

Modeling V2

April 28, 2024

```
[386]: from IPython.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))
```

<IPython.core.display.HTML object>

```
[387]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import scipy
import sklearn
import os
from sklearn.model_selection import train_test_split
```

```
[388]: def convert_to_categorical(time_str):
    hour = int(time_str.split(':')[0])

    if 0 <= hour < 3:
        return 'Late Night'
    elif 3 <= hour < 6:
        return 'Early Morning'
    elif 6 <= hour < 9:
        return 'Morning'
    elif 9 <= hour < 12:
        return 'Late Morning'
    elif 12 <= hour < 15:
        return 'Noon'
    elif 15 <= hour < 18:
        return 'Afternoon'
    elif 18 <= hour < 21:
        return 'Evening'
    else:
        return 'Night'
```

```
[389]: data = pd.read_csv("./SYR_ORIGIN_WEATHER_IMPUTED.csv")
```

```
[390]: data.head()
```

```

[390]: Carrier Code Date (MM/DD/YYYY) Origin Airport Scheduled Arrival Time \
0      B6      2010-01-01      JFK      00:01
1      B6      2010-01-01      JFK      08:55
2      MQ      2010-01-01      ORD      11:20
3      9E      2010-01-01      DTW      11:44
4      B6      2010-01-01      JFK      11:52

Scheduled Elapsed Time (Minutes) FLIGHT_STATUS month day season WeekDay \
0      76      LATE      1      1      winter      Friday
1      75      LATE      1      1      winter      Friday
2      100     ONTIME     1      1      winter      Friday
3      84      LATE      1      1      winter      Friday
4      71      LATE      1      1      winter      Friday

... SYR_wind_type SYR_wind_speed SYR_ceiling_height \
0 ...      N      17.4      741.370199
1 ...      C      0.0      548.800000
2 ...      C      0.0      841.400000
3 ...      C      0.0      1006.000000
4 ...      C      0.0      1006.000000

SYR_ceiling_det_code SYR_celing_CAVOK SYR_visibility_dist \
0      M      N      6110.0
1      M      N      2414.0
2      M      N      3138.4
3      M      N      3138.4
4      M      N      3138.4

SYR_visibility_variability SYR_air_temperature SYR_dew_point_temperature \
0      N      -4.8      -24.0
1      N      -17.0     -28.8
2      N      -19.4     -29.6
3      N      -19.8     -29.2
4      N      -19.8     -29.2

SYR_sea_level_pressure
0      10154.572817
1      10132.352069
2      10129.796558
3      10129.482076
4      10129.482076

[5 rows x 40 columns]

```

```

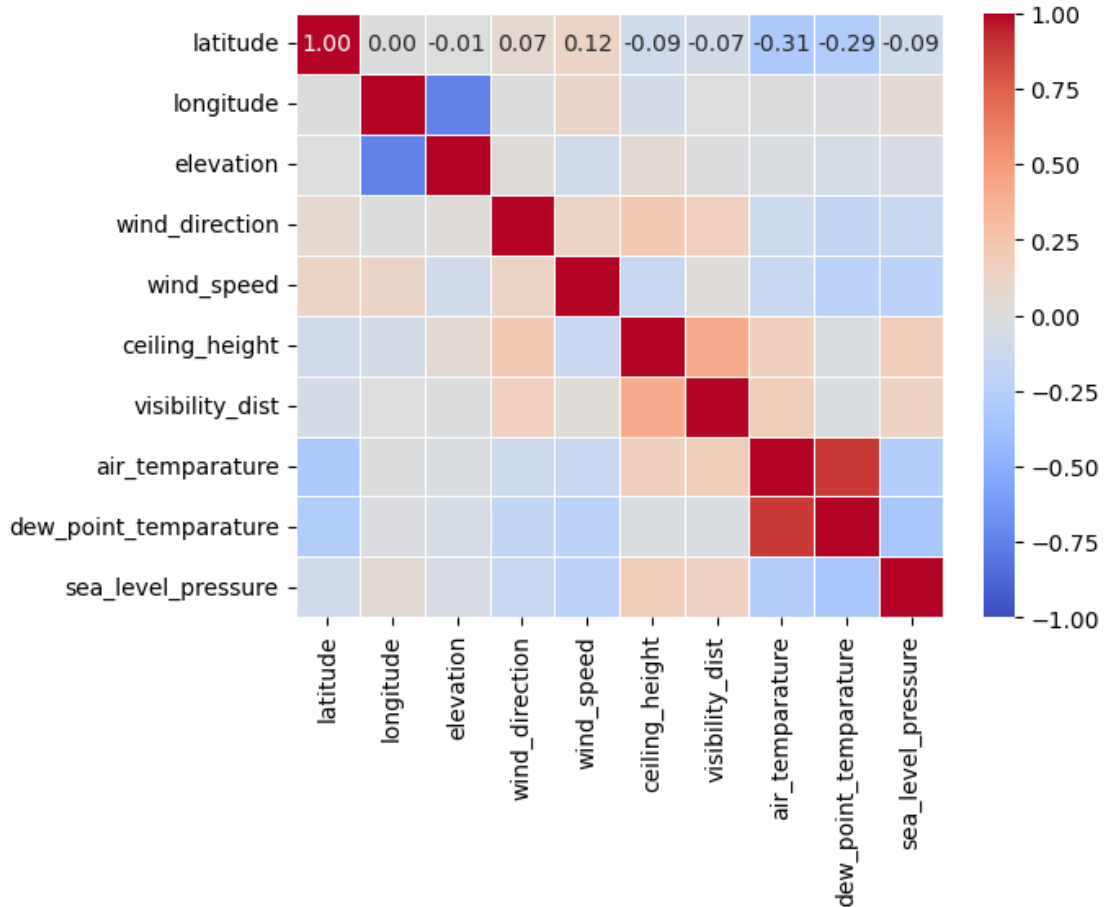
[391]: tmp = data[['latitude', 'longitude', 'elevation',
                  'wind_direction', 'wind_speed', 'ceiling_height', 'visibility_dist',
                  'air_temperature', 'dew_point_temperature',

```

```
'sea_level_pressure']]
```

```
[392]: sns.heatmap(tmp.corr(), annot=True, cmap='coolwarm', fmt=".2f", linewidths=.5,
↳ vmin=-1, vmax=1)
```

```
[392]: <Axes: >
```



```
[393]: data['SCHED_ARRV_TIME_CAT'] = data['Scheduled Arrival Time'].
↳ apply(convert_to_categorical)
```

```
[394]: data.shape
```

```
[394]: (113671, 41)
```

```
[395]: data.columns
```

```
[395]: Index(['Carrier Code', 'Date (MM/DD/YYYY)', 'Origin Airport',
↳ 'Scheduled Arrival Time', 'Scheduled Elapsed Time (Minutes)',
```

```

'FLIGHT_STATUS', 'month', 'day', 'season', 'WeekDay', 'UNIX_DATE',
'UNIX_TIMESTAMP', 'latitude', 'longitude', 'elevation',
'wind_direction', 'wind_type', 'wind_speed', 'ceiling_height',
'ceiling_det_code', 'celing_CAVOK', 'visibility_dist',
'visibility_variability', 'air_temparature', 'dew_point_temparature',
'sea_level_pressure', 'SYR_latitude', 'SYR_longitude', 'SYR_elevation',
'SYR_wind_direction', 'SYR_wind_type', 'SYR_wind_speed',
'SYR_ceiling_height', 'SYR_ceiling_det_code', 'SYR_celing_CAVOK',
'SYR_visibility_dist', 'SYR_visibility_variability',
'SYR_air_temparature', 'SYR_dew_point_temparature',
'SYR_sea_level_pressure', 'SCHED_ARRV_TIME_CAT'],
dtype='object')

```

```
[396]: # data = data[data['Origin Airport'].isin(['JFK','ORD','MCO'])]
```

```
[397]: data.shape
```

```
[397]: (113671, 41)
```

```
[398]: df = data.drop(columns=['Date (MM/DD/YYYY)', 'Scheduled Arrival Time', 'day',
    ↳ 'UNIX_DATE', 'UNIX_TIMESTAMP',
    ↳ 'longitude', 'dew_point_temparature',
    ↳ 'wind_type', 'ceiling_height', 'ceiling_det_code',
    ↳ 'celing_CAVOK', 'visibility_variability',
    ↳ 'SYR_longitude',
    ↳ 'SYR_wind_type', 'SYR_ceiling_height', 'SYR_ceiling_det_code',
    ↳ 'SYR_celing_CAVOK', 'SYR_visibility_variability',
    ↳ 'SYR_dew_point_temparature', 'SYR_latitude', 'SYR_elevation'])
```

```
[399]: df.isna().sum()
```

```

[399]: Carrier Code          0
Origin Airport              0
Scheduled Elapsed Time (Minutes)  0
FLIGHT_STATUS              0
month                      0
season                     0
WeekDay                    0
latitude                   0
elevation                  0
wind_direction             0
wind_speed                 0
visibility_dist            0
air_temparature            0
sea_level_pressure         0
SYR_wind_direction         0
SYR_wind_speed             0

```

```

SYR_visibility_dist          0
SYR_air_temperature          0
SYR_sea_level_pressure      0
SCHED_ARRV_TIME_CAT         0
dtype: int64

```

```
[400]: df.shape
```

```
[400]: (113671, 20)
```

```
[401]: df['SYR_sea_level_pressure'] = df['SYR_sea_level_pressure'].
        ↪fillna(df['SYR_sea_level_pressure'].mean())
```

```
[402]: df.isna().sum()
```

```

[402]: Carrier Code          0
Origin Airport              0
Scheduled Elapsed Time (Minutes)  0
FLIGHT_STATUS              0
month                      0
season                     0
WeekDay                    0
latitude                   0
elevation                  0
wind_direction             0
wind_speed                 0
visibility_dist            0
air_temperature            0
sea_level_pressure         0
SYR_wind_direction         0
SYR_wind_speed             0
SYR_visibility_dist        0
SYR_air_temperature        0
SYR_sea_level_pressure     0
SCHED_ARRV_TIME_CAT       0
dtype: int64

```

```

[403]: def mps_to_mph(speed_mps):
        return (speed_mps/10.0) * 2.23694

def meters_to_miles(distance_meters):
    return distance_meters * 0.000621371
def celsius_to_fahrenheit(temperature_celsius):
    return (temperature_celsius/10.0) * 9/5 + 32

def hectopascal_to_mb(v):
    return v/10.0

```

```
[404]: df.tail()
```

```
[404]:      Carrier Code Origin Airport  Scheduled Elapsed Time (Minutes)  \
113666          DL          DTW              83
113667          WN          BWI              70
113668          AA          DFW             181
113669          UA          DEN             198
113670          UA          EWR              74

      FLIGHT_STATUS  month  season WeekDay  latitude  elevation  \
113666        EARLY    12  winter  Sunday  42.23113      191.9
113667         LATE    12  winter  Sunday  39.17329       42.0
113668        ONTIME    12  winter  Sunday  32.56500      213.4
113669        ONTIME    12  winter  Sunday  39.84657     1647.2
113670        EARLY    12  winter  Sunday  40.68275       1.9

      wind_direction  wind_speed  visibility_dist  air_temperature  \
113666      240.000000        20.4          13143.4           34.8
113667      243.746270        23.6          16074.4           88.6
113668      199.866450        10.4          16093.0          184.0
113669      191.233489        19.4          16074.4           42.0
113670      286.000000        34.0          16074.4           67.0

      sea_level_pressure  SYR_wind_direction  SYR_wind_speed  \
113666      10126.074551                226.0           25.8
113667      10154.400000                226.0           25.8
113668      10167.225662                226.0           25.8
113669      10192.400000                226.0           25.8
113670      10155.536842                226.0           25.8

      SYR_visibility_dist  SYR_air_temperature  SYR_sea_level_pressure  \
113666          15030.8             -17.0          10154.321162
113667          15030.8             -17.0          10154.321162
113668          15030.8             -17.0          10154.321162
113669          15030.8             -17.0          10154.321162
113670          15030.8             -17.0          10154.321162

      SCHED_ARRV_TIME_CAT
113666          Night
113667          Night
113668          Night
113669          Night
113670          Night
```

```
[405]: df['SYR_wind_speed'] = df['SYR_wind_speed'].apply(mps_to_mph)
df['wind_speed'] = df['wind_speed'].apply(mps_to_mph)
df['visibility_dist'] = df['visibility_dist'].apply(meters_to_miles)
```

```

df['SYR_visibility_dist'] = df['SYR_visibility_dist'].apply(meters_to_miles)
df['air_temperature'] = df['air_temperature'].apply(celsius_to_fahrenheit)
df['SYR_air_temperature'] = df['SYR_air_temperature'].
    ↪apply(celsius_to_fahrenheit)
df['sea_level_pressure'] = df['sea_level_pressure'].apply(hectopascal_to_mb)
df['SYR_sea_level_pressure'] = df['SYR_sea_level_pressure'].
    ↪apply(hectopascal_to_mb)

```

```
[406]: df[df.season=='spring'].head(50)
```

```
[406]:
```

	Carrier Code	Origin Airport	Scheduled Elapsed Time (Minutes)	\
1666	B6	JFK	80	
1667	9E	DTW	87	
1668	00	ORD	106	
1669	B6	JFK	84	
1670	OH	JFK	94	
1671	MQ	ORD	105	
1672	B6	JFK	78	
1673	EV	ATL	137	
1674	XE	CLE	63	
1675	YV	IAD	75	
1676	XE	ORD	100	
1677	B6	MCO	160	
1678	EV	ATL	130	
1679	9E	DTW	87	
1680	OH	CVG	96	
1681	B6	JFK	88	
1682	XE	CLE	65	
1683	OH	JFK	92	
1684	EV	ATL	133	
1685	XE	EWR	69	
1686	MQ	ORD	105	
1687	XE	IAD	76	
1688	US	CLT	114	
1689	OH	DTW	83	
1690	YV	ORD	100	
1691	XE	CLE	65	
1692	OH	JFK	115	
1693	B6	MCO	159	
1694	MQ	ORD	105	
1695	US	PHL	72	
1696	XE	EWR	72	
1697	OH	DTW	87	
1698	EV	ATL	132	
1699	B6	JFK	80	
1700	9E	DTW	87	
1701	00	ORD	106	

1702	B6	JFK	84
1703	OH	JFK	94
1704	9E	DTW	83
1705	MQ	ORD	105
1706	B6	JFK	78
1707	EV	ATL	137
1708	YV	IAD	75
1709	XE	ORD	100
1710	B6	MCO	160
1711	EV	ATL	130
1712	9E	DTW	87
1713	OH	CVG	96
1714	B6	JFK	88
1715	XE	CLE	65

	FLIGHT_STATUS	month	season	WeekDay	latitude	elevation	\
1666	EARLY	3	spring	Monday	40.63915	3.4	
1667	ONTIME	3	spring	Monday	42.23130	192.3	
1668	EARLY	3	spring	Monday	41.96019	201.8	
1669	EARLY	3	spring	Monday	40.63915	3.4	
1670	LATE	3	spring	Monday	40.63915	3.4	
1671	LATE	3	spring	Monday	41.96019	201.8	
1672	ONTIME	3	spring	Monday	40.63915	3.4	
1673	EARLY	3	spring	Monday	33.63010	307.8	
1674	ONTIME	3	spring	Monday	41.40570	238.0	
1675	LATE	3	spring	Monday	38.93486	88.4	
1676	LATE	3	spring	Monday	41.96019	201.8	
1677	LATE	3	spring	Monday	28.43390	27.4	
1678	LATE	3	spring	Monday	33.63010	307.8	
1679	LATE	3	spring	Monday	42.23130	192.3	
1680	ONTIME	3	spring	Monday	39.04440	269.1	
1681	EARLY	3	spring	Monday	40.63915	3.4	
1682	LATE	3	spring	Monday	41.40570	238.0	
1683	EARLY	3	spring	Monday	40.63915	3.4	
1684	LATE	3	spring	Monday	33.63010	307.8	
1685	LATE	3	spring	Monday	40.68250	2.1	
1686	ONTIME	3	spring	Monday	41.96019	201.8	
1687	LATE	3	spring	Monday	38.93486	88.4	
1688	LATE	3	spring	Monday	35.22360	221.9	
1689	ONTIME	3	spring	Monday	42.23130	192.3	
1690	EARLY	3	spring	Monday	41.96019	201.8	
1691	LATE	3	spring	Monday	41.40570	238.0	
1692	LATE	3	spring	Monday	40.63915	3.4	
1693	EARLY	3	spring	Monday	28.43390	27.4	
1694	LATE	3	spring	Monday	41.96019	201.8	
1695	EARLY	3	spring	Monday	39.87327	3.0	
1696	LATE	3	spring	Monday	40.68250	2.1	

1697	EARLY	3	spring	Monday	42.23130	192.3
1698	ONTIME	3	spring	Monday	33.63010	307.8
1699	EARLY	3	spring	Tuesday	40.63915	3.4
1700	LATE	3	spring	Tuesday	42.23130	192.3
1701	EARLY	3	spring	Tuesday	41.96019	201.8
1702	ONTIME	3	spring	Tuesday	40.63915	3.4
1703	EARLY	3	spring	Tuesday	40.63915	3.4
1704	EARLY	3	spring	Tuesday	42.23130	192.3
1705	LATE	3	spring	Tuesday	41.96019	201.8
1706	EARLY	3	spring	Tuesday	40.63915	3.4
1707	LATE	3	spring	Tuesday	33.63010	307.8
1708	LATE	3	spring	Tuesday	38.93486	88.4
1709	EARLY	3	spring	Tuesday	41.96019	201.8
1710	LATE	3	spring	Tuesday	28.43390	27.4
1711	LATE	3	spring	Tuesday	33.63010	307.8
1712	LATE	3	spring	Tuesday	42.23130	192.3
1713	ONTIME	3	spring	Tuesday	39.04440	269.1
1714	EARLY	3	spring	Tuesday	40.63915	3.4
1715	LATE	3	spring	Tuesday	41.40570	238.0

	wind_direction	wind_speed	visibility_dist	air_temperature	\
1666	210.000000	8.545111	9.976609	37.940000	
1667	332.000000	5.816044	9.599809	35.456000	
1668	216.000000	9.439887	9.999724	33.008000	
1669	304.000000	19.103468	9.976609	36.176000	
1670	302.000000	18.835035	9.988166	36.392000	
1671	146.000000	9.663581	9.999724	31.820000	
1672	304.000000	18.611341	9.988166	36.788000	
1673	336.000000	6.710820	9.188337	30.560000	
1674	298.000000	5.368656	9.588252	34.160000	
1675	288.000000	12.616342	9.988166	39.200000	
1676	354.000000	8.724066	9.999724	30.488000	
1677	239.243002	2.684328	9.588252	47.300000	
1678	184.000000	7.158208	9.988166	33.980000	
1679	346.000000	8.276678	9.976609	34.232000	
1680	328.000000	6.487126	9.199895	34.376000	
1681	320.000000	24.203691	9.988166	41.432000	
1682	330.000000	9.842536	9.988166	36.104000	
1683	324.000000	24.651079	9.988166	42.836000	
1684	117.230769	6.934514	9.999724	39.956000	
1685	314.000000	22.145706	9.999724	44.204000	
1686	344.000000	8.008245	9.999724	31.352000	
1687	306.000000	19.998244	9.988166	44.996000	
1688	268.933096	2.057985	9.976609	44.168000	
1689	270.000000	8.500372	9.988166	38.372000	
1690	264.000000	5.950260	9.988166	33.224000	
1691	342.000000	11.542610	9.988166	35.996000	

1692	320.000000	24.606340	9.976609	47.984000
1693	148.282259	5.771305	9.988166	68.432000
1694	160.000000	8.947760	9.999724	35.132000
1695	310.000000	21.608840	9.988166	47.408000
1696	304.000000	20.266676	9.988166	47.156000
1697	332.000000	9.439887	9.988166	38.192000
1698	142.841359	1.879030	9.999724	55.184000
1699	324.000000	20.490370	9.976609	44.996000
1700	26.000000	8.276678	9.999724	26.672000
1701	339.166495	6.925095	9.986341	30.143158
1702	326.000000	8.276678	9.976609	36.572000
1703	330.000000	7.381902	9.988166	35.384000
1704	220.000000	5.279178	9.988166	27.248000
1705	350.000000	6.934514	9.999724	30.200000
1706	332.000000	6.934514	9.988166	35.384000
1707	104.000000	16.105968	6.518306	39.632000
1708	320.000000	4.071231	9.988166	36.968000
1709	344.000000	6.487126	9.988166	30.020000
1710	178.000000	15.434886	9.599809	59.972000
1711	20.000000	16.777050	1.850070	36.320000
1712	352.000000	7.829290	9.788209	29.336000
1713	24.000000	10.110969	7.400032	30.884000
1714	63.666667	9.395148	9.988166	43.232000
1715	91.777778	5.484035	8.166803	30.236000

	sea_level_pressure	SYR_wind_direction	SYR_wind_speed	\
1666	1003.380000	300.0	9.618842	
1667	1017.176569	298.0	8.052984	
1668	1021.027984	300.0	8.052984	
1669	1003.460000	300.0	8.052984	
1670	1003.180000	308.0	8.724066	
1671	1021.388574	312.0	9.171454	
1672	1003.220000	312.0	9.171454	
1673	1017.300000	320.0	9.618842	
1674	1017.859973	324.0	9.887275	
1675	1013.200000	330.0	10.110969	
1676	1022.635592	330.0	10.110969	
1677	1017.620000	336.0	10.826790	
1678	1017.780000	336.0	10.826790	
1679	1019.420000	336.0	10.826790	
1680	1019.575845	346.0	12.482125	
1681	1004.640000	346.0	12.482125	
1682	1019.632162	350.0	12.482125	
1683	1004.900000	284.0	13.376901	
1684	1017.800000	284.0	13.376901	
1685	1005.840000	352.0	12.661080	
1686	1023.540000	352.0	12.661080	

1687	1014.040000	348.0	12.437386
1688	1017.340000	348.0	12.437386
1689	1019.420000	342.0	12.437386
1690	1023.780000	336.0	12.705819
1691	1019.680000	326.0	11.497872
1692	1005.960000	330.0	11.229439
1693	1014.480000	330.0	11.229439
1694	1022.882353	330.0	11.229439
1695	1009.660000	330.0	11.229439
1696	1006.840000	328.0	10.737312
1697	1019.520000	326.0	8.947760
1698	1013.420000	326.0	8.947760
1699	1008.240000	322.0	7.829290
1700	1019.314288	280.0	6.934514
1701	1020.836264	280.0	6.263432
1702	1011.940000	280.0	6.263432
1703	1011.940000	288.0	5.368656
1704	1018.713089	290.0	5.368656
1705	1019.980000	290.0	5.368656
1706	1012.060000	290.0	5.368656
1707	1005.781262	292.0	5.592350
1708	1014.800000	308.0	5.144962
1709	1020.180000	308.0	5.144962
1710	1004.243120	312.0	4.876529
1711	1005.431706	312.0	4.876529
1712	1018.237542	312.0	4.876529
1713	1015.144890	322.0	4.160708
1714	1013.280000	322.0	4.160708
1715	1017.519681	318.0	4.608096

	SYR_visibility_dist	SYR_air_temparature	SYR_sea_level_pressure \
1666	9.976609	35.600	1006.760000
1667	8.199985	33.692	1008.654677
1668	7.891287	33.548	1008.678891
1669	7.891287	33.548	1008.678891
1670	8.091369	33.224	1008.953368
1671	8.091369	33.224	1009.140395
1672	8.091369	33.224	1009.140395
1673	8.939789	33.728	1009.959100
1674	9.539660	34.124	1010.277330
1675	9.739618	34.340	1010.457330
1676	9.739618	34.340	1010.457330
1677	9.999724	34.736	1010.737330
1678	9.999724	34.736	1010.737330
1679	9.999724	34.736	1010.737330
1680	9.988166	35.600	1011.400000
1681	9.988166	35.600	1011.400000

1682	9.988166	35.708	1011.502695
1683	9.999724	36.104	1011.707290
1684	9.999724	36.104	1011.707290
1685	9.999724	36.032	1011.949045
1686	9.999724	36.032	1011.949045
1687	9.988166	36.104	1012.108737
1688	9.988166	36.104	1012.108737
1689	9.988166	35.816	1012.246885
1690	8.488176	35.816	1012.372678
1691	4.299887	35.204	1012.742059
1692	3.345586	34.484	1013.150324
1693	4.545577	34.160	1013.460653
1694	4.545577	34.160	1013.460653
1695	4.545577	34.160	1013.460653
1696	5.545612	33.908	1013.559273
1697	8.799980	33.872	1014.036243
1698	8.799980	33.872	1014.036243
1699	8.999938	33.872	1014.697623
1700	9.988166	32.000	1014.920000
1701	9.976609	32.000	1014.840000
1702	9.976609	32.000	1014.840000
1703	9.988166	32.216	1014.760000
1704	9.988166	32.216	1014.860000
1705	9.988166	32.216	1014.860000
1706	9.988166	32.216	1014.860000
1707	9.976609	32.216	1014.940000
1708	9.988166	32.432	1015.320000
1709	9.988166	32.432	1015.320000
1710	9.988166	32.612	1015.480000
1711	9.988166	32.612	1015.480000
1712	9.988166	32.612	1015.480000
1713	9.988166	33.620	1015.560000
1714	9.988166	33.620	1015.560000
1715	9.988166	34.412	1015.320000

SCHED_ARRV_TIME_CAT

1666	Late Night
1667	Morning
1668	Late Morning
1669	Late Morning
1670	Late Morning
1671	Late Morning
1672	Late Morning
1673	Noon
1674	Noon
1675	Noon
1676	Noon

1677	Noon
1678	Noon
1679	Noon
1680	Afternoon
1681	Afternoon
1682	Afternoon
1683	Afternoon
1684	Afternoon
1685	Afternoon
1686	Afternoon
1687	Evening
1688	Evening
1689	Evening
1690	Evening
1691	Evening
1692	Night
1693	Night
1694	Night
1695	Night
1696	Night
1697	Night
1698	Night
1699	Late Night
1700	Morning
1701	Late Morning
1702	Late Morning
1703	Late Morning
1704	Late Morning
1705	Late Morning
1706	Late Morning
1707	Noon
1708	Noon
1709	Noon
1710	Noon
1711	Noon
1712	Noon
1713	Afternoon
1714	Afternoon
1715	Afternoon

[]:

[]:

[]:

[]:

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

```
[407]: from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.preprocessing import OneHotEncoder

class MultiColumnOneHotEncoder(BaseEstimator, TransformerMixin):
    def __init__(self, columns=None):
        self.columns = columns
        self.encoder = None

    def fit(self, X, y=None):
        self.encoder = OneHotEncoder(sparse_output=False, drop='first')
        self.encoder.fit(X[self.columns])
        return self

    def transform(self, X):
        X_encoded = X.copy()
        encoded_data = self.encoder.transform(X[self.columns])
        encoded_df = pd.DataFrame(encoded_data, columns=self.encoder.
        ↪get_feature_names_out(self.columns), index=X.index)

        # Drop the original columns
        X_encoded = X_encoded.drop(columns=self.columns)

        # Concatenate the encoded DataFrame with the original DataFrame,
        ↪preserving the index
        X_encoded = pd.concat([X_encoded, encoded_df], axis=1)
        return X_encoded

    def fit_transform(self, X, y=None):
        self.fit(X)
        return self.transform(X)

[408]: sample = df.head()

[409]: # sample.head().drop(columns=['FLIGHT_STATUS']).to_csv("sample_input.csv",
        ↪index=False)

[410]: encoder = MultiColumnOneHotEncoder(columns=['Carrier Code', 'Origin
        ↪Airport', 'season', 'SCHED_ARRV_TIME_CAT', 'month', 'WeekDay'])
        ↪#, 'wind_type', 'ceiling_det_code', 'celing_CAVOK', 'visibility_variability'
```

```
[411]: encoded_data = encoder.fit_transform(df.drop(columns=['FLIGHT_STATUS']))
```

```
[412]: encoded_data.head()
```

```
[412]: Scheduled Elapsed Time (Minutes)  latitude  elevation  wind_direction  \
0                76  40.63915         3.4      102.428571
1                75  40.63915         3.4      110.503072
2               100  41.96019       201.8      292.000000
3                84  42.23130       192.3      270.000000
4                71  40.63915         3.4      159.721841
```

```
      wind_speed  visibility_dist  air_temperature  sea_level_pressure  \
0      3.221194      6.966812      33.944      1018.725782
1      0.000000      5.400087      33.368      1015.367354
2      9.395148      9.999724       7.808      1024.780000
3     12.392648      6.599830      23.000      1018.248971
4      0.000000      4.794250      33.224      1014.594175
```

```
      SYR_wind_direction  SYR_wind_speed  ...  month_9  month_10  month_11  \
0          106.000000      3.892276  ...    0.0      0.0      0.0
1          103.912190      0.000000  ...    0.0      0.0      0.0
2          132.389021      0.000000  ...    0.0      0.0      0.0
3          144.543770      0.000000  ...    0.0      0.0      0.0
4          144.543770      0.000000  ...    0.0      0.0      0.0
```

```
      month_12  WeekDay_Monday  WeekDay_Saturday  WeekDay_Sunday  \
0          0.0          0.0          0.0          0.0
1          0.0          0.0          0.0          0.0
2          0.0          0.0          0.0          0.0
3          0.0          0.0          0.0          0.0
4          0.0          0.0          0.0          0.0
```

```
      WeekDay_Thursday  WeekDay_Tuesday  WeekDay_Wednesday
0          0.0          0.0          0.0
1          0.0          0.0          0.0
2          0.0          0.0          0.0
3          0.0          0.0          0.0
4          0.0          0.0          0.0
```

```
[5 rows x 83 columns]
```

```
[413]: trainX, testX, trainY, testY = train_test_split(
      encoded_data,
      df['FLIGHT_STATUS'],
      test_size=0.2,
      random_state=947,
      stratify=df['FLIGHT_STATUS'])
```

```
)
```

```
[414]: from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
enc_trainX = pd.DataFrame(scaler.fit_transform(trainX), index=trainX.index,
    ↪ columns=trainX.columns)
enc_testX = pd.DataFrame(scaler.transform(testX), index=testX.index,
    ↪ columns=testX.columns)
```

```
[415]: from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

def fit_and_evaluate(model, trainX, trainY, testX, testY):
    # Convert string labels to integers
    label_encoder = LabelEncoder()
    trainY_encoded = label_encoder.fit_transform(trainY)
    testY_encoded = label_encoder.transform(testY)

    model.fit(trainX, trainY_encoded)
    testY_pred = model.predict(testX)
    accuracy = accuracy_score(testY_encoded, testY_pred)
    report = classification_report(testY_encoded, testY_pred, output_dict=True)
    results = {'accuracy': accuracy, 'classification_report': report}
    return results

# Update other classification functions similarly...

# Random Forest
from sklearn.ensemble import RandomForestClassifier

def random_forest_classification(trainX, trainY, testX, testY,
    ↪ n_estimators=100, criterion='gini', max_depth=None):
    model = RandomForestClassifier(n_estimators=n_estimators,
    ↪ criterion=criterion, max_depth=max_depth)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# Support Vector Machines (SVM)
from sklearn.svm import SVC

def svm_classification(trainX, trainY, testX, testY, kernel='rbf', C=1.0):
    model = SVC(kernel=kernel, C=C)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# K-Nearest Neighbors (KNN)
```



```

from sklearn.neighbors import KNeighborsClassifier

def knn_classification(trainX, trainY, testX, testY, n_neighbors=5):
    model = KNeighborsClassifier(n_neighbors=n_neighbors)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# Gradient Boosting Machines (GBM)
from sklearn.ensemble import GradientBoostingClassifier

def gbm_classification(trainX, trainY, testX, testY, n_estimators=100,
    ↪learning_rate=0.1, max_depth=3):
    model = GradientBoostingClassifier(n_estimators=n_estimators,
    ↪learning_rate=learning_rate, max_depth=max_depth)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# Naive Bayes
from sklearn.naive_bayes import GaussianNB

def naive_bayes_classification(trainX, trainY, testX, testY):
    model = GaussianNB()
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# AdaBoost
from sklearn.ensemble import AdaBoostClassifier

def adaboost_classification(trainX, trainY, testX, testY, n_estimators=50,
    ↪learning_rate=1.0):
    model = AdaBoostClassifier(n_estimators=n_estimators,
    ↪learning_rate=learning_rate)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# XGBoost
from xgboost import XGBClassifier

def xgboost_classification(trainX, trainY, testX, testY, n_estimators=100,
    ↪learning_rate=0.1, max_depth=3):
    model = XGBClassifier(n_estimators=n_estimators,
    ↪learning_rate=learning_rate, max_depth=max_depth)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

def logistic_regression_classification(trainX, trainY, testX, testY,
    ↪penalty='l2', C=1.0, max_iter=1000, solver='lbfgs'):
    target_names = trainY.unique() if isinstance(trainY, pd.Series) else testY.
    ↪unique()

    # Initialize and train the Logistic Regression model

```

```

    model = LogisticRegression(penalty=penalty, C=C, max_iter=max_iter,
↪solver=solver, verbose=1 if max_iter > 300 else 0)
    model.fit(trainX, trainY)

    # Predict on the testing set
    testY_pred = model.predict(testX)

    # Calculate accuracy score
    accuracy = accuracy_score(testY, testY_pred)

    # Generate classification report
    report = classification_report(testY, testY_pred,
↪target_names=target_names, output_dict=True)

    results = {
        'accuracy': accuracy,
        'classification_report': report
    }

    return results

def decision_tree_classification(trainX, trainY, testX, testY,
↪criterion='gini', max_depth=None):

    # Get unique target names
    target_names = trainY.unique() if isinstance(trainY, pd.Series) else testY.
↪unique()

    # Initialize and train the Decision Tree model
    model = DecisionTreeClassifier(criterion=criterion,
↪max_depth=max_depth,min_samples_split=5)
    model.fit(trainX, trainY)

    # Predict on the testing set
    testY_pred = model.predict(testX)

    # Calculate accuracy score
    accuracy = accuracy_score(testY, testY_pred)

    # Generate classification report
    report = classification_report(testY, testY_pred,
↪target_names=target_names, output_dict=True)

    results = {
        'accuracy': accuracy,
        'classification_report': report
    }

```

```
}

return results
```

```
[416]: # logistic_regression_classification(enc_trainX, trainY, enc_testX, testY,
      ↪ solver='saga', max_iter=1500, C=50)
```

```
[417]: # decision_tree_classification(enc_trainX, trainY, enc_testX, testY,
      ↪ max_depth=11, criterion='entropy')
```

```
[418]: # random_forest_classification(enc_trainX, trainY, enc_testX, testY)
```

```
[419]: # knn_classification(enc_trainX, trainY, enc_testX, testY)
```

```
[420]: # gbm_classification(enc_trainX, trainY, enc_testX, testY)
```

```
[421]: # naive_bayes_classification(enc_trainX, trainY, enc_testX, testY)
```

```
[422]: # adaboost_classification(enc_trainX, trainY, enc_testX, testY)
```

```
[423]: # xgboost_classification(enc_trainX, trainY, enc_testX, testY)
```

```
[424]: from sklearn.model_selection import GridSearchCV, KFold
def hyperparameter_tuning(model, param_grid, trainX, trainY, cv=5):
    label_encoder = LabelEncoder()
    trainY_encoded = label_encoder.fit_transform(trainY)
    # Initialize K-Fold cross-validator
    kf = KFold(n_splits=cv, shuffle=True, random_state=42)

    # Perform grid search
    grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=kf,
    ↪ scoring='accuracy', verbose=3, n_jobs=10, )
    grid_search.fit(trainX, trainY_encoded)

    return grid_search
```

```
[425]: params = {'colsample_bytree': 0.7, 'gamma': 0.1, 'learning_rate': 0.1,
      ↪ 'max_depth': 7, 'n_estimators': 200, 'reg_lambda': 1.5, 'subsample': 0.9}
# param_grid = {
#     'n_estimators': [100, 200], # Number of boosting rounds
#     'max_depth': [3, 5, 7, 11], # Maximum tree depth
#     'learning_rate': [0.01, 0.1, 0.3], # Step size shrinkage
#     'subsample': [0.7, 0.9], # Subsample ratio of the training instances
#     'colsample_bytree': [0.7, 0.9], # Subsample ratio of columns when
      ↪ constructing each tree
```

```

#     'gamma': [0, 0.1], # Minimum loss reduction required to make a further
    ↪ partition on a leaf node of the tree
#     'reg_lambda': [1, 1.5, 2] # L2 regularization term on weights
# }
# model = XGBClassifier(**params)
# # best_model = hyperparameter_tuning(model, param_grid, enc_trainX, trainY)
# fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)
# Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)

```

```

[426]: params = {'bootstrap': False, 'criterion': 'entropy', 'max_depth': None,
    ↪ 'max_features': 'sqrt', 'min_samples_leaf': 4, 'min_samples_split': 2,
    ↪ 'n_estimators': 200}
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt'],
    'bootstrap': [True, False],
    'criterion': ['gini', 'entropy']
}
# model = RandomForestClassifier(**params)
# fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)
# best_model = hyperparameter_tuning(model, param_grid, enc_trainX, trainY)
# Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)

```

```

[427]: params = {'algorithm': 'SAMME.R', 'learning_rate': 1.0, 'n_estimators': 200}
param_grid = {
    'n_estimators': [50, 100, 200], # Number of estimators
    'learning_rate': [0.01, 0.1, 1.0], # Learning rate
    'algorithm': ['SAMME', 'SAMME.R'], # Algorithm
}

# model = AdaBoostClassifier(**params)
# fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)
# best_model = hyperparameter_tuning(model, param_grid, enc_trainX, trainY)
# # Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)

```

```

[428]: params = {'ccp_alpha': 0.07, 'criterion': 'gini', 'max_depth': None,
    ↪ 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2,
    ↪ 'splitter': 'best'}

```

```

param_grid = {
    'criterion': ['gini', 'entropy'], # Split criterion
    'splitter': ['best', 'random'], # Strategy to choose split at each node
    'max_depth': [None, 1, 3, 5], # Max depth of the tree
    'min_samples_split': [2, 5, 10], # Min samples required to split a node
    'min_samples_leaf': [1, 2, 4], # Min samples required at each leaf node
    'max_features': ['sqrt', 'log2'], # Max features to consider for split
    'ccp_alpha': [0.07, 0.01]
}

# model = DecisionTreeClassifier(**params)
# fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)
# best_model = hyperparameter_tuning(model, param_grid, enc_trainX, trainY)
# # Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)

```

```

[429]: # {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 200}
gbm_params = {
    'learning_rate': [0.01, 0.05, 0.1], # Learning rate
    'n_estimators': [50, 100, 200], # Number of boosting stages
    'max_depth': [3, 4, 5], # Maximum depth of the individual trees
}
# model = GradientBoostingClassifier()
# best_model = hyperparameter_tuning(model, gbm_params, enc_trainX, trainY)
# Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)

```

```

[430]: %%time
from sklearn.ensemble import VotingClassifier

lr = LogisticRegression(penalty='l2', C=50, max_iter=1500, solver='saga')
ada = AdaBoostClassifier(**{'algorithm': 'SAMME.R', 'learning_rate': 1.0,
    ↳ 'n_estimators': 200})
gbm = GradientBoostingClassifier(**{'learning_rate': 0.1, 'max_depth': 5,
    ↳ 'n_estimators': 200})
rf = RandomForestClassifier(**{'bootstrap': False, 'criterion': 'entropy',
    ↳ 'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 4,
    ↳ 'min_samples_split': 2, 'n_estimators': 200})
xgb = XGBClassifier(**{'colsample_bytree': 0.7, 'gamma': 0.1, 'learning_rate':
    ↳ 0.1, 'max_depth': 7, 'n_estimators': 200, 'reg_lambda': 1.5, 'subsample': 0.
    ↳ 9}, objective='multi:softprob')

# res = fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)

```

```

votingCLF = VotingClassifier(
    estimators=[
        ('rf', rf),
        ('ada', ada),
        ('xgb', xgb),
        ('lr', lr),
        ('gbm', gbm)
    ],
    voting='soft',
    weights=[4,5,7,2, 6]
)
# fit_and_evaluate(votingCLF, enc_trainX, trainY, enc_testX, testY)

```

CPU times: user 0 ns, sys: 86 µs, total: 86 µs
 Wall time: 88.2 µs

```

[435]: INPUT = encoded_data
TARGET=df['FLIGHT_STATUS']
label_encoder = LabelEncoder()
TARGET_ENCODED = label_encoder.fit_transform(TARGET)

```

```

[436]: from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
enc_trainX = pd.DataFrame(scaler.fit_transform(encoded_data),
    ↪index=encoded_data.index, columns=encoded_data.columns)

```

```

[437]: %%time
votingCLF.fit(enc_trainX, TARGET_ENCODED)

```

/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-packages/sklearn/ensemble/_weight_boosting.py:519: FutureWarning: The SAMME.R algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME algorithm to circumvent this warning.
 warnings.warn(

CPU times: user 7min 33s, sys: 2.1 s, total: 7min 35s
 Wall time: 6min 53s

```

[437]: VotingClassifier(estimators=[('rf',
                                     RandomForestClassifier(bootstrap=False,
                                                                criterion='entropy',
                                                                min_samples_leaf=4,
                                                                n_estimators=200)),
                                   ('ada', AdaBoostClassifier(n_estimators=200)),
                                   ('xgb',
                                    XGBClassifier(base_score=None, booster=None,
                                                    callbacks=None,
                                                    colsample_bylevel=None,

```

```

        colsample_bynode=None,
        colsample_bytree=0.7, device=None,
        early_stopping_rounds=None,
        ena...
        max_delta_step=None, max_depth=7,
        max_leaves=None,
        min_child_weight=None, missing=nan,
        monotone_constraints=None,
        multi_strategy=None,
        n_estimators=200, n_jobs=None,
        num_parallel_tree=None,
        objective='multi:softprob', ...)),
    ('lr',
     LogisticRegression(C=50, max_iter=1500,
                        solver='saga')),
    ('gbm',
     GradientBoostingClassifier(max_depth=5,
                                n_estimators=200))],
    voting='soft', weights=[4, 5, 7, 2, 6])

```

```
[ ]:
```

```
[ ]:
```

```
[438]: enc_trainX.shape
```

```
[438]: (113671, 83)
```

```
[439]: initi_first_task = pd.read_csv("./sample_input_first_task.csv")
```

```
[440]: initial_first_task_data = scaler.transform(encoder.transform(initi_first_task))
```

```
[441]: initial_first_task_data.shape
```

```
[441]: (12, 83)
```

```
[442]: votingCLF.predict(initial_first_task_data)
```

```

/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but RandomForestClassifier was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but AdaBoostClassifier was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-

```

```
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but LogisticRegression was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but GradientBoostingClassifier was fitted with feature names
  warnings.warn(
```

```
[442]: array([0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1])
```

```
[443]: label_encoder.inverse_transform(votingCLF.predict(initial_first_task_data))
```

```
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but RandomForestClassifier was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but AdaBoostClassifier was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but LogisticRegression was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but GradientBoostingClassifier was fitted with feature names
  warnings.warn(
```

```
[443]: array(['EARLY', 'EARLY', 'EARLY', 'EARLY', 'EARLY', 'LATE', 'EARLY',
'EARLY', 'LATE', 'EARLY', 'EARLY', 'LATE'], dtype=object)
```

```
[ ]:
```

```
[444]: initi_first_task = pd.read_csv("./final_input_first.csv")
```

```
[445]: initial_first_task_data = scaler.transform(encoder.transform(initi_first_task))
```

```
[446]: initial_first_task_data.shape
```

```
[446]: (12, 83)
```

```
[447]: votingCLF.predict(initial_first_task_data)
```

```
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but RandomForestClassifier was fitted with feature names
  warnings.warn(
```



```

/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but AdaBoostClassifier was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but LogisticRegression was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but GradientBoostingClassifier was fitted with feature names
  warnings.warn(

```

```
[447]: array([0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1])
```

```
[448]: label_encoder.inverse_transform(votingCLF.predict(initial_first_task_data))
```

```

/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but RandomForestClassifier was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but AdaBoostClassifier was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but LogisticRegression was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but GradientBoostingClassifier was fitted with feature names
  warnings.warn(

```

```
[448]: array(['EARLY', 'EARLY', 'LATE', 'EARLY', 'EARLY', 'EARLY', 'EARLY',
              'EARLY', 'EARLY', 'EARLY', 'EARLY', 'LATE'], dtype=object)
```

```
[ ]: 
```

```
[ ]: 
```

```
[ ]: 
```

```
[ ]: 
```

```
[ ]: 
```

1 2nd Model

```
[449]: data1 = pd.read_csv("./SYR_ORIGIN_1HOP.csv")
data2 = pd.read_csv("./SYR_ORIGIN_2HOP.csv")
data3 = pd.read_csv("./SYR_ORIGIN_3HOP.csv")
```

```
[450]: data1['SCHED_ARRV_TIME_CAT'] = data1['Scheduled Arrival Time'].
        ↪apply(convert_to_categorical)
data2['SCHED_ARRV_TIME_CAT'] = data2['Scheduled Arrival Time'].
        ↪apply(convert_to_categorical)
data3['SCHED_ARRV_TIME_CAT'] = data3['Scheduled Arrival Time'].
        ↪apply(convert_to_categorical)
```

```
[451]: data1['SYR_sea_level_pressure'] = data1['SYR_sea_level_pressure'].
        ↪fillna(data1['SYR_sea_level_pressure'].mean())
data2['SYR_sea_level_pressure'] = data2['SYR_sea_level_pressure'].
        ↪fillna(data2['SYR_sea_level_pressure'].mean())
data3['SYR_sea_level_pressure'] = data3['SYR_sea_level_pressure'].
        ↪fillna(data3['SYR_sea_level_pressure'].mean())
```

```
[452]: def convert(df):
        df['SYR_wind_speed'] = df['SYR_wind_speed'].apply(mps_to_mph)
        df['wind_speed'] = df['wind_speed'].apply(mps_to_mph)
        df['visibility_dist'] = df['visibility_dist'].apply(meters_to_miles)
        df['SYR_visibility_dist'] = df['SYR_visibility_dist'].apply(meters_to_miles)
        df['air_temperature'] = df['air_temperature'].apply(celsius_to_fahrenheit)
        df['SYR_air_temperature'] = df['SYR_air_temperature'].
        ↪apply(celsius_to_fahrenheit)
        df['sea_level_pressure'] = df['sea_level_pressure'].apply(hectopascal_to_mb)
        df['SYR_sea_level_pressure'] = df['SYR_sea_level_pressure'].
        ↪apply(hectopascal_to_mb)
        return df
```

```
[453]: data1 = convert(data1)
data2 = convert(data2)
data3 = convert(data3)
```

```
[454]: data1.shape
```

```
[454]: (113671, 42)
```

```
[455]: data2.shape
```

```
[455]: (227342, 42)
```

```
[456]: data3.shape
```

[456]: (341013, 42)

```
[457]: cols_to_drop = ['Date (MM/DD/YYYY)', 'Scheduled Arrival Time', 'day',  
    ↪ 'UNIX_DATE', 'UNIX_TIMESTAMP',  
    ↪ 'longitude', 'dew_point_temperature',  
    ↪ 'wind_type', 'ceiling_height', 'ceiling_det_code',  
    ↪ 'celing_CAVOK', 'visibility_variability',  
    ↪ 'SYR_longitude',  
    ↪ 'SYR_wind_type', 'SYR_ceiling_height', 'SYR_ceiling_det_code',  
    ↪ 'SYR_celing_CAVOK', 'SYR_visibility_variability',  
    ↪ 'SYR_dew_point_temperature', 'SYR_latitude', 'SYR_elevation']
```

```
[458]: df1 = data1.drop(columns=cols_to_drop)  
df2 = data2.drop(columns=cols_to_drop)  
df3 = data3.drop(columns=cols_to_drop)
```

```
[459]: df1.shape, df2.shape, df3.shape
```

[459]: ((113671, 21), (227342, 21), (341013, 21))

```
[460]: from sklearn.base import BaseEstimator, TransformerMixin  
from sklearn.preprocessing import OneHotEncoder  
  
class MultiColumnOneHotEncoder(BaseEstimator, TransformerMixin):  
    def __init__(self, columns=None):  
        self.columns = columns  
        self.encoder = None  
  
    def fit(self, X, y=None):  
        self.encoder = OneHotEncoder(sparse_output=False, drop='first')  
        self.encoder.fit(X[self.columns])  
        return self  
  
    def transform(self, X):  
        X_encoded = X.copy()  
        encoded_data = self.encoder.transform(X[self.columns])  
        encoded_df = pd.DataFrame(encoded_data, columns=self.encoder.  
    ↪ get_feature_names_out(self.columns), index=X.index)  
  
        # Drop the original columns  
        X_encoded = X_encoded.drop(columns=self.columns)  
  
        # Concatenate the encoded DataFrame with the original DataFrame,  
    ↪ preserving the index  
        X_encoded = pd.concat([X_encoded, encoded_df], axis=1)  
        return X_encoded
```

```
def fit_transform(self, X, y=None):
    self.fit(X)
    return self.transform(X)
```

1.1 START 1 HOP

```
[461]: encoder = MultiColumnOneHotEncoder(columns=['Carrier Code', 'Origin_
↳Airport', 'season', 'SCHED_ARRV_TIME_CAT', 'month', 'WeekDay', 'PREV_STAT'])_
↳#, 'wind_type', 'ceiling_det_code', 'celing_CAVOK', 'visibility_variability'
```

```
[462]: encoded_data = encoder.fit_transform(df1.drop(columns=['FLIGHT_STATUS']))
```

```
[463]: encoded_data.head()
```

```
[463]: Scheduled Elapsed Time (Minutes)  latitude  elevation  wind_direction  \
0                76  40.63915         3.4      102.428571
1                75  40.63915         3.4      110.503072
2               100  41.96019        201.8      292.000000
3                84  42.23130        192.3      270.000000
4                71  40.63915         3.4      159.721841
```

```
    wind_speed  visibility_dist  air_temparature  sea_level_pressure  \
0      3.221194      6.966812         33.944      1018.725782
1      0.000000      5.400087         33.368      1015.367354
2      9.395148      9.999724          7.808      1024.780000
3     12.392648      6.599830         23.000      1018.248971
4      0.000000      4.794250         33.224      1014.594175
```

```
    SYR_wind_direction  SYR_wind_speed  ...  month_11  month_12  \
0          106.000000         3.892276  ...      0.0      0.0
1          103.912190         0.000000  ...      0.0      0.0
2          132.389021         0.000000  ...      0.0      0.0
3          144.543770         0.000000  ...      0.0      0.0
4          144.543770         0.000000  ...      0.0      0.0
```

```
    WeekDay_Monday  WeekDay_Saturday  WeekDay_Sunday  WeekDay_Thursday  \
0              0.0              0.0              0.0              0.0
1              0.0              0.0              0.0              0.0
2              0.0              0.0              0.0              0.0
3              0.0              0.0              0.0              0.0
4              0.0              0.0              0.0              0.0
```

```
    WeekDay_Tuesday  WeekDay_Wednesday  PREV_STAT_LATE  PREV_STAT_ONTIME
0              0.0              0.0              0.0              1.0
1              0.0              0.0              1.0              0.0
2              0.0              0.0              1.0              0.0
3              0.0              0.0              0.0              1.0
```

```
4          0.0          0.0          1.0          0.0

[5 rows x 85 columns]
```

```
[464]: df1.shape
```

```
[464]: (113671, 21)
```

```
[465]: trainX, testX, trainY, testY = train_test_split(
        encoded_data,
        df1['FLIGHT_STATUS'],
        test_size=0.2,
        random_state=947,
        stratify=df1['FLIGHT_STATUS']
    )
```

```
[466]: from sklearn.preprocessing import StandardScaler
        scaler = StandardScaler()
        enc_trainX = pd.DataFrame(scaler.fit_transform(trainX), index=trainX.index,
        ↪columns=trainX.columns)
        enc_testX = pd.DataFrame(scaler.transform(testX), index=testX.index,
        ↪columns=testX.columns)
```

```
[467]: from sklearn.tree import DecisionTreeClassifier
        from sklearn.preprocessing import LabelEncoder
        from sklearn.linear_model import LogisticRegression
        from sklearn.metrics import accuracy_score, classification_report

        def fit_and_evaluate(model, trainX, trainY, testX, testY):
            # Convert string labels to integers
            label_encoder = LabelEncoder()
            trainY_encoded = label_encoder.fit_transform(trainY)
            testY_encoded = label_encoder.transform(testY)

            model.fit(trainX, trainY_encoded)
            testY_pred = model.predict(testX)
            accuracy = accuracy_score(testY_encoded, testY_pred)
            report = classification_report(testY_encoded, testY_pred, output_dict=True)
            results = {'accuracy': accuracy, 'classification_report': report}
            return results

        # Update other classification functions similarly...

        # Random Forest
        from sklearn.ensemble import RandomForestClassifier
```

```

def random_forest_classification(trainX, trainY, testX, testY,
    ↪n_estimators=100, criterion='gini', max_depth=None):
    model = RandomForestClassifier(n_estimators=n_estimators,
    ↪criterion=criterion, max_depth=max_depth)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# Support Vector Machines (SVM)
from sklearn.svm import SVC

def svm_classification(trainX, trainY, testX, testY, kernel='rbf', C=1.0):
    model = SVC(kernel=kernel, C=C)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# K-Nearest Neighbors (KNN)
from sklearn.neighbors import KNeighborsClassifier

def knn_classification(trainX, trainY, testX, testY, n_neighbors=5):
    model = KNeighborsClassifier(n_neighbors=n_neighbors)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# Gradient Boosting Machines (GBM)
from sklearn.ensemble import GradientBoostingClassifier

def gbm_classification(trainX, trainY, testX, testY, n_estimators=100,
    ↪learning_rate=0.1, max_depth=3):
    model = GradientBoostingClassifier(n_estimators=n_estimators,
    ↪learning_rate=learning_rate, max_depth=max_depth)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# Naive Bayes
from sklearn.naive_bayes import GaussianNB

def naive_bayes_classification(trainX, trainY, testX, testY):
    model = GaussianNB()
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# AdaBoost
from sklearn.ensemble import AdaBoostClassifier

def adaboost_classification(trainX, trainY, testX, testY, n_estimators=50,
    ↪learning_rate=1.0):
    model = AdaBoostClassifier(n_estimators=n_estimators,
    ↪learning_rate=learning_rate)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# XGBoost

```

```

from xgboost import XGBClassifier

def xgboost_classification(trainX, trainY, testX, testY, n_estimators=100,
    ↪learning_rate=0.1, max_depth=3):
    model = XGBClassifier(n_estimators=n_estimators,
    ↪learning_rate=learning_rate, max_depth=max_depth)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

def logistic_regression_classification(trainX, trainY, testX, testY,
    ↪penalty='l2', C=1.0, max_iter=1000, solver='lbfgs'):
    target_names = trainY.unique() if isinstance(trainY, pd.Series) else testY.
    ↪unique()

    # Initialize and train the Logistic Regression model
    model = LogisticRegression(penalty=penalty, C=C, max_iter=max_iter,
    ↪solver=solver, verbose=1 if max_iter > 300 else 0)
    model.fit(trainX, trainY)

    # Predict on the testing set
    testY_pred = model.predict(testX)

    # Calculate accuracy score
    accuracy = accuracy_score(testY, testY_pred)

    # Generate classification report
    report = classification_report(testY, testY_pred,
    ↪target_names=target_names, output_dict=True)

    results = {
        'accuracy': accuracy,
        'classification_report': report
    }

    return results

def decision_tree_classification(trainX, trainY, testX, testY,
    ↪criterion='gini', max_depth=None):

    # Get unique target names
    target_names = trainY.unique() if isinstance(trainY, pd.Series) else testY.
    ↪unique()

    # Initialize and train the Decision Tree model
    model = DecisionTreeClassifier(criterion=criterion,
    ↪max_depth=max_depth, min_samples_split=5)
    model.fit(trainX, trainY)

```

```

# Predict on the testing set
testY_pred = model.predict(testX)

# Calculate accuracy score
accuracy = accuracy_score(testY, testY_pred)

# Generate classification report
report = classification_report(testY, testY_pred,
↪target_names=target_names, output_dict=True)

results = {
    'accuracy': accuracy,
    'classification_report': report
}

return results

```

logistic_regression_classification(enc_trainX, trainY, enc_testX, testY, solver='saga', max_iter=1500, C=50)

decision_tree_classification(enc_trainX, trainY, enc_testX, testY, max_depth=11, criterion='entropy')

random_forest_classification(enc_trainX, trainY, enc_testX, testY)

knn_classification(enc_trainX, trainY, enc_testX, testY)

gbm_classification(enc_trainX, trainY, enc_testX, testY)

naive_bayes_classification(enc_trainX, trainY, enc_testX, testY)

adaboost_classification(enc_trainX, trainY, enc_testX, testY)

xgboost_classification(enc_trainX, trainY, enc_testX, testY)

```

[468]: from sklearn.model_selection import GridSearchCV, KFold
def hyperparameter_tuning(model, param_grid, trainX, trainY, cv=5):
    label_encoder = LabelEncoder()
    trainY_encoded = label_encoder.fit_transform(trainY)
    # Initialize K-Fold cross-validator
    kf = KFold(n_splits=cv, shuffle=True, random_state=42)

    # Perform grid search
    grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=kf,
↪scoring='accuracy', verbose=3, n_jobs=10, )
    grid_search.fit(trainX, trainY_encoded)

    return grid_search

```



```
[469]: params = {'colsample_bytree': 0.7, 'gamma': 0.1, 'learning_rate': 0.1,
↳ 'max_depth': 7, 'n_estimators': 200, 'reg_lambda': 1.5, 'subsample': 0.9}
# param_grid = {
#     'n_estimators': [100, 200], # Number of boosting rounds
#     'max_depth': [3, 5, 7, 11], # Maximum tree depth
#     'learning_rate': [0.01, 0.1, 0.3], # Step size shrinkage
#     'subsample': [0.7, 0.9], # Subsample ratio of the training instances
#     'colsample_bytree': [0.7, 0.9], # Subsample ratio of columns when
↳ constructing each tree
#     'gamma': [0, 0.1], # Minimum loss reduction required to make a further
↳ partition on a leaf node of the tree
#     'reg_lambda': [1, 1.5, 2] # L2 regularization term on weights
# }
# model = XGBClassifier(**params)
# # best_model = hyperparameter_tuning(model, param_grid, enc_trainX, trainY)
# fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)
# Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)
```

```
[470]: params = {'bootstrap': False, 'criterion': 'entropy', 'max_depth': None,
↳ 'max_features': 'sqrt', 'min_samples_leaf': 4, 'min_samples_split': 2,
↳ 'n_estimators': 200}
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt'],
    'bootstrap': [True, False],
    'criterion': ['gini', 'entropy']
}
# model = RandomForestClassifier(**params)
# fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)
# best_model = hyperparameter_tuning(model, param_grid, enc_trainX, trainY)
# Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)
```

```
[471]: params = {'algorithm': 'SAMME.R', 'learning_rate': 1.0, 'n_estimators': 200}
param_grid = {
    'n_estimators': [50, 100, 200], # Number of estimators
    'learning_rate': [0.01, 0.1, 1.0], # Learning rate
    'algorithm': ['SAMME', 'SAMME.R'], # Algorithm
}

# model = AdaBoostClassifier(**params)
```

```
# fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)
# best_model = hyperparameter_tuning(model, param_grid, enc_trainX, trainY)
# # Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)
```

```
[472]: params = {'ccp_alpha': 0.07, 'criterion': 'gini', 'max_depth': None,
↳ 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2,
↳ 'splitter': 'best'}

param_grid = {
    'criterion': ['gini', 'entropy'], # Split criterion
    'splitter': ['best', 'random'], # Strategy to choose split at each node
    'max_depth': [None, 1, 3, 5], # Max depth of the tree
    'min_samples_split': [2, 5, 10], # Min samples required to split a node
    'min_samples_leaf': [1, 2, 4], # Min samples required at each leaf node
    'max_features': ['sqrt', 'log2'], # Max features to consider for split
    'ccp_alpha': [0.07, 0.01]
}

# model = DecisionTreeClassifier(**params)
# fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)
# best_model = hyperparameter_tuning(model, param_grid, enc_trainX, trainY)
# # Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)
```

```
[473]: gbm_params = {
    'learning_rate': [0.01, 0.05, 0.1], # Learning rate
    'n_estimators': [50, 100, 200], # Number of boosting stages
    'max_depth': [3, 4, 5], # Maximum depth of the individual trees
}

# model = GradientBoostingClassifier()
# best_model = hyperparameter_tuning(model, gbm_params, enc_trainX, trainY)
# # Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)
```

```
[474]: enc_trainX.shape
```

```
[474]: (90936, 85)
```

```
[480]: %%time
from sklearn.ensemble import VotingClassifier

lr = LogisticRegression(penalty='l2', C=50, max_iter=1500, solver='saga')
```

```

ada = AdaBoostClassifier(**{'algorithm': 'SAMME.R', 'learning_rate': 1.0,
    ↪ 'n_estimators': 200})
gbm = GradientBoostingClassifier(**{'learning_rate': 0.1, 'max_depth': 5,
    ↪ 'n_estimators': 200})
rf = RandomForestClassifier(** {'bootstrap': False, 'criterion': 'entropy',
    ↪ 'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 4,
    ↪ 'min_samples_split': 2, 'n_estimators': 200})
xgb = XGBClassifier(**{'colsample_bytree': 0.7, 'gamma': 0.1, 'learning_rate':
    ↪ 0.1, 'max_depth': 7, 'n_estimators': 200, 'reg_lambda': 1.5, 'subsample': 0.
    ↪ 9}, objective='multi:softprob')

# res = fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)

votingCLF = VotingClassifier(
    estimators=[
        ('rf', rf),
        ('ada', ada),
        ('xgb', xgb),
        ('lr', lr),
        ('gbm', gbm)
    ],
    voting='soft',
    weights=[4,5,7,2, 6]
)
# fit_and_evaluate(votingCLF, enc_trainX, trainY, enc_testX, testY)

```

CPU times: user 87 µs, sys: 0 ns, total: 87 µs
 Wall time: 89.4 µs

[]:

[481]: final = pd.read_csv("./sample_input_2nd_task.csv")

[477]: final_data = scaler.transform(encoder.transform(final))

[478]: final_data.shape

[478]: (33, 85)

[479]: votingCLF.predict(final_data)

 AttributeError

Traceback (most recent call last)

Cell In[479], line 1

----> 1 votingCLF.predict(final_data)

```

File ~/anaconda3/envs/mldl/lib/python3.9/site-packages/sklearn/ensemble/_voting
->py:383, in VotingClassifier.predict(self, X)
    381 check_is_fitted(self)
    382 if self.voting == "soft":
--> 383     maj = np.argmax(self.predict_proba(X), axis=1)
    385 else: # 'hard' voting
    386     predictions = self._predict(X)

```

```

File ~/anaconda3/envs/mldl/lib/python3.9/site-packages/sklearn/ensemble/_voting
->py:424, in VotingClassifier.predict_proba(self, X)
    410 """Compute probabilities of possible outcomes for samples in X.
    411
    412 Parameters
    (...)
    420     Weighted average probability for each class per sample.
    421 """
    422 check_is_fitted(self)
    423 avg = np.average(
--> 424     self._collect_proba(X), axis=0, weights=self._weights_not_none
    425 )
    426 return avg

```

```

File ~/anaconda3/envs/mldl/lib/python3.9/site-packages/sklearn/ensemble/_voting
->py:399, in VotingClassifier._collect_proba(self, X)
    397 def _collect_proba(self, X):
    398     """Collect results from clf.predict calls."""
--> 399     return np.asarray([clf.predict_proba(X) for clf in self.estimators_])

AttributeError: 'VotingClassifier' object has no attribute 'estimators_'

```

```

[280]: label_encoder = LabelEncoder()
       trainY_encoded = label_encoder.fit_transform(trainY)

```

```

[281]: label_encoder.inverse_transform(votingCLF.predict(final_data))

```

```

/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but RandomForestClassifier was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but AdaBoostClassifier was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but LogisticRegression was fitted with feature names
  warnings.warn(

```

```
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but GradientBoostingClassifier was fitted with feature names
  warnings.warn(
```

```
[281]: array(['LATE', 'LATE', 'LATE', 'EARLY', 'EARLY', 'EARLY', 'EARLY',
            'EARLY', 'EARLY', 'EARLY', 'EARLY', 'LATE', 'EARLY', 'EARLY',
            'EARLY', 'LATE', 'LATE', 'LATE', 'EARLY', 'EARLY', 'EARLY',
            'EARLY', 'EARLY', 'EARLY', 'LATE', 'LATE', 'LATE', 'EARLY',
            'EARLY', 'EARLY', 'EARLY', 'EARLY', 'EARLY'], dtype=object)
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

1.2 START 2 HOP

```
[491]: encoder = MultiColumnOneHotEncoder(columns=['Carrier Code', 'Origin_
↳Airport', 'season', 'SCHED_ARRV_TIME_CAT', 'month', 'WeekDay', 'PREV_STAT'])_
↳#, 'wind_type', 'ceiling_det_code', 'celing_CAVOK', 'visibility_variability'
```

```
[492]: encoded_data = encoder.fit_transform(df2.drop(columns=['FLIGHT_STATUS']))
```

```
[493]: encoded_data.head()
```

```
[493]: Scheduled Elapsed Time (Minutes)  latitude  elevation  wind_direction  \
0                                     76  40.63915         3.4      102.428571
1                                     76  40.63915         3.4      102.428571
2                                     75  40.63915         3.4      110.503072
3                                     75  40.63915         3.4      110.503072
4                                    100  41.96019        201.8      292.000000

      wind_speed  visibility_dist  air_temparature  sea_level_pressure  \
0      3.221194      6.966812      33.944      1018.725782
1      3.221194      6.966812      33.944      1018.725782
2      0.000000      5.400087      33.368      1015.367354
3      0.000000      5.400087      33.368      1015.367354
4      9.395148      9.999724       7.808      1024.780000
```

	SYR_wind_direction	SYR_wind_speed	...	month_11	month_12	\
0	106.000000	3.892276	...	0.0	0.0	
1	106.000000	3.892276	...	0.0	0.0	
2	103.912190	0.000000	...	0.0	0.0	
3	103.912190	0.000000	...	0.0	0.0	
4	132.389021	0.000000	...	0.0	0.0	

	WeekDay_Monday	WeekDay_Saturday	WeekDay_Sunday	WeekDay_Thursday	\
0	0.0	0.0	0.0	0.0	
1	0.0	0.0	0.0	0.0	
2	0.0	0.0	0.0	0.0	
3	0.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	0.0	

	WeekDay_Tuesday	WeekDay_Wednesday	PREV_STAT_LATE	PREV_STAT_ONTIME
0	0.0	0.0	0.0	1.0
1	0.0	0.0	0.0	1.0
2	0.0	0.0	1.0	0.0
3	0.0	0.0	0.0	1.0
4	0.0	0.0	1.0	0.0

[5 rows x 85 columns]

```
[494]: encoded_data.shape
```

```
[494]: (227342, 85)
```

```
[495]: trainX, testX, trainY, testY = train_test_split(
    encoded_data,
    df2['FLIGHT_STATUS'],
    test_size=0.2,
    random_state=947,
    stratify=df2['FLIGHT_STATUS']
)
```

```
[496]: from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
enc_trainX = pd.DataFrame(scaler.fit_transform(trainX), index=trainX.index,
    ↪columns=trainX.columns)
enc_testX = pd.DataFrame(scaler.transform(testX), index=testX.index,
    ↪columns=testX.columns)
```

```
[497]: from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report
```

```

def fit_and_evaluate(model, trainX, trainY, testX, testY):
    # Convert string labels to integers
    label_encoder = LabelEncoder()
    trainY_encoded = label_encoder.fit_transform(trainY)
    testY_encoded = label_encoder.transform(testY)

    model.fit(trainX, trainY_encoded)
    testY_pred = model.predict(testX)
    accuracy = accuracy_score(testY_encoded, testY_pred)
    report = classification_report(testY_encoded, testY_pred, output_dict=True)
    results = {'accuracy': accuracy, 'classification_report': report}
    return results

# Update other classification functions similarly...

# Random Forest
from sklearn.ensemble import RandomForestClassifier

def random_forest_classification(trainX, trainY, testX, testY,
    ↪n_estimators=100, criterion='gini', max_depth=None):
    model = RandomForestClassifier(n_estimators=n_estimators,
    ↪criterion=criterion, max_depth=max_depth)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# Support Vector Machines (SVM)
from sklearn.svm import SVC

def svm_classification(trainX, trainY, testX, testY, kernel='rbf', C=1.0):
    model = SVC(kernel=kernel, C=C)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# K-Nearest Neighbors (KNN)
from sklearn.neighbors import KNeighborsClassifier

def knn_classification(trainX, trainY, testX, testY, n_neighbors=5):
    model = KNeighborsClassifier(n_neighbors=n_neighbors)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# Gradient Boosting Machines (GBM)
from sklearn.ensemble import GradientBoostingClassifier

def gbm_classification(trainX, trainY, testX, testY, n_estimators=100,
    ↪learning_rate=0.1, max_depth=3):
    model = GradientBoostingClassifier(n_estimators=n_estimators,
    ↪learning_rate=learning_rate, max_depth=max_depth)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

```

```

# Naive Bayes
from sklearn.naive_bayes import GaussianNB

def naive_bayes_classification(trainX, trainY, testX, testY):
    model = GaussianNB()
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# AdaBoost
from sklearn.ensemble import AdaBoostClassifier

def adaboost_classification(trainX, trainY, testX, testY, n_estimators=50,
    ↪learning_rate=1.0):
    model = AdaBoostClassifier(n_estimators=n_estimators,
    ↪learning_rate=learning_rate)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# XGBoost
from xgboost import XGBClassifier

def xgboost_classification(trainX, trainY, testX, testY, n_estimators=100,
    ↪learning_rate=0.1, max_depth=3):
    model = XGBClassifier(n_estimators=n_estimators,
    ↪learning_rate=learning_rate, max_depth=max_depth)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

def logistic_regression_classification(trainX, trainY, testX, testY,
    ↪penalty='l2', C=1.0, max_iter=1000, solver='lbfgs'):
    target_names = trainY.unique() if isinstance(trainY, pd.Series) else testY.
    ↪unique()

    # Initialize and train the Logistic Regression model
    model = LogisticRegression(penalty=penalty, C=C, max_iter=max_iter,
    ↪solver=solver, verbose=1 if max_iter > 300 else 0)
    model.fit(trainX, trainY)

    # Predict on the testing set
    testY_pred = model.predict(testX)

    # Calculate accuracy score
    accuracy = accuracy_score(testY, testY_pred)

    # Generate classification report
    report = classification_report(testY, testY_pred,
    ↪target_names=target_names, output_dict=True)

```



```

    results = {
        'accuracy': accuracy,
        'classification_report': report
    }

    return results

def decision_tree_classification(trainX, trainY, testX, testY,
    ↪criterion='gini', max_depth=None):

    # Get unique target names
    target_names = trainY.unique() if isinstance(trainY, pd.Series) else testY.
    ↪unique()

    # Initialize and train the Decision Tree model
    model = DecisionTreeClassifier(criterion=criterion,
    ↪max_depth=max_depth,min_samples_split=5)
    model.fit(trainX, trainY)

    # Predict on the testing set
    testY_pred = model.predict(testX)

    # Calculate accuracy score
    accuracy = accuracy_score(testY, testY_pred)

    # Generate classification report
    report = classification_report(testY, testY_pred,
    ↪target_names=target_names, output_dict=True)

    results = {
        'accuracy': accuracy,
        'classification_report': report
    }

    return results

```

```

[498]: # logistic_regression_classification(enc_trainX, trainY, enc_testX, testY,
    ↪solver='saga', max_iter=1500, C=50)

```

```

[499]: # decision_tree_classification(enc_trainX, trainY, enc_testX, testY,
    ↪max_depth=11, criterion='entropy')

```

```

[500]: # random_forest_classification(enc_trainX, trainY, enc_testX, testY)

```

```

[501]: # knn_classification(enc_trainX, trainY, enc_testX, testY)

```

```

[502]: # gbm_classification(enc_trainX, trainY, enc_testX, testY)

```

```
[503]: # naive_bayes_classification(enc_trainX, trainY, enc_testX, testY)

[504]: # adaboost_classification(enc_trainX, trainY, enc_testX, testY)

[505]: # xgboost_classification(enc_trainX, trainY, enc_testX, testY)

[506]: from sklearn.model_selection import GridSearchCV, KFold
def hyperparameter_tuning(model, param_grid, trainX, trainY, cv=5):
    label_encoder = LabelEncoder()
    trainY_encoded = label_encoder.fit_transform(trainY)
    # Initialize K-Fold cross-validator
    kf = KFold(n_splits=cv, shuffle=True, random_state=42)

    # Perform grid search
    grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=kf,
    ↪scoring='accuracy', verbose=3, n_jobs=10, )
    grid_search.fit(trainX, trainY_encoded)

    return grid_search

[507]: params = {'colsample_bytree': 0.7, 'gamma': 0.1, 'learning_rate': 0.1,
    ↪'max_depth': 7, 'n_estimators': 200, 'reg_lambda': 1.5, 'subsample': 0.9}
# param_grid = {
#     'n_estimators': [100, 200], # Number of boosting rounds
#     'max_depth': [3, 5, 7, 11], # Maximum tree depth
#     'learning_rate': [0.01, 0.1, 0.3], # Step size shrinkage
#     'subsample': [0.7, 0.9], # Subsample ratio of the training instances
#     'colsample_bytree': [0.7, 0.9], # Subsample ratio of columns when
    ↪constructing each tree
#     'gamma': [0, 0.1], # Minimum loss reduction required to make a further
    ↪partition on a leaf node of the tree
#     'reg_lambda': [1, 1.5, 2] # L2 regularization term on weights
# }
# model = XGBClassifier(**params)
# best_model = hyperparameter_tuning(model, param_grid, enc_trainX, trainY)
# fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)
# Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)

[508]: params = {'bootstrap': False, 'criterion': 'entropy', 'max_depth': None,
    ↪'max_features': 'sqrt', 'min_samples_leaf': 4, 'min_samples_split': 2,
    ↪'n_estimators': 200}
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
```

```

    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt'],
    'bootstrap': [True, False],
    'criterion': ['gini', 'entropy']
}

# model = RandomForestClassifier(**params)
# fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)
# best_model = hyperparameter_tuning(model, param_grid, enc_trainX, trainY)
# Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)

```

```

[509]: params = {'algorithm': 'SAMME.R', 'learning_rate': 1.0, 'n_estimators': 200}
param_grid = {
    'n_estimators': [50, 100, 200], # Number of estimators
    'learning_rate': [0.01, 0.1, 1.0], # Learning rate
    'algorithm': ['SAMME', 'SAMME.R'], # Algorithm
}

# model = AdaBoostClassifier(**params)
# fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)
# best_model = hyperparameter_tuning(model, param_grid, enc_trainX, trainY)
# # Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)

```

```

[510]: params = {'ccp_alpha': 0.07, 'criterion': 'gini', 'max_depth': None,
    ↪ 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2,
    ↪ 'splitter': 'best'}

param_grid = {
    'criterion': ['gini', 'entropy'], # Split criterion
    'splitter': ['best', 'random'], # Strategy to choose split at each node
    'max_depth': [None, 1, 3, 5], # Max depth of the tree
    'min_samples_split': [2, 5, 10], # Min samples required to split a node
    'min_samples_leaf': [1, 2, 4], # Min samples required at each leaf node
    'max_features': ['sqrt', 'log2'], # Max features to consider for split
    'ccp_alpha': [0.07, 0.01]
}

# model = DecisionTreeClassifier(**params)
# fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)
# best_model = hyperparameter_tuning(model, param_grid, enc_trainX, trainY)
# # Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)

```

```
[511]: gbm_params = {
    'learning_rate': [0.01, 0.05, 0.1], # Learning rate
    'n_estimators': [50, 100, 200], # Number of boosting stages
    'max_depth': [3, 4, 5], # Maximum depth of the individual trees
}
# model = GradientBoostingClassifier()
# best_model = hyperparameter_tuning(model, gbm_params, enc_trainX, trainY)
# # Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)
```

```
[517]: %%time
from sklearn.ensemble import VotingClassifier

lr = LogisticRegression(penalty='l2', C=50, max_iter=1500, solver='saga')
ada = AdaBoostClassifier(**{'algorithm': 'SAMME.R', 'learning_rate': 1.0,
    ↪ 'n_estimators': 200})
gbm = GradientBoostingClassifier(**{'learning_rate': 0.1, 'max_depth': 5,
    ↪ 'n_estimators': 200})
rf = RandomForestClassifier(** {'bootstrap': False, 'criterion': 'entropy',
    ↪ 'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 4,
    ↪ 'min_samples_split': 2, 'n_estimators': 200})
xgb = XGBClassifier(**{'colsample_bytree': 0.7, 'gamma': 0.1, 'learning_rate':
    ↪ 0.1, 'max_depth': 7, 'n_estimators': 200, 'reg_lambda': 1.5, 'subsample': 0.
    ↪ 9}, objective='multi:softprob')

# res = fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)

votingCLF = VotingClassifier(
    estimators=[
        ('rf', rf),
        ('ada', ada),
        ('xgb', xgb),
        ('lr', lr),
        ('gbm', gbm)
    ],
    voting='soft',
    weights=[4,5,7,2, 6]
)
# fit_and_evaluate(votingCLF, enc_trainX, trainY, enc_testX, testY)
```

CPU times: user 111 µs, sys: 1 µs, total: 112 µs
 Wall time: 117 µs

```
[518]: INPUT = encoded_data
TARGET=df2['FLIGHT_STATUS']
label_encoder = LabelEncoder()
```

```
TARGET_ENCODED = label_encoder.fit_transform(TARGET)

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
enc_trainX = pd.DataFrame(scaler.fit_transform(encoded_data),
    ↪ index=encoded_data.index, columns=encoded_data.columns)
```

```
[519]: enc_trainX.shape
```

```
[519]: (227342, 85)
```

```
[520]: %%time
votingCLF.fit(enc_trainX, TARGET_ENCODED)
```

```
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/ensemble/_weight_boosting.py:519: FutureWarning: The SAMME.R
algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
```

```
warnings.warn(
```

```
CPU times: user 14min 18s, sys: 4.07 s, total: 14min 22s
```

```
Wall time: 13min 12s
```

```
[520]: VotingClassifier(estimators=[('rf',
    RandomForestClassifier(bootstrap=False,
                           criterion='entropy',
                           min_samples_leaf=4,
                           n_estimators=200)),
    ('ada', AdaBoostClassifier(n_estimators=200)),
    ('xgb',
     XGBClassifier(base_score=None, booster=None,
                   callbacks=None,
                   colsample_bylevel=None,
                   colsample_bynode=None,
                   colsample_bytree=0.7, device=None,
                   early_stopping_rounds=None,
                   ena...
                   max_delta_step=None, max_depth=7,
                   max_leaves=None,
                   min_child_weight=None, missing=nan,
                   monotone_constraints=None,
                   multi_strategy=None,
                   n_estimators=200, n_jobs=None,
                   num_parallel_tree=None,
                   objective='multi:softprob', ...)),
    ('lr',
     LogisticRegression(C=50, max_iter=1500,
                       solver='saga'))],
```

```

        ('gbm',
         GradientBoostingClassifier(max_depth=5,
                                   n_estimators=200))),
        voting='soft', weights=[4, 5, 7, 2, 6])

```

```
[527]: final = pd.read_csv("./sample_input_2nd_task.csv")
```

```
[528]: final_data = scaler.transform(encoder.transform(final))
```

```
[529]: final_data.shape
```

```
[529]: (33, 85)
```

```
[530]: votingCLF.predict(final_data)
```

```

/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but RandomForestClassifier was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but AdaBoostClassifier was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but LogisticRegression was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but GradientBoostingClassifier was fitted with feature names
  warnings.warn(

```

```
[530]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
              0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0])
```

```
[531]: label_encoder = LabelEncoder()
trainY_encoded = label_encoder.fit_transform(trainY)
```

```
[532]: label_encoder.inverse_transform(votingCLF.predict(final_data))
```

```

/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but RandomForestClassifier was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but AdaBoostClassifier was fitted with feature names
  warnings.warn(

```

```

/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but LogisticRegression was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but GradientBoostingClassifier was fitted with feature names
  warnings.warn(

```

```

[532]: array(['EARLY', 'EARLY', 'EARLY', 'EARLY', 'EARLY', 'EARLY', 'EARLY',
'EARLY', 'EARLY', 'EARLY', 'EARLY', 'LATE', 'EARLY', 'EARLY',
'EARLY', 'LATE', 'LATE', 'LATE', 'EARLY', 'EARLY', 'EARLY',
'EARLY', 'EARLY', 'EARLY', 'LATE', 'LATE', 'LATE', 'EARLY',
'EARLY', 'EARLY', 'EARLY', 'EARLY', 'EARLY'], dtype=object)

```

```

[533]: final = pd.read_csv("./final_input_second.csv")

```

```

[534]: final_data = scaler.transform(encoder.transform(final))

```

```

[535]: final_data.shape

```

```

[535]: (33, 85)

```

```

[536]: votingCLF.predict(final_data)

```

```

/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but RandomForestClassifier was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but AdaBoostClassifier was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but LogisticRegression was fitted with feature names
  warnings.warn(
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,
but GradientBoostingClassifier was fitted with feature names
  warnings.warn(

```

```

[536]: array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1])

```

```

[537]: label_encoder = LabelEncoder()
trainY_encoded = label_encoder.fit_transform(trainY)

```

```
[538]: label_encoder.inverse_transform(votingCLF.predict(final_data))
```

```
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-  
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,  
but RandomForestClassifier was fitted with feature names  
  warnings.warn(  
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-  
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,  
but AdaBoostClassifier was fitted with feature names  
  warnings.warn(  
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-  
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,  
but LogisticRegression was fitted with feature names  
  warnings.warn(  
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-  
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,  
but GradientBoostingClassifier was fitted with feature names  
  warnings.warn(  

```

```
[538]: array(['EARLY', 'EARLY', 'EARLY', 'EARLY', 'EARLY', 'EARLY', 'EARLY',  
            'LATE', 'LATE', 'EARLY', 'EARLY', 'EARLY', 'EARLY', 'EARLY',  
            'EARLY', 'EARLY', 'EARLY', 'EARLY', 'EARLY', 'EARLY', 'EARLY',  
            'EARLY', 'EARLY', 'LATE', 'EARLY', 'EARLY', 'EARLY', 'EARLY',  
            'EARLY', 'EARLY', 'LATE', 'LATE', 'LATE'], dtype=object)
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

1.3 START 3 HOP

```
[310]: encoder = MultiColumnOneHotEncoder(columns=['Carrier Code', 'Origin_  
        ↳Airport', 'season', 'SCHED_ARRV_TIME_CAT', 'month', 'WeekDay', 'PREV_STAT'])_  
        ↳#, 'wind_type', 'ceiling_det_code', 'celing_CAVOK', 'visibility_variability'
```

```
[311]: encoded_data = encoder.fit_transform(df3.drop(columns=['FLIGHT_STATUS']))
```



```
[312]: encoded_data.head()
```

```
[312]: Scheduled Elapsed Time (Minutes)  latitude  elevation  wind_direction  \
0          76  40.63915          3.4      102.428571
1          76  40.63915          3.4      102.428571
2          76  40.63915          3.4      102.428571
3          75  40.63915          3.4      110.503072
4          75  40.63915          3.4      110.503072

    wind_speed  visibility_dist  air_temparature  sea_level_pressure  \
0    3.221194      6.966812          33.944      1018.725782
1    3.221194      6.966812          33.944      1018.725782
2    3.221194      6.966812          33.944      1018.725782
3    0.000000      5.400087          33.368      1015.367354
4    0.000000      5.400087          33.368      1015.367354

    SYR_wind_direction  SYR_wind_speed  ...  month_11  month_12  \
0          106.00000      3.892276  ...      0.0      0.0
1          106.00000      3.892276  ...      0.0      0.0
2          106.00000      3.892276  ...      0.0      0.0
3          103.91219      0.000000  ...      0.0      0.0
4          103.91219      0.000000  ...      0.0      0.0

    WeekDay_Monday  WeekDay_Saturday  WeekDay_Sunday  WeekDay_Thursday  \
0              0.0              0.0              0.0              0.0
1              0.0              0.0              0.0              0.0
2              0.0              0.0              0.0              0.0
3              0.0              0.0              0.0              0.0
4              0.0              0.0              0.0              0.0

    WeekDay_Tuesday  WeekDay_Wednesday  PREV_STAT_LATE  PREV_STAT_ONTIME
0              0.0              0.0              0.0              1.0
1              0.0              0.0              0.0              1.0
2              0.0              0.0              0.0              1.0
3              0.0              0.0              1.0              0.0
4              0.0              0.0              0.0              1.0
```

```
[5 rows x 85 columns]
```

```
[313]: encoded_data.shape
```

```
[313]: (341013, 85)
```

```
[314]: trainX, testX, trainY, testY = train_test_split(
    encoded_data,
    df3['FLIGHT_STATUS'],
    test_size=0.2,
```

```

        random_state=947,
        stratify=df3['FLIGHT_STATUS']
    )

```

```

[315]: from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
enc_trainX = pd.DataFrame(scaler.fit_transform(trainX), index=trainX.index,
    ↪ columns=trainX.columns)
enc_testX = pd.DataFrame(scaler.transform(testX), index=testX.index,
    ↪ columns=testX.columns)

```

```

[316]: from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

def fit_and_evaluate(model, trainX, trainY, testX, testY):
    # Convert string labels to integers
    label_encoder = LabelEncoder()
    trainY_encoded = label_encoder.fit_transform(trainY)
    testY_encoded = label_encoder.transform(testY)

    model.fit(trainX, trainY_encoded)
    testY_pred = model.predict(testX)
    accuracy = accuracy_score(testY_encoded, testY_pred)
    report = classification_report(testY_encoded, testY_pred, output_dict=True)
    results = {'accuracy': accuracy, 'classification_report': report}
    return results

# Update other classification functions similarly...

# Random Forest
from sklearn.ensemble import RandomForestClassifier

def random_forest_classification(trainX, trainY, testX, testY,
    ↪ n_estimators=100, criterion='gini', max_depth=None):
    model = RandomForestClassifier(n_estimators=n_estimators,
    ↪ criterion=criterion, max_depth=max_depth)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# Support Vector Machines (SVM)
from sklearn.svm import SVC

def svm_classification(trainX, trainY, testX, testY, kernel='rbf', C=1.0):
    model = SVC(kernel=kernel, C=C)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

```

```

# K-Nearest Neighbors (KNN)
from sklearn.neighbors import KNeighborsClassifier

def knn_classification(trainX, trainY, testX, testY, n_neighbors=5):
    model = KNeighborsClassifier(n_neighbors=n_neighbors)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# Gradient Boosting Machines (GBM)
from sklearn.ensemble import GradientBoostingClassifier

def gbm_classification(trainX, trainY, testX, testY, n_estimators=100,
    ↪learning_rate=0.1, max_depth=3):
    model = GradientBoostingClassifier(n_estimators=n_estimators,
    ↪learning_rate=learning_rate, max_depth=max_depth)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# Naive Bayes
from sklearn.naive_bayes import GaussianNB

def naive_bayes_classification(trainX, trainY, testX, testY):
    model = GaussianNB()
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# AdaBoost
from sklearn.ensemble import AdaBoostClassifier

def adaboost_classification(trainX, trainY, testX, testY, n_estimators=50,
    ↪learning_rate=1.0):
    model = AdaBoostClassifier(n_estimators=n_estimators,
    ↪learning_rate=learning_rate)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

# XGBoost
from xgboost import XGBClassifier

def xgboost_classification(trainX, trainY, testX, testY, n_estimators=100,
    ↪learning_rate=0.1, max_depth=3):
    model = XGBClassifier(n_estimators=n_estimators,
    ↪learning_rate=learning_rate, max_depth=max_depth)
    return fit_and_evaluate(model, trainX, trainY, testX, testY)

def logistic_regression_classification(trainX, trainY, testX, testY,
    ↪penalty='l2', C=1.0, max_iter=1000, solver='lbfgs'):
    target_names = trainY.unique() if isinstance(trainY, pd.Series) else testY.
    ↪unique()

```

```

    # Initialize and train the Logistic Regression model
    model = LogisticRegression(penalty=penalty, C=C, max_iter=max_iter,
    ↪solver=solver, verbose=1 if max_iter > 300 else 0)
    model.fit(trainX, trainY)

    # Predict on the testing set
    testY_pred = model.predict(testX)

    # Calculate accuracy score
    accuracy = accuracy_score(testY, testY_pred)

    # Generate classification report
    report = classification_report(testY, testY_pred,
    ↪target_names=target_names, output_dict=True)

    results = {
        'accuracy': accuracy,
        'classification_report': report
    }

    return results

def decision_tree_classification(trainX, trainY, testX, testY,
    ↪criterion='gini', max_depth=None):

    # Get unique target names
    target_names = trainY.unique() if isinstance(trainY, pd.Series) else testY.
    ↪unique()

    # Initialize and train the Decision Tree model
    model = DecisionTreeClassifier(criterion=criterion,
    ↪max_depth=max_depth, min_samples_split=5)
    model.fit(trainX, trainY)

    # Predict on the testing set
    testY_pred = model.predict(testX)

    # Calculate accuracy score
    accuracy = accuracy_score(testY, testY_pred)

    # Generate classification report
    report = classification_report(testY, testY_pred,
    ↪target_names=target_names, output_dict=True)

    results = {

```

```

        'accuracy': accuracy,
        'classification_report': report
    }

    return results

```

```

[317]: # logistic_regression_classification(enc_trainX, trainY, enc_testX, testY,
      ↪ solver='saga', max_iter=1500, C=50)

```

```

[318]: # decision_tree_classification(enc_trainX, trainY, enc_testX, testY,
      ↪ max_depth=11, criterion='entropy')

```

```

[319]: # random_forest_classification(enc_trainX, trainY, enc_testX, testY)

```

```

[320]: # knn_classification(enc_trainX, trainY, enc_testX, testY)

```

```

[321]: # gbm_classification(enc_trainX, trainY, enc_testX, testY)

```

```

[322]: # naive_bayes_classification(enc_trainX, trainY, enc_testX, testY)

```

```

[323]: # adaboost_classification(enc_trainX, trainY, enc_testX, testY)

```

```

[324]: # xgboost_classification(enc_trainX, trainY, enc_testX, testY)

```

```

[325]: from sklearn.model_selection import GridSearchCV, KFold
def hyperparameter_tuning(model, param_grid, trainX, trainY, cv=5):
    label_encoder = LabelEncoder()
    trainY_encoded = label_encoder.fit_transform(trainY)
    # Initialize K-Fold cross-validator
    kf = KFold(n_splits=cv, shuffle=True, random_state=42)

    # Perform grid search
    grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=kf,
    ↪ scoring='accuracy', verbose=3, n_jobs=10, )
    grid_search.fit(trainX, trainY_encoded)

    return grid_search

```

```

[326]: params = {'colsample_bytree': 0.7, 'gamma': 0.1, 'learning_rate': 0.1,
      ↪ 'max_depth': 7, 'n_estimators': 200, 'reg_lambda': 1.5, 'subsample': 0.9}
# param_grid = {
#     'n_estimators': [100, 200], # Number of boosting rounds
#     'max_depth': [3, 5, 7, 11], # Maximum tree depth
#     'learning_rate': [0.01, 0.1, 0.3], # Step size shrinkage
#     'subsample': [0.7, 0.9], # Subsample ratio of the training instances
#     'colsample_bytree': [0.7, 0.9], # Subsample ratio of columns when
      ↪ constructing each tree

```

```

#     'gamma': [0, 0.1], # Minimum loss reduction required to make a further
    ↪ partition on a leaf node of the tree
#     'reg_lambda': [1, 1.5, 2] # L2 regularization term on weights
# }
# model = XGBClassifier(**params)
# best_model = hyperparameter_tuning(model, param_grid, enc_trainX, trainY)
# fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)
# Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)

```

```

[327]: params = {'bootstrap': False, 'criterion': 'entropy', 'max_depth': None,
    ↪ 'max_features': 'sqrt', 'min_samples_leaf': 4, 'min_samples_split': 2,
    ↪ 'n_estimators': 200}
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt'],
    'bootstrap': [True, False],
    'criterion': ['gini', 'entropy']
}
# model = RandomForestClassifier(**params)
# fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)
# best_model = hyperparameter_tuning(model, param_grid, enc_trainX, trainY)
# Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)

```

```

[328]: params = {'algorithm': 'SAMME.R', 'learning_rate': 1.0, 'n_estimators': 200}
param_grid = {
    'n_estimators': [50, 100, 200], # Number of estimators
    'learning_rate': [0.01, 0.1, 1.0], # Learning rate
    'algorithm': ['SAMME', 'SAMME.R'], # Algorithm
}

# model = AdaBoostClassifier(**params)
# fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)
# best_model = hyperparameter_tuning(model, param_grid, enc_trainX, trainY)
# # Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)

```

```

[329]: params = {'ccp_alpha': 0.07, 'criterion': 'gini', 'max_depth': None,
    ↪ 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2,
    ↪ 'splitter': 'best'}

```

```

param_grid = {
    'criterion': ['gini', 'entropy'], # Split criterion
    'splitter': ['best', 'random'], # Strategy to choose split at each node
    'max_depth': [None, 1, 3, 5], # Max depth of the tree
    'min_samples_split': [2, 5, 10], # Min samples required to split a node
    'min_samples_leaf': [1, 2, 4], # Min samples required at each leaf node
    'max_features': ['sqrt', 'log2'], # Max features to consider for split
    'ccp_alpha': [0.07, 0.01]
}

# model = DecisionTreeClassifier(**params)
# fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)
# best_model = hyperparameter_tuning(model, param_grid, enc_trainX, trainY)
# # Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)

```

```

[330]: gbm_params = {
    'learning_rate': [0.01, 0.05, 0.1], # Learning rate
    'n_estimators': [50, 100, 200], # Number of boosting stages
    'max_depth': [3, 4, 5], # Maximum depth of the individual trees
}

# model = GradientBoostingClassifier()
# best_model = hyperparameter_tuning(model, gbm_params, enc_trainX, trainY)
# # Print best parameters and best score
# print("Best parameters found: ", best_model.best_params_)
# print("Best accuracy score found: ", best_model.best_score_)

```

```

[331]: %%time
from sklearn.ensemble import VotingClassifier

lr = LogisticRegression(penalty='l2', C=50, max_iter=1500, solver='saga')
ada = AdaBoostClassifier(**{'algorithm': 'SAMME.R', 'learning_rate': 1.0,
    ↪ 'n_estimators': 200})
gbm = GradientBoostingClassifier(**{'learning_rate': 0.1, 'max_depth': 5,
    ↪ 'n_estimators': 200})
rf = RandomForestClassifier(**{'bootstrap': False, 'criterion': 'entropy',
    ↪ 'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 4,
    ↪ 'min_samples_split': 2, 'n_estimators': 200})
xgb = XGBClassifier(**{'colsample_bytree': 0.7, 'gamma': 0.1, 'learning_rate':
    ↪ 0.1, 'max_depth': 7, 'n_estimators': 200, 'reg_lambda': 1.5, 'subsample': 0.
    ↪ 9}, objective='multi:softprob')

# res = fit_and_evaluate(model, enc_trainX, trainY, enc_testX, testY)

votingCLF = VotingClassifier(

```

```

    estimators=[
        ('rf', rf),
        ('ada', ada),
        ('xgb', xgb),
        ('lr', lr),
        ('gbm', gbm)
    ],
    voting='soft',
    weights=[4,5,7,2, 6]
)
fit_and_evaluate(votingCLF, enc_trainX, trainY, enc_testX, testY)

```

/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-packages/sklearn/ensemble/_weight_boosting.py:519: FutureWarning: The SAMME.R algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME algorithm to circumvent this warning.

```
warnings.warn(
```

CPU times: user 18min 21s, sys: 1.58 s, total: 18min 22s

Wall time: 16min 38s

```

[331]: {'accuracy': 0.667111417386332,
       'classification_report': {'0': {'precision': 0.6275531067731314,
                                         'recall': 0.9302577845839715,
                                         'f1-score': 0.7494955466731785,
                                         'support': 31344.0},
                                '1': {'precision': 0.705195563339171,
                                         'recall': 0.5903049257232212,
                                         'f1-score': 0.6426557429377028,
                                         'support': 20464.0},
                                '2': {'precision': 0.924295010845987,
                                         'recall': 0.2598963098505642,
                                         'f1-score': 0.40571292549393,
                                         'support': 16395.0},
                                'accuracy': 0.667111417386332,
                                'macro avg': {'precision': 0.7523478936527631,
                                                'recall': 0.5934863400525856,
                                                'f1-score': 0.5992880717016037,
                                                'support': 68203.0},
                                'weighted avg': {'precision': 0.7221817704454317,
                                                  'recall': 0.667111417386332,
                                                  'f1-score': 0.6347984539078082,
                                                  'support': 68203.0}}}

```

```
[332]: final = pd.read_csv("./sample_input_2nd_task.csv")
```

```
[333]: final_data = scaler.transform(encoder.transform(final))
```



```
[334]: final_data.shape
```

```
[334]: (33, 85)
```

```
[335]: votingCLF.predict(final_data)
```

```
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-  
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,  
but RandomForestClassifier was fitted with feature names  
  warnings.warn(  
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-  
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,  
but AdaBoostClassifier was fitted with feature names  
  warnings.warn(  
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-  
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,  
but LogisticRegression was fitted with feature names  
  warnings.warn(  
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-  
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,  
but GradientBoostingClassifier was fitted with feature names  
  warnings.warn(  

```

```
[335]: array([0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0,  
          0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0])
```

```
[336]: label_encoder = LabelEncoder()  
trainY_encoded = label_encoder.fit_transform(trainY)
```

```
[337]: label_encoder.inverse_transform(votingCLF.predict(final_data))
```

```
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-  
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,  
but RandomForestClassifier was fitted with feature names  
  warnings.warn(  
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-  
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,  
but AdaBoostClassifier was fitted with feature names  
  warnings.warn(  
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-  
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,  
but LogisticRegression was fitted with feature names  
  warnings.warn(  
/home/numan947/anaconda3/envs/mldl/lib/python3.9/site-  
packages/sklearn/base.py:493: UserWarning: X does not have valid feature names,  
but GradientBoostingClassifier was fitted with feature names  
  warnings.warn(  

```

```
[337]: array(['EARLY', 'EARLY', 'LATE', 'EARLY', 'EARLY', 'EARLY', 'EARLY',  
        'EARLY', 'EARLY', 'LATE', 'LATE', 'LATE', 'EARLY', 'EARLY',  
        'EARLY', 'LATE', 'LATE', 'LATE', 'EARLY', 'EARLY', 'EARLY',  
        'EARLY', 'EARLY', 'EARLY', 'LATE', 'LATE', 'LATE', 'EARLY',  
        'EARLY', 'EARLY', 'EARLY', 'EARLY', 'EARLY'], dtype=object)
```

[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	

[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	
[]:	