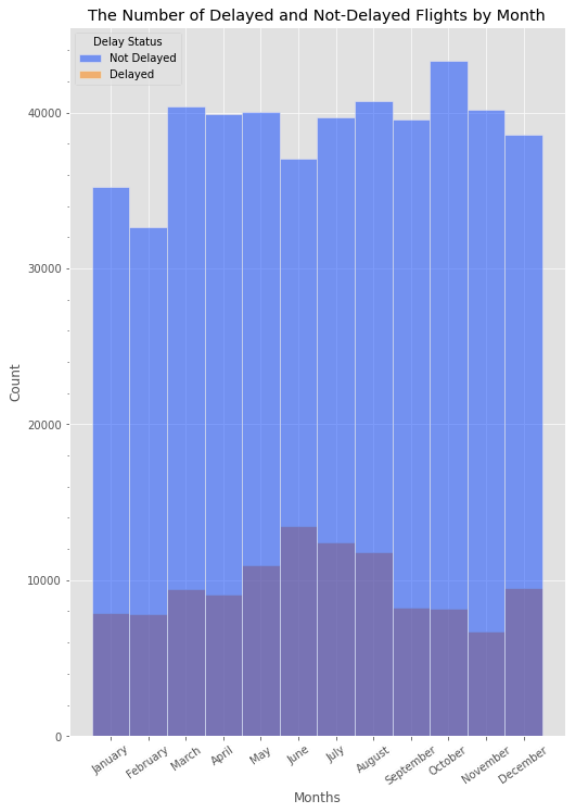
Out[14]:
```
1     43189
2     40457
3     49824
4     49038
5     51038
6     50529
7     52122
8     52538
9     47786
10    51477
11    46880
12    48061
Name: MONTH, dtype: int64
```

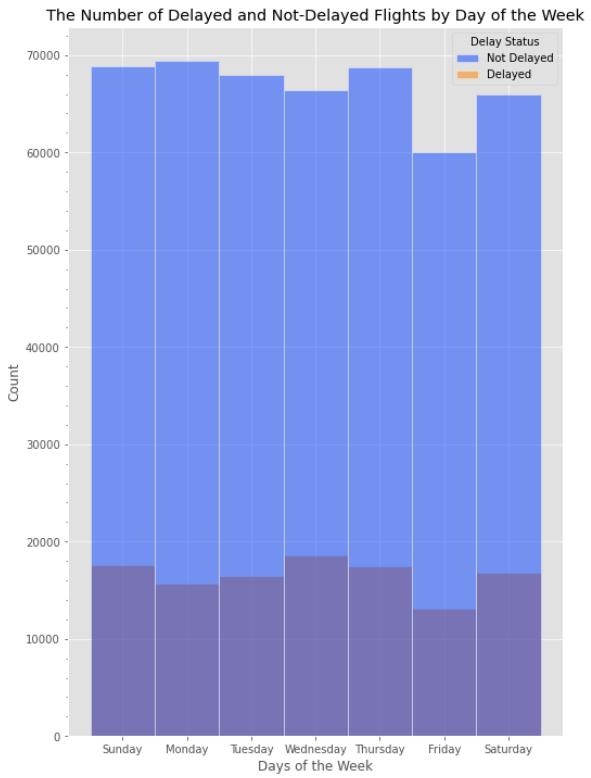
In [15]:
```
# Break down the records by days of the week
data2['DAY_OF_WEEK'].value_counts().sort_index()
```

Out[15]:
```
1    86421
2    85043
3    84471
4    84973
5    86153
6    73156
7    82722
Name: DAY_OF_WEEK, dtype: int64
```

In [16]:
```
# Plot feature distributions

# Create month and days list objects
months = ['January', 'February', 'March', 'April', 'May', 'June',
          'July', 'August', 'September', 'October', 'November', 'December']
days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Sat

# Visualize the delay status by month
fig, ax = plt.subplots(1, 2, figsize=(18, 12))
plot1 = sns.histplot(data2, x='MONTH', hue='DELAYED', ax=ax[0], palette='brigh

# Change the legend labels
new_title = 'Delay Status'
plot1.legend_.set_title(new_title)
new_labels = ['Not Delayed', 'Delayed']
for t, l in zip(plot1.legend_.texts, new_labels):
    t.set_text(l)

# Add minor gridlines
minor_locator = AutoMinorLocator(5)
ax[0].yaxis.set_minor_locator(minor_locator)
ax[0].set_axisbelow(True)
plt.grid(which='both')
ax[0].tick_params(which="both", bottom=True)

# Change x-tick labels to months (written form) and rotate the labels
old_labels = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
ax[0].set_xticks(old_labels)
ax[0].set_xticklabels(months)
ax[0].tick_params(axis='x', labelrotation=35)
ax[0].set_xlabel("Months")
ax[0].set_title("The Number of Delayed and Not-Delayed Flights by Month")

# Visualize the delay status by days of the week
plot2 = sns.histplot(data2, x='DAY_OF_WEEK', hue='DELAYED', ax=ax[1], palette=

# Change the legend labels
new_title = 'Delay Status'
plot2.legend_.set_title(new_title)
new_labels = ['Not Delayed', 'Delayed']
for t, l in zip(plot2.legend_.texts, new_labels):
    t.set_text(l)

# Add minor gridlines
minor_locator = AutoMinorLocator(5)
ax[1].yaxis.set_minor_locator(minor_locator)
ax[1].set_axisbelow(True)
plt.grid(which='both')

# Change x-tick labels to days (written form)
old_labels = [1, 2, 3, 4, 5, 6, 7]
ax[1].set_xticks(old_labels)
ax[1].set_xticklabels(days)
ax[1].set_xlabel("Days of the Week")
```

```
ax[1].set_title("The Number of Delayed and Not-Delayed Flights by Day of the W
```





Based on the visuals above, I see that there definitely exists a class imbalance in the dataset. In regard to the features, the first plot suggests that the summer months tend to see the highest number of delays, while the colder months seem to see less. In the second plot, the delays are highest on Wednesdays; although, the data seems to have low variation between the days of the week.

The next few features are the `'DEP_TIME_BLK'` and the `'DISTANCE_GROUP'` columns. The first feature lists the time blocks for United Airlines flights, ranging from midnight (`'0001'`) to 11:59pm, and separated into hour blocks (except for the first block). The second feature lists all 11 distance groups, where the first group travels the shortest distance, and the eleventh group travels the farthest.

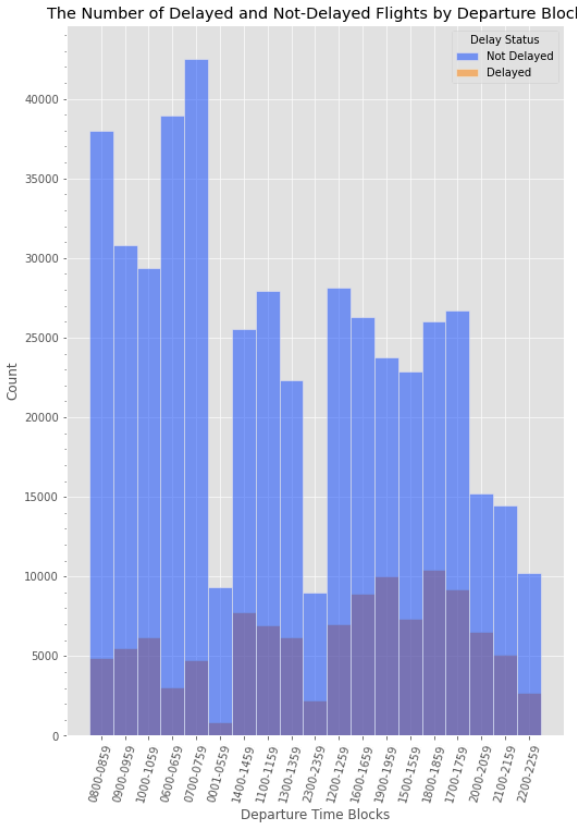
In [17]:
```
# Break down the records by departure time blocks
data2['DEP_TIME_BLK'].value_counts().sort_index()
```

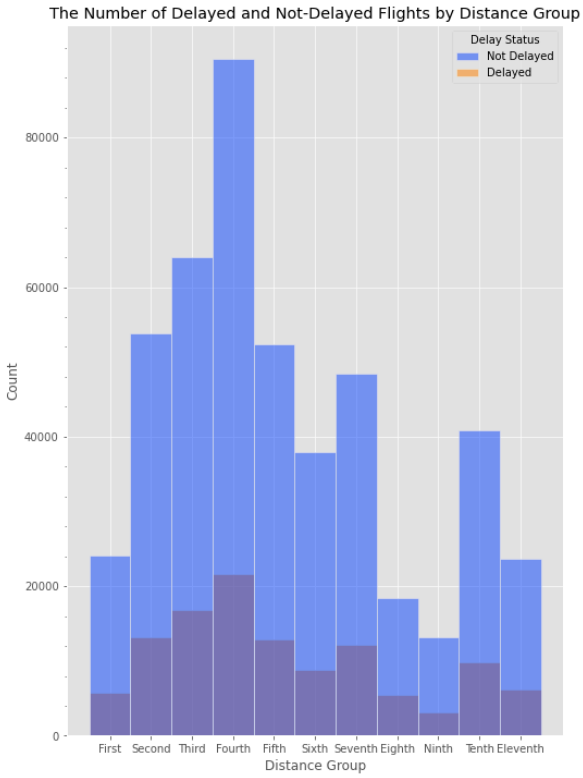
Out[17]:
```
0001-0559    10189
0600-0659    42020
0700-0759    47223
0800-0859    42906
0900-0959    36314
1000-1059    35549
1100-1159    34927
1200-1259    35124
1300-1359    28503
1400-1459    33245
1500-1559    30224
1600-1659    35246
1700-1759    35880
1800-1859    36416
1900-1959    33797
2000-2059    21716
2100-2159    19542
2200-2259    12919
2300-2359    11199
Name: DEP_TIME_BLK, dtype: int64
```

In [18]:
```
# Break down the records by distance group
data2['DISTANCE_GROUP'].value_counts().sort_index()
```

Out[18]:
```
1      29818
2      66963
3      80728
4     112096
5      65293
6      46784
7      60650
8      23843
9      16293
10     50679
11     29792
Name: DISTANCE_GROUP, dtype: int64
```

In [19]:
```
# Plot feature distributions
fig, ax = plt.subplots(1, 2, figsize=(18, 12))

# Visualize the delay status by departure time block
plot3 = sns.histplot(data2, x='DEP_TIME_BLK', hue='DELAYED', ax=ax[0], palette

# Change legend labels
new_title = 'Delay Status'
plot3.legend_.set_title(new_title)
new_labels = ['Not Delayed', 'Delayed']
for t, l in zip(plot3.legend_.texts, new_labels):
    t.set_text(l)

# Add minor gridlines
minor_locator = AutoMinorLocator(5)
ax[0].yaxis.set_minor_locator(minor_locator)
ax[0].set_axisbelow(True)
plt.grid(which='both')

# Rotate x-tick labels
ax[0].tick_params(axis='x', labelrotation=75)
ax[0].tick_params(which="both", bottom=True)
ax[0].set_xlabel("Departure Time Blocks")
ax[0].set_title("The Number of Delayed and Not-Delayed Flights by Departure Bl

# Visualize the delay status by distance group
plot4 = sns.histplot(data2, x='DISTANCE_GROUP', hue='DELAYED', ax=ax[1], palet

# Change legend labels
new_title = 'Delay Status'
plot4.legend_.set_title(new_title)
new_labels = ['Not Delayed', 'Delayed']
for t, l in zip(plot4.legend_.texts, new_labels):
    t.set_text(l)

# Add minor gridlines
minor_locator = AutoMinorLocator(5)
ax[1].yaxis.set_minor_locator(minor_locator)
ax[1].set_axisbelow(True)
plt.grid(which='both')

# Change x-tick labels to distance group (written form)
num_xlabels = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
str_xlabels = ['First', 'Second', 'Third', 'Fourth', 'Fifth', 'Sixth', 'Sevent
ax[1].set_xticks(num_xlabels)
ax[1].set_xticklabels(str_xlabels)
ax[1].set_xlabel("Distance Group")
ax[1].set_title("The Number of Delayed and Not-Delayed Flights by Distance Gro
```

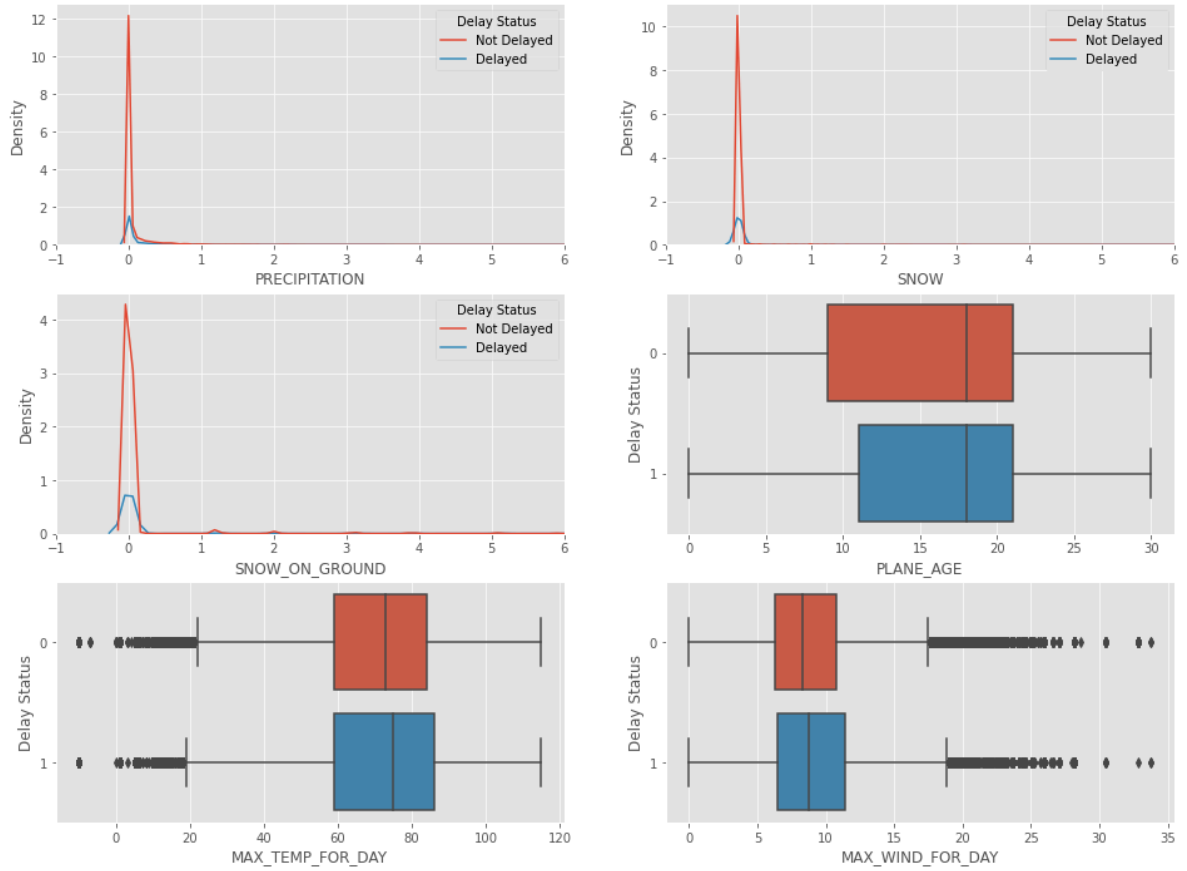




From what I see, the later time blocks (i.e., starting at 4pm to about 8pm) face the most delays. However, in the second visual, the third and fourth distance groups experience more delays, which is far less than those in higher distance groups.

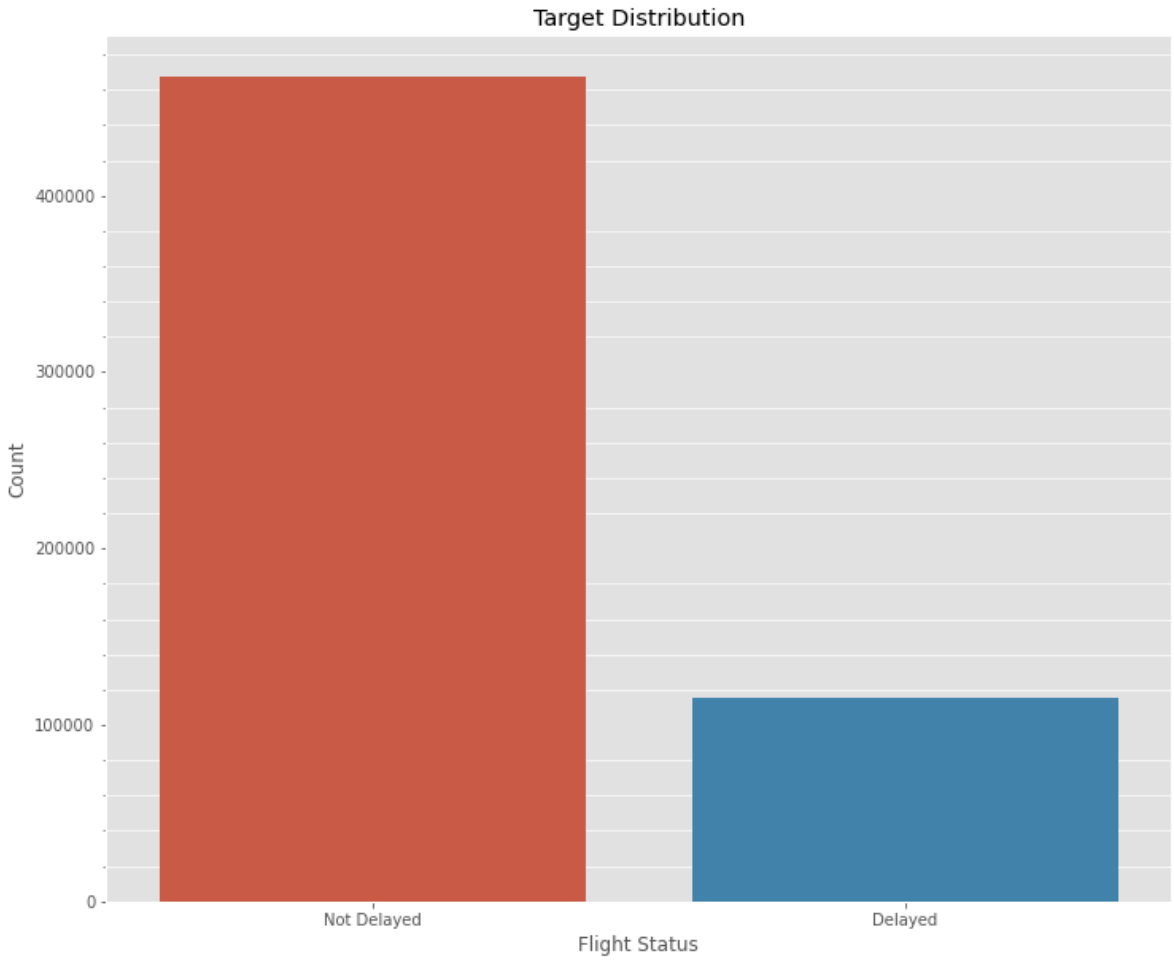
To be able to plot the rest of the features, I use both kdeplots and boxplots because the data within those columns are numerical and continuous in nature. I see that all six graphs contain outliers that affect the distribution one way or another, as well.

In [20]:
```
# Plot the remaining continuous variables with the delay status
fig, ax = plt.subplots(3, 2, figsize=(16, 12))
kde1 = sns.kdeplot(data=data2, x="PRECIPITATION", hue="DELAYED", ax=ax[0,0])
kde2 = sns.kdeplot(data=data2, x="SNOW", hue="DELAYED", ax=ax[0,1])
kde3 = sns.kdeplot(data=data2, x="SNOW_ON_GROUND", hue="DELAYED", ax=ax[1,0])
box1 = sns.boxplot(data=data2, x="PLANE_AGE", y="DELAYED", orient='h', ax=ax[1
box2 = sns.boxplot(data=data2, x="MAX_TEMP_FOR_DAY", y="DELAYED", orient='h',
box3 = sns.boxplot(data=data2, x="MAX_WIND_FOR_DAY", y="DELAYED", orient='h',

# Change the legend in the kdeplots for better comprehension
kdeplots = [kde1, kde2, kde3]
for plot in kdeplots:
    new_title = 'Delay Status'
    plot.legend_.set_title(new_title)
    new_labels = ['Not Delayed', 'Delayed']
    for t, l in zip(plot.legend_.texts, new_labels):
        t.set_text(l)
    plot.set_xlim(-1, 6)

# Change the y-axis label for the boxplots
boxplots = [box1, box2, box3]
for boxplot in boxplots:
    boxplot.set(ylabel='Delay Status')
```



It's time to examine the target, and confirm the class imbalance I mentioned earlier. First, I will plot my target as a bar plot, then as a pie chart. I plot both to show both the count and weight of each label in the dataset, respectively.

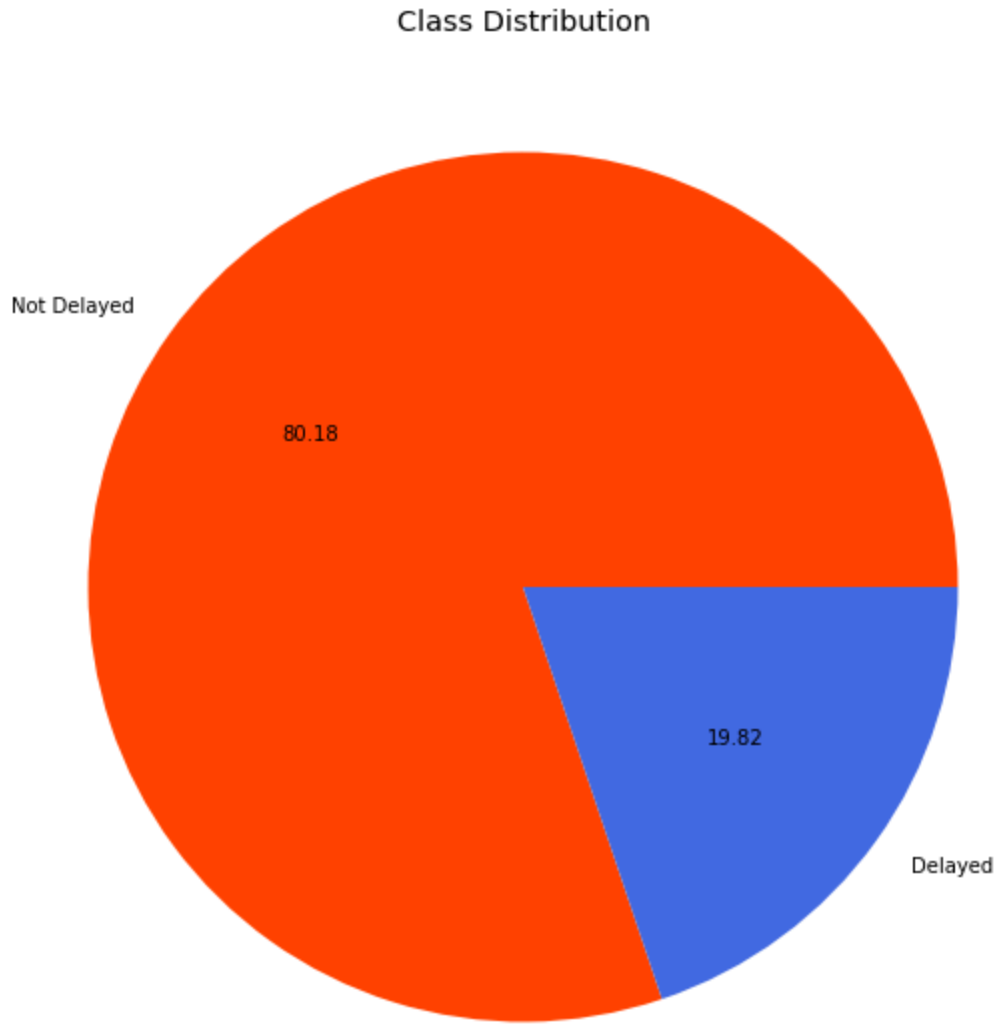
In [21]:
```
# Visualize the class (target) distribution
fig, ax = plt.subplots(figsize=(12,10))

# Add minor gridlines
minor_locator = AutoMinorLocator(5)
ax.yaxis.set_minor_locator(minor_locator)
ax.set_axisbelow(True)
plt.grid(which='both')

# Plot the target
sns.countplot(data=data2, x='DELAYED', orient='v')
ax.set_title('Target Distribution')

# Change x-tick labels and axis labels
old_labels = [0, 1]
new_xlabels = ['Not Delayed', 'Delayed']
ax.set_xticks(old_labels)
ax.set_xticklabels(new_xlabels)
ax.set_xlabel("Flight Status")
ax.set_ylabel("Count");
```



The bar plot above further confirms my speculation of a class imbalance - showing over 460,000 flights as `'Not Delayed'` and a little under 120,000 as `'Delayed'` flights. The pie plot below reinforces this observation, but displays the weight of each label in the data. I see that out of 600,000 flights, close to 20% are delayed - this is quite high considering the high

amount of flights taken in a year through United. According to the Bureau of Transportation, 18.72% of flights in 2019 were delayed (all carriers included), meaning United exceeded the

In [22]:
```
# Create a pie plot to further visualize the class distribution
fig, ax = plt.subplots(figsize=(12, 10))
data2['DELAYED'].value_counts().plot.pie(autopct='%.2f', title='Class Distribu
                                         labels=['Not Delayed', 'Delayed'], col
ax.yaxis.set_visible(False);
```



I feel I've gained sufficient insight into how my data is structured and distributed; so, now it's time to begin modeling and building my algorithm.

Further Preprocessing:

In order to properly model my dataframe, I need to encode the categorical features into quanitative data. I use pandas's `get_dummies` function to encode my five categorical columns into multiple, numerical columns; then list the first 5 records to confirm the transformation.

In [23]:
```
# Encode categorical features
cols_to_encode = ['MONTH', 'DAY_OF_WEEK', 'DEP_TIME_BLK', 'DISTANCE_GROUP', 'D
data2_enc = pd.get_dummies(data2, columns=cols_to_encode)
```

In [24]:
```
# List the first five records to confirm the transformation
data2_enc.head()
```

Out[24]:

	PLANE_AGE	PRECIPITATION	SNOW	SNOW_ON_GROUND	MAX_TEMP_FOR_DAY	MAX_WIN
21	6	0.0	0.0	0.0	65.0	
22	22	0.0	0.0	0.0	65.0	
23	3	0.0	0.0	0.0	65.0	
24	19	0.0	0.0	0.0	65.0	
25	4	0.0	0.0	0.0	65.0	

5 rows × 140 columns

The dataset above will serve as my final dataframe, but I will also divide it into a training, testing and validation set. I will split the data 75/25, with 25% of the dataset reserved for the test set, and a seed set for reproducibility. Now, I will split the test data even further (by half, actually) into a smaller test set and a newly-formed validation set to iterate through my modeling process. I check the shape multiple times to be sure the data stays intact.

In [25]:
```
# Split the dataset into training and testing sets
y = data2_enc['DELAYED']
X = data2_enc.drop('DELAYED', axis=1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.25, rando
X_train.shape
```

Out[25]:
```
(437204, 139)
```

In [26]:
```
# Split the test dataset in half to create a validation dataset
X_test, X_val, y_test, y_val = train_test_split(X_test, y_test, test_size=.5,
X_val.shape
```

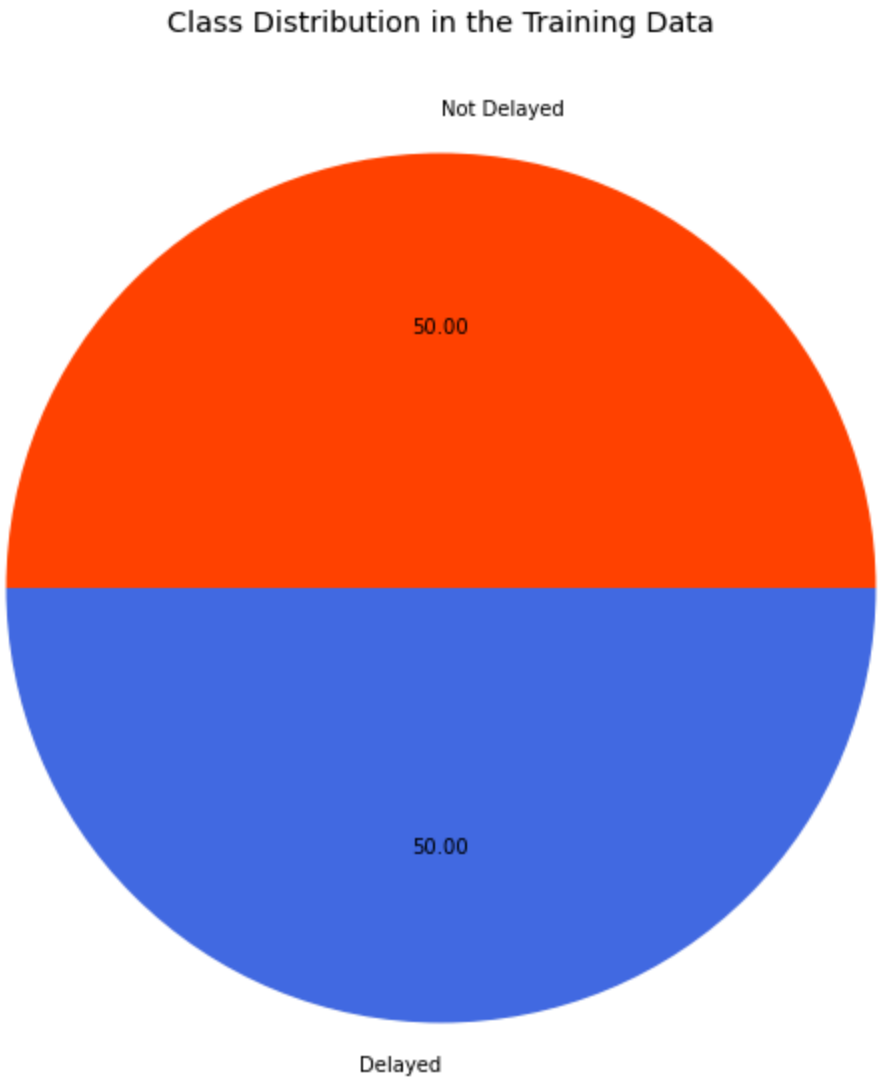
Out[26]:
```
(72868, 139)
```

Given the initial class imbalance, I need to apply a sampling technique - specifically, oversampling (SMOTE) - to augment the minority class (`'Delayed'`) in my target to an even ratio with the majority class (`'Not Delayed'`). I use the `Counter` function to show this change. I also plot the pie chart to display the new class weights, but this time expecting an even ratio.

In [27]:
```
# Count the number of classes before oversampling
counter = Counter(y_train)
print('Before sampling: ', counter)

# Instantiate the SMOTE function and oversample the training data
smt = SMOTE(sampling_strategy=1, random_state=seed)
X_train_res, y_train_res = smt.fit_resample(X_train, y_train)

# Count the number of classes after oversampling
counter_res = Counter(y_train_res)
print('After sampling: ', counter_res)
```

```
Before sampling:  Counter({0: 350513, 1: 86691})
After sampling:  Counter({0: 350513, 1: 350513})
```

In [28]:
```
# Visualize the balanced class distribution
fig, ax = plt.subplots(figsize=(12, 10))
y_train_res.value_counts().plot.pie(autopct='%.2f', title='Class Distribution
                                                labels=['Not Delayed', 'Delayed'], col
ax.yaxis.set_visible(False);
```



As shown above, the SMOTE method worked! To see just how much my training data has grown, I print the shape of the training data below. I see that the dataset now has **701,026 observations** to train my model on, and that the features remained intact. This increase in observations should improve my modeling performance and help me avoid any issues or controversy in results. The downside to this is the long run times I'm sure to experience due to my computational constraints.

In [29]:
```
# Print the balanced training data's shape
X_train_res.shape
```

Out[29]:
```
(701026, 139)
```

Modeling

Now it's time to begin modeling! I will create a few baseline models, to start, then cross-validate 4 different classifiers, and evaluate them based on precision, recall, and f1 scores. As a reminder, my main focus is to improve precision, which effects the number of false positive predictions. The false positive count, in this case, represents the amount of on-time flights predicted as delays. This could be detrimental if not addressed properly; the model could spread misinformation, and cause passengers to miss their on-time flights. This would lead to further client disapproval, and hurt United's sales and reputation. However, I will still track the recall score, and the f1 score, which is the harmonic mean of precision and recall. The baseline model with the best trio of average metric scores will be selected to undergo hyperparameter tuning and further evaluation.

Baseline Modeling:

I will begin by using scikit-learn's `make_pipeline` function in order to pass a scaler and a classifier into my pipeline(s). I use this function over the library's long form version to avoid naming the estimators. I cross-validate these pipelines in 3 folds, then take the average of each score and print them for each classifier.

In [30]:
```
# Make two lists, one containing classifier names and the other containing the
classifier_names = ['Decision Tree', 'Random Forest', 'AdaBoost', 'Gradient Bo
classifiers = [DecisionTreeClassifier(random_state=seed), RandomForestClassifi
               AdaBoostClassifier(random_state=seed), GradientBoostingClassifi

# Loop through the two lists to create a pipeline, cross-validate the model, a
for name, classifier in zip(classifier_names, classifiers):
    pipe = make_pipeline((MinMaxScaler()),
                         (classifier))
    score = cross_validate(pipe, X_train_res, y_train_res, cv=3, scoring=['pre
    model_scores = []
    keys = ['test_precision', 'test_recall', 'test_f1']
    for key in keys:
        model_scores.append(np.mean(score[key]))
    print("Validation precision, recall and f1 scores for {}: {}\n".format(nam
```

```
Validation precision, recall and f1 scores for Decision Tree: [0.823074189016
5016, 0.7770016943473458, 0.7616622487592548]

Validation precision, recall and f1 scores for Random Forest: [0.911645704981
1837, 0.7696011907997504, 0.7838231443353849]

Validation precision, recall and f1 scores for AdaBoost: [0.825750469687249,
0.6985395193651334, 0.7139101078676141]

Validation precision, recall and f1 scores for Gradient Boosting: [0.80721267
15505892, 0.7331089008313593, 0.7262921242468844]
```

Based on the precision scores listed above, it seems that the Random Forest classifier outperforms the field, which eliminates those models from further iteration. The Random Forest classifier is 91.2% precise, while the other scores don't even break 90%! Fortunately, I can use the other two scores (recall and f1) to support my decision. Now, the Random Forest classifier's recall score is slightly lower than the Decision Tree's (0.769 vs. 0.777, respectively), but that 0.008 difference seems negligible enough to select the former as the best baseline model, especially when its f1 score is higher than the Decision Tree's score (0.784 vs. 0.762).

91.2% validation precision is fantastic as a baseline, but also could be a sign that my model is overfitting. My next step will be trying to improve this start with some hyperparameter tuning.

Tuning the "Best" Baseline Model:

In order to keep in line with the business objectives, I need to set the `scoring` parameter to the three metrics I have been tracking so far. I will create a dictionary object that holds these three scores, and pass it into the `GridSearchCV` function. I will also pass a pipeline (long form this time) into the function, that will apply `MinMaxScaler` and my chosen classifier over the folds (in this case, 3) from the exhaustive grid search process. Because I'm using scikit-learn's long form pipeline function, it's vital I name each estimator in the steps parameter. My last step before fitting the grid search object is to create a parameter grid for the function to circulate

through. It's important I don't forget to put the classifier name I choose as a prefix followed by two underscores before each parameter name in order to avoid error. Once I've set all the relevant parameters, I fit the grid search object.

> Note: Due to the number of fits, and depending on the computational

In [31]:
```
# Create a dictionary of relevant scoring metrics
scorers = {'precision_score': make_scorer(precision_score),
           'recall_score': make_scorer(recall_score),
           'f1_score': make_scorer(f1_score)}

# Create a Pipeline with a scaler and classifier step
rf_pipe = Pipeline(steps=[('scaler', MinMaxScaler()),
                          ('classifier', RandomForestClassifier(random_state=s

# Create a parameter grid for the Random Forest Classifier
rf_param_grid = {'classifier__n_estimators': [50, 100, 125],
                 'classifier__max_depth': [None, 10, 20],
                 'classifier__min_samples_leaf': [5, 10, 15]}

# Instantiate the GridSearchCV function with the 'refit' parameter set to 'pre
rf_gridsearch = GridSearchCV(rf_pipe, rf_param_grid, scoring=scorers,
                             refit='precision_score', verbose=3, cv=3)
```

In [32]:
```
# Fit the GridSearchCV object
rf_gridsearch.fit(X_train_res, y_train_res)
```

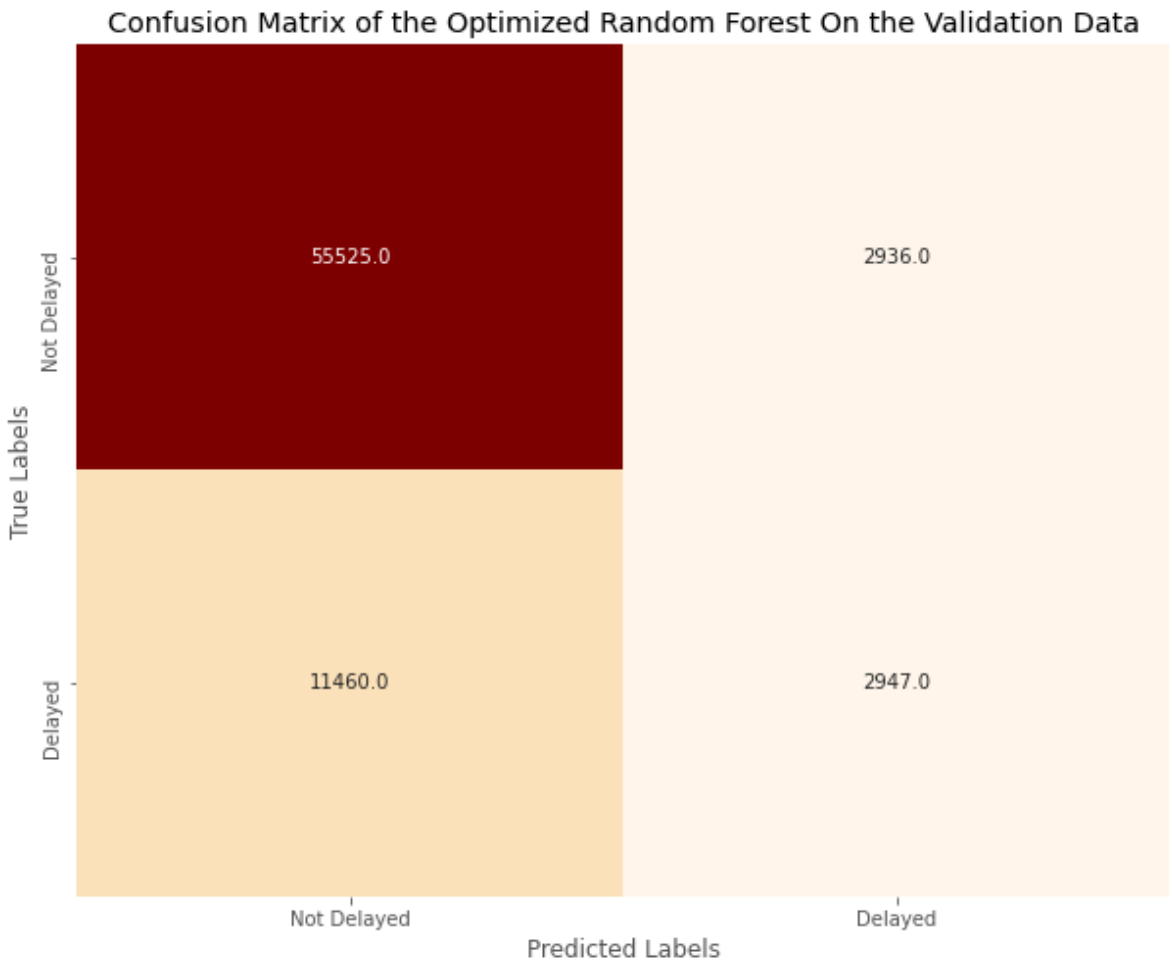
```
Fitting 3 folds for each of 27 candidates, totalling 81 fits
[CV] classifier__max_depth=None, classifier__min_samples_leaf=5, classifie
r__n_estimators=50

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent wo
rkers.

[CV]  classifier__max_depth=None, classifier__min_samples_leaf=5, classifi
er__n_estimators=50, f1_score=0.430, precision_score=0.977, recall_score=
0.276, total= 1.4min
[CV] classifier__max_depth=None, classifier__min_samples_leaf=5, classifie
r__n_estimators=50

[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:  1.4min remaining:
0.0s

[CV]  classifier__max_depth=None, classifier__min_samples_leaf=5, classifi
er__n_estimators=50, f1_score=0.929, precision_score=0.902, recall_score=
0.957, total= 1.6min
[CV] classifier__max_depth=None, classifier__min_samples_leaf=5, classifie
```

Now that the grid search has fully run, I can use the `best_params_` and `best_score_` attributes to find just that - the best parameters found through my grid search and the best cross-validated score of the best estimator, respectively. I see that my grid search determined 100 estimators and a minimum number of 5 samples per leaf in the Random Forest classifier

In [33]:
```
# Print the 'best' parameters and the best score for the model
print('Best params for Random Forest Classifier refit for {}:'.format('precisi
print(rf_gridsearch.best_params_)
print('\n')
print('Best score for Random Forest Classifier refit for {}:'.format('precisio
print(rf_gridsearch.best_score_)
```

```
Best params for Random Forest Classifier refit for precision_score:
{'classifier__max_depth': None, 'classifier__min_samples_leaf': 5, 'classifie
r__n_estimators': 100}


Best score for Random Forest Classifier refit for precision_score:
0.9305787737258301
```

In [34]:
```
# Plot the confusion matrix
y_hat_val = rf_gridsearch.predict(X_val)
plt.figure(figsize=(10, 8))
sns.heatmap(confusion_matrix(y_val, y_hat_val), annot=True,
            fmt='.1f', xticklabels=['Not Delayed', 'Delayed'],
            yticklabels=['Not Delayed', 'Delayed'], cmap='OrRd', cbar=False)
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title('Confusion Matrix of the Optimized Random Forest On the Validation D
plt.show();
```



From initial observation, I can see this model needs more training, especially if my hope is to reduce the false positive count while also predicting delays. The false positive count is 2,936, which is a decent start; however, there are 11,460 predictions misclassified as not delayed, which is a problem. I print the classification report below to get a full look into the grid search's metric scores, including recall and f1.

In [35]:
```
# Print the classification report
print(classification_report(y_val, y_hat_val))
```

```
              precision    recall  f1-score   support

           0       0.83      0.95      0.89     58461
           1       0.50      0.20      0.29     14407

    accuracy                           0.80     72868
   macro avg       0.66      0.58      0.59     72868
weighted avg       0.76      0.80      0.77     72868
```

I'll use the visual and scores above as my baseline to compare against the next step of my modeling phase: deep learning.

Deep Learning:

Before I begin creating neural networks to train my data on, it's important I normalize the input datasets so that they are all on the same scale; otherwise, the models will generate flawed results. I will use `MinMaxScaler` once again, and transform all three input datasets, but only fitting on the training data.

In [36]:
```
# Use the MinMaxScaler function to scale the data
cols = X_train_res.columns

mmscaler = MinMaxScaler()
X_train_scaled = mmscaler.fit_transform(X_train_res)
X_val_scaled = mmscaler.transform(X_val)
X_test_scaled = mmscaler.transform(X_test)

X_train = pd.DataFrame(X_train_scaled, columns=cols)
X_val = pd.DataFrame(X_val_scaled, columns=cols)
X_test = pd.DataFrame(X_test_scaled, columns=cols)

# Check & reconfirm the training data's shape
X_train.shape
```

Out[36]:
```
(701026, 139)
```

I see from the last line of code that the shape of the training data has remained intact, and that my data is ready to be modeled. To begin, I build a small baseline neural network - specifically, with 2 hidden layers and 1 output layer). I also set some regularization to help counter any potential overfitting to start.

Seeing as how this is a classification problem, it's important I select loss and optimizer functions that work well in binary classification. From my knowledge, binary cross-entropy and Adam optimization work best in these situations, so I pass them along with a metrics parameter in the compiler function. The summary below shows the number of parameters that exist (and will be trained) in the network, which in this case is over 10,000 parameters!

In [37]:
```
# Create a small-layered baseline neural network
neural_network = Sequential()
neural_network.add(Dense(64, activation='relu',
                         kernel_regularizer=regularizers.l2(0.01), input_sh
neural_network.add(Dense(16, activation='relu'))
neural_network.add(Dense(1, activation='sigmoid'))

# Compile the baseline network
neural_network.compile(loss='binary_crossentropy', optimizer='adam', metrics=[

# List a summary of the baseline network
neural_network.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 64)                8960
_________________________________________________________________
dense_1 (Dense)              (None, 16)                1040
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 17
=================================================================
Total params: 10,017
Trainable params: 10,017
Non-trainable params: 0
_________________________________________________________________
```

I fit the baseline network to the training data, and run it for 32 epochs alongside the validation data.

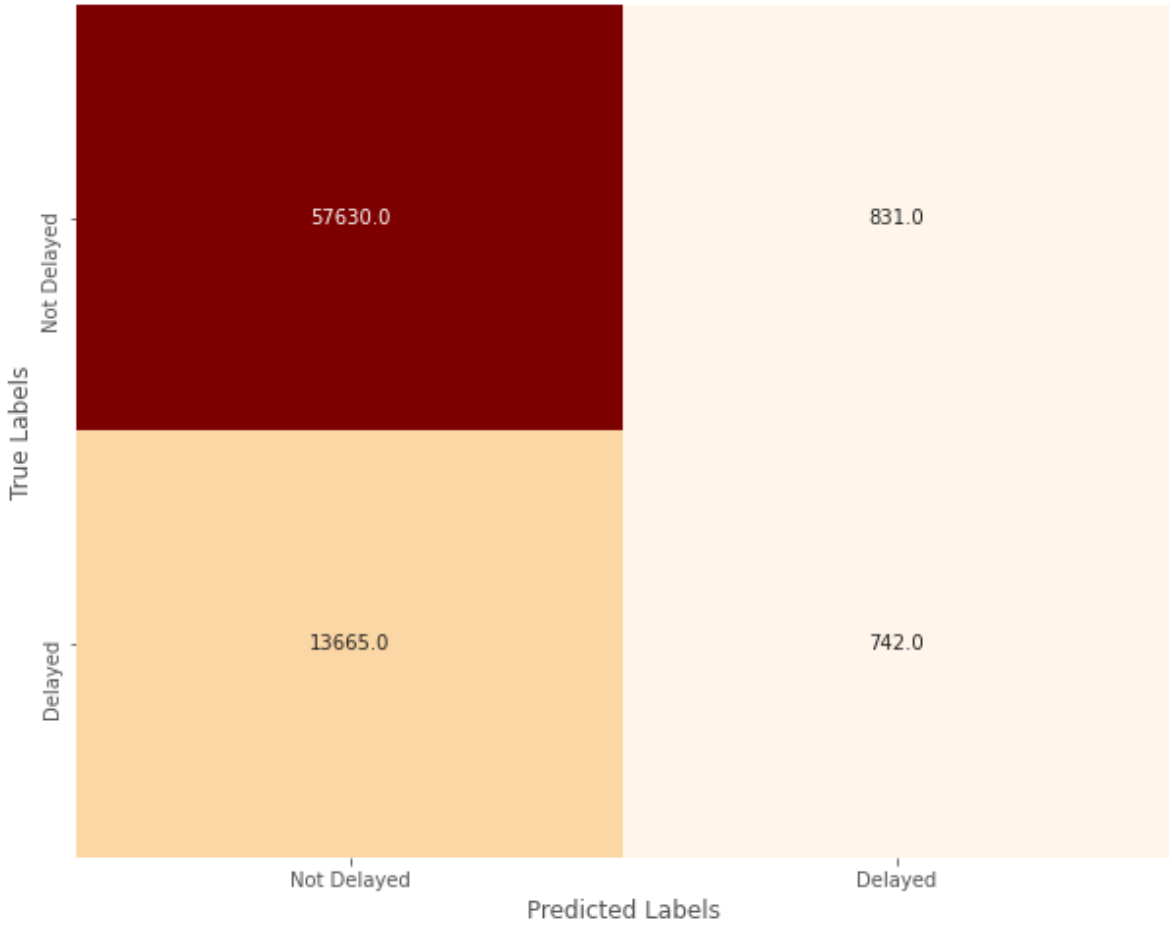
In [38]:
```
# Fit the baseline network
baseline = neural_network.fit(X_train, y_train_res, epochs=32, batch_size=256,
                                        validation_data=(X_val, y_val))
```

```
Epoch 1/32
2739/2739 [==============================] - 8s 3ms/step - loss: 0.3857 -
accuracy: 0.8540 - val_loss: 0.5059 - val_accuracy: 0.7974
Epoch 2/32
2739/2739 [==============================] - 9s 3ms/step - loss: 0.3403 -
accuracy: 0.8636 - val_loss: 0.5010 - val_accuracy: 0.7996
Epoch 3/32
2739/2739 [==============================] - 8s 3ms/step - loss: 0.3363 -
accuracy: 0.8647 - val_loss: 0.4968 - val_accuracy: 0.7963
Epoch 4/32
2739/2739 [==============================] - 8s 3ms/step - loss: 0.3334 -
accuracy: 0.8650 - val_loss: 0.4927 - val_accuracy: 0.8004
Epoch 5/32
2739/2739 [==============================] - 9s 3ms/step - loss: 0.3314 -
accuracy: 0.8657 - val_loss: 0.4955 - val_accuracy: 0.8023
Epoch 6/32
2739/2739 [==============================] - 9s 3ms/step - loss: 0.3295 -
accuracy: 0.8664 - val_loss: 0.5090 - val_accuracy: 0.7860
Epoch 7/32
```

Once all the epochs have been run through, I evaluate the network on both the training and validation data to get a look into the baseline networks accuracy and loss. Then I generate predictions and map the values to plot a confusion matrix, and get a further look into how the model performed.

In [39]:
```
# Evaluate the loss and accuracy scores for the training and validation datase
print(f'Training data results:\n{neural_network.evaluate(X_train_res, y_train_
print('\n')
print(f'Validation data results:\n{neural_network.evaluate(X_val, y_val)}')

# Generate predictions and "round" the values
baseline_preds = neural_network.predict(X_val)
baseline_preds[baseline_preds > 0.5] = 1
baseline_preds[baseline_preds < 0.5] = 0

# Plot a confusion matrix of the validation data
plt.figure(figsize=(10, 8))
sns.heatmap(confusion_matrix(y_val, baseline_preds), annot=True,
           fmt='.1f', xticklabels=['Not Delayed', 'Delayed'],
           yticklabels=['Not Delayed', 'Delayed'], cmap='OrRd', cbar=False)
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()
```

```
21908/21908 [==============================] - 19s 870us/step - loss: 8.9353
- accuracy: 0.5019
Training data results:
[8.935270309448242, 0.5019385814666748]


2278/2278 [==============================] - 2s 1ms/step - loss: 0.4834 - acc
uracy: 0.8011
Validation data results:
[0.4834330677986145, 0.8010649681091309]
```



I see above that the validation accuracy is much higher than the training data's score (80.1% vs. 50.2%, respectively). Although this is only a baseline network, that is still very poor performance, and implies my network needs to be tuned and retrained. The confusion matrix above further suggests the model performs poorly. When compared to the tuned Random Forest classifier, the model above predicts less false positives, which is great; however, it is predicting far less true positives. So in actuality, it is not predicting true delayed flights as well as I'd like.

Below is the classification report for the baseline network to see how the model performed based on the three metrics I've been tracking thus far (precision, recall and f1 score). However, for simplicity's sake, I also create a function further below that will take in the true and predicted labels, and generate the three scores.

In [40]:
```
# Print the classification report
print(classification_report(y_val, baseline_preds))
```

```
              precision    recall  f1-score   support

           0       0.81      0.99      0.89     58461
           1       0.47      0.05      0.09     14407

    accuracy                           0.80     72868
   macro avg       0.64      0.52      0.49     72868
weighted avg       0.74      0.80      0.73     72868
```

In [41]:
```
# Create a function that will calculate the three relevant metric scores
def model_metrics(y_true, y_preds):

    precision = precision_score(y_true, y_preds)
    recall = recall_score(y_true, y_preds)
    f1 = f1_score(y_true, y_preds)

    print('Precision score:', round(precision * 100, 2),'%')
    print('Recall score:', round(recall * 100, 2),'%')
    print('F1 score:', round(f1 * 100, 2),'%')

# Run the function with the appropriate arguments passed in
model_metrics(y_val, baseline_preds)
```

```
Precision score: 47.17 %
Recall score: 5.15 %
F1 score: 9.29 %
```

It seems all three scores dropped when compared to the machine learning algorithms. The precision score is the only one to decrease slightly when compared to the Random Forest classifier (47.2% vs 50.0%). Conversely, the recall and f1 scores dropped greatly, falling under 10% in this iteration. This means the model's sensitivity, or ability to predict positive results, has decreased. Therefore, I will focus on improving this area while maintaining low false positive numbers.

I will create another neural network; however, it will have 3 hidden layers, instead of two (and the units will be adjusted, accordingly). I apply the same compiler parameters, and print the summary again to see how many more parameters I am training. This denser network will train 11,313 total parameters.

In [42]:
```
# Create another, more dense neural network
neural_network2 = Sequential()
neural_network2.add(Dense(64, activation='relu',
                          kernel_regularizer=regularizers.l2(0.01), input_sh
neural_network2.add(Dense(32, activation='relu'))
neural_network2.add(Dense(8, activation='relu'))
neural_network2.add(Dense(1, activation='sigmoid'))

# Compile the model
neural_network2.compile(loss='binary_crossentropy', optimizer='adam', metrics=

# List a summary of the model
neural_network2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 64)                8960
_________________________________________________________________
dense_4 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_5 (Dense)              (None, 8)                 264
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 9
=================================================================
Total params: 11,313
Trainable params: 11,313
Non-trainable params: 0
_________________________________________________________________
```

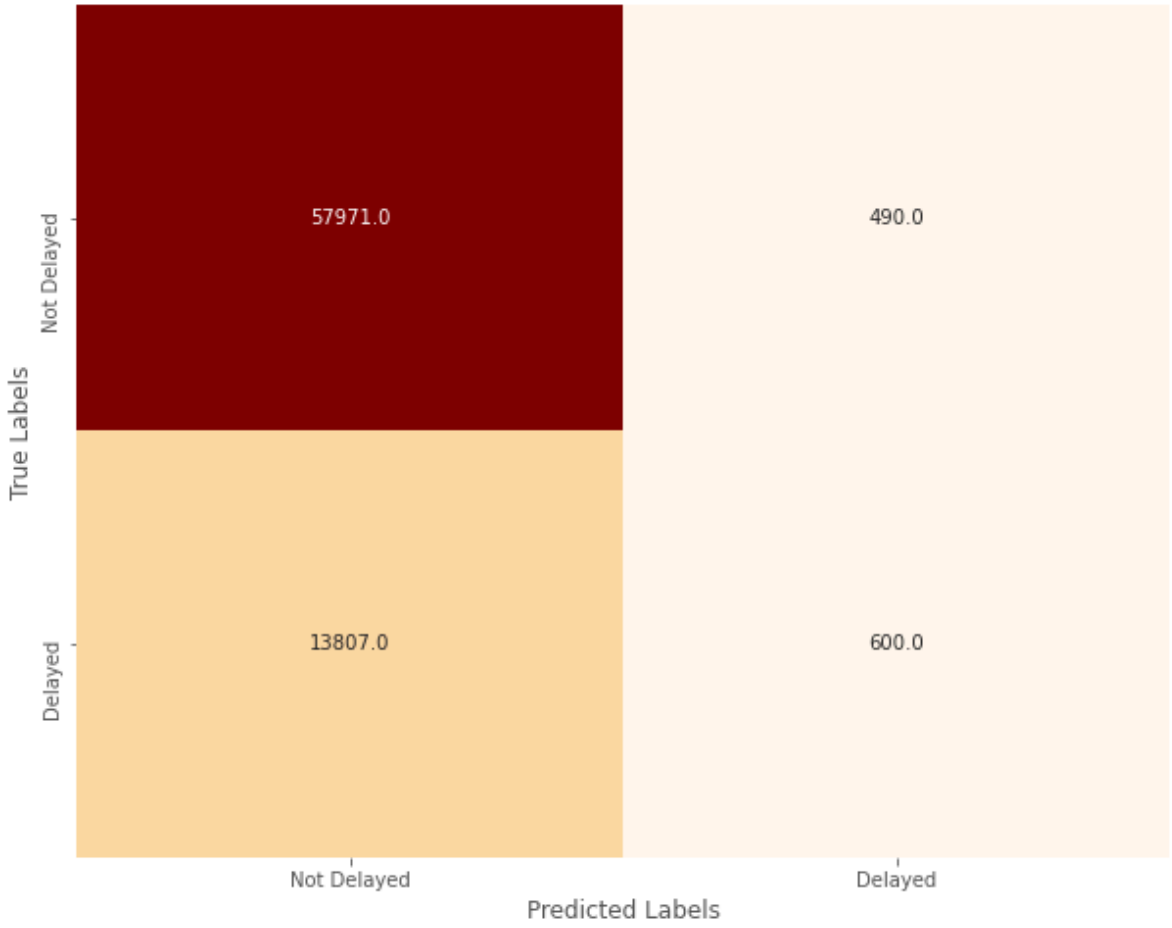
I fit the model this time to **50 epochs** in order to increase my training performance, and in the hope that my precision will increase as well. I will retain the batch size and evaluate on the validation data again.

In [43]:
```
# Fit the model
results = neural_network2.fit(X_train, y_train_res, epochs=50, batch_size=256,
                                    validation_data=(X_val, y_val))
```

```
accuracy: 0.8689 - val_loss: 0.4713 - val_accuracy: 0.8017
Epoch 45/50
2739/2739 [==============================] - 10s 4ms/step - loss: 0.3075 -
accuracy: 0.8690 - val_loss: 0.4695 - val_accuracy: 0.8033
Epoch 46/50
2739/2739 [==============================] - 10s 4ms/step - loss: 0.3075 -
accuracy: 0.8689 - val_loss: 0.4710 - val_accuracy: 0.8028
Epoch 47/50
2739/2739 [==============================] - 9s 3ms/step - loss: 0.3073 -
accuracy: 0.8690 - val_loss: 0.4698 - val_accuracy: 0.8036
Epoch 48/50
2739/2739 [==============================] - 9s 3ms/step - loss: 0.3076 -
accuracy: 0.8688 - val_loss: 0.4691 - val_accuracy: 0.8040
Epoch 49/50
2739/2739 [==============================] - 7s 3ms/step - loss: 0.3073 -
accuracy: 0.8690 - val_loss: 0.4715 - val_accuracy: 0.8021
Epoch 50/50
2739/2739 [==============================] - 9s 3ms/step - loss: 0.3074 -
accuracy: 0.8690 - val_loss: 0.4697 - val_accuracy: 0.8038
```

In [44]:
```
# Evaluate the loss and accuracy scores for the training and validation datase
print(f'Training data results:\n{neural_network2.evaluate(X_train_res, y_train
print('\n')
print(f'Validation data results:\n{neural_network2.evaluate(X_val, y_val)}')

# Generate predictions and "round" the values
y_preds2 = neural_network2.predict(X_val)
y_preds2[y_preds2 > 0.5] = 1
y_preds2[y_preds2 < 0.5] = 0

# Plot a confusion matrix of the validation data
plt.figure(figsize=(10, 8))
sns.heatmap(confusion_matrix(y_val, y_preds2), annot=True,
            fmt='.1f', xticklabels=['Not Delayed', 'Delayed'],
            yticklabels=['Not Delayed', 'Delayed'], cmap='OrRd', cbar=False)
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()
```

```
21908/21908 [==============================] - 22s 1ms/step - loss: 3.2982 -
accuracy: 0.4935
Training data results:
[3.2982168197631836, 0.49347811937332153]


2278/2278 [==============================] - 3s 1ms/step - loss: 0.4697 - acc
uracy: 0.8038
Validation data results:
[0.46969321370124817, 0.8037959337234497]
```



Based on the training and validation accuracies, I see that my model got worse! The numbers in my confusion matrix were worse as well. My false positive count was reduced from 831 to 490, which is opposite of what I aimed for, and will require more training. Unfortunately, my true positive count decreased as well, which is a problem. I print out the three metrics once again to get a better idea of my model's performance.

In [45]:
```
# Run the metrics function
model_metrics(y_val, y_preds2)
```

```
Precision score: 55.05 %
Recall score: 4.16 %
F1 score: 7.74 %
```

The three scores above tell me a few things. First, the precision score slightly increased to 55.1% and the remaining two scores decreased slightly, which is to be expected after seeing the true positive cases decrease. I will create one more neural network to try to improve my model's performance once again. Below I create another network, reverting back to my baseline architecture but with a smaller regularizer (0.001).

In [46]:
```
# Create a neural network similar to the baseline with a smaller regularizer
neural_network3 = Sequential()
neural_network3.add(Dense(64, activation='relu',
                          kernel_regularizer=regularizers.l2(0.001), input_s
neural_network3.add(Dense(16, activation='relu'))
neural_network3.add(Dense(1, activation='sigmoid'))

# Compile the baseline model
neural_network3.compile(loss='binary_crossentropy', optimizer='adam', metrics=

# List a summary of the model
neural_network3.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_7 (Dense)              (None, 64)                8960
_________________________________________________________________
dense_8 (Dense)              (None, 16)                1040
_________________________________________________________________
dense_9 (Dense)              (None, 1)                 17
=================================================================
Total params: 10,017
Trainable params: 10,017
Non-trainable params: 0
_________________________________________________________________
```

This iteration will differ from the other two networks I built by increasing the epochs to 128, setting a stopping condition with the `EarlyStopping` function, and saving that checkpoint as my "best model" with the `ModelCheckpoint` function. I will pass this stopping condition as I fit my model, evaluating it once again on my validation data.

In [47]:
```
# Create a file name object for the best model checkpoint
model_filepath = 'best_model.h5'

# Set early stopping and model checkpoint conditions
early_stopping = [EarlyStopping(monitor='val_loss', patience=8),
                  ModelCheckpoint(filepath=model_filepath, monitor='val_loss',
```

In [48]:
```
# Fit the model with more epochs
results2 = neural_network3.fit(X_train, y_train_res, epochs=128, batch_size=25
                                validation_data=(X_val, y_val))
```

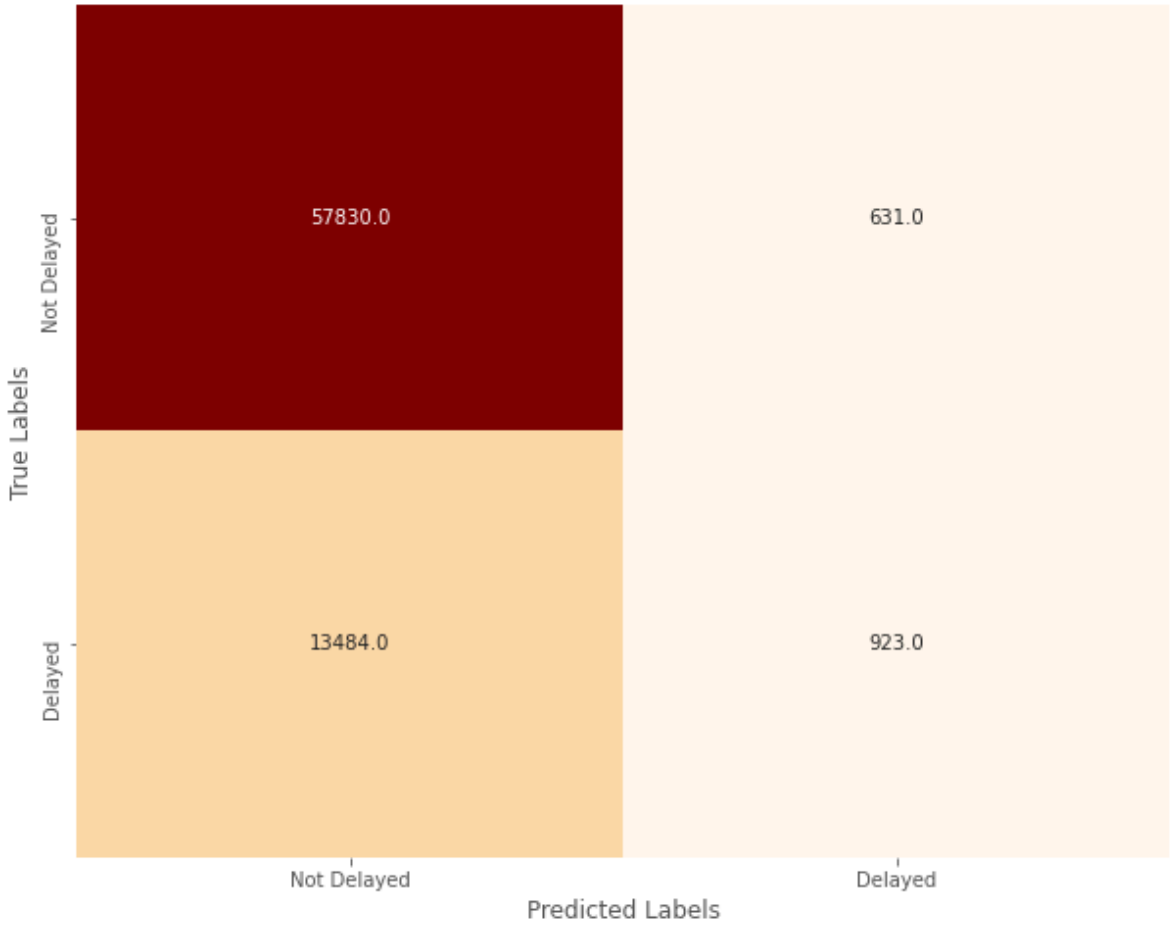
```
accuracy: 0.8728 - val_loss: 0.4621 - val_accuracy: 0.8061
Epoch 29/128
2739/2739 [==============================] - 9s 3ms/step - loss: 0.2986 -
accuracy: 0.8730 - val_loss: 0.4628 - val_accuracy: 0.8051
Epoch 30/128
2739/2739 [==============================] - 8s 3ms/step - loss: 0.2987 -
accuracy: 0.8728 - val_loss: 0.4695 - val_accuracy: 0.8014
Epoch 31/128
2739/2739 [==============================] - 8s 3ms/step - loss: 0.2986 -
accuracy: 0.8730 - val_loss: 0.4636 - val_accuracy: 0.8034
Epoch 32/128
2739/2739 [==============================] - 8s 3ms/step - loss: 0.2986 -
accuracy: 0.8728 - val_loss: 0.4625 - val_accuracy: 0.8052
Epoch 33/128
2739/2739 [==============================] - 8s 3ms/step - loss: 0.2983 -
accuracy: 0.8728 - val_loss: 0.4625 - val_accuracy: 0.8055
Epoch 34/128
2739/2739 [==============================] - 8s 3ms/step - loss: 0.2986 -
accuracy: 0.8730 - val_loss: 0.4621 - val_accuracy: 0.8057
```

I see that my model has stopped after 34 epochs! Now I will take that saved model, load it, and evaluate the model on the training and validation data to view its accuracy. I will also plot the confusion matrix, and see how the model performed.

In [49]:
```
# Load the best (saved) model
saved_model = load_model(model_filepath)

# Calculate the loss and accuracy scores for both training datasets
results_train = saved_model.evaluate(X_train, y_train_res)
print(f'Training Loss: {results_train[0]:.3} \nTraining Accuracy: {results_tra

print('----------')

results_val = saved_model.evaluate(X_val, y_val)
print(f'Validation Loss: {results_val[0]:.3} \nValidation Accuracy: {results_v
```

```
21908/21908 [==============================] - 20s 929us/step - loss: 0.2971
- accuracy: 0.8733
Training Loss: 0.297
Training Accuracy: 0.873
----------
2278/2278 [==============================] - 2s 919us/step - loss: 0.4620 - a
ccuracy: 0.8063
Validation Loss: 0.462
Validation Accuracy: 0.806
```

In [50]:
```
# Generate predictions and "round" the values
y_val_preds = saved_model.predict(X_val)
y_val_preds[y_val_preds > 0.5] = 1
y_val_preds[y_val_preds < 0.5] = 0

# Plot a confusion matrix of the validation data
plt.figure(figsize=(10, 8))
sns.heatmap(confusion_matrix(y_val, y_val_preds), annot=True,
            fmt='.1f', xticklabels=['Not Delayed', 'Delayed'],
            yticklabels=['Not Delayed', 'Delayed'], cmap='OrRd', cbar=False)
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()
```



This model has definitely improved since its baseline. Although the other metrics leave something to be desired, I can at least see that my model is improving, especially in its precision, and its true positive predictions. The false positive cases amount to 631, which makes this good enough to serve as my final model (which is luckily already saved). I will take another look at the metrics scores. I see that the model is 59.4% precise, has a recall of 6.41%, and its f1 score is at 11.57%.

In [51]:
```
# Calculate the relevant metrics
model_metrics(y_val, y_val_preds)
```

```
Precision score: 59.4 %
Recall score: 6.41 %
F1 score: 11.57 %
```

Evaluation

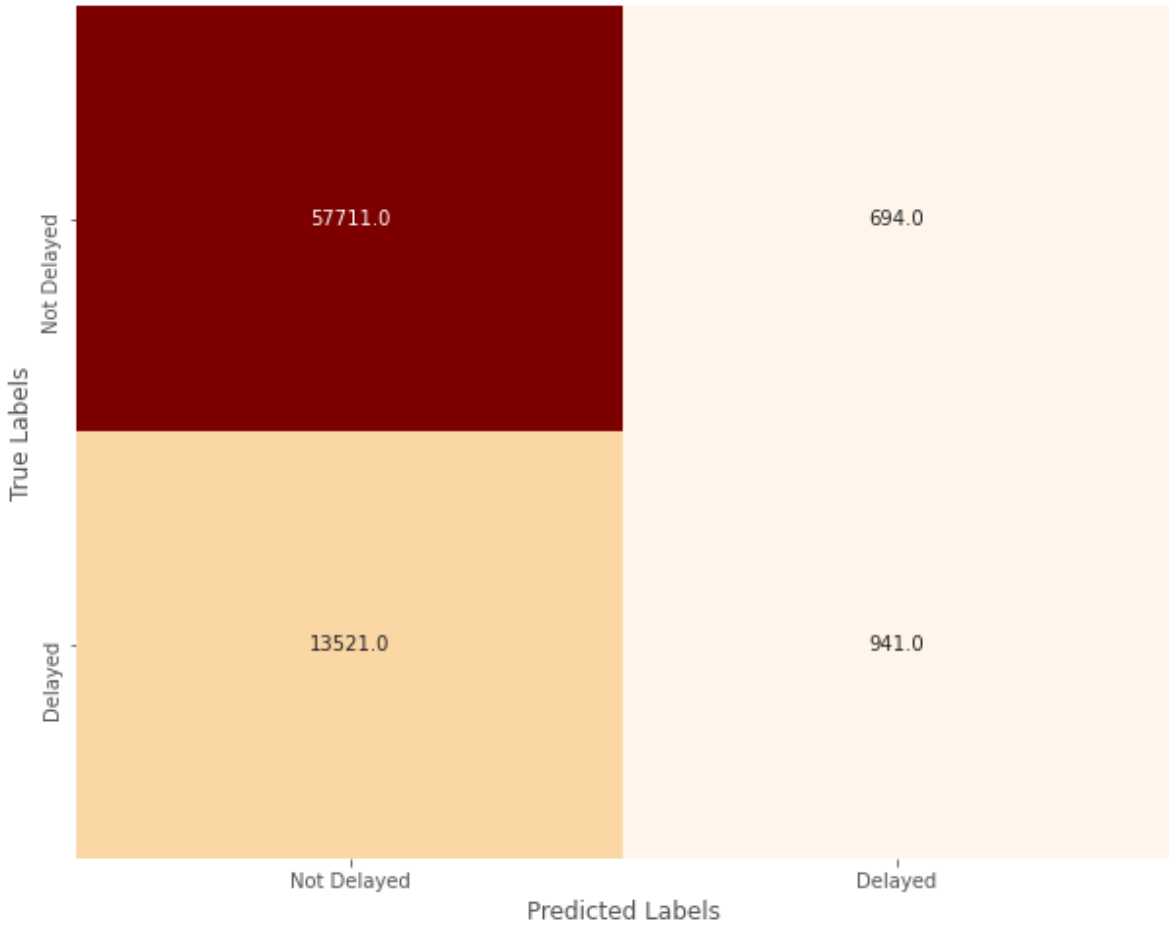
Now that I have my "best" model, I will see how it performs on my test data. I will generate the predictions, and print the three scores plus its accuracy to get a full idea of how efficient my model is. Fortunately, the model has been already saved and has been included in the repository as my means of deployment.

In [52]:
```
# Evaluate the test data and list the relevant metrics
results_test = saved_model.evaluate(X_test, y_test)
y_hat_test = saved_model.predict(X_test)
y_hat_test[y_hat_test > 0.5] = 1
y_hat_test[y_hat_test < 0.5] = 0
print('Generated {} predictions'.format(len(y_hat_test)))
print(f'Testing Loss: {results_test[0]:.3} \nTesting Accuracy: {results_test[1
model_metrics(y_test, y_hat_test)
```

```
2278/2278 [==============================] - 2s 940us/step - loss: 0.4633 - a
ccuracy: 0.8049
Generated 72867 predictions
Testing Loss: 0.463
Testing Accuracy: 0.805
Precision score: 57.55 %
Recall score: 6.51 %
F1 score: 11.69 %
```

My model's metrics are a semblance to how it performed on my validation data - the model is 57.6% precise, has a recall of 6.5%, and an f1 score of 11.7%. The good news is my model is consistent, and doesn't generate results wildly different from what I expected. However, there's major room for improvement.

In [53]:
```
# Plot a confusion matrix of the test data
plt.figure(figsize=(10, 8))
sns.heatmap(confusion_matrix(y_test, y_hat_test), annot=True,
            fmt='.1f', xticklabels=['Not Delayed', 'Delayed'],
            yticklabels=['Not Delayed', 'Delayed'], cmap='OrRd', cbar=False)
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show();
```



The confusion matrix above shows similar results to the last plot - there are 694 false positive predictions, which is still better than the past models I built and evaluated. The number of true positive cases increases as I trained my networks, which gives me some hope that the model can be improved to levels that are acceptable for mass deployment.

Conclusion

This analysis leads to the following conclusions:

1. The neural networks performed better than the machine learning algorithms I tested in terms of precision, and is the path I will explore further as I aim to improve my performance.
2. The model is 57.6% precise when testing and classifying flights as delayed or not delayed.

Limitations/Further Work

This project is limited in a few ways. First and foremost, I built my models under heavy computational constraints. Given the nature of the data, it is necessary to train models on computers that can process large and full datasets in quicker time. For example, the grid search I performed earlier took almost 3 hours to run, which hinders me from further testing and modifications. Another drawback is the class imbalance. I applied sampling methods to augment the minority class, which gave me more data to work with, but also increased my run times. Therefore, gathering more data that fall in the minority class would greatly improve my precision. Lastly, and this ties with my computational constraints, I could have used a wider range of hyperparameters to perform my grid search with. Unfortunately, with limited resources, it would have taken me hours or even days to fully perform this search. However, if I had, I could have found a better set of hyperparameters that would improve my metrics and predictions.

Further analyses could yield a more effective predictor, and possibly improve the algorithm's performance. Some possible courses of action I could take include:

1. Training my model with better, stronger computer(s).
2. Gathering more data with an emphasis on balancing the minority class to avoid sampling.
3. Evaluating on different metrics to see how it impacts my model's performance.

Sources

Link to original dataset: https://www.kaggle.com/datasets/threnjen/2019-airline-delays-and-cancellations/data?select=full_data_flightdelay.csv (https://www.kaggle.com/datasets/threnjen/2019-airline-delays-and-cancellations/data?select=full_data_flightdelay.csv)

Wall Street Journal annual airlines ranking report: https://www.wsj.com/lifestyle/travel/best-airlines-us-2023-36e9ea20 (https://www.wsj.com/lifestyle/travel/best-airlines-us-2023-36e9ea20)

Bureau of Transportation Statistics: https://www.transtats.bts.gov/HomeDrillChart.asp (https://www.transtats.bts.gov/HomeDrillChart.asp)

Federal Aviation Administration (FAA): https://aspm.faa.gov/aspmhelp/index/Types_of_Delay.html (https://aspm.faa.gov/aspmhelp/index/Types_of_Delay.html)