Ogunlaiye Sope

Customer segmentation
With python

Introduction

Customer segmentation is the process of grouping customers based on shared characteristics. This process makes it easier to direct your marketing and sales efforts in a more efficient and effective way that leads to customer loyalty and boost conversions.

This project looks at data from a fictional bank in the Uk. It contains data about the accounts opened and other details for the period of a year. The aim of this project is to

show the steps involved in the process of analyzing data with python, so it's very beginner friendly.

The analysis shows the following:

- Job class distribution.
- The relationship between account balance and job class.
- Grouping per region (group count).
- Account balance per region.
- Account balance per age.
- Gender distribution.
- Number of accounts opened as per date distribution.

Data source:

This data set was from an Edx course I audited when learning Power BI. You can download the csv file from my Github

Methodology:

This is supposed to be an instructional article to take the reader step by step on the data analytics process with python in Jupyter notebook. For the .ipynb file you can always visit my Github. Without much ado let's get to it!

Step1

The very first thing to do is to import all relevant python libraries we will need for this analysis. Below is a screenshot of this:

In [1]:
```
import pandas as pd
import numpy as np
import datetime
from datetime import date, timedelta
import plotly.graph_objects as go
import plotly.express as px
import plotly.io as pio
from plotly.subplots import make_subplots
pio.renderers.default='notebook'
pio.templates.default = "plotly_white"
import ipywidgets as widgets
```

We use the pandas library for data cleaning and modeling. The numpy library is also imported because it is useful for dealing with arrays and matrix data structures. Numpy also contains many useful mathematical functions important in the analysis process. The remaining libraries are python libraries like plotly that aid in the visualizations we'll do in the course of this project.

Step 2:

Next, we'll use the .read_csv() query to load the csv file (BankCustomersUK.csv) and also use .head() to view the first few lines of the file. We'll also use the .info() query to get a summary of the number of rows and columns and their data types. Here is the pandas syntax and output :

```
In [2]:  1 customer = pd.read_csv('BankCustomersUK.csv')

In [3]:  1 customer.info()
         2 customer.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4014 entries, 0 to 4013
Data columns (total 9 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Customer ID         4014 non-null   int64
 1   Name                4014 non-null   object
 2   Surname             4014 non-null   object
 3   Gender              4014 non-null   object
 4   Age                 4014 non-null   int64
 5   Region              4014 non-null   object
 6   Job Classification  4014 non-null   object
 7   Date Joined         4014 non-null   object
 8   Balance             4014 non-null   float64
dtypes: float64(1), int64(2), object(6)
memory usage: 282.4+ KB
```

Out[3]:

	Customer ID	Name	Surname	Gender	Age	Region	Job Classification	Date Joined	Balance
0	100000001	Simon	Walsh	Male	21	England	White Collar	05.Jan.15	113810.15
1	400000002	Jasmine	Miller	Female	34	Northern Ireland	Blue Collar	06.Jan.15	36919.73
2	100000003	Liam	Brown	Male	46	England	White Collar	07.Jan.15	101536.83
3	300000004	Trevor	Parr	Male	32	Wales	White Collar	08.Jan.15	1421.52
4	100000005	Deirdre	Pullman	Female	38	England	Blue Collar	09.Jan.15	35639.79

Looking at the output, I can already tell that there isn't much to clean up. However, there are a few things that also stand out. Firstly, the dtype of the 'Date Joined' column is wrong so we will have to convert it to the right dtype. Secondly, there are some irrelevant columns we won't be needing for the analysis so we'll be deleting them. Next, we'll check that there

are no null values with the isnull() function. All these actions are part of the data cleaning process we'll be exploring in the next step.

Step 3:

Welcome to the data cleaning step! We have already mentioned some of the actions we are going to take to make our data analyzable.

First of all let's check for null values using the isnull().sum() function. The purpose of this is to ensure that there aren't any missing values in the dataset. Since the data contains over 4,000 rows, it'll be strenuous to confirm this at a glance. That is why we use the isnull() function. Here is the syntax and the output for this:

```
In [5]:  1 #Checking for null values
         2 customer.isnull().sum()

Out[5]: Customer ID          0
        Name                 0
        Surname              0
        Gender               0
        Age                  0
        Region               0
        Job Classification   0
        Date Joined          0
        Balance              0
        dtype: int64
```

We can see that there are no null values. This means that there are no empty rows and all rows are filled. Next we go ahead to change the dtype of the 'Date Joined' column. All data types are divided into dtypes. For example strings usually are of the object dtype, whole numbers belong to the int64 dtype, decimal numbers are of the float dtype and Dates belong to the Date or Datetime64 dtype. However, the date column in our data shows 'Date Joined' appearing as an object dtype (check the result of the .info() query in step1). This means that we have to correct this using the to_datetime() function. To read more about pandas dtypes click here. Here is the syntax for changing the object dtype to a datetime64 dtype:

```
In [4]:
#converting to date
customer['Date Joined'] = pd.to_datetime(customer['Date Joined'])
customer['Date Joined']
```

```
Out[4]: 0      2015-01-05
        1      2015-01-06
        2      2015-01-07
        3      2015-01-08
        4      2015-01-09
                  ...
        4009   2015-12-30
        4010   2015-12-30
        4011   2015-12-30
        4012   2015-12-30
        4013   2015-12-30
        Name: Date Joined, Length: 4014, dtype: datetime64[ns]
```

We can see from the output, the dtype has been converted to datetime64.

Next, we'll shorten the column name of the 'Job Classification' column to 'Job Class' just for convenience and also to show you an example of how to rename columns with python. For this, we use the .rename() function. Here is the syntax and the output:

```
[7]:
#rename the job classification column to something shorter
customer.rename(columns={"Job Classification": "Job Class"}, inplace=True)
customer.head(1)
```

[7]:

	Customer ID	Name	Surname	Gender	Age	Region	Job Class	Date Joined	Balance
0	100000001	Simon	Walsh	Male	21	England	White Collar	2015-01-05	113810.15

We can see it's pretty easy to change column names using the .rename() function.

We'll now delete the irrelevant columns that don't have any role to play in our analysis. We will use the .drop() query to do this. The columns are the customer ID, Name and Surname columns. Here is the syntax and output:

```
In [8]:
#delete irrelevat columns
customer = customer.drop(['Customer ID', 'Name', 'Surname'], axis=1)
```

```
In [9]:
#confirm deletion
customer.head(1)
```

Out[9]:

	Gender	Age	Region	Job Class	Date Joined	Balance
0	Male	21	England	White Collar	2015-01-05	113810.15

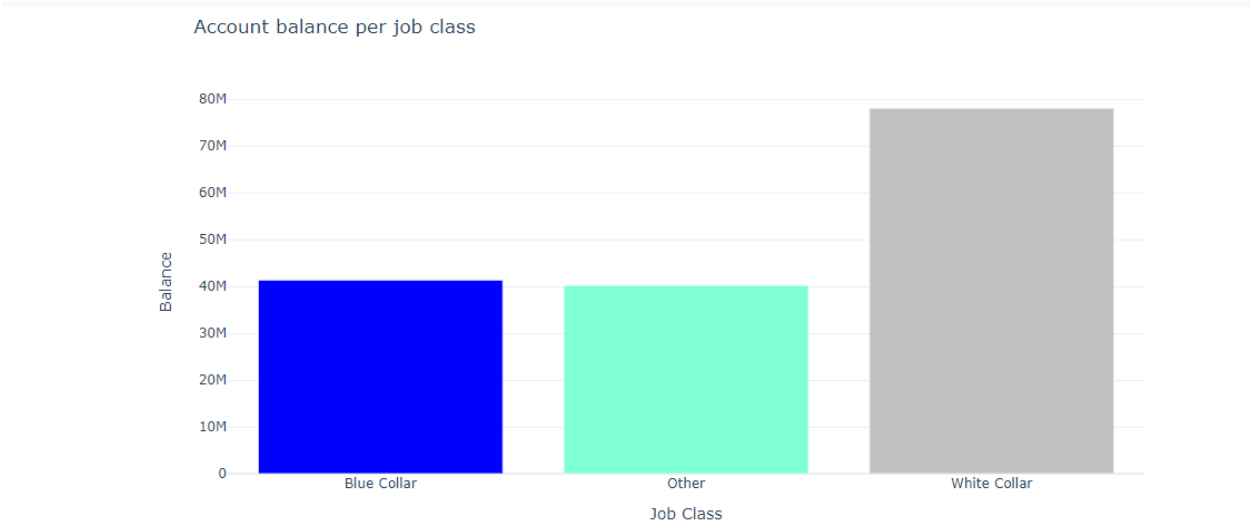
When compared to the previous output you'll see that this new output has dropped some columns like earlier mentioned.

Step 4:

The data is now ready to be visualized.

a. Let's look at the relationship between Account balance and job class. We'll use a simple bar chart for this. Here is the syntax and the output

```
In [11]:
1 customer2 = customer.groupby('Job Class')['Balance'].sum()
2 fig = px.bar(customer2, x = customer2.index, y = 'Balance', title = 'Account balance per job class')
3 fig.update_traces(marker_color=['blue', 'aquamarine', 'silver'])
4 fig.show('notebook')
```



Observations:

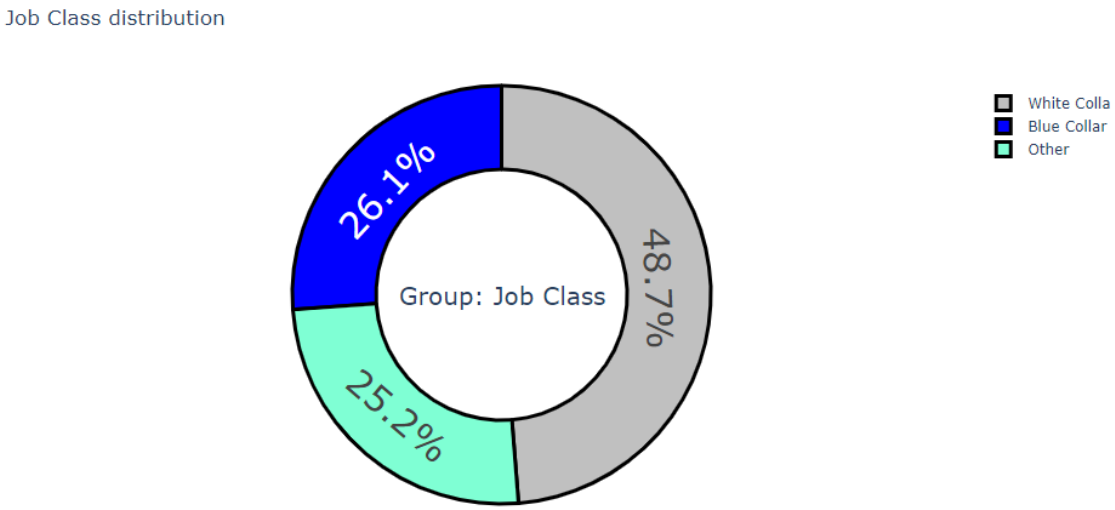
I. White collar: White collar workers have the highest bank balances with a total value of over £78M.
II. Blue collar: The blue collars have the next highest bank balances with over £41M.

III. Other: The other group has the lowest total account balances with a little above £40M.

Just because a group has the largest bank balance doesn't mean that they are the largest group by population. They could just have richer people. Is that the case in this scenario? Next I'll do a donut chart to confirm this

b. A Donut chart visualization showing the job class distribution. Here is the syntax and output:

```
Job = customer['Job Class'].value_counts()
label = Job.index
counts = Job.values
colors = ['silver', 'blue', 'aquamarine']
fig = go.Figure(data=[go.Pie(labels=label, values=counts)])
fig.update_layout(title_text='Job Class distribution', annotations=[dict(text='Group: Job Class', x=0.5, y=0.5,
                                                                         font_size=20, showarrow=False)])
fig.update_traces(hole= .6, hoverinfo='label+value', textinfo='percent', textfont_size=30,
                  marker=dict(colors=colors, line=dict(color='black', width=3)))
fig.show('notebook')
```



Observations:

I. White collar: The largest segment of account openers are the white collars which make up almost half of all the accounts opened for that year (48.7%)

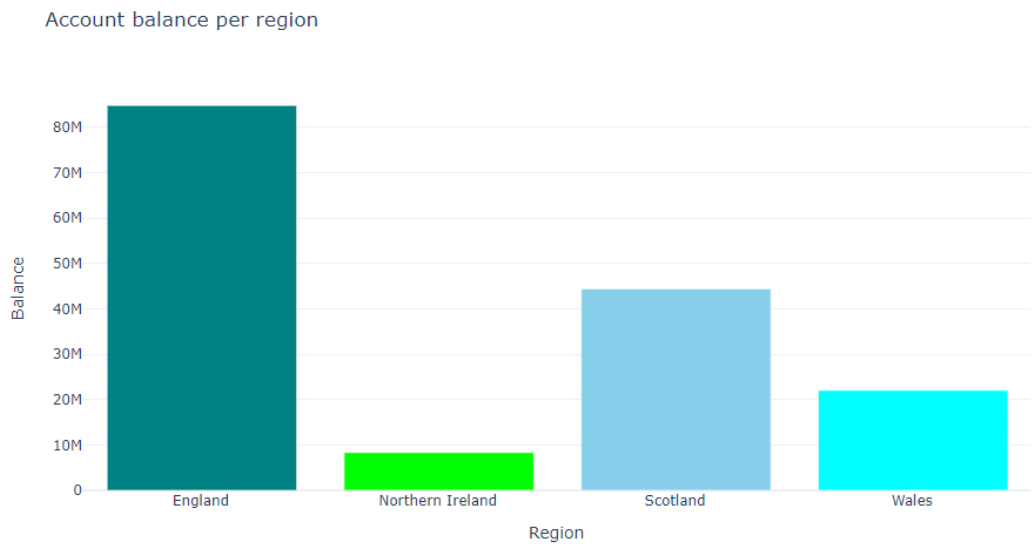
II. Blue Collar: The blue collars have the second largest percentage of account openers with 26.1%

III. Other: The others follow the blue collars closely with a 0.9% difference in the share percentage (25.2%)

From the above, it is clear to see that the biggest customer base is the white collar group so it would make sense to design financial products that will cater to that demographic of customers which are most likely salary earners.

c. Account balance per region. We use a bar chart for this visualization. Here is the syntax:

```
In [13]:
region = customer.groupby('Region')['Balance'].sum()
fig = px.bar(region, x = region.index, y = 'Balance', title = 'Account balance per region')
fig.update_traces(marker_color=['teal', 'lime', 'skyblue', 'cyan'])
fig.show('notebook')
```



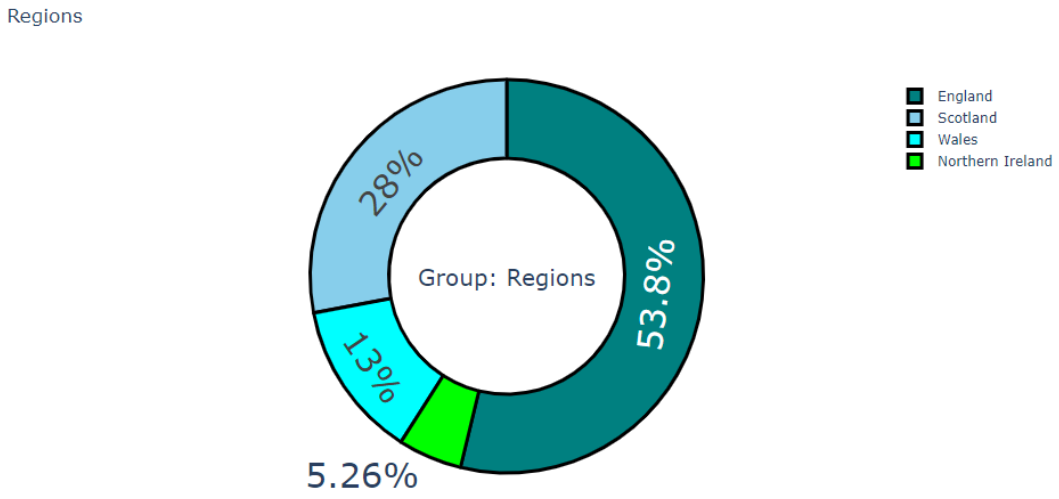
Observations:

The region with the highest account balance is England £84.83M, followed by Scotland £44.4M, Wales £22.04M and Northern Ireland respectively £8.3M.

Like I said above, just because a region has the largest bank balance doesn't mean that region is the most populous. We do a donut chart to confirm this

d. Grouping by region. We will use a donut chart for this visualization. Here's the syntax and the output:

```
Region = customer['Region'].value_counts()
label = Region.index
counts = Region.values
colors = ['teal','skyblue', 'cyan', 'lime']
fig = go.Figure(data=[go.Pie(labels=label, values=counts)])
fig.update_layout(title_text='Regions', annotations=[dict(text='Group: Regions', x=0.5, y=0.5, font_size=20,
                                                          showarrow=False)])
fig.update_traces(hole= .6, hoverinfo='label+value', textinfo='percent', textfont_size=30,
                  marker=dict(colors=colors, line=dict(color='black', width=3)))
fig.show('notebook')
```



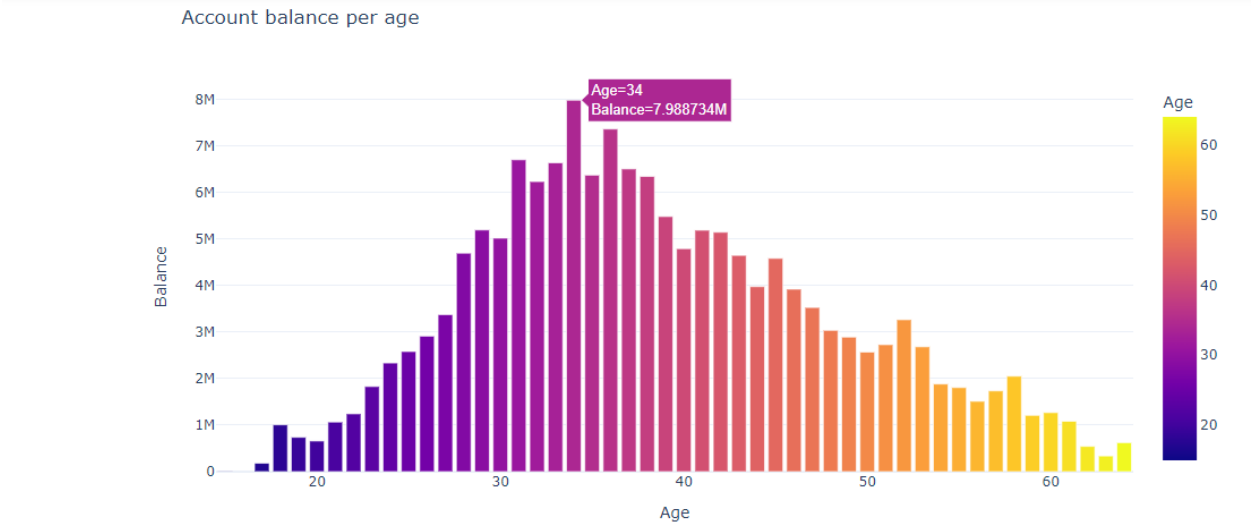
Observations:

It has been rightfully noted above that England is the region with the most accounts opened, followed by Scotland, Wales and Northern Ireland

Next, Relationship between Age and account balance

e. Account balance by age. Here's the syntax and the output:

```
In [15]:
1 age = customer.groupby('Age')['Balance'].sum()
2 fig = px.bar(age, x = age.index, y = 'Balance', title = 'Account balance per age', color = age.index)
3
4 fig.show('notebook')
```

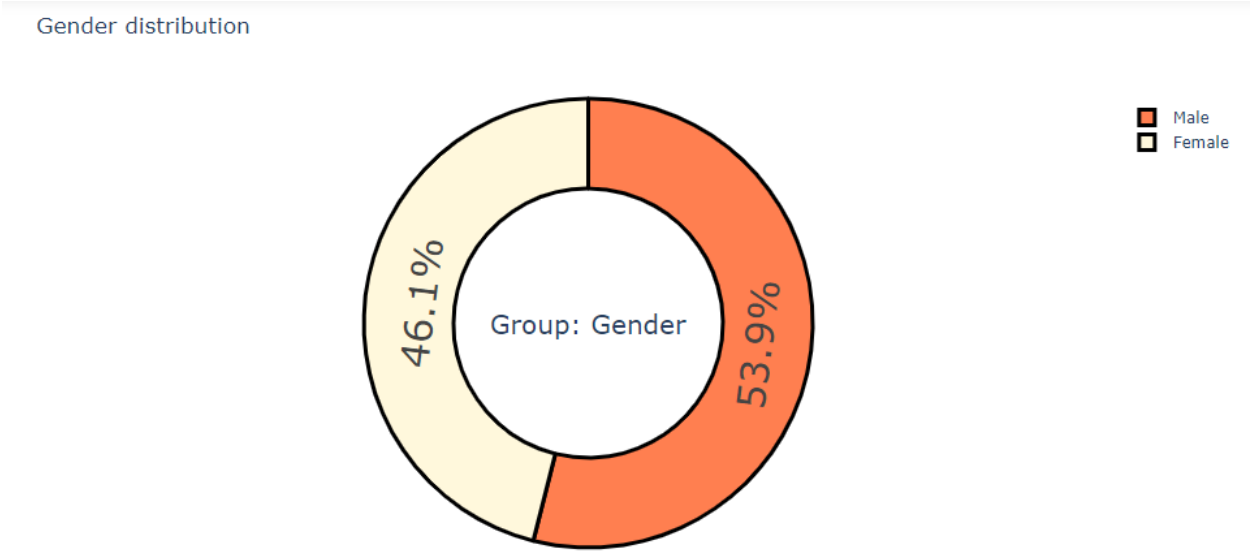


Observations:

We can see that the segment of customers with the highest account balances is the age group ranging from the late 20's to the mid 40's. Ages 17 to 27 and ages 48 and above are the group segments with a low account balance. This information is useful in knowing which age range to focus marketing efforts on. It can also help in the research stage of a product design.

f. Gender distribution using a donut chart. Here's the syntax and the output:

```
In [16]:
Gender = customer['Gender'].value_counts()
label = Gender.index
counts = Gender.values
colors = ['coral', 'cornsilk']
fig = go.Figure(data=[go.Pie(labels=label, values=counts)])
fig.update_layout(title_text='Gender distribution', annotations=[dict(text='Group: Gender', x=0.5, y=0.5, font_size=20,
                                                                      showarrow=False)])
fig.update_traces(hole= .6, hoverinfo='label+value', textinfo='percent', textfont_size=30,
                  marker=dict(colors=colors, line=dict(color='black', width=3)))
fig.show('notebook')
```

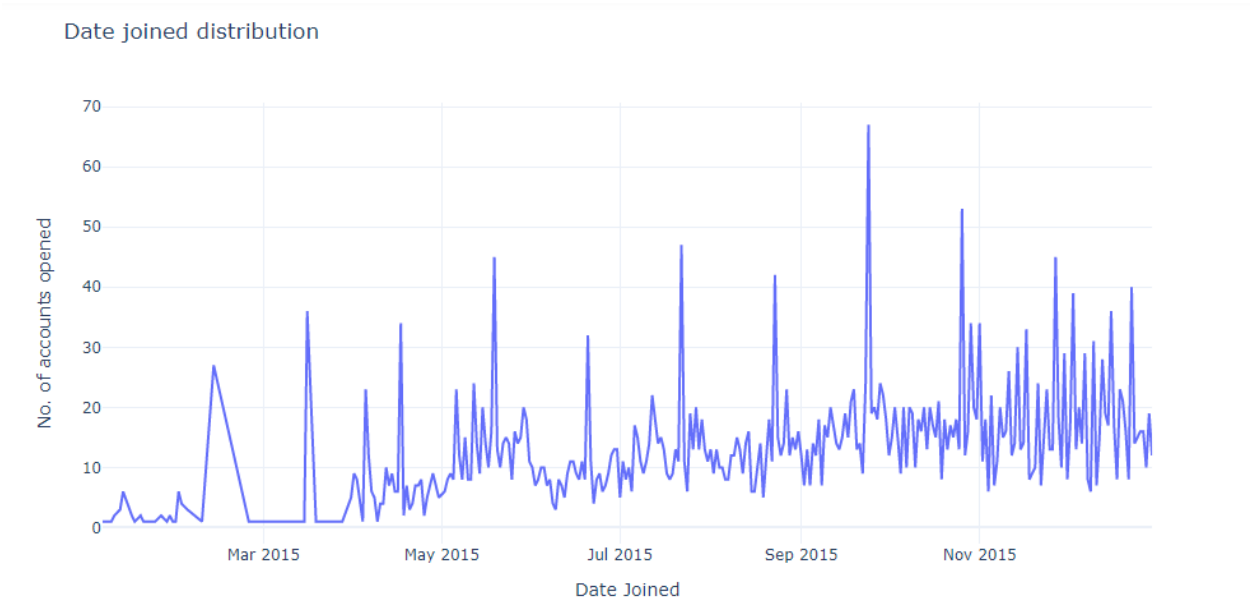


Observation:

The gender distribution is almost equal with just a 7.8% difference.

g. Number of accounts opened as per date distribution. We are using a line chart for this visualization: Here is the syntax and output:

```
In [17]:
date1 = customer.groupby('Date Joined')['Balance'].count().rename('No. of accounts opened').reset_index()
fig = px.line(date1, x= 'Date Joined', y='No. of accounts opened', title= 'Date joined distribution')
fig.show('notebook')
```



Observation:

We can see the account opening was slow during the beginning of the year but gradually increases as the year progresses. Being that this is just a year's data we cannot use this as a periodic pattern of account opening.

Conclusion:

We were able to do some analysis on the various segments of the bank customers i.e by sex, by age and by job classification. We were able to determine that England has the highest customer base and that a bulk of the bank's customers come from the white collar job types. This means that it would make a lot of sense if banking products are designed to cater to such an audience. The age demographic with the biggest account balances are between the late 20's to the mid 40's. These analysis sheds some insight on the data and makes us understand the customer demography better.