Case Study 1: Fake Instagram Accounts

We will demonstrate how to work with data in Python in this Jupyter notebook. In this case study, we will explore the relationship between categorical variables in a dataset of fake and real Instagram accounts to demonstrate how we might consider the "full data science pipeline" approach to making informed, business decisions when it comes to identifying fake and real accounts.

1. Reading in the data from a csv file in a compressed folder

The data that we will be using in this analysis will be in the fake_insta.csv file, which is located in the same folder as this Jupyter notebook. In order to extract and view the data, we need to import a Python package:

- **pandas** - Python package for reading and processing flat data files.

We'll be using **pandas** regularly throughout out course.

In [1]:
```
import pandas as pd
```

We can use the pandas "read_csv" function to read the file into a pandas data frame (df).

In [2]:
```
# extract data file and read into a data frame
df = pd.read_csv('fake_insta.csv')
```

2. Learning the scope of the data

One of my first steps when I start interacting with a dataset is to learn more about the "scope" of the data. I like to know more about what is in the data, including the size of the data set, what defines a row (observational unit), how many and what variables are recorded, and what types of values those variables take.

There are many ways to learn about a data file. Some require knowledge about the data source. Others we can answer using our analytical tools.

2.1. Previewing the data

First, we could ask python to print the full data frame (which is how Pandas reads and stores the data set).

In [3]:
```
df
```

Out[3]:

	has_a_profile_pic	number_of_words_in_name	num_characters_in_bio	number_of_posts	number_of_follo
0	yes	1	30	35	
1	yes	5	64	3	
2	yes	2	82	319	
3	yes	1	143	273	1
4	yes	1	76	6	
...	...	...	...	...	

	has_a_profile_pic	number_of_words_in_name	num_characters_in_bio	number_of_posts	number_of_follo
115	yes	1	0	13	
116	yes	1	0	4	
117	yes	2	0	3	
118	no	1	0	1	
119	yes	1	0	3	

120 rows × 7 columns

Because there are so many rows, many of the middle rows are skipped. We could get similar information using the **`head()`** function, which prints just the first few lines of data.

By default, `head()` prints 5 rows. We can specify the number of rows that we'd like printed using an argument inside of the head function.

In [4]:
```
df.head()
```

Out[4]:

	has_a_profile_pic	number_of_words_in_name	num_characters_in_bio	number_of_posts	number_of_follow
0	yes	1	30	35	
1	yes	5	64	3	
2	yes	2	82	319	
3	yes	1	143	273	14
4	yes	1	76	6	

In [5]:
```
df.head(12)
```

Out[5]:

	has_a_profile_pic	number_of_words_in_name	num_characters_in_bio	number_of_posts	number_of_follo
0	yes	1	30	35	
1	yes	5	64	3	
2	yes	2	82	319	
3	yes	1	143	273	14
4	yes	1	76	6	
5	yes	1	0	6	
6	yes	1	132	9	
7	yes	2	0	19	
8	yes	2	96	17	
9	yes	1	78	9	
10	yes	1	0	53	
11	yes	1	78	97	

The result of `head()` is still a dataframe.

In [6]:
```
type(df.head(12))
```

Out[6]:
```
pandas.core.frame.DataFrame
```

2.2. Understanding Rows & Columns

How many rows and columns are there in the data frame? The **`shape`** attribute gives us this information.

In [7]:
```
df.shape
```

Out[7]:
```
(120, 7)
```

What type of object did we just create?

In [8]:
```
type(df.shape)
```

Out[8]:
```
tuple
```

This tells us that there are 120 rows (observations) and 7 columns (variables) in the dataset.

When we were previewing the data, we see that some of the middle columns were skipped, specifically in the printed version of the data.

We can extract the variables in the dataframe using pandas.

In [9]:
```
print(df.columns.values)
```

```
['has_a_profile_pic' 'number_of_words_in_name' 'num_characters_in_bio'
 'number_of_posts' 'number_of_followers' 'number_of_follows'
 'account_type']
```

What type of object did we just print?

In [10]:
```
type(df.columns.values)
```

Out[10]:
```
numpy.ndarray
```

We can also view the column labels & row labels of the dataframe.

In [11]:
```
df.columns
```

Out[11]:
```
Index(['has_a_profile_pic', 'number_of_words_in_name', 'num_characters_in_bio',
       'number_of_posts', 'number_of_followers', 'number_of_follows',
       'account_type'],
      dtype='object')
```

In [12]:
```
type(df.columns)
```

Out[12]:
```
pandas.core.indexes.base.Index
```

In [13]:
```
df.index
```

Out[13]:
```
RangeIndex(start=0, stop=120, step=1)
```

In [14]:
```
type(df.index)
```

Out[14]:
```
pandas.core.indexes.range.RangeIndex
```

In this case, our row labels are not the most informative.

Looking at our variables in the dataframe, it seems reasonable that each row corresponds to an Instagram account/profile. An observational unit is an Instagram account.

Now, let's inspect what types of objects are contained in each of the variables in the dataframe.

In [15]:
```
df.dtypes
```

Out[15]:
```
has_a_profile_pic          object
number_of_words_in_name     int64
num_characters_in_bio       int64
number_of_posts             int64
number_of_followers         int64
number_of_follows           int64
account_type               object
dtype: object
```

The `has_a_profile_pic` and the `account_type` columns in df are comprised of 'object' data type observations. The other columns are comprise of 'int64' data type observations.

Looking back at the first few rows of our dataframe, we can understand better what these mean.

In [16]:
```
df.head()
```

Out[16]:

	has_a_profile_pic	number_of_words_in_name	num_characters_in_bio	number_of_posts	number_of_follow
0	yes	1	30	35	
1	yes	5	64	3	
2	yes	2	82	319	
3	yes	1	143	273	14
4	yes	1	76	6	

3. Summary Statistics for a Categorical Variable

Suppose that we'd like to analyze a categorical variable. We'll look at the `has_a_profile_pic` variable in the Instagram dataframe.

3.1. Isolating a Variable

We can isolate a single column from the dataframe using brackets after the name of the dataframe and including the name of the column we would like to isolate.

In [17]:
```
df['has_a_profile_pic']
```

Out[17]:
```
0      yes
1      yes
2      yes
3      yes
4      yes
      ...
115    yes
116    yes
```

```
117    yes
118     no
119    yes
Name: has_a_profile_pic, Length: 120, dtype: object
```

What type of object is this column?

In [18]:
```
type(df['has_a_profile_pic'])
```

Out[18]:
```
pandas.core.series.Series
```

The single column that we have isolated from df is a **pandas series** object. We can think of this as a dataframe with just one column.

3.2. Count Calculation

We can then summarize this variable by counting the number of observations at each level. We do this with the pandas function **`.value_counts()`**.

In [19]:
```
df['has_a_profile_pic'].value_counts()
```

Out[19]:
```
yes    91
no     29
Name: has_a_profile_pic, dtype: int64
```

In [20]:
```
type(df['has_a_profile_pic'].value_counts())
```

Out[20]:
```
pandas.core.series.Series
```

In [21]:
```
df['has_a_profile_pic'].shape
```

Out[21]:
```
(120,)
```

The isolated column is now a series with only one column. Notice how this affects the results of the `shape` attribute.

3.3. Percentage/Proportion Calculation

Most **functions** in Python have a series of **parameters** that can be set within the function.

- Some parameters in a given function do not need to be specified. In this case, the function uses the 'defaul' value prescribed for that parameter. You can always Google a Python function and read the documentation created for that function to determine what the default value is for eadh parameter.
- Some parameters in a given function do need to be specified. If you do not specify it, Python will throw you an error.

By setting the **normalize** parameter equal to True in the **`.value_counts()`** function, we can calculate percentages/proportions instead of counts.

In [22]:
```
df['has_a_profile_pic'].value_counts(normalize=True)
```

Out[22]:
```
yes    0.758333
no     0.241667
Name: has_a_profile_pic, dtype: float64
```

In [23]:
```
type(True)
```

Out[23]:
```
bool
```

In [24]:
```
type(False)
```

Out[24]:
```
bool
```

The True & False that we use in Python and saw for the normalize parameter are Booleans.

4. Visualizating a Categorical Variable

In order to visualize the data, we import two graphics modules that we will use frequently throughout the course:

- **matplotlib.pyplot** - Basic python graphics functions
- **seaborn** - Enhanced graphics functions with additional styles and capabilities

In [25]:
```
import matplotlib.pyplot as plt    # basic graphing package
import seaborn as sns; sns.set()   # enhanced graphic package
```

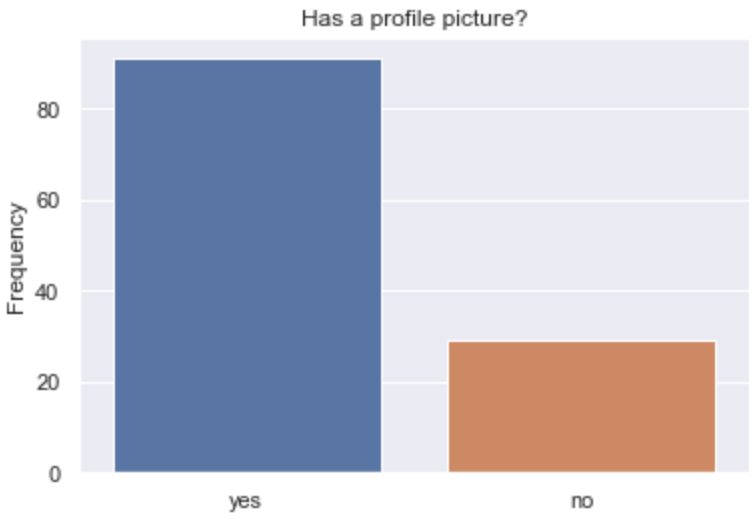
We can create barplots or bar graphs of a categorical variable using seaborn (abbreviated as sns). We can choose whether to use the counts or the proportions.

First, we create a new series with the summary measures from 3 above. We can assign (save) this to a named object, as below.

In [26]:
```
has_pic_counts = df['has_a_profile_pic'].value_counts()
display(has_pic_counts.shape, has_pic_counts)
```

```
(2,)
yes    91
no     29
Name: has_a_profile_pic, dtype: int64
```

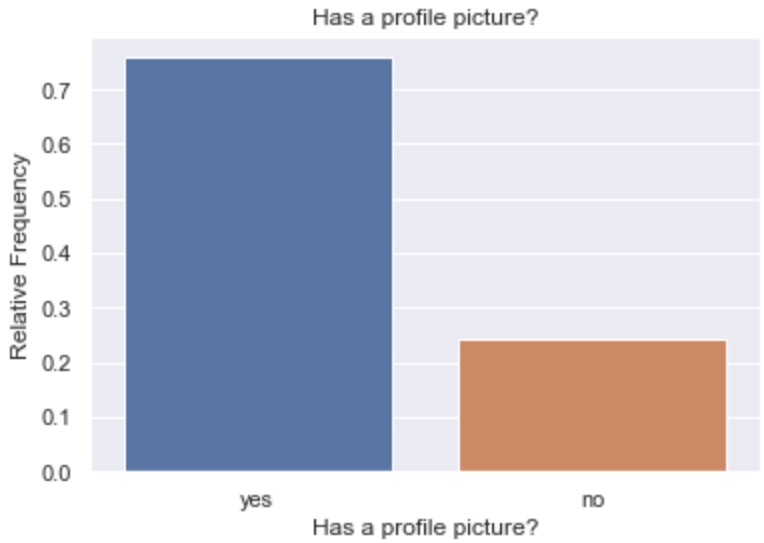
In [27]:
```
sns.barplot(x=has_pic_counts.index, y=has_pic_counts)
plt.title('Has a profile picture?')
plt.ylabel('Frequency')
plt.show()
```



In [28]:
```
has_pic_counts_perc = df['has_a_profile_pic'].value_counts(normalize=True)
display(has_pic_counts_perc.shape, has_pic_counts_perc)
```

```
(2,)
yes    0.758333
no     0.241667
Name: has_a_profile_pic, dtype: float64
```

In [29]:
```
sns.barplot(x=has_pic_counts_perc.index, y=has_pic_counts_perc)
plt.title('Has a profile picture?')
plt.xlabel('Has a profile picture?')
plt.ylabel('Relative Frequency')
plt.show()
```



5. Summary Statistics for Two Categorical Variables

While we may sometimes want to describe a single variable, we often would like to describe the relationship between two variables.

How do the frequencies and relative frequencies of having/not having a profile picture (variable 1) vary based on whether the Instagram account is fake or not (variable 2)? Let's look at the **cross-tabulation** of the two variables.

5.1. Count Calculation

Now, we can't simply count for one variable. Instead, we need to count the number of observations at each combination of level types for the categorical variables. We can do this with the **`crosstab`** function.

In [30]:
```
pd.crosstab(df['account_type'], df['has_a_profile_pic'])
```

Out[30]:

has_a_profile_pic	no	yes
account_type		
fake	29	31
real	0	60

In [31]:
```
pd.crosstab(df['has_a_profile_pic'], df['account_type'])
```

Out[31]:

has_a_profile_pic	fake	real
no	29	0
yes	31	60

5.2. Percentage Calculation

First, we might be interested in calculating the proportion of all respondents that fall into each combination as performed below.

In [32]:
```
pd.crosstab(df['has_a_profile_pic'], df['account_type'], normalize=True)
```

Out[32]:

has_a_profile_pic	fake	real
no	0.241667	0.0
yes	0.258333	0.5

Often, we might be interested in a more nuanced calculation. We might want to split our data for the levels of one variable, and then look at the percentage distribution of the other variable.

For example, what percent of fake accounts have (or conversely don't have) a profile picture? What percent of real accounts have a profile picture?

In [33]:
```
pd.crosstab(df['account_type'], df['has_a_profile_pic'], normalize='index')
```

Out[33]:

account_type	no	yes
fake	0.483333	0.516667
real	0.000000	1.000000

What percent of accounts with a profile picture are real (or conversely fake)? What percent of accounts without a profile picture are real?

In [34]:
```
pd.crosstab(df['has_a_profile_pic'], df['account_type'], normalize='index')
```

Out[34]:

has_a_profile_pic	fake	real
no	1.000000	0.000000
yes	0.340659	0.659341

6. Visualizing Two Categorical Variables

Suppose that we want to visualize the relationship between these two categorical variables. What are our options?

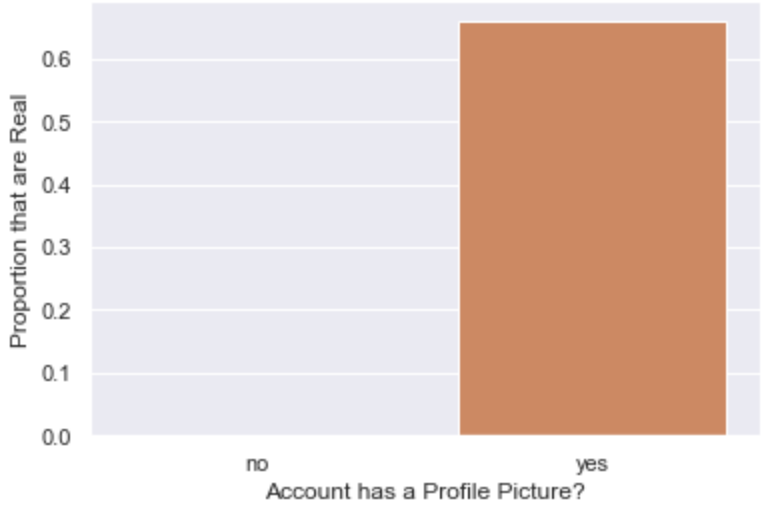
6.1. Create separate barplots for each level of a variable

In [35]:
```
temp = pd.crosstab(df['has_a_profile_pic'], df['account_type'], normalize='index')
temp
```

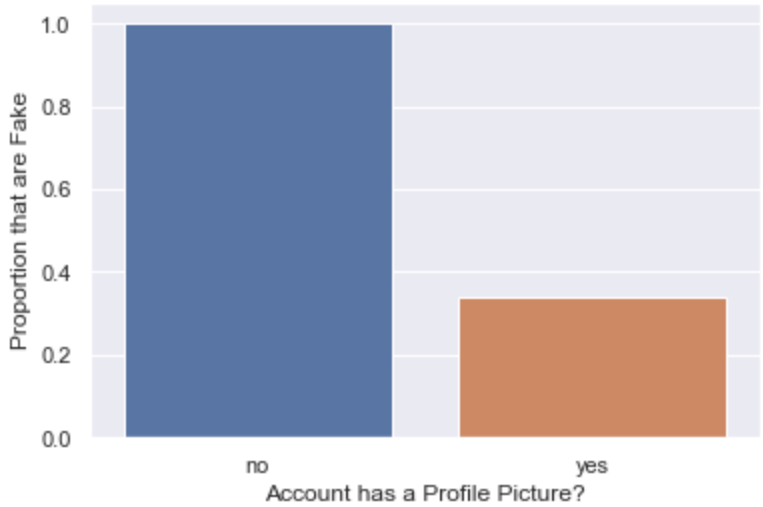
Out[35]:

has_a_profile_pic	account_type	fake	real
no		1.000000	0.000000
yes		0.340659	0.659341

In [36]:
```
sns.barplot(x=temp.index, y='real', data=temp)
plt.ylabel("Proportion that are Real")
plt.xlabel("Account has a Profile Picture?")
plt.show()
```



In [37]:
```
sns.barplot(x=temp.index, y='fake', data=temp)
plt.ylabel("Proportion that are Fake")
plt.xlabel("Account has a Profile Picture?")
plt.show()
```

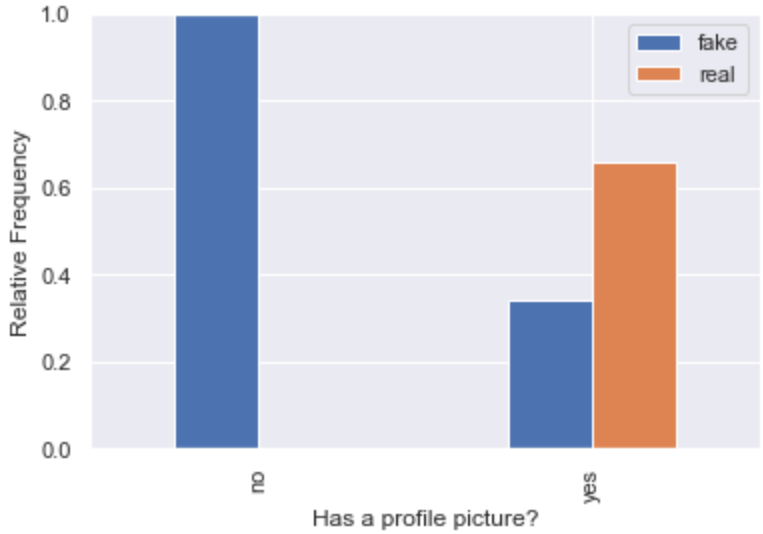


6.2. Side-by-Side Barplots

We could alternatively combine the information from the two above graphs into one graph.

We'll create two groups (having a profile picture or not) and display bars for the proportion of real or fake accounts within each group.

In [38]:
```
temp.plot.bar()
plt.legend(loc='upper right')
plt.xlabel('Has a profile picture?')
plt.ylabel('Relative Frequency')
plt.ylim([0,1])
plt.show()
```



In [39]:
```
temp2 = pd.crosstab(df['account_type'], df['has_a_profile_pic'], normalize='index')
temp2
```

Out[39]:

has_a_profile_pic	no	yes
account_type		
fake	0.483333	0.516667
real	0.000000	1.000000

In [40]:
```
temp2.plot.bar()
plt.legend(loc='upper center')
plt.xlabel('Account type')
plt.ylabel('Relative Frequency')
plt.show()
```


no
yes
Relative Frequency
1.0
0.8
0.6
0.4
0.2
0.0
fake
real
Account type
