Machine Learning Foundations

Feature Engineering

A critical part of the successful Machine Learning project is coming up with a good set of features to train on. This process is called feature engineering, and it involves three steps: feature transformation (transforming the original features), feature selection (selecting the most useful features to train on), and feature extraction (combining existing features to produce more useful ones). In this notebook we will explore different tools in Feature Engineering.

Objectives

After completing this lab you will be able to:

- Understand the types of Feature Engineering
  - Feature Transformation
    - Dealing with Categorical Variables
      - One Hot Encoding
      - Label Encoding
    - Date Time Transformations
  - Feature Selection
  - Feature Extraction using Principal Component Analysis

Setup

For this lab, we will be using the following libraries:

- `pandas` for managing the data.
- `numpy` for mathematical operations.
- `seaborn` for visualizing the data.
- `matplotlib` for visualizing the data.
- `plotly.express` for visualizing the data.
- `sklearn` for machine learning and machine-learning-pipeline related functions.

Installing Required Libraries

The following required modules are pre-installed in the Skills Network Labs environment. However if you run this notebook commands in a different Jupyter environment (e.g. Watson Studio or Ananconda) you will need to install these libraries by removing the `#` sign before `!mamba` in the code cell below.

In [2]:
```
# All Libraries required for this lab are listed below.
!mamba install -qy pandas==1.3.4 numpy==1.21.4 seaborn==0.9.0 matplotlib==3.5.0 scikit
# Note: If your environment doesn't support "!mamba install", use "!pip install"
```

```
Could not solve for environment specs
The following packages are incompatible
├─ matplotlib 3.5.0  is installable with the potential options
│  ├─ matplotlib [2.2.2|3.1.2|...|3.5.3] would require
│  │  └─ pyqt [ |>=5.6,<6.0a0 ] with the potential options
│  │     ├─ pyqt 5.6.0 would require
│  │     │  └─ qt 5.6.*  with the potential options
│  │     │     ├─ qt 5.6.2 would require
│  │     │     │  ├─ gst-plugins-base >=1.12.2,<1.13.0a0 , which requires
│  │     │     │  │  └─ gstreamer [>=1.12.2,<1.13.0a0 |>=1.12.4,<1.13.0a0 ], which re
quires
│  │     │     │  │     └─ glib >=2.53.6,<3.0a0 , which can be installed;
│  │     │     │  └─ openssl 1.0.* , which can be installed;
│  │     │     ├─ qt 5.6.2 would require
│  │     │     │  ├─ glib >=2.53.6,<3.0a0 , which can be installed;
│  │     │     │  └─ openssl >=1.0.2n,<1.0.3a , which can be installed;
│  │     │     ├─ qt 5.6.3 would require
│  │     │     │  ├─ glib >=2.56.1,<3.0a0 , which can be installed;
│  │     │     │  └─ openssl >=1.0.2o,<1.0.3a , which can be installed;
│  │     │     ├─ qt 5.6.3 would require
│  │     │     │  ├─ glib >=2.56.1,<3.0a0 , which can be installed;
│  │     │     │  └─ openssl >=1.0.2p,<1.0.3a , which can be installed;
│  │     │     └─ qt [5.6.3|5.9.7] would require
│  │     │        ├─ fontconfig >=2.13.0,<3.0a0  with the potential options
│  │     │        │  ├─ fontconfig 2.14.2 would require
│  │     │        │  │  └─ freetype >=2.12.1,<3.0a0 , which can be installed;
│  │     │        │  ├─ fontconfig [2.13.0|2.13.1] would require
│  │     │        │  │  └─ libuuid >=1.0.3,<2.0a0 , which can be installed;
│  │     │        │  ├─ fontconfig 2.14.1 would require
│  │     │        │  │  └─ libuuid >=1.41.5,<2.0a0 , which can be installed;
│  │     │        │  └─ fontconfig 2.14.1 would require
│  │     │        │     └─ freetype >=2.10.4,<3.0a0 , which can be installed;
│  │     │        └─ glib >=2.56.2,<3.0a0 , which can be installed;
│  │     ├─ pyqt [5.15.10|5.15.7|5.9.2] would require
│  │     │  └─ python >=3.10,<3.11.0a0 , which can be installed;
│  │     ├─ pyqt [5.15.10|5.15.7] would require
│  │     │  └─ python >=3.11,<3.12.0a0 , which can be installed;
│  │     ├─ pyqt 5.15.10 would require
│  │     │  └─ python >=3.12,<3.13.0a0 , which can be installed;
│  │     ├─ pyqt [5.15.10|5.15.7|5.9.2] would require
│  │     │  └─ python >=3.8,<3.9.0a0 , which can be installed;
│  │     ├─ pyqt [5.15.10|5.15.7|5.9.2] would require
│  │     │  └─ python >=3.9,<3.10.0a0 , which can be installed;
│  │     ├─ pyqt 5.15.7 would require
│  │     │  └─ qtwebkit 5.* , which requires
│  │     │     └─ glib >=2.69.1,<3.0a0 , which can be installed;
│  │     ├─ pyqt [5.6.0|5.9.2] would require
│  │     │  └─ python >=2.7,<2.8.0a0 , which can be installed;
│  │     ├─ pyqt [5.6.0|5.9.2] would require
│  │     │  └─ python >=3.5,<3.6.0a0 , which can be installed;
│  │     ├─ pyqt [5.6.0|5.9.2] would require
│  │     │  └─ python >=3.6,<3.7.0a0 , which can be installed;
│  │     └─ pyqt 5.9.2 would require
│  │        └─ qt [5.9.* |>=5.9.6,<5.10.0a0 ] with the potential options
│  │           ├─ qt [5.6.3|5.9.7], which can be installed (as previously explained);
│  │           ├─ qt 5.9.6 would require
│  │           │  ├─ glib >=2.56.1,<3.0a0 , which can be installed;
│  │           │  └─ openssl 1.0.* , which can be installed;
│  │           └─ qt [5.9.4|5.9.5] would require
│  │              └─ openssl 1.0.* , which can be installed;
```

```
│   ├─ matplotlib [3.5.0|3.5.1|...|3.8.0] would require
│   │  └─ python >=3.10,<3.11.0a0 , which can be installed;
│   ├─ matplotlib [3.1.1|3.1.2|...|3.7.2] would require
│   │  └─ python >=3.8,<3.9.0a0 , which can be installed;
│   └─ matplotlib [3.3.4|3.4.2|...|3.8.0] would require
│      └─ python >=3.9,<3.10.0a0 , which can be installed;
├─ numpy 1.21.4  does not exist (perhaps a typo or a missing channel);
└─ seaborn 0.9.0  is installable with the potential options
   ├─ seaborn 0.9.0 would require
   │  └─ matplotlib >=1.4.3  with the potential options
   │     ├─ matplotlib [2.2.2|3.1.2|...|3.5.3], which can be installed (as previously
explained);
   │     ├─ matplotlib [3.5.0|3.5.1|...|3.8.0], which can be installed (as previously
explained);
   │     ├─ matplotlib [3.1.1|3.1.2|...|3.7.2], which can be installed (as previously
explained);
   │     ├─ matplotlib [3.3.4|3.4.2|...|3.8.0], which can be installed (as previously
explained);
   │     ├─ matplotlib [2.0.2|2.1.0|...|2.2.3] would require
   │     │  └─ python >=2.7,<2.8.0a0 , which can be installed;
   │     ├─ matplotlib [2.0.2|2.1.0|...|3.0.0] would require
   │     │  └─ python >=3.5,<3.6.0a0 , which can be installed;
   │     ├─ matplotlib [2.0.2|2.1.0|...|3.3.4] would require
   │     │  └─ python >=3.6,<3.7.0a0 , which can be installed;
   │     ├─ matplotlib [2.2.3|3.0.0|...|3.1.2] would require
   │     │  └─ pyqt 5.9.* , which can be installed (as previously explained);
   │     ├─ matplotlib [3.6.2|3.7.1|3.7.2|3.8.0] would require
   │     │  └─ python >=3.11,<3.12.0a0 , which can be installed;
   │     └─ matplotlib 3.8.0 would require
   │        └─ python >=3.12,<3.13.0a0 , which can be installed;
   ├─ seaborn 0.9.0 would require
   │  └─ python >=2.7,<2.8.0a0 , which can be installed;
   ├─ seaborn 0.9.0 would require
   │  └─ python >=3.5,<3.6.0a0 , which can be installed;
   └─ seaborn 0.9.0 would require
      └─ python >=3.6,<3.7.0a0 , which can be installed.
```

In [3]:
```
!mamba install -qy openpyxl
```

```
Preparing transaction: ...working... done
Verifying transaction: ...working... done
Executing transaction: ...working... done
```

In [4]:
```
# Surpress warnings from using older version of sklearn:
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
```

In [5]:
```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import plotly.express as px

from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
```

Reading and understanding our data

For this lab, we will be using the airlines_data.xlsx file, hosted on IBM Cloud object. This dataset contains the prices of flight tickets for various airlines between the months of March and June of 2019 and between various cities. This dataset is often used for prediction analysis of the flight prices which are influenced by various factors, such as name of the airline, date of journey, route, departure and arrival times, the source and the destination of the trip, duration and other parameters.

In this notebook, we will use the airlines dataset to perform feature engineering on some of its independent variables.

Let's start by reading the data into *pandas* data frame and looking at the first 5 rows using the `head()` method.

In [21]:
```
data = pd.read_excel('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.clc
data.head()
```

Out[21]:

	Airline	Date_of_Journey	Source	Destination	Route	Dep_Time	Arrival_Time	Duration	Total_St
0	IndiGo	24/03/2019	Banglore	New Delhi	BLR → DEL	22:20	01:10 22 Mar	2h 50m	non-s
1	Air India	1/05/2019	Kolkata	Banglore	CCU → IXR → BBI → BLR	05:50	13:15	7h 25m	2 st
2	Jet Airways	9/06/2019	Delhi	Cochin	DEL → LKO → BOM → COK	09:25	04:25 10 Jun	19h	2 st
3	IndiGo	12/05/2019	Kolkata	Banglore	CCU → NAG → BLR	18:05	23:30	5h 25m	1 s
4	IndiGo	01/03/2019	Banglore	New Delhi	BLR → NAG → DEL	16:50	21:35	4h 45m	1 s

By using the `info` function, we will take a look at the types of data that our dataset contains.

In [7]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10683 entries, 0 to 10682
Data columns (total 11 columns):
 #   Column           Non-Null Count  Dtype 
---  ------           --------------  ----- 
 0   Airline          10683 non-null  object
 1   Date_of_Journey  10683 non-null  object
 2   Source           10683 non-null  object
 3   Destination      10683 non-null  object
 4   Route            10682 non-null  object
 5   Dep_Time         10683 non-null  object
 6   Arrival_Time     10683 non-null  object
 7   Duration         10683 non-null  object
 8   Total_Stops      10682 non-null  object
 9   Additional_Info  10683 non-null  object
 10  Price            10683 non-null  int64 
dtypes: int64(1), object(10)
memory usage: 918.2+ KB
```

As we see from the output above, we mostly have object data types, except for the 'price' column, which is an integer.

The `describe()` function provides the statistical information about the numerical variables. In our case, it is the 'price' variable.

In [9]:
```
data.describe().T
```

Out[9]:

	count	mean	std	min	25%	50%	75%	max
Price	10683.0	9087.064121	4611.359167	1759.0	5277.0	8372.0	12373.0	79512.0

Next, we will check for any null values.

In [10]:
```
data.isnull().sum()
```

Out[10]:
```
Airline            0
Date_of_Journey    0
Source             0
Destination        0
Route              1
Dep_Time           0
Arrival_Time       0
Duration           0
Total_Stops        1
Additional_Info    0
Price              0
dtype: int64
```

Now that we have found some null points, we need to either remove them from our dataset or fill them with something else. In this case, we will use `fillna()` and `method='ffill'`, which fills the last observed non-null value forward until another non-null value is encountered.

In [11]:
```
data = data.fillna(method='ffill')
```

Feature Transformation

Feature Transformation means transforming our features to the functions of the original features. For example, feature encoding, scaling, and discretization (the process of transforming continuous variables into discrete form, by creating bins or intervals) are the most common forms of data transformation.

Dealing with Categorical Variables

Categorical variables represent qualitative data with no apparent inherent mathematical meaning. Therefore, for any machine learning analysis, all the categorical data must be transformed into the numerical data types. First, we'll start with 'Airlines' column, as it contains categorical values. We will use `unique()` method to obtain all the categories in this column.

In [12]:
```
data['Airline'].unique().tolist()
```

Out[12]:
```
['IndiGo',
 'Air India',
 'Jet Airways',
 'SpiceJet',
 'Multiple carriers',
 'GoAir',
 'Vistara',
 'Air Asia',
 'Vistara Premium economy',
 'Jet Airways Business',
 'Multiple carriers Premium economy',
 'Trujet']
```

From the above list, we notice that some of the airline names are being repeated. For example, 'Jet Airways' and 'Jet Airways Business'. This means that some of the airlines are subdivided into separate parts. We will combine these 'two-parts' airlines to make our categorical features more consistent with the rest of the variables.

Here, we will use the *numpy* `where()` function to locate and combine the two categories.

In [19]:
```
data['Airline'] = np.where(data['Airline']=='Vistara Premium economy', 'Vistara', data
data['Airline'] = np.where(data['Airline']=='Jet Airways Business', 'Jet Airways', dat
```

The code above is using the NumPy library to modify the 'Airline' column in a DataFrame named `data`. Let's break down the code:

```
data['Airline'] = np.where(data['Airline']=='Jet Airways Business', 'Jet
Airways', data['Airline'])
```

Here's what each part of the code is doing:

1. `data['Airline']`: This part references the 'Airline' column in the DataFrame `data`. It selects the entire column.

2. `np.where(...)`: This is a NumPy function that is used for conditional assignment. It has the following structure:

    - The first argument is the condition to be checked (`data['Airline']=='Jet Airways Business'`).
    - The second argument is the value to be assigned when the condition is true (`'Jet Airways'`).
    - The third argument is the value to be assigned when the condition is false (`data['Airline']` - i.e., keep the original value).
3. The entire line effectively says: "If the value in the 'Airline' column is 'Jet Airways Business', replace it with 'Jet Airways'; otherwise, keep the original value."

So, after this line of code executes, the 'Airline' column in the DataFrame `data` will have the values modified according to the specified condition. This is a common technique in data manipulation to clean or transform data based on certain conditions.

Exercise 1

In this exercise, use `np.where()` function to combine 'Multiple carriers Premium economy' and 'Multiple carriers' categories, like shown in the code above. Print the newly created list using `unique().tolist()` functions.

In [22]:
```
# Enter your code and run the cell
data['Airline'] = np.where(data['Airline']=='Multiple carriers Premium economy','Multi
data['Airline'].unique().tolist()
```

Out[22]:
```
['IndiGo',
 'Air India',
 'Jet Airways',
 'SpiceJet',
 'Multiple carriers',
 'GoAir',
 'Vistara',
 'Air Asia',
 'Vistara Premium economy',
 'Jet Airways Business',
 'Trujet']
```

▶ **Solution** (Click Here)

One Hot Encoding

Now, to be recognized by a machine learning algorithms, our categorical variables should be converted into numerical ones. One way to do this is through *one hot encoding*. To learn more about this process, please visit this documentation.

We will use, `get_dummies()` method to do this transformation. In the next cell, we will transform 'Airline', 'Source', and 'Destination' into their respective numeric variables. We will put all the transformed data into a 'data1' data frame.

In [23]:
```
data1 = pd.get_dummies(data=data, columns = ['Airline', 'Source', 'Destination'])
```

In [24]:
```
data1.head()
```

Out[24]:

	Date_of_Journey	Route	Dep_Time	Arrival_Time	Duration	Total_Stops	Additional_Info	Price	Ai
0	24/03/2019	BLR → DEL	22:20	01:10 22 Mar	2h 50m	non-stop	No info	3897	
1	1/05/2019	CCU → IXR → BBI → BLR	05:50	13:15	7h 25m	2 stops	No info	7662	
2	9/06/2019	DEL → LKO → BOM → COK	09:25	04:25 10 Jun	19h	2 stops	No info	13882	
3	12/05/2019	CCU → NAG → BLR	18:05	23:30	5h 25m	1 stop	No info	6218	
4	01/03/2019	BLR → NAG → DEL	16:50	21:35	4h 45m	1 stop	No info	13302	

5 rows × 30 columns

Below, we will compare our original data frame with the transformed one.

In [25]:
```
data.shape
```

Out[25]:
```
(10683, 11)
```

In [26]:
```
data1.shape
```

Out[26]:
```
(10683, 30)
```

As we can see, we went from 11 original features in our dataset to 38. This is because *Pandas* `get_dummies()` approach when applied to a column with different categories (e.g. different airlines) will produce a new column (variable) for each unique categorical value (for each unique airline). It will place a one in the column corresponding to the categorical value present for that observation.

Exercise 2

In this exercise, use `value_counts()` to determine the values distribution of the 'Total_Stops' parameter.

In [28]:
```
# Enter your code and run the cell
data['Total_Stops'].value_counts()
```

Out[28]:
```
1 stop      5625
non-stop    3491
2 stops     1520
3 stops       45
4 stops        1
Name: Total_Stops, dtype: int64
```

▶ **Solution** (Click Here)

Label Encoding

Since 'Total_Stops' is originally a categorical data type, we also need to convert it into numerical one. For this, we can perform a label encoding, where values are manually assigned to the corresponding keys, like "0" to a "non-stop", using the `replace()` function.

In [29]:
```
data1.replace({"non-stop":0,"1 stop":1,"2 stops":2,"3 stops":3,"4 stops":4},inplace=Tr
data1.head()
```

Out[29]:

	Date_of_Journey	Route	Dep_Time	Arrival_Time	Duration	Total_Stops	Additional_Info	Price	Ai
0	24/03/2019	BLR → DEL	22:20	01:10 22 Mar	2h 50m	0.0	No info	3897	
1	1/05/2019	CCU → IXR → BBI → BLR	05:50	13:15	7h 25m	2.0	No info	7662	
2	9/06/2019	DEL → LKO → BOM → COK	09:25	04:25 10 Jun	19h	2.0	No info	13882	
3	12/05/2019	CCU → NAG → BLR	18:05	23:30	5h 25m	1.0	No info	6218	
4	01/03/2019	BLR → NAG → DEL	16:50	21:35	4h 45m	1.0	No info	13302	

5 rows × 30 columns

Date Time Transformations

Transforming the 'Duration' time column

Here, we will take a closer look at the `Duration` variable. Duration is the time taken by a plane to reach its destination. It is the difference between the `Dep_Time` and `Arrival_Time`. In our dataset, the `Duration` is expressed as a string, in hours and minutes. To be recognized by machine learning algorithms, we also need to transform it into numerical type.

The code below will iterate through each record in `Duration` column and split it into hours and minutes, as two additional separate columns. Also, we want to add the `Duration_hours` (in minutes) to the `Duration_minutes` column to obtain a `Duration_Total_mins` time, in minutes. The total duration time column will be useful feature for any regression type of analysis.

In [30]:

```
duration = list(data1['Duration'])
for i in range(len(duration)) :
    if len(duration[i].split()) != 2:
        if 'h' in duration[i] :
```

```
            duration[i] = duration[i].strip() + ' 0m'
        elif 'm' in duration[i] :
            duration[i] = '0h {}'.format(duration[i].strip())
dur_hours = []
dur_minutes = []

for i in range(len(duration)) :
    dur_hours.append(int(duration[i].split()[0][:-1]))
    dur_minutes.append(int(duration[i].split()[1][:-1]))


data1['Duration_hours'] = dur_hours
data1['Duration_minutes'] =dur_minutes
data1.loc[:,'Duration_hours'] *= 60
data1['Duration_Total_mins']= data1['Duration_hours']+data1['Duration_minutes']
```

This code above is designed to process a column named 'Duration' in a DataFrame (`data1`). The 'Duration' column contains time durations in the format "Xh Ym" where X is the number of hours and Y is the number of minutes. However, the code is handling cases where the format might not be consistent, and it's converting the durations into total minutes.

Let's break down the code step by step:

1. **Initialize a List:**

   ```
   duration = list(data1['Duration'])
   ```

   This line creates a list named `duration` containing the values from the 'Duration' column of the DataFrame.

2. **Handle Inconsistent Format:**

   ```
   for i in range(len(duration)):
       if len(duration[i].split()) != 2:
           if 'h' in duration[i]:
               duration[i] = duration[i].strip() + ' 0m'
           elif 'm' in duration[i]:
               duration[i] = '0h {}'.format(duration[i].strip())
   ```

   This loop iterates through each element in the `duration` list. If the duration does not have the format "Xh Ym" (i.e., it doesn't contain both hours and minutes), it's trying to handle these cases by adding '0h' or '0m' accordingly.We can break it further to look deeply into it. Let's break it down:

   > **Loop through each element in the `duration` list:**
   >
   > ```
   > for i in range(len(duration)):
   > ```
   >
   > This loop iterates through each element of the `duration` list.
   >
   > **Check if the format is inconsistent (not 'Xh Ym'):**
   >
   > ```
   > if len(duration[i].split()) != 2:
   > ```
   >
   > It checks if the current duration doesn't have the format "Xh Ym" by splitting the string and checking if the resulting list has a length different from 2.

> **Handle cases where hours ('h') or minutes ('m') are missing:**
>
> ```
> if 'h' in duration[i]:
>         duration[i] = duration[i].strip() + ' 0m'
>     elif 'm' in duration[i]:
>         duration[i] = '0h {}'.format(duration[i].strip())
> ```
>
> - If 'h' (hours) is present but 'm' (minutes) is missing, it adds ' 0m' to the end of the duration.
> - If 'm' (minutes) is present but 'h' (hours) is missing, it adds '0h ' to the beginning of the duration.

This part of the code ensures that each 'Duration' value ends up with a consistent format of "Xh Ym" before further processing. It's a form of data cleaning to handle variations in the format of the 'Duration' values.

1. **Extract Hours and Minutes:**

   ```
   dur_hours = []
   dur_minutes = []
   for i in range(len(duration)):
       dur_hours.append(int(duration[i].split()[0][:-1]))
       dur_minutes.append(int(duration[i].split()[1][:-1]))
   ```

   Certainly! This portion of the code is responsible for extracting the hours and minutes from each 'Duration' value and appending them to separate lists (`dur_hours` and `dur_minutes`). Let's break down the code:

```
dur_hours.append(int(duration[i].split()[0][:-1]))
dur_minutes.append(int(duration[i].split()[1][:-1]))
```

`duration[i].split()` **:**

This part splits the 'Duration' value into a list of strings using whitespace as the separator. For example, if the 'Duration' value is "5h 30m", `split()` would produce the list `['5h', '30m']` .

> **Extracting Hours:**
>
> ```
> dur_hours.append(int(duration[i].split()[0][:-1]))
> ```
>
> - `duration[i].split()[0]` extracts the first element of the split list, which corresponds to the hours part.
> - `[:-1]` is used to exclude the last character, which is 'h' (indicating hours).
> - `int(...)` converts the result to an integer before appending it to the `dur_hours` list.
>
> **Extracting Minutes:**
>
> ```
> dur_minutes.append(int(duration[i].split()[1][:-1]))
> ```

> - `duration[i].split()[1]` extracts the second element of the split list, which corresponds to the minutes part.
> - `[:-1]` is used to exclude the last character, which is 'm' (indicating minutes).
> - `int(...)` converts the result to an integer before appending it to the `dur_minutes` list.

In summary, these lines of code are extracting the hours and minutes from each 'Duration' value, converting them to integers, and then appending them to separate lists. These lists will be used later to create new columns in the DataFrame to represent the duration in hours and minutes separately. This loop extracts the hours and minutes from each duration in the modified `duration` list and stores them in separate lists `dur_hours` and `dur_minutes`.

1. **Convert Hours to Minutes:**

   ```
   data1['Duration_hours'] = dur_hours
   data1['Duration_minutes'] = dur_minutes
   data1.loc[:, 'Duration_hours'] *= 60
   ```

   This part adds two new columns, 'Duration_hours' and 'Duration_minutes', to the DataFrame `data1`. It also converts the 'Duration_hours' to minutes by multiplying by 60.

2. **Calculate Total Duration in Minutes:**

   ```
   data1['Duration_Total_mins'] = data1['Duration_hours'] +
   data1['Duration_minutes']
   ```

   This line creates a new column named 'Duration_Total_mins' in the DataFrame `data1` that represents the total duration in minutes by adding the converted hours and minutes.

In summary, this code is processing a 'Duration' column with varying formats, extracting hours and minutes, converting hours to minutes, and creating a new column with the total duration in minutes.

Print 'data1' data frame to see the newly created columns.

In [32]:
```
data1[['Duration_hours','Duration_minutes','Duration_Total_mins']]
```

Out[32]:

	Duration_hours	Duration_minutes	Duration_Total_mins
0	120	50	170
1	420	25	445
2	1140	0	1140
3	300	25	325
4	240	45	285
...	...	...	...
10678	120	30	150
10679	120	35	155
10680	180	0	180
10681	120	40	160
10682	480	20	500

10683 rows × 3 columns

As you have noticed, three new columns were created: `Duration_hours`, `Duration_minutes`, and `Duration_Total_mins` - all numerical values.

Transforming the 'Departure' and 'Arrival' Time Columns

Now, we will transform the 'Dep_Time' and 'Arrival_Time' columns to the appropriate date and time format. We will use *pandas* `to_datetime()` function for this.

We will split the 'Dep_Time' and 'Arrival_Time' columns into their corresponding hours and minutes columns.

In [38]:

```
data1["Dep_Hour"]= pd.to_datetime(data1['Dep_Time']).dt.hour
data1["Dep_Min"]= pd.to_datetime(data1['Dep_Time']).dt.minute
```

The code above is using the Pandas library in Python to extract the hour and minute components from a column named 'Dep_Time' in a DataFrame named `data1`. It involves converting the 'Dep_Time' values to datetime objects and then extracting the hour and minute components.

Let's break down the code:

```
data1["Dep_Hour"] = pd.to_datetime(data1['Dep_Time']).dt.hour
data1["Dep_Min"] = pd.to_datetime(data1['Dep_Time']).dt.minute
```

1. **`pd.to_datetime(data1['Dep_Time'])`:** This part converts the 'Dep_Time' column to Pandas datetime objects. The `to_datetime` function in Pandas is used for this purpose.
2. **`.dt.hour` and `.dt.minute`:** After converting 'Dep_Time' to datetime objects, `.dt.hour` is used to extract the hour component, and `.dt.minute` is used to extract the

minute component.

3. **Creating New Columns:**

```
data1["Dep_Hour"] = ...
data1["Dep_Min"] = ...
```

   These lines create two new columns in the DataFrame `data1` named 'Dep_Hour' and 'Dep_Min' to store the extracted hour and minute components, respectively.

In summary, these lines of code create new columns 'Dep_Hour' and 'Dep_Min' in the DataFrame `data1` containing the hour and minute components of the 'Dep_Time' values. This can be useful for further analysis or visualization based on the departure times.

Exercise 3

Now, let's transform the 'Arrival_Time' column.

In [39]:
```
# Enter your code and run the cell
data1['Arrival_Hour'] = pd.to_datetime(data1['Arrival_Time']).dt.hour
data1['Arrival_Min'] = pd.to_datetime(data1['Arrival_Time']).dt.minute
```

▶ **Solution** (Click Here)

In [41]:
```
data1[['Arrival_Hour','Arrival_Min','Dep_Hour','Dep_Min']]
```

Out[41]:

	Arrival_Hour	Arrival_Min	Dep_Hour	Dep_Min
0	1	10	22	20
1	13	15	5	50
2	4	25	9	25
3	23	30	18	5
4	21	35	16	50
...	...	...	...	...
10678	22	25	19	55
10679	23	20	20	45
10680	11	20	8	20
10681	14	10	11	30
10682	19	15	10	55

10683 rows × 4 columns

Splitting 'Departure/Arrival_Time' into Time Zones

To further transform our 'Departure/Arrival_Time' column, we can break down the 24 hours format for the departure and arrival time into 4 different time zones: night, morning, afternoon,

and evening. This might be an interesting feature engineering technique to see what time of a day has the most arrivals/departures.

One way to do this is transformation is by using *pandas* `cut()` function.

In [40]:
```
data1['dep_timezone'] = pd.cut(data1.Dep_Hour, [0,6,12,18,24], labels=['Night','Mornir
data1['dep_timezone']
```

Out[40]:
```
0          Evening
1            Night
2          Morning
3        Afternoon
4        Afternoon
           ...
10678      Evening
10679      Evening
10680      Morning
10681      Morning
10682      Morning
Name: dep_timezone, Length: 10683, dtype: category
Categories (4, object): ['Night' < 'Morning' < 'Afternoon' < 'Evening']
```

This code is using the Pandas library in Python to create a new column named 'dep_timezone' in the DataFrame `data1` based on the values in the existing 'Dep_Hour' column. It categorizes the departure hours into different time zones (Night, Morning, Afternoon, and Evening) using the `pd.cut` function.

Let's break down the code:

```
data1['dep_timezone'] = pd.cut(data1.Dep_Hour, [0, 6, 12, 18, 24], labels=
['Night', 'Morning', 'Afternoon', 'Evening'])
```

1. **`pd.cut(data1.Dep_Hour, [0, 6, 12, 18, 24], labels=['Night', 'Morning', 'Afternoon', 'Evening'])`:**
   - `pd.cut` is a Pandas function used for binning values into discrete intervals.
   - `data1.Dep_Hour` is the column that is being binned (in this case, the departure hours).
   - `[0, 6, 12, 18, 24]` defines the bin edges. The values will be placed into bins based on these edges.
   - `labels=['Night', 'Morning', 'Afternoon', 'Evening']` specifies the labels to assign to each bin.
2. **`data1['dep_timezone'] = ...`:** This line creates a new column named 'dep_timezone' in the DataFrame `data1` and assigns the bin labels to each corresponding departure hour.

Exercise 4

Now, let's transform the 'Arrival_Time' column into its corresponding time zones, as shown in the example above.

In [44]:
```
# Enter your code and run the cell
data1['arr_timezones']= pd.cut(data1.Arrival_Hour,[0,6,12,18,24],labels=['Night','Morr
data1['arr_timezones']
```

Out[44]:
```
0            Night
1        Afternoon
2            Night
3          Evening
4          Evening
           ...
10678      Evening
10679      Evening
10680      Morning
10681    Afternoon
10682      Evening
Name: arr_timezones, Length: 10683, dtype: category
Categories (4, object): ['Night' < 'Morning' < 'Afternoon' < 'Evening']
```

▶ **Solution** (Click Here)

Transforming the 'Date_of_Journey' Column

Similar to the departure/arrival time, we will now extract some information from the 'date_of_journey' column, which is also an object type and can not be used for any machine learning algorithm yet.

So, we will extract the month information first and store it under the 'Month' column name.

In [45]:
```
data1['Month']= pd.to_datetime(data1["Date_of_Journey"], format="%d/%m/%Y").dt.month
```

Exercise 5

Now, let's create 'Day' and 'Year' columns in a similar way.

In [49]:
```
# Enter your code and run the cell
data1['Day'] = pd.to_datetime(data1["Date_of_Journey"], format="%d/%m/%Y").dt.day
data1['Year'] = pd.to_datetime(data1["Date_of_Journey"], format="%d/%m/%Y").dt.year
```

▶ **Solution** (Click Here)

Additionally, we can extract the day of the weak name by using `dt.day_name()` function.

In [50]:
```
data1['day_of_week'] = pd.to_datetime(data1['Date_of_Journey']).dt.day_name()
```

In [51]:
```
data1[['Day','Month','Year','day_of_week']]
```

Out[51]:

	Day	Month	Year	day_of_week
0	24	3	2019	Sunday
1	1	5	2019	Saturday
2	9	6	2019	Friday
3	12	5	2019	Thursday
4	1	3	2019	Thursday
...	...	...	...	...
10678	9	4	2019	Wednesday
10679	27	4	2019	Saturday
10680	27	4	2019	Saturday
10681	1	3	2019	Thursday
10682	9	5	2019	Thursday

10683 rows × 4 columns

Feature Selection

Here, we will select only those attributes which best explain the relationship of the independent variables with respect to the target variable, 'price'. There are many methods for feature selection, building the heatmap and calculating the correlation coefficients scores are the most commonly used ones.

First, we will select only the relevant and newly transformed variables (and exclude variables such as 'Route', 'Additional_Info', and all the original categorical variables), and place them into a 'new_data' data frame.

We will print all of our data1 columns.

In [52]:

```
data1.columns
```

Out[52]:

```
Index(['Date_of_Journey', 'Route', 'Dep_Time', 'Arrival_Time', 'Duration',
       'Total_Stops', 'Additional_Info', 'Price', 'Airline_Air Asia',
       'Airline_Air India', 'Airline_GoAir', 'Airline_IndiGo',
       'Airline_Jet Airways', 'Airline_Jet Airways Business',
       'Airline_Multiple carriers', 'Airline_SpiceJet', 'Airline_Trujet',
       'Airline_Vistara', 'Airline_Vistara Premium economy', 'Source_Banglore',
       'Source_Chennai', 'Source_Delhi', 'Source_Kolkata', 'Source_Mumbai',
       'Destination_Banglore', 'Destination_Cochin', 'Destination_Delhi',
       'Destination_Hyderabad', 'Destination_Kolkata', 'Destination_New Delhi',
       'Duration_hours', 'Duration_minutes', 'Duration_Total_mins',
       'Arrival_Hour', 'Arrival_Min', 'Dep_Hour', 'Dep_Min', 'dep_timezone',
       'arr_timezones', 'Month', 'Day', 'Year', 'day_of_week'],
      dtype='object')
```

In [53]:

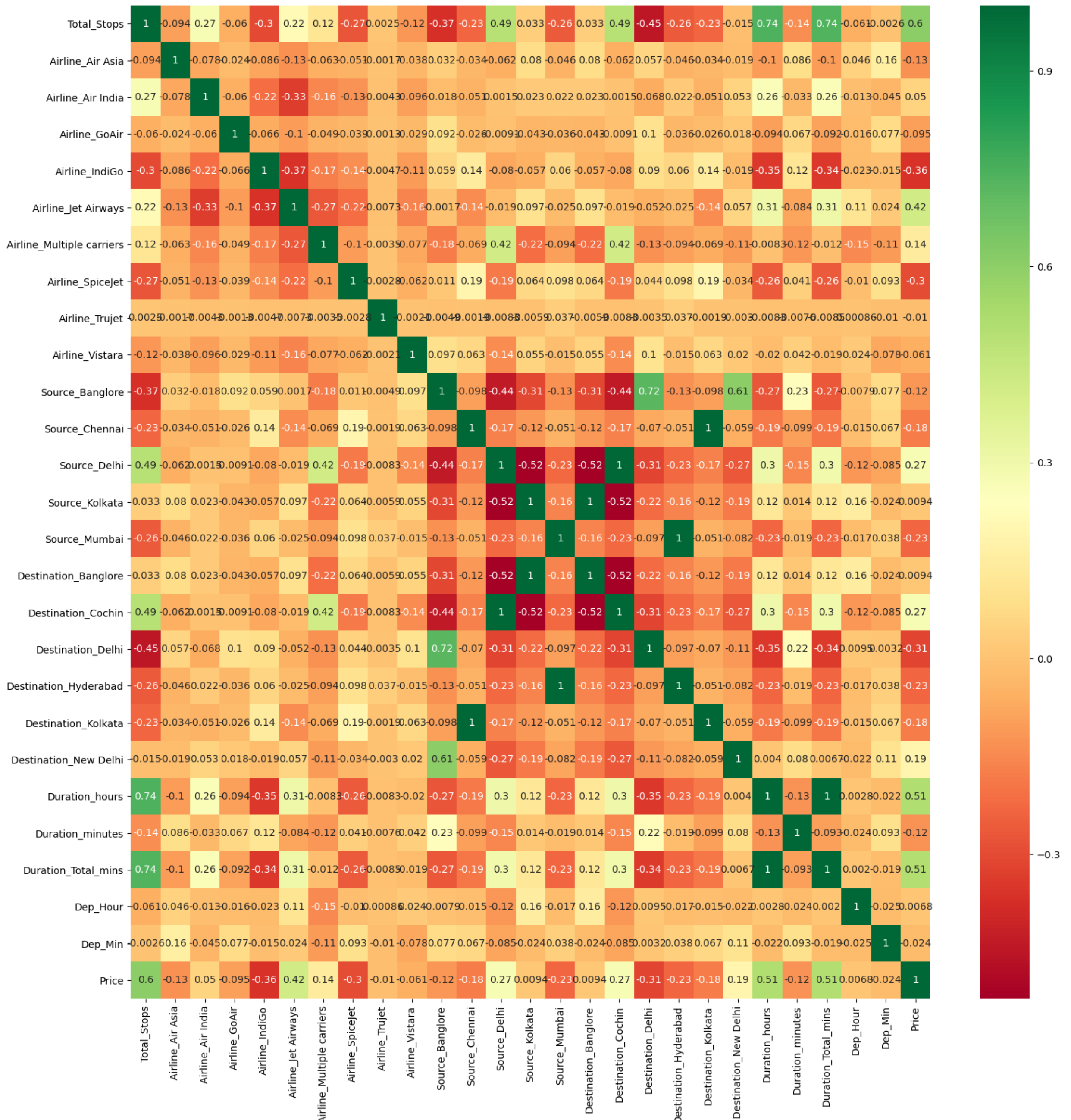
```
new_data = data1.loc[:,['Total_Stops', 'Airline_Air Asia',
       'Airline_Air India', 'Airline_GoAir', 'Airline_IndiGo',
```

```
       'Airline_Jet Airways', 'Airline_Multiple carriers', 'Airline_SpiceJet',
       'Airline_Trujet', 'Airline_Vistara', 'Source_Banglore',
       'Source_Chennai', 'Source_Delhi', 'Source_Kolkata', 'Source_Mumbai',
       'Destination_Banglore', 'Destination_Cochin', 'Destination_Delhi',
       'Destination_Hyderabad', 'Destination_Kolkata', 'Destination_New Delhi',
       'Duration_hours', 'Duration_minutes', 'Duration_Total_mins', 'Dep_Hour',
       'Dep_Min', 'dep_timezone', 'Price']]
```

Now we will construct a `heatmap()`, using the *seaborn* library with a newly formed data frame, 'new_data'.

In [54]:
```
plt.figure(figsize=(18,18))
sns.heatmap(new_data.corr(),annot=True,cmap='RdYlGn')

plt.show()
```



From the heatmap above, extreme green means highly positively correlated features (relationship between two variables in which both variables move in the same direction), extreme red means negatively correlated features (relationship between two variables in which an increase in one variable is associated with a decrease in the other).

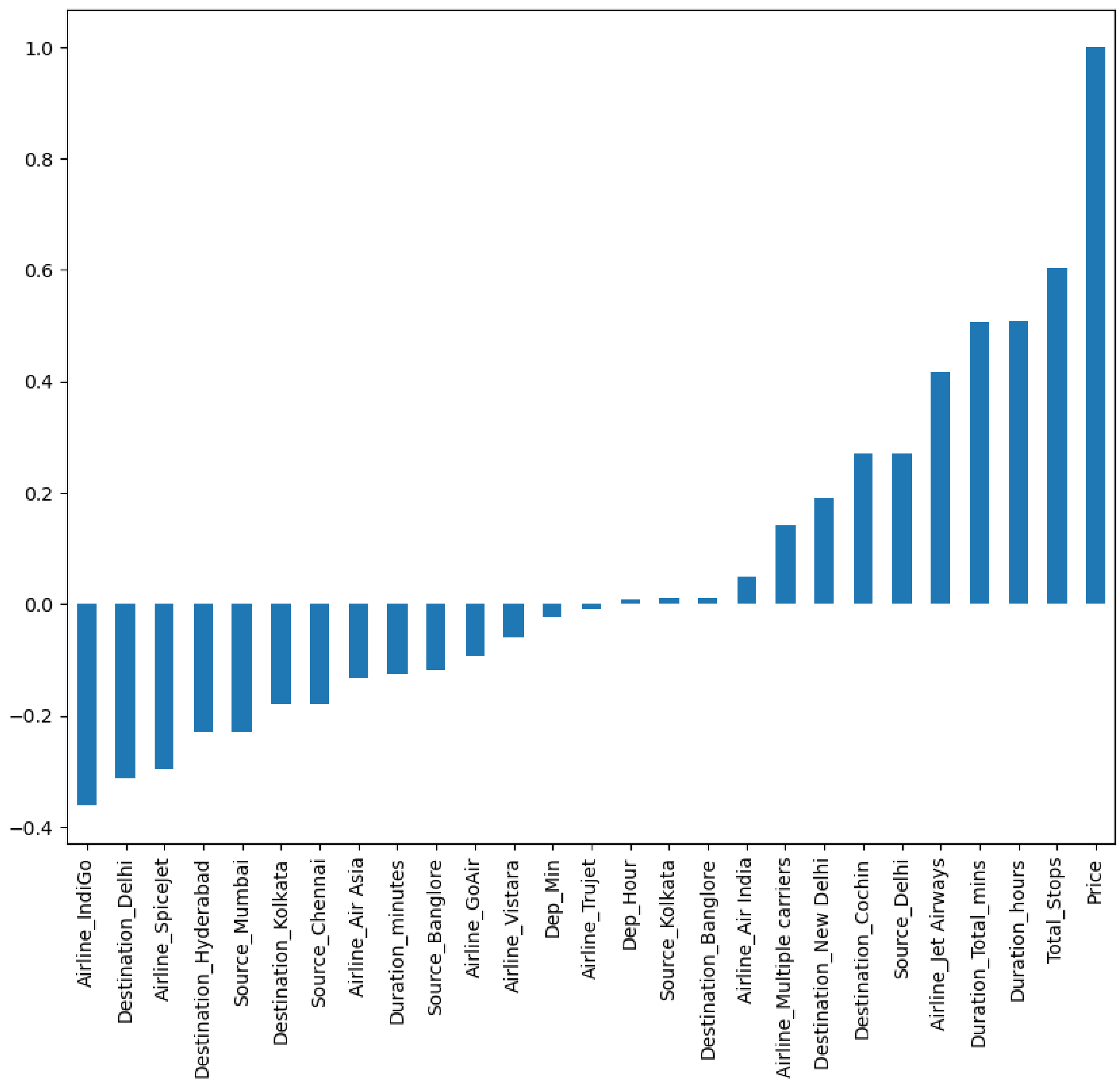
Now, we can use the `corr()` function to calculate and list the correlation between all independent variables and the 'price'.

In [55]:
```
features = new_data.corr()['Price'].sort_values()
features
```

Out[55]:
```
Airline_IndiGo              -0.361048
Destination_Delhi           -0.313401
Airline_SpiceJet            -0.296552
Destination_Hyderabad       -0.230745
Source_Mumbai               -0.230745
Destination_Kolkata         -0.179216
Source_Chennai              -0.179216
Airline_Air Asia            -0.133044
Duration_minutes            -0.124874
Source_Banglore             -0.118026
Airline_GoAir               -0.095146
Airline_Vistara             -0.060646
Dep_Min                     -0.024492
Airline_Trujet              -0.010380
Dep_Hour                     0.006819
Source_Kolkata               0.009377
Destination_Banglore         0.009377
Airline_Air India            0.050346
Airline_Multiple carriers    0.141087
Destination_New Delhi        0.189785
Destination_Cochin           0.270619
Source_Delhi                 0.270619
Airline_Jet Airways          0.416135
Duration_Total_mins          0.506371
Duration_hours               0.508672
Total_Stops                  0.603897
Price                        1.000000
Name: Price, dtype: float64
```

We can also plot these correlation coefficients for easier visualization.

In [57]:
```
features.plot(kind='bar',figsize=(10,8))
plt.show()
```



From the graph above, we can deduct some of the highly correlated features and select only those ones for any future analysis.

Feature Extraction using Principal Component Analysis (Optional)

PCA with Scikit-Learn

Dimentionality reduction is part of the feature extraction process that combines the existing features to produce more useful ones. The goal of dimensionality reduction is to simplify the data without loosing too much information. Principal Component Analysis (PCA) is one of the most popular dimensionality reduction algorithms. First, it identifies the hyperplane that lies closest to the data, and then it projects the data onto it. In this way, a few multidimensional features are merged into one.

In the following portion of the lab, we will use `scikit-learn` library to perform some PCA on our data. To learn more about `scikit-learn` PCA, please visit this documentation.

First, we must scale our data using the `StandardScaler()` function. We will assign all the independent variables to x, and the dependent variable, 'price', to y.

In [58]:
```
x = data1.loc[:,['Total_Stops', 'Airline_Air Asia',
       'Airline_Air India', 'Airline_GoAir', 'Airline_IndiGo',
       'Airline_Jet Airways', 'Airline_Multiple carriers', 'Airline_SpiceJet',
       'Airline_Trujet', 'Airline_Vistara', 'Source_Banglore',
       'Source_Chennai', 'Source_Delhi', 'Source_Kolkata', 'Source_Mumbai',
       'Destination_Banglore', 'Destination_Cochin', 'Destination_Delhi',
       'Destination_Hyderabad', 'Destination_Kolkata', 'Destination_New Delhi',
       'Duration_hours', 'Duration_minutes', 'Duration_Total_mins', 'Dep_Hour',
       'Dep_Min']]
```

In [59]:
```
y= data1.Price
```

In [62]:
```
scaler = StandardScaler()
x=scaler.fit_transform(x.astype(np.float64))
x
```

Out[62]:
```
array([[-1.22066609, -0.17544122, -0.44291155, ..., -0.93158255,
         1.65425948, -0.23505036],
       [ 1.74143186, -0.17544122,  2.25778713, ..., -0.39007152,
        -1.30309491,  1.36349161],
       [ 1.74143186, -0.17544122, -0.44291155, ...,  0.97847452,
        -0.60724682,  0.0313733 ],
       ...,
       [-1.22066609, -0.17544122, -0.44291155, ..., -0.91189124,
        -0.78120884, -0.23505036],
       [-1.22066609, -0.17544122, -0.44291155, ..., -0.95127386,
        -0.25932278,  0.29779696],
       [ 1.74143186, -0.17544122,  2.25778713, ..., -0.28176932,
        -0.4332848 ,  1.62991527]])
```

Once the data is scaled, we can apply the `fit_transform()` function to reduce the dimensionality of the dataset down to two dimensions.

In [65]:
```
print(np.isnan(x).sum())
```

```
1
```

In [66]:
```
x = np.nan_to_num(x)
```

In [67]:
```
pca = PCA(n_components = 2)
pca.fit_transform(x)
```

Out[67]:
```
array([[-2.87631608, -0.55645701],
       [ 0.31883409,  2.39182774],
       [ 3.05947064, -0.52675097],
       ...,
       [-2.24761482, -0.58799069],
       [-2.69663415, -0.285652  ],
       [ 1.92522086, -1.10484483]])
```

Explained Variance Ratio

Another useful piece of information in PCA is the explained variance ratio of each principal component, available via the `explained_variance_ratio_` function. The ratio indicates the proportion of the dataset's variance that lies along each principal component. Let's look at the explained variance ratio of each of our two components.

In [68]:
```
explained_variance=pca.explained_variance_ratio_
explained_variance
```

Out[68]:
```
array([0.17545413, 0.12112304])
```

The first component constitutes 17.54% of the variance and second component constitutes 12.11% of the variance between the features.

Exercise 6 (Optional)

In this exercise, experiment with the number of components to see how many dimensions our dataset could be reduced to in order to explain most of the variability between the features. Additionally, you can plot the components using bar plot to see how much variability each component represents.

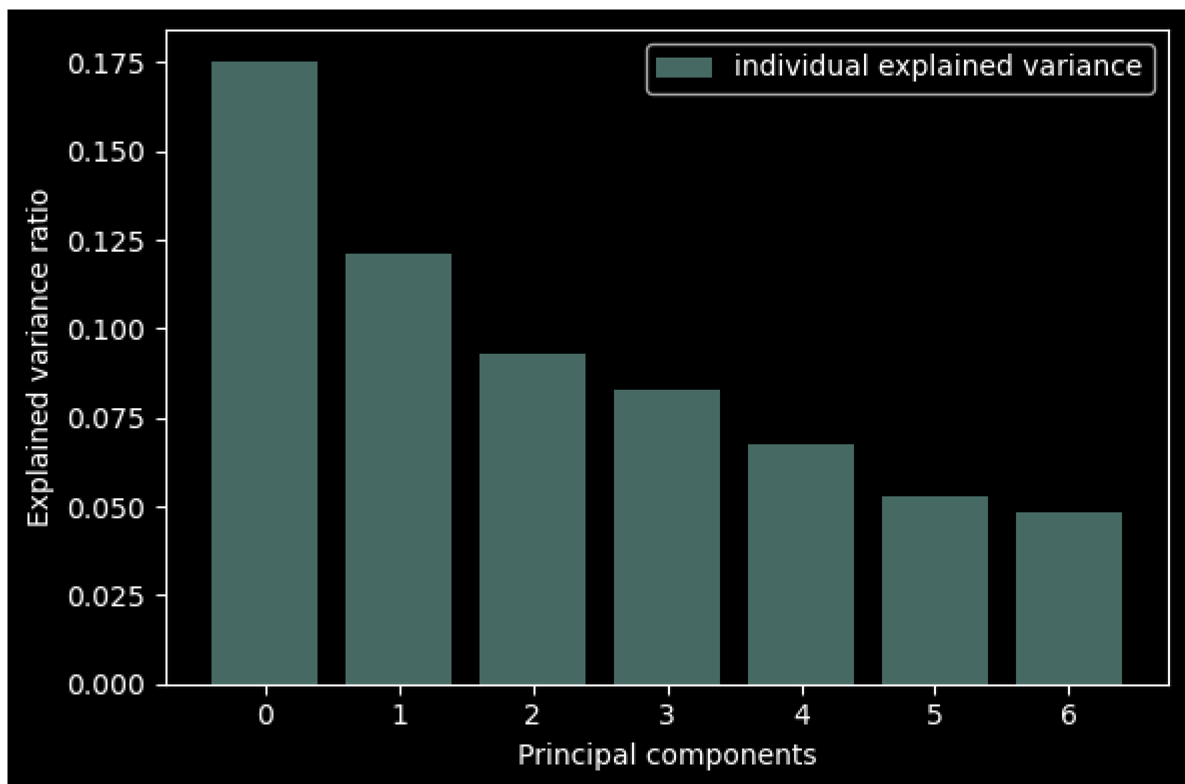
In [75]:
```
# Enter your code and run the cell
pca = PCA(n_components = 7)
pca.fit_transform(x)
explained_variance=pca.explained_variance_ratio_
explained_variance
```

Out[75]:
```
array([0.17545413, 0.12112304, 0.09263209, 0.0827907 , 0.06737639,
       0.0527503 , 0.04818182])
```

In [78]:
```
# Enter your code and run the cell
with plt.style.context('dark_background'):


    plt.figure(figsize=(6, 4))

    plt.bar(range(7), explained_variance, alpha=0.5, align='center',
    label='individual explained variance')
    plt.ylabel('Explained variance ratio')
    plt.xlabel('Principal components')
    plt.legend(loc='best')
    plt.tight_layout()
```



▶ **Solution_part1** (Click Here)

▶ **Solution_part2** (Click Here)

Choosing the Right Number of Dimensions

Instead of arbitrary choosing the number of dimensions to reduce down to, it is simpler to choose the number of dimensions that add up to a sufficiently large proportion of the variance, let's say 95%.

The following code performs PCA without reducing dimensionality, then computes the minimum number of dimensions required to preserve 95% of the variance.

In [79]:
```
pca = PCA()
pca.fit(x)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >=0.95) + 1
```

In [80]:
```
d
```

Out[80]:
```
16
```

There are 16 components required to meet 95% variance. Therefore, we could set n_components = 16 and run PCA again. However, there is better way, instead of specifying the number of principal components you want to preserve, you can set n_components to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve.

In [81]:
```
pca = PCA(n_components=0.95)
x_reduced = pca.fit_transform(x)
```

There is also a graphical way to determine the number of principal components in your analysis. It is to plot the explained variance as a function of the number of dimensions. There will usually be an elbow in the curve, where the explained variance stops growing fast. That point is usually the optimal point for the number of principal components.

In [ ]:
```
px.area(
    x=range(1, cumsum.shape[0] + 1),
    y=cumsum,
    labels={"x": "# Components", "y": "Explained Variance"}
)
```

Congratulations! - You have completed the lab