project_with_output

January 10, 2021

1 FDS Mini Project

WARNING: Before making any git commit to this notebook please clear all output in this notebook

1.1 1. Cleaning the data

1.1.1 Invalid Columns:

- delete unnamed column which was serving as index (index already exists - duplicated column)
- delete last column (contains only NaN values) - 'Unnamed 21'

1.1.2 NaN values:

- check number of NaN values/location of NaN values
- leave NaN values that are required in order not to lose data (for example: a cancelled flight will always have NaN values for DEP_TIME, ARR_TIME, ARR_DEL15, DEP_DEL15 - as the flight did not happen)
- delete NaN values that would incommodate analysis and plotting later on (for example, flight timings that are simply missing without the flight having been cancelled)

1.1.3 Times conversion (Note: 00:00 timings all represent cancelled flights)

- observation –> no flight leaves at 00:00, all *00:00 date/time values belong to flights that have been cancelled*
- converted DEP_TIME and ARR_TIME to 4-character string of the format: hhmm (error when attempting to convert to date/time)
- added two extra columns: ARR_TIME_MINS and DEP_TIME_MINS representing the arrival and departure time in minutes for easier calculations

1.1.4 Irrelevant columns (to this project) to be removed/ duplicated data:

- Remove both OP_CARRIER_AIRLINE_ID and OP_CARRIER
- Remove ORIGIN_AIRPORT_SEQ_ID
- Remove DEST_AIRPORT_SEQ_ID

[1]:
```
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
import seaborn as sns
#Importing sklearn functions
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.model_selection import cross_val_score
from sklearn.cluster import KMeans
```

[2]:

```
#--------------------------------------- Load dataset␣
↪-------------------------------------------#
flight_data_path = os.path.join(os.getcwd(), 'datasets', 'flight_jan_2019.csv.
↪gz')
flight_data = pd.read_csv(flight_data_path, compression = 'gzip')

# Delete 'Unnamed 1' and 'Unnamed 21'
del flight_data['Unnamed: 0']
del flight_data['Unnamed: 21']
flight_data

#---------------------------------------- Check for 'NaN' values␣
↪-------------------------------#

# for col in flight_data.columns:
#    print(col, ' :',flight_data[col].isna().sum())

    # NA VALUES: TAIL_NUM  : 2543
    #            DEP_TIME  : 16352
    #            DEP_DEL15  : 16355
    #            ARR_TIME  : 17061
    #            ARR_DEL15  : 18022
    #            Unnamed: 21  : 583985

# Dealing with DEP_TIME and ARR_TIME Nan values
flight_data[np.isnan(flight_data.DEP_TIME)] # Observation: cancelled flights␣
↪have Nan values for DEP_TIME, ARR_TIME, DEP_DEL15,ARR_DEL15
# NaN values therefore make sense in this case, eliminating rows with NaN␣
↪values with plotting can be done by filtering:
#                       flight_data[~np.
↪isnan(flight_data['DEP_TIME'])]['DEP_TIME'].isna().sum()

# Eliminate rows with NaN values in place for DEP/ARR_DELL15 AND ARR_TIME where␣
↪the DEP_TIME is registered (timings simply missing)
```

```
indices_to_eliminate = list(flight_data[(~np.
↪isnan(flight_data['DEP_TIME']))][np.isnan(flight_data['DEP_DEL15'])].index.
↪values) + list(flight_data[(~np.isnan(flight_data['DEP_TIME']))][np.
↪isnan(flight_data['ARR_TIME'])].index.values) + list(flight_data[(~np.
↪isnan(flight_data['DEP_TIME']))][np.isnan(flight_data['ARR_DEL15'])].index.
↪values)
flight_data = flight_data.drop(indices_to_eliminate)

#--------------------------------------Modifying data␣
↪types----------------------------------#
flight_data.dtypes
# CANCELLED/DIVERTED to integer value
flight_data['CANCELLED'] = flight_data['CANCELLED'].astype(int)
flight_data['DIVERTED'] = flight_data['DIVERTED'].astype(int)
flight_data.dtypes
flight_data
# Modifying timings date/time format
#flight_data['DEP_TIME'] = pd.to_datetime(flight_data['DEP_TIME'],␣
↪format='%H%M').dt.time

# OBSERVATION: flights with value 0.0 - keeping in mind that timings are␣
↪currently floats - are all NaN values - so no flight leaves at 00:00 (those␣
↪are simply cancelled values)
len(flight_data[(flight_data['DEP_TIME'] == 0.0)][flight_data['CANCELLED'] ==␣
↪1]['DEP_TIME']) - flight_data[flight_data['DEP_TIME'] == 0.0]['DEP_TIME'].
↪isna().sum()
len(flight_data[(flight_data['DEP_TIME'] == 0.0)][flight_data['CANCELLED'] ==␣
↪1]['DEP_TIME']) - flight_data[flight_data['DEP_TIME'] == 0.0]['DEP_TIME'].
↪isna().sum()

# Convert DEP_TIME and ARR_TIME to int and add new columns: DEP_TIME_MINS and␣
↪ARR_TIME_MINS for easy calculations
def convert_minutes(x):
    minutes = int(x[2])*10 + int(x[3])
    hr_minutes = (int(x[0])*10 + int(x[1]))*60
    return minutes+hr_minutes

def fill_in(x):
    if (len(x) == 4):
        return x
    if (len(x) == 3):
        return '0' + x
    if (len(x) == 2):
        return '00' + x
    if (len(x) == 1):
        return '000' + x
```

```python
    if (len(x) == 0):
        return '000' + x
    return '0000'

flight_data['DEP_TIME'] = flight_data['DEP_TIME'].fillna(0)
flight_data['DEP_TIME'] = flight_data['DEP_TIME'].astype(int)
flight_data['DEP_TIME'] = flight_data['DEP_TIME'].astype(str)
flight_data['DEP_TIME'] = flight_data['DEP_TIME'].apply(fill_in)
flight_data['DEP_TIME_MINS'] = flight_data['DEP_TIME'].apply(convert_minutes)
flight_data['ARR_TIME'] = flight_data['ARR_TIME'].fillna(0)
flight_data['ARR_TIME'] = flight_data['ARR_TIME'].astype(int)
flight_data['ARR_TIME'] = flight_data['ARR_TIME'].astype(str)
flight_data['ARR_TIME'] = flight_data['ARR_TIME'].apply(fill_in)
flight_data['ARR_TIME_MINS'] = flight_data['ARR_TIME'].apply(convert_minutes)

#-------------------------------ATTEMPT AT CONVERTING TO DATE/
 ↪TIME-----------------#
def fill_in(x):
    if (len(x) == 4):
        return x
    if (len(x) == 3):
        return '0' + x
    if (len(x) == 2):
        return '00' + x
    if (len(x) == 1):
        return '000' + x
    if (len(x) == 0):
        return '000' + x
    return '0000'

#def convert_time(x):
#    return datetime.datetime.strptime(x,'%H%M' )

#flight_data['DEP_TIME'] = flight_data['DEP_TIME'].apply(fill_in)
#flight_data['ARR_TIME'] = flight_data['ARR_TIME'].apply(fill_in)
#flight_data['DEP_TIME'] = flight_data['DEP_TIME'].apply(convert_time)
#flight_data['DEP_TIME'] = flight_data['DEP_TIME'].apply(check)
#flight_data['DEP_TIME'] = pd.to_datetime(flight_data['DEP_TIME'], format=)


#------------------------------------Eliminating extra␣
 ↪columns------------------------------#

flight_data['OP_UNIQUE_CARRIER'].nunique()  # 17
flight_data['OP_CARRIER_AIRLINE_ID'].nunique()  # 17
flight_data['OP_CARRIER'].nunique() # 17
# Remove both OP_CARRIER_AIRLINE_ID and OP_CARRIER
```

```
del flight_data['OP_CARRIER_AIRLINE_ID']
del flight_data['OP_CARRIER']

flight_data['TAIL_NUM'].nunique() # 5445
flight_data['ORIGIN_AIRPORT_ID'].nunique() # 346
flight_data['ORIGIN_AIRPORT_SEQ_ID'].nunique() # 346
# Remove ORIGIN_AIRPORT_SEQ_ID
del flight_data['ORIGIN_AIRPORT_SEQ_ID']

flight_data['DEST_AIRPORT_ID'].nunique() # 346
flight_data['DEST_AIRPORT_SEQ_ID'].nunique() # 346
# Remove DEST_AIRPORT_SEQ_ID
del flight_data['DEST_AIRPORT_SEQ_ID']

del flight_data['ORIGIN_AIRPORT_ID']
del flight_data['DEST_AIRPORT_ID']

flight_data.head()
```

[2]:

```
   DAY_OF_MONTH  DAY_OF_WEEK OP_UNIQUE_CARRIER TAIL_NUM  OP_CARRIER_FL_NUM  \
0             1            2                9E   N8688C               3280
1             1            2                9E   N348PQ               3281
2             1            2                9E   N8896A               3282
3             1            2                9E   N8886A               3283
4             1            2                9E   N8974C               3284

  ORIGIN DEST DEP_TIME  DEP_DEL15 DEP_TIME_BLK ARR_TIME  ARR_DEL15  CANCELLED  \
0    GNV  ATL     0601        0.0    0600-0659     0722        0.0          0
1    MSP  CVG     1359        0.0    1400-1459     1633        0.0          0
2    DTW  CVG     1215        0.0    1200-1259     1329        0.0          0
3    TLH  ATL     1521        0.0    1500-1559     1625        0.0          0
4    ATL  FSM     1847        0.0    1900-1959     1940        0.0          0

   DIVERTED  DISTANCE  DEP_TIME_MINS  ARR_TIME_MINS
0         0     300.0            361            442
1         0     596.0            839            993
2         0     229.0            735            809
3         0     223.0            921            985
4         0     579.0           1127           1180
```

1.2 2. Data Analysis Preparation / Pre-processing

1.2.1 Data selection:

- As we only need reliable data, which the flight were not cancelled, the normal_flight is filtered from the original dataset
- By combining or processing some of the columns, the data would be more concise and brief

[3]:
```python
normal_flight = flight_data[flight_data['CANCELLED'] == 0.0].
↪drop(columns=['CANCELLED','DEP_TIME','DEP_TIME_BLK','ARR_TIME','DIVERTED','TAIL_NUM'])
normal_flight['FLIGHT_NUM'] = normal_flight.apply(lambda x :␣
↪x['OP_UNIQUE_CARRIER'] + str(x['OP_CARRIER_FL_NUM']), axis=1)
normal_flight['TRAVEL_TIME'] = normal_flight.apply(lambda x :␣
↪x['ARR_TIME_MINS'] - x['DEP_TIME_MINS'], axis=1)
normal_flight.drop(columns=['OP_CARRIER_FL_NUM','ARR_TIME_MINS'],inplace=True)
```

[4]:
```python
normal_flight.head()
```

[4]:
```
   DAY_OF_MONTH  DAY_OF_WEEK OP_UNIQUE_CARRIER ORIGIN DEST  DEP_DEL15  \
0             1            2                9E    GNV  ATL        0.0
1             1            2                9E    MSP  CVG        0.0
2             1            2                9E    DTW  CVG        0.0
3             1            2                9E    TLH  ATL        0.0
4             1            2                9E    ATL  FSM        0.0

   ARR_DEL15  DISTANCE  DEP_TIME_MINS FLIGHT_NUM  TRAVEL_TIME
0        0.0     300.0            361     9E3280           81
1        0.0     596.0            839     9E3281          154
2        0.0     229.0            735     9E3282           74
3        0.0     223.0            921     9E3283           64
4        0.0     579.0           1127     9E3284           53
```

1.2.2 Data Transfer

- Transfering the categorical data to relative delay rate(Better observation)

[5]:
```python
# group by each column and calculate delay rate (DEP_DR and ARR_DR) of each␣
↪attribute
cols =␣
↪['DAY_OF_MONTH','DAY_OF_WEEK','OP_UNIQUE_CARRIER','FLIGHT_NUM','ORIGIN','DEST']
for col in cols:
    dep_name, arr_name = 'DEP_DR_'+col, 'ARR_DR_'+col
    stat = normal_flight[[col, 'DEP_DEL15', 'ARR_DEL15']].groupby(col).
↪transform('mean')
    normal_flight[dep_name] = stat['DEP_DEL15']
    normal_flight[arr_name] = stat['ARR_DEL15']
normal_flight.drop(columns=cols,inplace=True)
```

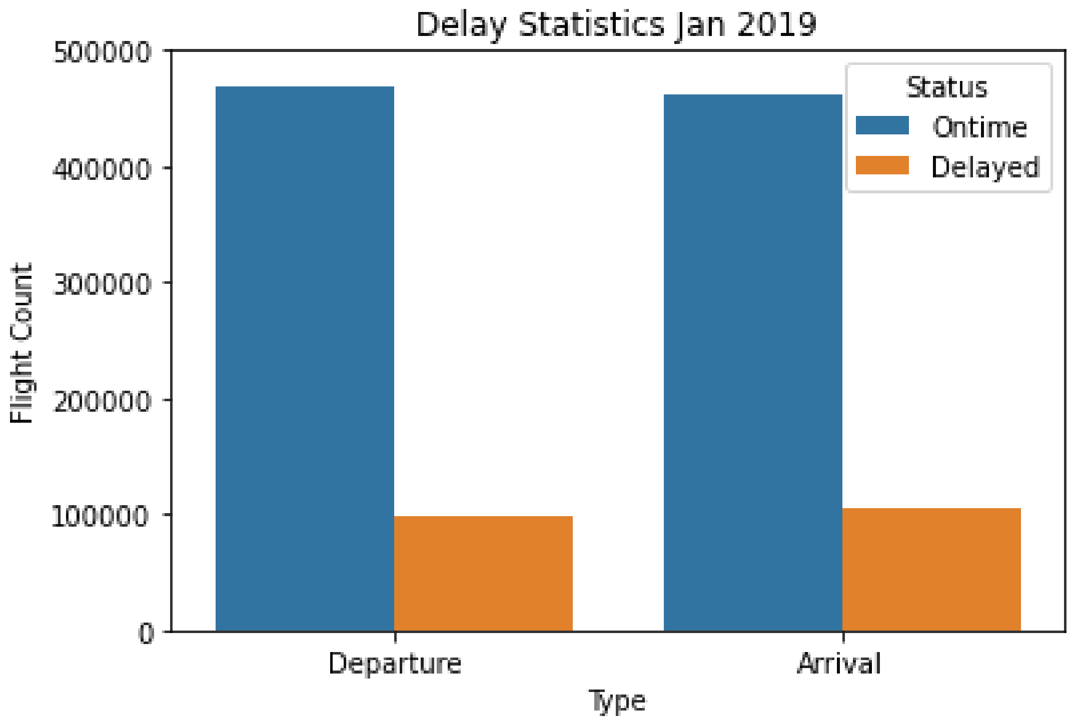
[6]:
```python
normal_flight.head()
```

[6]:
```
   DEP_DEL15  ARR_DEL15  DISTANCE  DEP_TIME_MINS  TRAVEL_TIME  \
0        0.0        0.0     300.0            361           81
1        0.0        0.0     596.0            839          154
```

```
2        0.0        0.0     229.0           735            74
3        0.0        0.0     223.0           921            64
4        0.0        0.0     579.0          1127            53

   DEP_DR_DAY_OF_MONTH  ARR_DR_DAY_OF_MONTH  DEP_DR_DAY_OF_WEEK  \
0             0.211332             0.215423            0.158906
1             0.211332             0.215423            0.158906
2             0.211332             0.215423            0.158906
3             0.211332             0.215423            0.158906
4             0.211332             0.215423            0.158906

   ARR_DR_DAY_OF_WEEK  DEP_DR_OP_UNIQUE_CARRIER  ARR_DR_OP_UNIQUE_CARRIER  \
0            0.167916                  0.188134                  0.202462
1            0.167916                  0.188134                  0.202462
2            0.167916                  0.188134                  0.202462
3            0.167916                  0.188134                  0.202462
4            0.167916                  0.188134                  0.202462

   DEP_DR_FLIGHT_NUM  ARR_DR_FLIGHT_NUM  DEP_DR_ORIGIN  ARR_DR_ORIGIN  \
0           0.173077           0.153846       0.130682       0.136364
1           0.129630           0.148148       0.155117       0.151762
2           0.072727           0.090909       0.187218       0.205004
3           0.127273           0.090909       0.103734       0.136929
4           0.029412           0.058824       0.133775       0.130364

   DEP_DR_DEST  ARR_DR_DEST
0     0.120077     0.127642
1     0.209911     0.206849
2     0.209911     0.206849
3     0.120077     0.127642
4     0.113772     0.131737
```

[7]:
```
dep_ontime_cnt, dep_delay_cnt = np.sum(normal_flight['DEP_DEL15'] == 0.0), np.
↪sum(normal_flight['DEP_DEL15'] == 1.0)
arr_ontime_cnt, arr_delay_cnt = np.sum(normal_flight['ARR_DEL15'] == 0.0), np.
↪sum(normal_flight['ARR_DEL15'] == 1.0)
# plt.title('Delay Statistics')
stat_df = pd.DataFrame({'Type': ['Departure','Departure','Arrival','Arrival'],␣
↪'Status': ['Ontime','Delayed','Ontime','Delayed'], 'Flight Count':␣
↪[dep_ontime_cnt, dep_delay_cnt, arr_ontime_cnt, arr_delay_cnt]})
plt.ylim((0,500000))
plt.title('Delay Statistics Jan 2019')
sns.barplot(data=stat_df, x='Type', y='Flight Count', hue='Status')
```

[7]:
```
<AxesSubplot:title={'center':'Delay Statistics Jan 2019'}, xlabel='Type',
ylabel='Flight Count'>
```



1.3 3. PCA Analysis

1.3.1 Training preparation

- Cancelled flights are removed from original dataset as they are not relevant to delay prediction
- Dataset is split up into training data(60%), validation data(20%) and test data(20%)

1.3.2 Implement of PCA

- Choose n = 6, PCA would help us to do Dimension reduction
- (Reduce the computational overhead of the algorithm and Reserve most of the data: 80%)

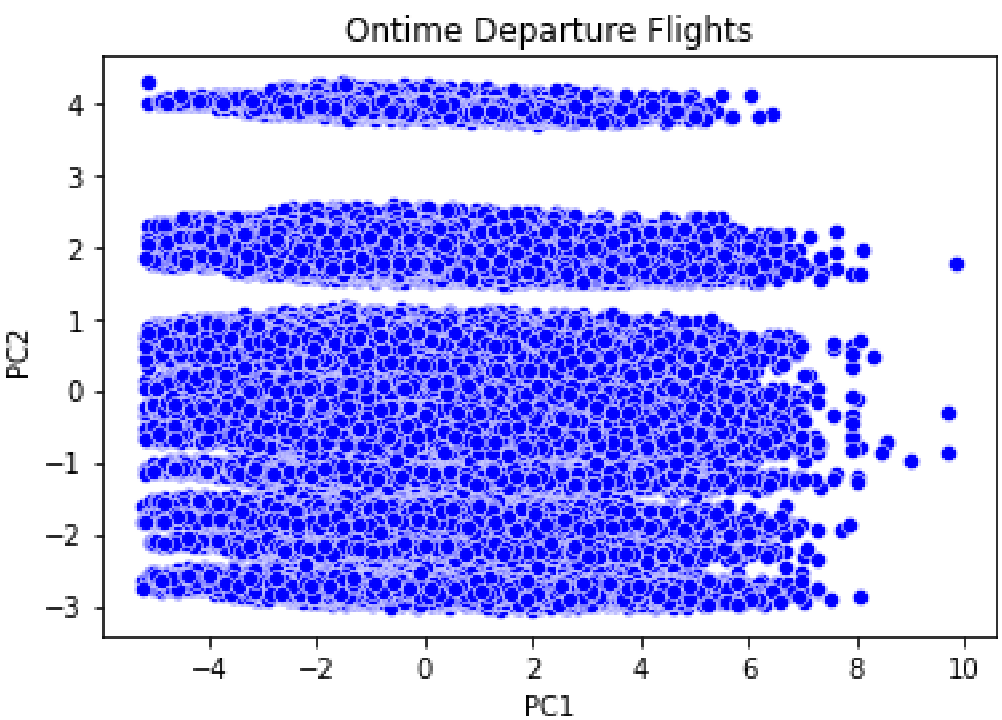
[8]:
```
# split up test and train data for normal flight
all_dep = normal_flight['DEP_DEL15']
all_arr = normal_flight['ARR_DEL15']
all_data = normal_flight.drop(columns=['DEP_DEL15','ARR_DEL15'])
all_data = StandardScaler().fit_transform(all_data)
pca = PCA(n_components=6).fit(all_data)
all_data = pca.transform(all_data)
train_data, test_data = train_test_split(all_data, train_size=0.8,
 ↪random_state=42)
train_data, val_data = train_test_split(train_data, train_size=0.75,
 ↪random_state=42)
train_dep, test_dep = train_test_split(all_dep, train_size=0.8, random_state=42)
```

```
train_dep, val_dep = train_test_split(train_dep, train_size=0.75,␣
↪random_state=42)
train_arr, test_arr = train_test_split(all_arr, train_size=0.8, random_state=42)
train_arr, val_arr = train_test_split(train_arr, train_size=0.75,␣
↪random_state=42)
print('Data reserved by PCA (in percentage):', np.sum(pca.
↪explained_variance_ratio_))
```

```
Data reserved by PCA (in percentage): 0.803934661761031
```

[9]:
```
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.title('Ontime Departure Flights')
sns.scatterplot(x=train_data[train_dep == 0.0][:,0], y=train_data[train_dep ==␣
↪0.0][:,1], color='b')
```

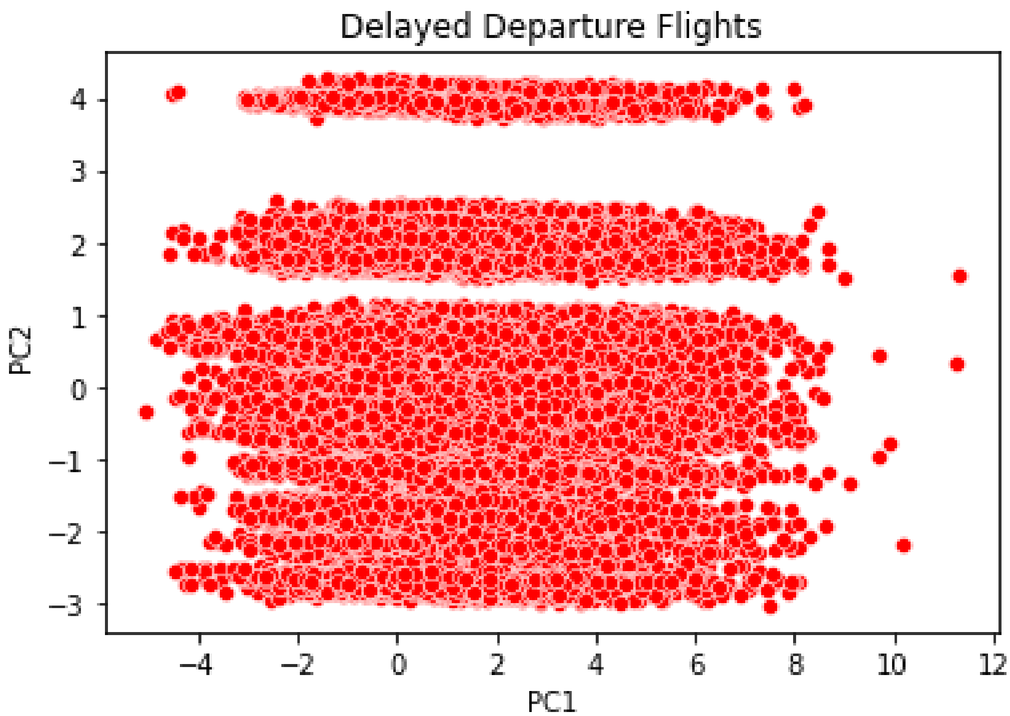
[9]:
```
<AxesSubplot:title={'center':'Ontime Departure Flights'}, xlabel='PC1',
ylabel='PC2'>
```



[10]:
```
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.title('Delayed Departure Flights')
```

```
sns.scatterplot(x=train_data[train_dep == 1.0][:,0], y=train_data[train_dep ==␣
↪1.0][:,1], color='red')
```

[10]: <AxesSubplot:title={'center':'Delayed Departure Flights'}, xlabel='PC1', ylabel='PC2'>



1.4 4. Flight Delay Prediction

1.4.1 Implement of KNN:

- View the different results with different k-value, choose the best one(observation) among all of them.
- We would use False Positive and False Negative to see the correctness of result
- False Positive : Prediction is True, but the truth is False
- False Negative : Prediction is False, but the truth is True

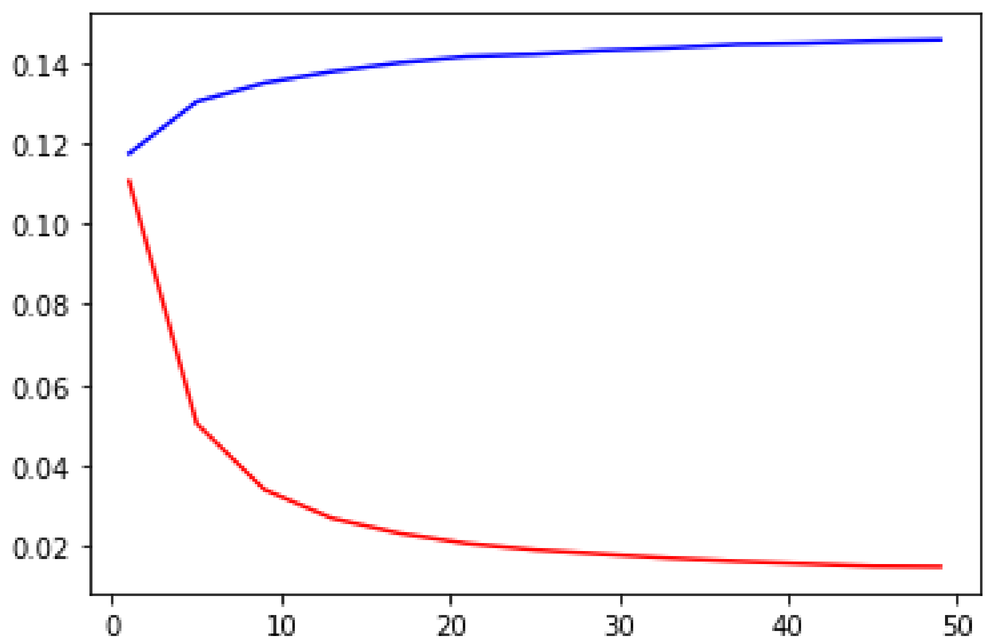
[11]:
```
# Calculate the accuracy of prediction against validation data given k-value
def test_accuracy(k, mode='DEP'):
    print('Running KNN with k =',k)
    train_target = train_dep if mode == 'DEP' else train_arr
    val_target = val_dep if mode == 'DEP' else val_arr
    knn = KNeighborsClassifier(n_neighbors=k, weights='distance', n_jobs=-1).
↪fit(train_data, train_target)
```

```
    prediction = knn.predict(val_data)
    # false positive, prediction > target
    fp = np.sum(prediction > val_target) / len(val_data)
    # false negative, prediction < target
    fn = np.sum(prediction < val_target) / len(val_data)
    return [k,fp,fn]
```

[12]:

```
run_result = np.array([test_accuracy(k, mode='DEP') for k in range(1,50,4)])
plt.plot(run_result[:,0], run_result[:,1], 'r') # False positive
plt.plot(run_result[:,0], run_result[:,2], 'b') # False negative
plt.show()
```

```
Running KNN with k = 1
Running KNN with k = 5
Running KNN with k = 9
Running KNN with k = 13
Running KNN with k = 17
Running KNN with k = 21
Running KNN with k = 25
Running KNN with k = 29
Running KNN with k = 33
Running KNN with k = 37
Running KNN with k = 41
Running KNN with k = 45
Running KNN with k = 49
```

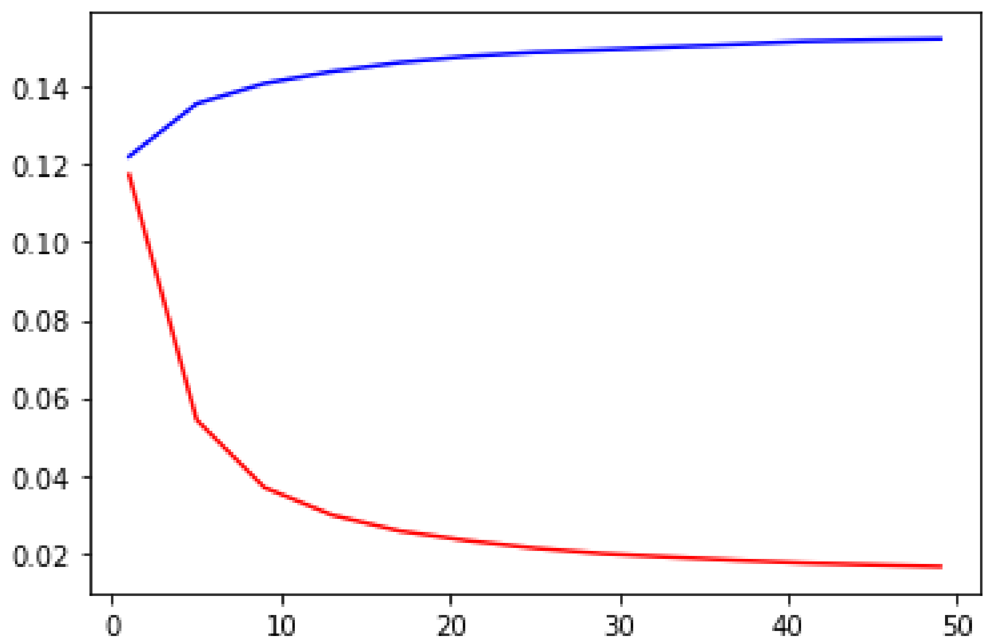


[13]:
```
dep_knn = KNeighborsClassifier(n_neighbors=9, weights='distance', n_jobs=-1).
    ↪fit(train_data, train_dep)
dep_prediction = dep_knn.predict(test_data)
dep_fp = np.sum(dep_prediction > test_dep) / len(test_data)
dep_fn = np.sum(dep_prediction < test_dep) / len(test_data)
dep_fp, dep_fn # (false positive rate, false negative rate) for departure delay
```

[13]:
```
(0.03442792398823249, 0.13351532338572172)
```

[14]:
```
run_result = np.array([test_accuracy(k, mode='ARR') for k in range(1,50,4)])
plt.plot(run_result[:,0], run_result[:,1], 'r') # False positive
plt.plot(run_result[:,0], run_result[:,2], 'b') # False negative
plt.show()
```

```
Running KNN with k = 1
Running KNN with k = 5
Running KNN with k = 9
Running KNN with k = 13
Running KNN with k = 17
Running KNN with k = 21
Running KNN with k = 25
Running KNN with k = 29
Running KNN with k = 33
Running KNN with k = 37
Running KNN with k = 41
Running KNN with k = 45
Running KNN with k = 49
```



[15]:
```
arr_knn = KNeighborsClassifier(n_neighbors=9, weights='distance', n_jobs=-1).
 ↪fit(train_data, train_arr)
arr_prediction = arr_knn.predict(test_data)
arr_fp = np.sum(arr_prediction > test_arr) / len(test_data)
arr_fn = np.sum(arr_prediction < test_arr) / len(test_data)
arr_fp, arr_fn # (false positive rate, false negative rate) for arrival delay
```

[15]:
```
(0.03706059561987049, 0.14210242682851412)
```

1.5 5. Analysis

1.5.1 Dataset URL:

- https://www.kaggle.com/divyansh22/flight-delay-prediction
- The data of detail of flight is collected, we could use the details to predict the flight will delay or not.

1.5.2 QUESTIONS:

- The delay of the flight is annoying, it would usually cause a series of time conflict. Therefore, we're wondering that what if we could predict the delay of the flight, then we could preplan the schedule and use the time more properly

1.5.3 TOOL:

- it is a prediction problem, we are trying to predict whether or not the flight will delay.
- we use the the distance between the origin and destination, total travel minutes, the time block and etc. to predict the probability of delay.
- And the reason why we choose them is because they are relative to the delay(e.g. Knowing the distance from one city to another and travel time could be seen as reference to predict the flight will delay or not)
- We used standardised data, which could be more precise and accurate.
- With what we said in (4,Implement of KNN), false positive and false negative are used to see the result.

1.5.4 ANALYSIS & FINDINGS:

- Even we have a quite good prediction through the regression model, but it still have some problem. From the original data, we know that the sample number of flight which not dalayed is totally greater than that of flight which delayed, it is also a reason why the probability of the false positive is smaller than that of false negative.
- graph can be seen in the bottom of (4. Flight Delay Prediction)

1.5.5 FUTURE DIRECTIONS:

- There still a lot of event which has not been considered might happened before departure of the flight(the number and weight of luggage, missing passengers and etc.),