

## Project: Hotel Bookings

### Dataset Source: Kaggle Datasets

First, I import the libraries and then the data set. The data looks like this:



	hotel	is_canceled	lead_time	arrival_date_year	arrival_date_month	arrival_date_week_number	arrival_date_day_of_month	stays_in_weekend_nights
0	Resort Hotel	0	342	2015	July	27	1	
1	Resort Hotel	0	737	2015	July	27	1	
2	Resort Hotel	0	7	2015	July	27	1	
3	Resort Hotel	0	13	2015	July	27	1	
4	Resort Hotel	0	14	2015	July	27	1	
...	...	...	...	...	...	...	...	...
119385	City Hotel	0	23	2017	August	35	30	

*The data frame*

Then, I check to see if there are missing values. And yes. There are 4 features that contain missing values:



Out[7]:	company	112593
	agent	16340
	country	488
	children	4
	lead_time	0
	arrival_date_year	0
	arrival_date_month	0
	arrival_date_week_number	0
	is_canceled	0
	market_segment	0
	arrival_date_day_of_month	0
	stays_in_weekend_nights	0

*Missing values*

Now, I have to try to fill in (or drop) the missing value. In the Kaggle™ guide for this dataset it is written:

**agent**: ID of the travel agency that made the booking.

**company**: ID of the company/entity that made the booking or responsible for paying the booking. ID is presented instead of designation for anonymity reasons.

Therefore, since the "agent" and "company" are merely IDs, we can fill in their missing values by method "fillna()" with a simple digit, I chose "0".

Now for the "country", I fill in the missing values by the mode metric. That way, I can find which countries are making the most reservations.

Now for the "children", I fill in the missing values by the mean metric. Since most of the people had "0" children.

Also, I check the people (guests) who registered for the hotel bookings, that is "adults", "children", and "babies".

Out[54]: (180, 32)

0 guests!

So there are 180 rows with zero guests! These rows will not give us any valuable information, so I delete them by "drop()" method.

Now, the type of data in the data frame should be checked and if necessary, they should be converted. For example, float to integer:

```
Out[86]: hotel                      object
is_canceled                  int64
lead_time                     int64
arrival_date_year              int64
arrival_date_month             object
arrival_date_week_number       int64
arrival_date_day_of_month      int64
stays_in_weekend_nights        int64
stays_in_week_nights           int64
adults                         int64
children                       float64
babies                          int64
meal                            object
country                         object
market_segment                  object
distribution_channel            object
is_repeated_guest               int64
previous_cancellations          int64
previous_bookings_not_canceled   int64
reserved_room_type              object
assigned_room_type              object
booking_changes                 int64
deposit_type                    object
agent                           float64
company                         float64
days_in_waiting_list            int64
customer_type                   object
adr                            float64
```

data types

As we saw in the previously, columns "children", "company", and "agent" all had a ".0", i.e. they were "float64" type. But it is not needed. So we simply convert them to "int64" using "astype('int64')":

Out[88]:

	children	company	agent
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	304
4	0	0	240
...	...	...	...
119385	0	0	394
119386	0	0	9
119387	0	0	9
119388	0	0	89
119389	0	0	9

119210 rows × 3 columns

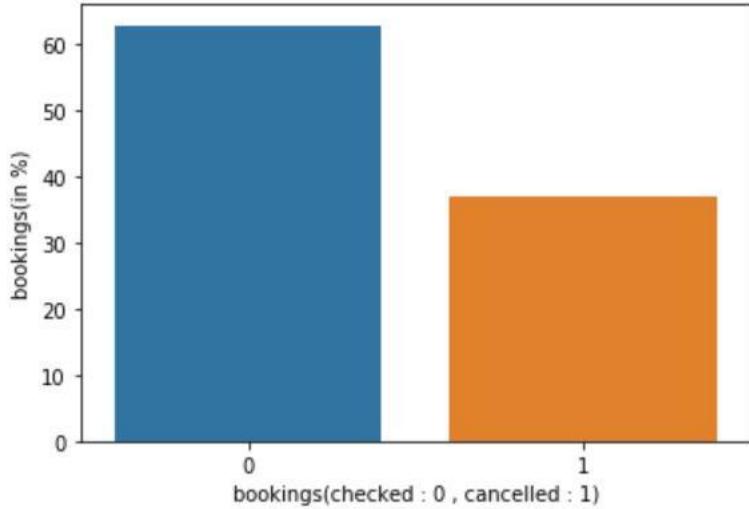
float64 to int64

So, let's see how many people cancelled their reservations and by which percent. Thus, if "is\_canceled" = 0 then people were checked-in. But if "is\_canceled" = 1 then people cancelled their reservations:

Out[92]: 0 75011  
1 44199  
Name: is\_canceled, dtype: int64

cancelled V.S checked ins

That I plotted using "seaborn" bar plot:

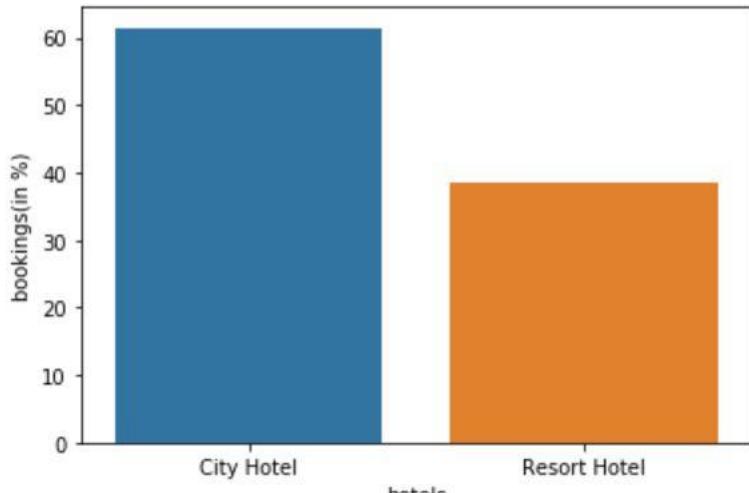


*Bar plot of the % of cancellations*

So about 63% got checked in. And about 37% cancelled their reservations.

Now, I want to analyze the “hotel” column:

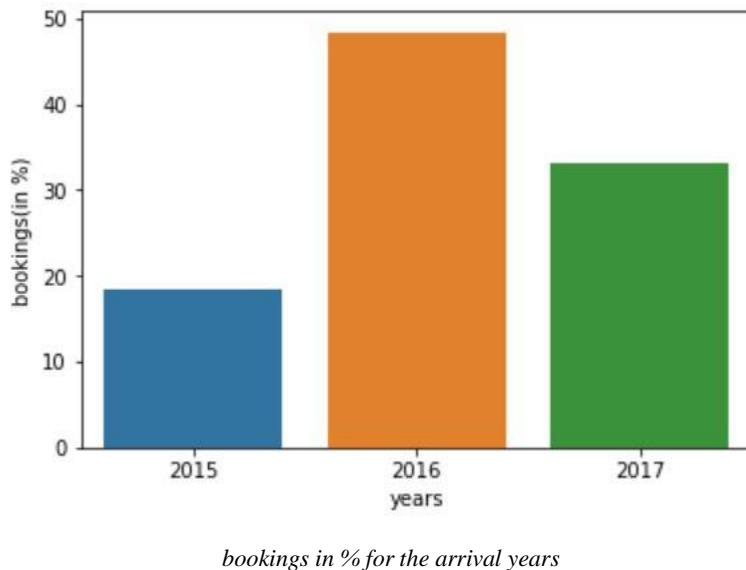
According to the code, from "0 to 40046" rows were “Resort Hotel” and also from " 40047 to end" rows were “City Hotel”. So the bookings in percent for the two types of hotels are:



*bookings in %for hotel types*

so about 61% booked a city hotel. And about 39% booked a resort hotel.

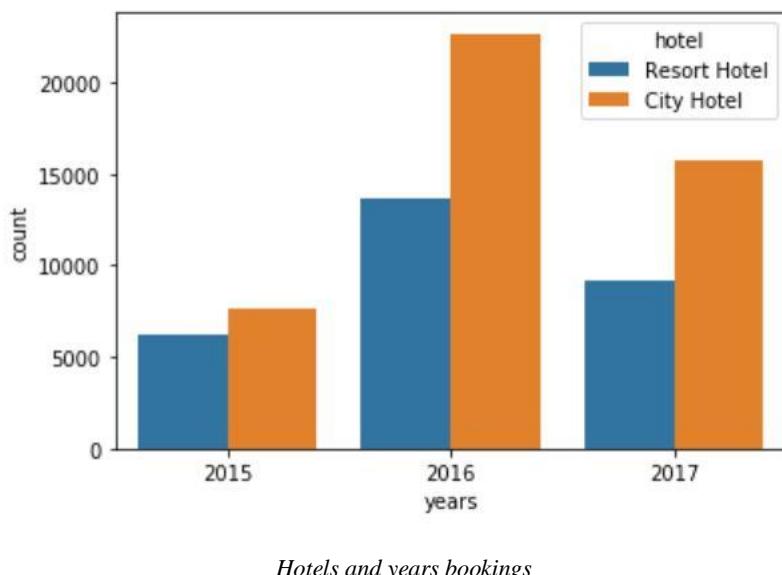
Now, I want to analyze the “arrival\_date\_year” column:



*bookings in % for the arrival years*

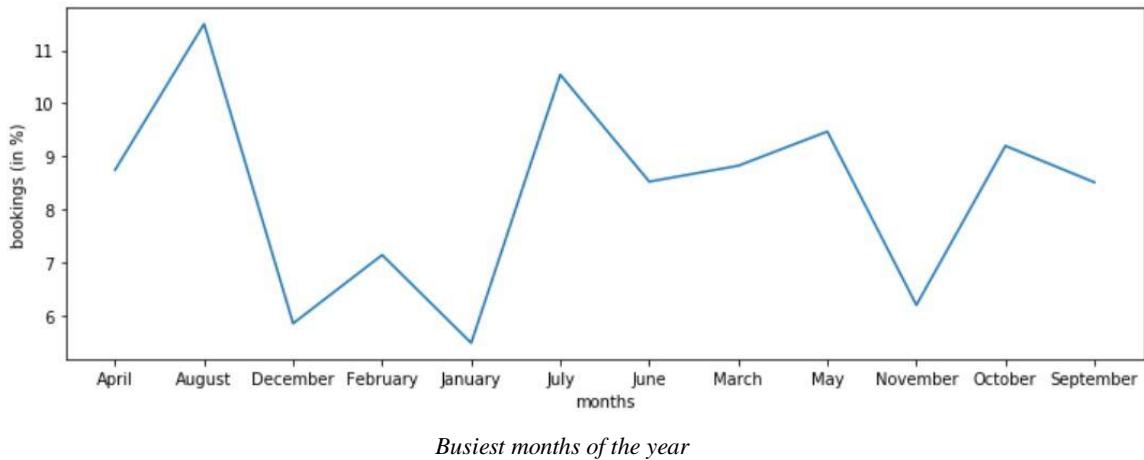
so about 50% of the bookings were for year 2016. But then it decreased by 15% for the next year, 2017.

Also I plotted a count plot for both the hotel types and the years:



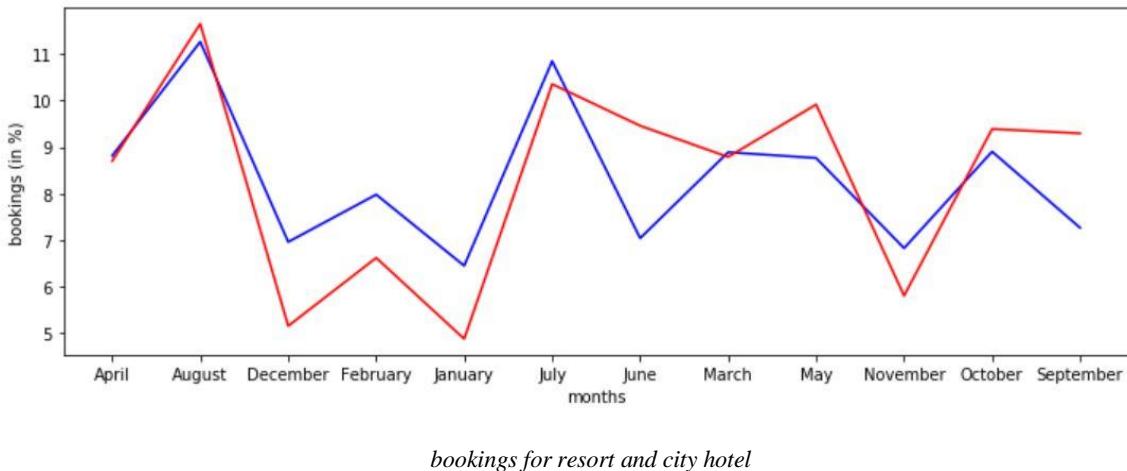
*Hotels and years bookings*

Now, I tried to discover the busiest month of the year:

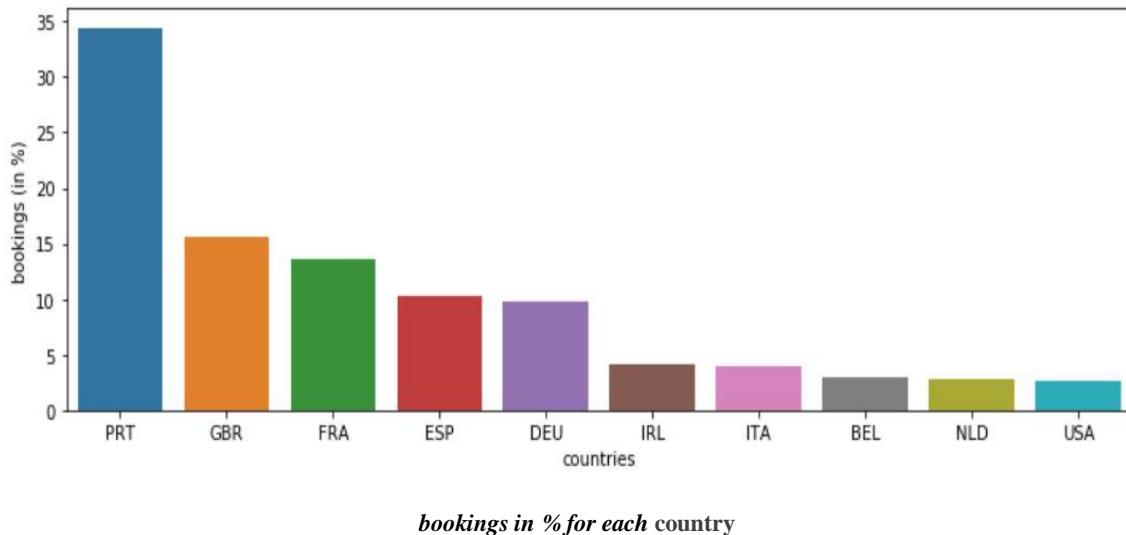


And as we see above, most of the bookings are for Summer (July and August). Also the least of the bookings are for the start (January) and the end (December) of the year.

I have plotted the trend for both the Resort and the City hotel. It was quite similar for both, except for slight differences. The blue line is for the Resort hotel. And the red line is for the City hotel.

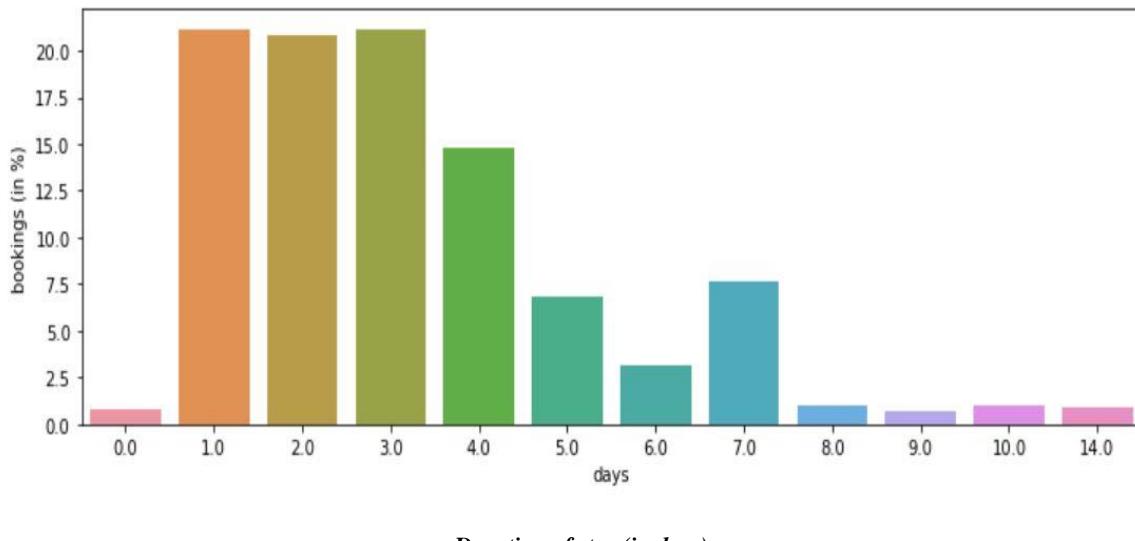


Next, I tried to find from which country does the guests come from the most:



So the top four countries who did most of the reservations and didn't cancel are: Portugal(PRT), Britain(GBR), France(FRA), and Spain(ESP).

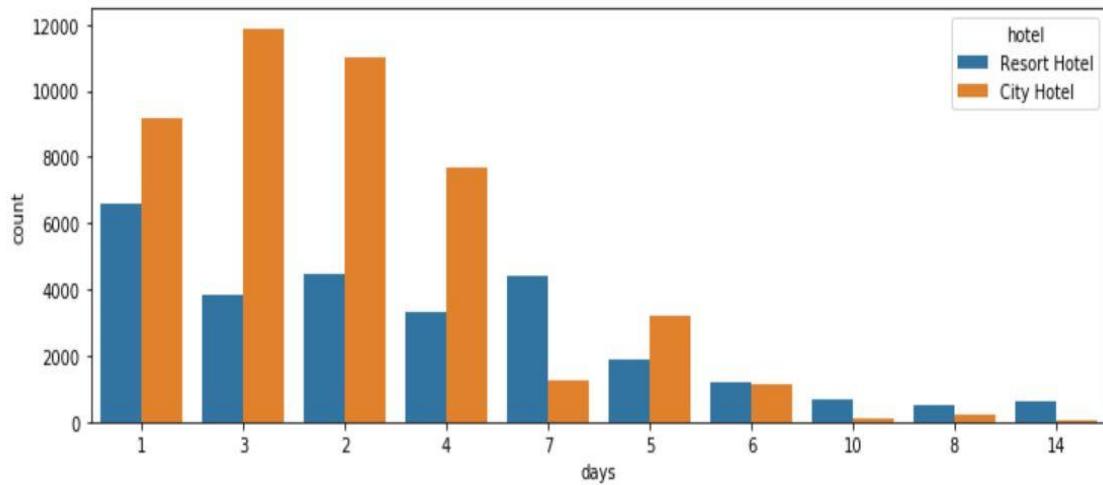
Next, I tried to find what is the most popular duration of stay (in days) among the people. And the result was that most of the people stay for 1, 2, and 3 nights. Which contains about 60% of all the people who didn't cancel their reservations.



Also the most popular duration of stay for different hotels are:

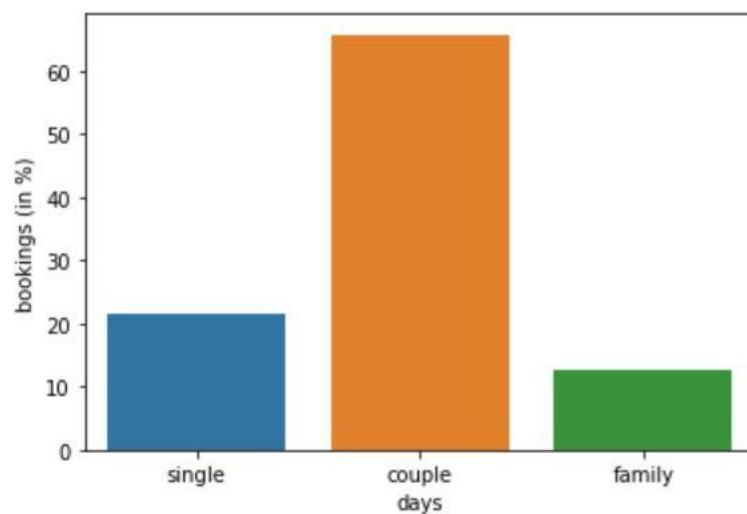
Resort hotel: 1, 7, 2, 3 days respectively

City hotel: 3, 2, 1, 4 days respectively



*duration of stay for different hotels*

Furthermore, the highest bookings were for couples (number of people = 2). That is shown below. Nearly 67% of the reservations were done by couples.

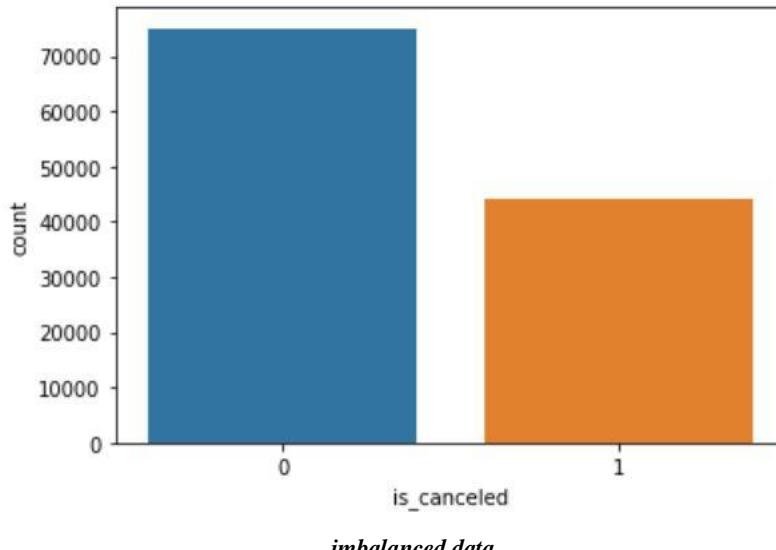


*bookings in % for different number of people*

Now it is time for **Feature Engineering**:

I want to first check if the data is balanced or not:

Out[203]: <matplotlib.axes.\_subplots.AxesSubplot at 0x179b2a82e88>

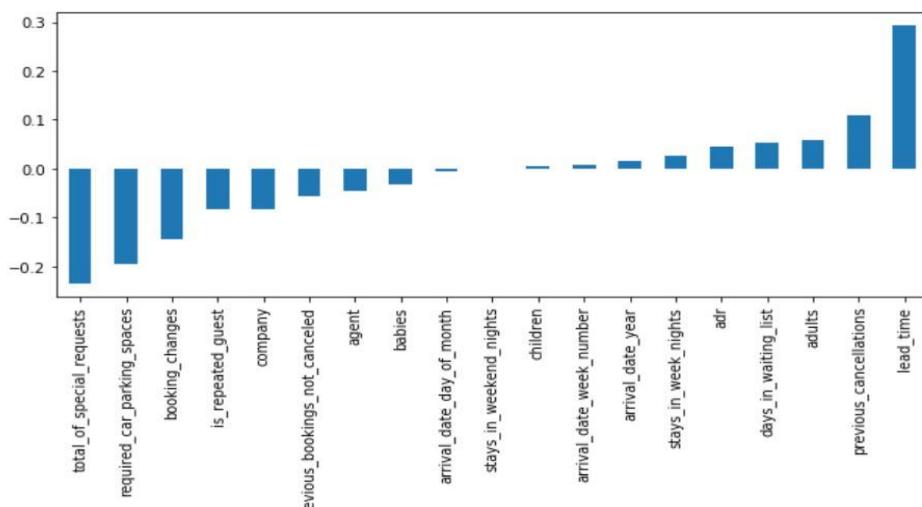


so this is an imbalanced problem. And most of the people have NOT cancelled.

One way to show the correlations for the label “is\_canceled” is using Matplotlib library:

```
plt.figure(figsize = (10, 4))
df.corr()['is_canceled'].sort_values().drop('is_canceled').plot(kind = 'bar')
```

Out[204]: <matplotlib.axes.\_subplots.AxesSubplot at 0x179b3719f48>



*Correlation for the label feature*

In order not to lose my original data frame "df" so that it is not effected by wrong feature engineering, I make a copy of my original data frame "df" as "df\_copy" by using "copy()" method.

Next, I want to analyze the room type:

```
In [29]: df_copy['reserved_room_type']

Out[29]: 0      C
          1      C
          2      A
          3      A
          4      A
          ..
         119385    A
         119386    E
         119387    D
         119388    A
         119389    A
Name: reserved_room_type, Length: 119210, dtype: object

reserved room type
```

And:

```
In [30]: df_copy['assigned_room_type']

Out[30]: 0      C
          1      C
          2      C
          3      A
          4      A
          ..
         119385    A
         119386    E
         119387    D
         119388    A
         119389    A
Name: assigned_room_type, Length: 119210, dtype: object

assigned room type
```

So since I am only interested to see if the "reserved\_room\_type" is different from "assigned\_room\_type", I make a new feature "room" to analyze it further. So that if the same room is assigned to the guest, then "room = 1 ". Otherwise, "room = 0". After this step is done I drop features "reserved\_room\_type" and "assigned\_room\_type", because they are merely alphabetical code of the rooms and do not give us any valuable information.

```
In [39]: df_copy['room']

Out[39]: 0      1
          1      1
          2      0
          3      1
          4      1
          ..
119385    1
119386    1
119387    1
119388    1
119389    1
Name: room, Length: 119210, dtype: int64
```

*new feature room*

Another feature is "previous\_cancellations" and "previous\_bookings\_not\_canceled". These two features can simply be combined to make a new feature "total\_canceled". So if canceled > not-canceled then "total\_canceled = 1". Otherwise, it is "0". After this step is done, I drop features "previous\_cancellations" and "previous\_bookings\_not\_canceled".

```
In [44]: df_copy['total_canceled']

Out[44]: 0      0
          1      0
          2      0
          3      0
          4      0
          ..
119385    0
119386    0
119387    0
119388    0
119389    0
Name: total_canceled, Length: 119210, dtype: int64
```

*new feature total canceled*

Now, let's see features "reservation\_status" and "reservation\_status\_date":

```
df_copy['reservation_status']
```

```
Out[47]: 0      Check-Out
         1      Check-Out
         2      Check-Out
         3      Check-Out
         4      Check-Out
         ...
        119385  Check-Out
        119386  Check-Out
        119387  Check-Out
        119388  Check-Out
        119389  Check-Out
Name: reservation_status, Length: 119210, dtype: object
```

*reservation status feature*

And:

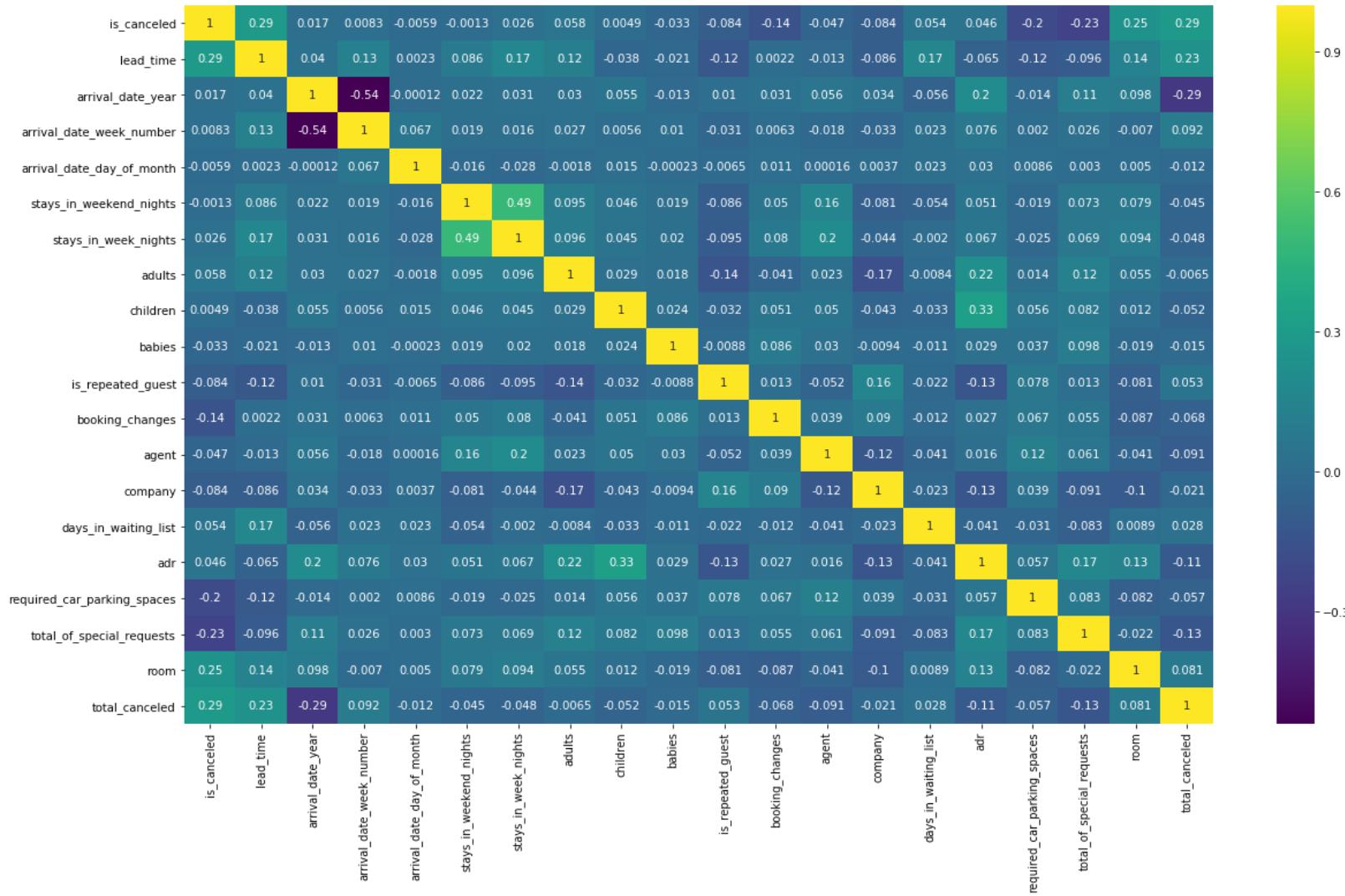
```
In [48]: df_copy['reservation_status_date']
```

```
Out[48]: 0      2015-07-01
         1      2015-07-01
         2      2015-07-02
         3      2015-07-02
         4      2015-07-03
         ...
        119385  2017-09-06
        119386  2017-09-07
        119387  2017-09-07
        119388  2017-09-07
        119389  2017-09-07
Name: reservation_status_date, Length: 119210, dtype: object
```

*reservation status date*

Since these two features give us only repeated and not valuable information, I drop them too. (we have feature “is\_canceled” as the label and also new feature “total\_canceled”)

Now, I see the heat map of the correlations to decide if any there are any other irrelevant features with very low correlation to the label “is\_canceled”:



heat map

So there are some features in the heat map that have really low correlations. I decide to drop the features with correlations of less than 0.01 (to be more precise, between -0.01 and 0.01). These features include: "arrival\_date\_year", "arrival\_date\_week\_number", "arrival\_date\_day\_of\_month".

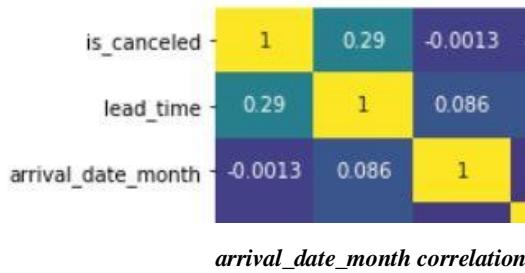
Also "adr" is in 'float64'. I convert it to 'int64' too.

Moreover, as we see above, there are still two features "stays\_in\_weekend\_nights" and "children" that I didn't drop previously. That is because their relevant features "stays\_in\_week\_nights" and "adult" have good enough correlations to prevent them from being dropped. So I keep these two features.

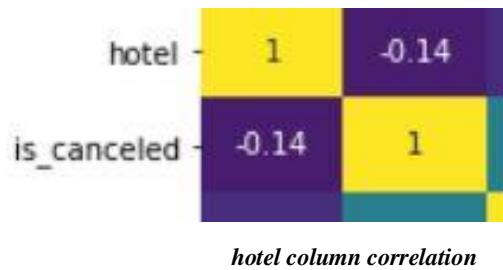
'hotel', 'arrival\_date\_month', 'meal', 'country', 'market\_segment', 'distribution\_channel', 'deposit\_type', 'customer\_type' are the remainder categorical columns.

So I will Label-encode these categorical columns, and if their correlation with the main label “is\_canceled” was low I will drop them. Else, I may One-hot-encode them or just let them be in the label-encoded form and use them to train the model.

Therefore, I started with “arrival\_date\_month” and Label-encoded it. Because there is a Sequence quality between months of the year in column “arrival\_date\_month”, we can use a Label-encoder for it. Then I saw the heat map again and its correlation was -0.0013 which is too low to keep this column. So I dropped “arrival\_date\_month”.



Next, is the “hotel” column. This column only contains two unique values ‘Resort hotel’ and ‘City hotel’. So I first Label-encode it. Then, see its correlation:



So since the correlation for "hotel" is high enough (-0.14), I keep the "hotel" column.

For the “market\_segment” column, the correlation after Label-encoding is (0.059). So I keep this column.

For the “distribution\_channel” column, the correlation after Label-encoding is (0.17). So I keep this column.

For the “deposit\_type” column, the correlation after Label-encoding is (0.47). So I keep this column.

For the “customer\_type” column, the correlation after Label-encoding is (0.068). So I keep this column.

For the “country” column, the correlation after Label-encoding is (0.27). So I keep this column.

For the “meal” column, the correlation after Label-encoding is (-0.017). So I keep this column.

Next, I converted these above mentioned categorical columns (7 columns) to One-hot-encoding by “get\_dummies()” method. For instance, for the “market\_segment” column we have:

```
one_hot1 = pd.get_dummies(df_copy2['market_segment'])
df_copy2 = df_copy2.drop('market_segment', axis = 1)
df_copy2 = df_copy2.join(one_hot1)
```

**Note:** So what I did for the encoding section was to first Label-encode the categorical columns and check their correlation. If the correlation was lower than (0.01) then I would drop that very column which only happened for column “arrival\_date\_month”. After that, I One-hot-encoded the remaining columns (by “get\_dummies()” method) so that I am able to capture the hidden relationships.

**Note:** my original data frame was “df”. Later for some preprocessing, I made a copy of “df” and I named this copy “df\_copy”. The reason was that I wanted to have the original data frame “df” intact, if by any chance I find a method for that specific preprocessing bad and faulty. That way, I have the chance to start over again with the original data frame “df”. Later on in the code, I had to do some encoding to convert categorical features to numerical features. Again for the exact same reason, I made another copy of “df\_copy” and named it “df\_copy2”. So for the rest of the code (from train-test-split to the end) I work with “df\_copy2”.

So right now, the data frame (“df\_copy2”) has 221 columns. But we should reduce this number if possible by seeing the correlations again. So if one of the column’s correlation with the label “is\_canceled” is less than 0.01 then I drop it.

```
In [102]: df_copy3.shape
```

```
Out[102]: (119210, 221)
```

*df\_copy3 before final cleaning*

**Note:** Again here, we HAVE TO make a copy of "df\_copy2" and name it "df\_copy3". The reason is that when we want to drop the columns (columns with low correlation in below),

the index of the chosen columns MUST NOT change. And for that, we have to make a copy of the current data frame.

```
for i in range(1, 220): #since we have 221 columns in general.
    if ((correlations['is_canceled'].iloc[i] < 0.01) and (correlations['is_canceled'].iloc[i] > -0.01)):
        df_copy3 = df_copy3.drop(df_copy2.columns[i], axis = 1)

# #or similarly:
# flag = 0
# while(flag != 220): #since we have 221 columns in general.
#     flag += 1
#     if ((correlations['is_canceled'].iloc[flag] < 0.01) and (correlations['is_canceled'].iloc[flag] > -0.01)):
#         df_copy3 = df_copy3.drop(df_copy2.columns[flag], axis = 1)
```

After dropping according to the above code, there will be only (77) out of (221) columns left! This is our final clean data that is ready to be fed into the model after the train-test-split!

In [104]: df\_copy3.shape

Out[104]: (119210, 77)

*df\_copy3 after final cleaning*

The preprocessing phase is finished. Now, I have to split the data (“df\_copy3”) into the train set and the test set:

```
from sklearn.model_selection import train_test_split

x = df_copy3.drop('is_canceled', axis = 1).values
y = df_copy3['is_canceled'].values

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state = 1)
```

Now, I should define the models. As suggested in the Project description, I chose to implement an ensemble method Random Forest as “model 1” and a neural network ANN as “model 2”. Then I measured the confusion matrix and accuracy for both of the models.

## Model 1: Random Forest

```
from sklearn.ensemble import RandomForestClassifier

forest = RandomForestClassifier(n_estimators = 10, criterion = 'entropy', random_
state = 0) #n_estimators: the number of trees (the more complex the dataset, the
more trees we need!)      #Tunning: increasing the "n_estimators" to increase the a
ccuracy
forest.fit(x_train, y_train)

[[13882  1081]
 [ 1849  7030]]
0.8771076251992282
```

In [ ]: ┌ #So the accuracy for the ensemble method "Random Forest" is about 88%.

*Random Forest (Model 1) accuracy and confusion matrix*

So the accuracy for the ensemble method "Random Forest" is about 88%.

## Model 2: ANN

The architecture of the ANN model is as the following:

```
In [200]: ┌ ann.summary()

Model: "sequential_5"
=====
Layer (type)          Output Shape         Param #
=====
dense_15 (Dense)     multiple            7700
dense_16 (Dense)     multiple            10100
dense_17 (Dense)     multiple            101
=====
Total params: 17,901
Trainable params: 17,901
Non-trainable params: 0
```

I trained the model through 100 epochs:

```
Epoch 100/100
95368/95368 [=====] - 8s 80us/sample - loss: 0.2027 - acc: 0.9073
Out[172]: <tensorflow.python.keras.callbacks.History at 0x179bbba3f48>
ANN model accuracy after training for 100 epochs
```

The model's accuracy is about 90%. But I check that through the following code lines as well:

```
[[13400 1563]
 [ 1875 7004]]
0.8558006878617566
```

```
In [201]: #So the accuracy in the ANN model is 85%.
```

*ANN model confusion matrix and accuracy*

Hence, the accuracy in the ANN model is about 85%.

**Result:** In general, for both models 1 and 2 we reached quite fine accuracies. Therefore with a mean accuracy of about  $\frac{88\%+85\%}{2} = 86.5\%$  percent, we are able to predict the cancelation probability of hotel bookings!

As an extra task, I also checked whether overfitting happened to the ANN model or not by using the “EarlyStopping” class of the “callbacks” library:

```
#With early stopping:

from tensorflow.keras.callbacks import EarlyStopping
early_stop = EarlyStopping(monitor = 'val_loss', patience = 1)

ann_early.fit(x = x_train, y = y_train, epochs = 100, validation_data = [x_test,
y_test], callbacks = [early_stop])
```

```

Train on 95368 samples, validate on 23842 samples
Epoch 1/100
95368/95368 [=====] - 7s 71us/sample - loss: 0.3499 - acc: 0.8333 - val_loss: 0.3319 - val_acc: 0.8
452
Epoch 2/100
95368/95368 [=====] - 6s 61us/sample - loss: 0.3238 - acc: 0.8450 - val_loss: 0.3266 - val_acc: 0.8
439
Epoch 3/100
95368/95368 [=====] - 5s 54us/sample - loss: 0.3164 - acc: 0.8500 - val_loss: 0.3153 - val_acc: 0.8
516
Epoch 4/100
95368/95368 [=====] - 5s 52us/sample - loss: 0.3100 - acc: 0.8540 - val_loss: 0.3128 - val_acc: 0.8
534
Epoch 5/100
95368/95368 [=====] - 5s 52us/sample - loss: 0.3047 - acc: 0.8567 - val_loss: 0.3124 - val_acc: 0.8
561
Epoch 6/100
95368/95368 [=====] - 5s 55us/sample - loss: 0.3011 - acc: 0.8591 - val_loss: 0.3091 - val_acc: 0.8
559
Epoch 7/100
95368/95368 [=====] - 6s 61us/sample - loss: 0.2968 - acc: 0.8613 - val_loss: 0.3073 - val_acc: 0.8
579
Epoch 8/100
95368/95368 [=====] - 5s 56us/sample - loss: 0.2938 - acc: 0.8633 - val_loss: 0.3095 - val_acc: 0.8
579

```

**Out[192]:** <tensorflow.python.keras.callbacks.History at 0x179bc003b48>

*Early stop model fitting*

So the accuracy is 80% by the early stopping method. The metrics are as follow:

**In [194]:** █ metrics = pd.DataFrame(ann\_early.history.history)  
metrics

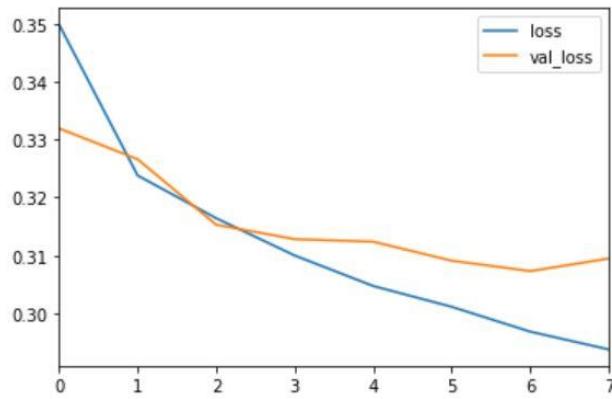
**Out[194]:**

	loss	acc	val_loss	val_acc
0	0.349857	0.833330	0.331926	0.845231
1	0.323802	0.845000	0.326605	0.843931
2	0.316429	0.850023	0.315296	0.851648
3	0.309989	0.853955	0.312816	0.853368
4	0.304729	0.856681	0.312371	0.856052
5	0.301135	0.859146	0.309091	0.855885
6	0.296848	0.861337	0.307296	0.857940
7	0.293779	0.863308	0.309472	0.857940

*metrics for the early stopping method to evaluate the possibility of overfitting*

And now, the loss for the training data and the test data looks like the following in a plot:

```
In [195]: metrics[['loss', 'val_loss']].plot()  
Out[195]: <matplotlib.axes._subplots.AxesSubplot at 0x179bc234a88>
```



```
In [202]: #No overfitting then!
```

*loss diagram for the training set V.S the test set*

Since the loss for the test set (“val\_loss” or the orange line) has not increased at any point and is always descending, then we can make sure that overfitting has not occurred!