Starbucks Capstone Project - Report

Alyson Weidmann

Introduction

Starbucks is a global coffee chain with millions of customers worldwide. The diversity of these customers is reflected not just in their demographics, location, and coffee preferences, but in how they respond to promotions and advertisements. The Starbucks app is a platform to reach the broadest possible customer base and entice them to choose Starbucks as their caffeinator of choice. This is achieved through seamless online ordering, customer service, and special offers. Special offers, such as discounts, BOGO, and bonus points, can boost sales by encouraging a user to purchase a product that they otherwise might not have.

Different promotional offers can appeal to certain customers based on their spending habits, demographic factors, and the medium by which they receive the offer (email, social media, etc.). To help Starbucks better target its offers to the customers most likely to "complete" them (i.e., redeem the offer alongside a transaction), we aim to build a model to predict whether an offer will be successful or not using features derived from data provided by the company.

Here, we go through the process of exploratory data analysis, data cleaning, feature engineering, model building and refinement, and finally, the evaluation of the model and analysis of the results. At the end of this study, we aim to have a better idea of which customers redeem promotional offers, and what kinds of offers are most successful.

Data

There are 3 main sources of data provided by Starbucks, in .json format. We will read them in as dataframes using the pandas library:

In [2]:
```
import pandas as pd
import numpy as np
import math
import json
import matplotlib.pyplot as plt
import seaborn as sns
```

In [3]:
```
# read in the json files
portfolio = pd.read_json('data/portfolio.json', orient='records', lines=True)
profile = pd.read_json('data/profile.json', orient='records', lines=True)
transcript = pd.read_json('data/transcript.json', orient='records', lines=True)
```

The first of the 3 datasets we will explore is the "portfolio" dataset. Portfolio contains metadata about the offers themselves – the types of offers (BOGO, discount, or informational), the reward amount, and what types of channels through which the offer is advertised. Let's view the portfolio dataset:

```
In [4]: portfolio.shape
Out[4]: (10, 6)

In [5]: portfolio
```

Out[5]:

	reward	channels	difficulty	duration	offer_type	id
0	10	[email, mobile, social]	10	7	bogo	ae264e3637204a6fb9bb56bc8210ddfd
1	10	[web, email, mobile, social]	10	5	bogo	4d5c57ea9a6940dd891ad53e9dbe8da0
2	0	[web, email, mobile]	0	4	informational	3f207df678b143eea3cee63160fa8bed
3	5	[web, email, mobile]	5	7	bogo	9b98b8c7a33c4b65b9aebfe6a799e6d9
4	5	[web, email]	20	10	discount	0b1e1539f2cc45b7b9fa7c272da2e1d7
5	3	[web, email, mobile, social]	7	7	discount	2298d6c36e964ae4a3e7e9706d1fb8c2
6	2	[web, email, mobile, social]	10	10	discount	fafdcd668e3743c1bb461111dcafc2a4
7	0	[email, mobile, social]	0	3	informational	5a8bc65990b245e5a138643cd4eb9837
8	5	[web, email, mobile, social]	5	5	bogo	f19421c1d4aa40978ebb69ca19b0e20d
9	2	[web, email, mobile]	10	7	discount	2906b810c7d4411798c6938adc9daaa5

This is a short dataset, as there are only a few distinct types of offers. We can see that the offers vary by type, difficulty, reward amount, duration, and the channels.

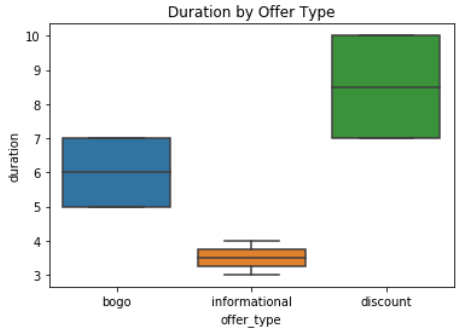
We want to do a little more cleaning of this data before we can do some initial analysis. The portfolio data set contains a single column called "channels", which contains a list of different mediums through which the offer can be viewed. This is best split into multiple columns, with each channel containing a 1 or a 0 to indicate if it was used in the associated offer or not.

```
In [9]: def f(channel_str, row):
            return 1 if channel_str in row else 0

In [10]: channels_list = ['email', 'mobile', 'social', 'web']

         for i in channels_list:
             portfolio[i] = portfolio['channels'].apply(lambda x: f(i, x))

In [11]: portfolio.head()
```

Out[11]:

	reward	channels	difficulty	duration	offer_type	id	email	mobile	social	web
0	10	[email, mobile, social]	10	7	bogo	ae264e3637204a6fb9bb56bc8210ddfd	1	1	1	0
1	10	[web, email, mobile, social]	10	5	bogo	4d5c57ea9a6940dd891ad53e9dbe8da0	1	1	1	1
2	0	[web, email, mobile]	0	4	informational	3f207df678b143eea3cee63160fa8bed	1	1	0	1
3	5	[web, email, mobile]	5	7	bogo	9b98b8c7a33c4b65b9aebfe6a799e6d9	1	1	0	1
4	5	[web, email]	20	10	discount	0b1e1539f2cc45b7b9fa7c272da2e1d7	1	0	0	1

Some initial analysis of this data reveals interesting information about the offers. Here, we can see that discount type offers have the longest duration. This could be meaningful in whether an offer is successful – if the user has time to see and use it before it expires.



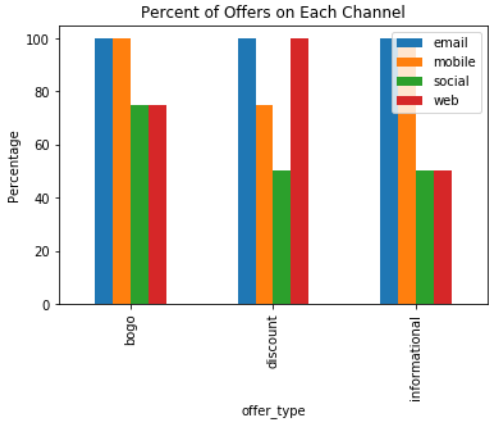
Next, we look at the relationship between offer type and channel. Are some offers featured more prominently than others on different channels? And how might that affect which customers view them?

```
In [13]: offer_counts = portfolio['offer_type'].value_counts().to_dict()

In [14]: channel_groups = portfolio.groupby(['offer_type']).agg({'web':'sum',
                                                  'email':'sum',
                                                  'mobile': 'sum',
                                                  'social':'sum'}).reset_index()

In [15]: channel_groups['counts'] = portfolio['offer_type'].map(offer_counts)

In [16]: channel_groups = channel_groups.assign(web=channel_groups.web / channel_groups.counts * 100,
                                email=channel_groups.email / channel_groups.counts * 100,
                                mobile=channel_groups.mobile / channel_groups.counts * 100,
                                social=channel_groups.social / channel_groups.counts * 100)

In [17]: channel_groups
```

Out[17]:

	offer_type	web	email	mobile	social	counts
0	bogo	75.0	100.0	100.0	75.0	4
1	discount	100.0	100.0	75.0	50.0	4
2	informational	50.0	100.0	100.0	50.0	2

Here we can see the percentage of offers appearing on each channel. This is easier to read in plot form:



BOGO makes use of the most channels for its offers, with 100% of offers appearing on both email and mobile, and 75% appearing on social and web. Both Discount and Informational offers use social media less frequently.

Next is the profile dataset. This contains demographic and other basic information about the users who are registered with the app, such as when they became a member and what their income level is.

In viewing the table, we can see right away that there is some missing data. We will have to do more cleaning here before visualizing this dataset and incorporating it into our model.

```
In [19]: profile.shape
Out[19]: (17000, 5)
```

```
In [20]: profile.head(10)
```

Out[20]:

	gender	age	id	became_member_on	income
0	None	118	68be06ca386d4c31939f3a4f0e3dd783	20170212	NaN
1	F	55	0610b486422d4921ae7d2bf64640c50b	20170715	112000.0
2	None	118	38fe809add3b4fcf9315a9694bb96ff5	20180712	NaN
3	F	75	78afa995795e4d85b5d9ceeca43f5fef	20170509	100000.0
4	None	118	a03223e636434f42ac4c3df47e8bac43	20170804	NaN
5	M	68	e2127556f4f64592b11af22de27a7932	20180426	70000.0
6	None	118	8ec6ce2a7e7949b1bf142def7d0e0586	20170925	NaN
7	None	118	68617ca6246f4fbc85e91a2a49552598	20171002	NaN
8	M	65	389bc3fa690240e798340f5a15918d5c	20180209	53000.0
9	None	118	8974fc5686fe429db53ddde067b88302	20161122	NaN

Let's get an overview of the data and see exactly how much is missing:

```
In [21]: profile.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17000 entries, 0 to 16999
Data columns (total 5 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   gender            14825 non-null  object 
 1   age               17000 non-null  int64  
 2   id                17000 non-null  object 
 3   became_member_on  17000 non-null  int64  
 4   income            14825 non-null  float64
dtypes: float64(1), int64(2), object(2)
memory usage: 664.2+ KB
```

```
In [22]: profile.isnull().sum()
Out[22]: gender              2175
         age                    0
         id                     0
         became_member_on       0
         income              2175
         dtype: int64
```

Looks like gender and income both have the same amount of nulls. We will have to check to see if they are the same rows. Looking at the distribution of the numerical data, we also see an odd outlier – the max age in this dataset is 118:

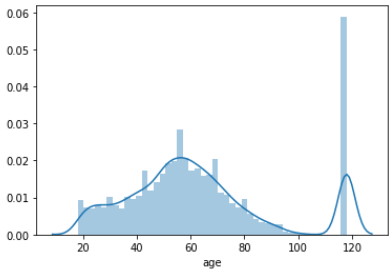
```
In [23]: profile.describe()
```

Out[23]:

	age	became_member_on	income
count	17000.000000	1.700000e+04	14825.000000
mean	62.531412	2.016703e+07	65404.991568
std	26.738580	1.167750e+04	21598.299410
min	18.000000	2.013073e+07	30000.000000
25%	45.000000	2.016053e+07	49000.000000
50%	58.000000	2.017080e+07	64000.000000
75%	73.000000	2.017123e+07	80000.000000
max	118.000000	2.018073e+07	120000.000000

It seems unlikely that 118 year olds are using the Starbucks app. It's not impossible that there are coffee-loving centenarians out there. How many 118-year-olds are in our dataset?

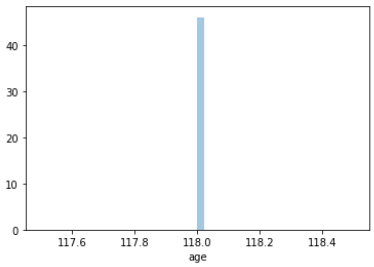
```
In [24]: sns.distplot(profile['age'])
Out[24]: <matplotlib.axes._subplots.AxesSubplot at 0x1a260cb6d0>
```




```
In [25]: profile[profile['age']==118].count()
Out[25]: gender                0
         age                2175
         id                 2175
         became_member_on   2175
         income                0
         dtype: int64
```

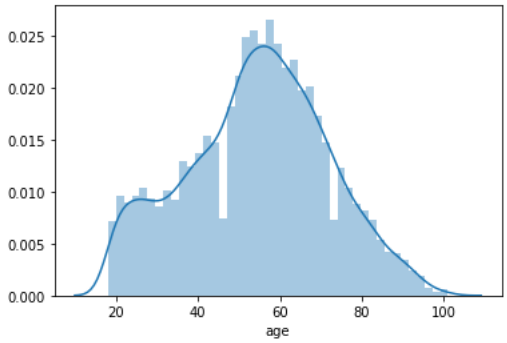
It looks like there are many rows with 118, but 118 appears to be the only major outlier in the age column. The number of rows with age 118 equals the number of rows with nulls for the gender and income columns. If these rows are all the same, we can toss them out.

```
In [26]: bad_data = profile[profile['gender'].isnull()]

In [27]: # All the null gender and income columns are contained in the same rows
         bad_data.info()
         <class 'pandas.core.frame.DataFrame'>
         Int64Index: 2175 entries, 0 to 16994
         Data columns (total 5 columns):
          #   Column            Non-Null Count  Dtype
         ---  ------            --------------  -----
          0   gender            0 non-null      object
          1   age               2175 non-null   int64
          2   id                2175 non-null   object
          3   became_member_on  2175 non-null   int64
          4   income            0 non-null      float64
         dtypes: float64(1), int64(2), object(2)
         memory usage: 102.0+ KB

In [28]: # All the rows with outlier age of 118 are also contained in this data set.
         sns.distplot(bad_data['age'])
         /Users/alysonweidmann/opt/anaconda3/lib/python3.7/site-packages/seaborn/distributions.py:288: UserWarning: Data must have variance to com
         pute a kernel density estimate.
           warnings.warn(msg, UserWarning)
Out[28]: <matplotlib.axes._subplots.AxesSubplot at 0x1a2601e290>
```

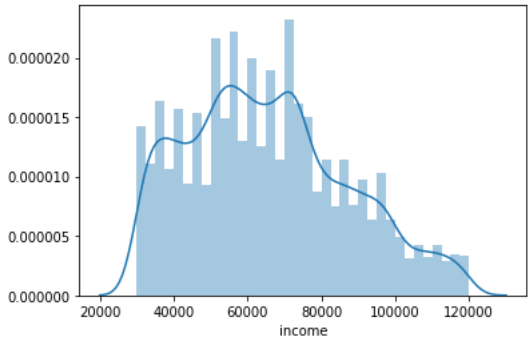


As there are 2175 rows with unusable data (i.e., unreasonable age, unknown gender and income), we can assume this data can be safely discarded from the dataset. Now, the distributions of age and income look more reasonable.

```
In [32]: sns.distplot(profile['age'])
Out[32]: <matplotlib.axes._subplots.AxesSubplot at 0x1a25fe1e90>
```

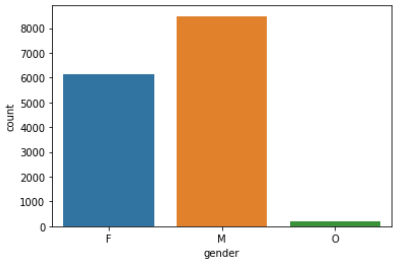



```
In [33]: sns.distplot(profile['income'])
Out[33]: <matplotlib.axes._subplots.AxesSubplot at 0x1a25f94890>
```

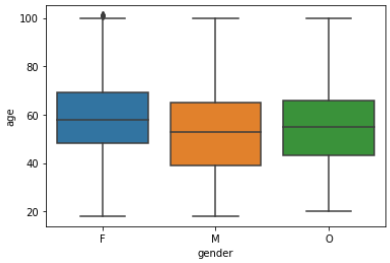


The majority of users are male, and the median age of users appears to be between 50 and 60 years old.

```
In [34]: sns.countplot(profile['gender'])
Out[34]: <matplotlib.axes._subplots.AxesSubplot at 0x1a25f15cd0>
```




```
In [35]: sns.boxplot(profile['gender'], profile['age'])
Out[35]: <matplotlib.axes._subplots.AxesSubplot at 0x1a25e9cdd0>
```



As a final piece of cleaning for the profile data set, we need to adjust the became_member_on field from scientific notation to datetime. Currently, the dates are in a “datekey” format (YYYYMMDD). If we use the pd.to_datetime method, it will not recognize the proper dates from this format. So we must use the datetime library to convert the datekey to a string, a string to a date.

```
In [39]: from datetime import datetime
```

```
In [40]: profile['became_member_on'] = profile['became_member_on'].apply(lambda x:
                                                                        datetime.strptime(str(x), '%Y%m%d'
                                                                                          ).strftime('%m/%d/%Y')
                                                                        )
```

```
In [41]: profile.head(10)
```

Out[41]:

	gender	age	id	became_member_on	income
1	F	55	0610b486422d4921ae7d2bf64640c50b	07/15/2017	112000.0
3	F	75	78afa995795e4d85b5d9ceeca43f5fef	05/09/2017	100000.0
5	M	68	e2127556f4f64592b11af22de27a7932	04/26/2018	70000.0
8	M	65	389bc3fa690240e798340f5a15918d5c	02/09/2018	53000.0
12	M	58	2eeac8d8feae4a8cad5a6af0499a211d	11/11/2017	51000.0
13	F	61	aa4862eba776480b8bb9c68455b8c2e1	09/11/2017	57000.0
14	M	26	e12aeaf2d47d42479ea1c4ac3d8286c6	02/13/2014	46000.0
15	F	62	31dda685af34476cad5bc968bdb01c53	02/11/2016	71000.0
16	M	49	62cf5e10845442329191fc246e7bcea3	11/13/2014	52000.0
18	M	57	6445de3b47274c759400cd68131d91b4	12/31/2017	42000.0

Let's engineer some additional time-based features based on when each customer became a Starbucks member. There may be some seasonality here that could be useful down the line in our model.

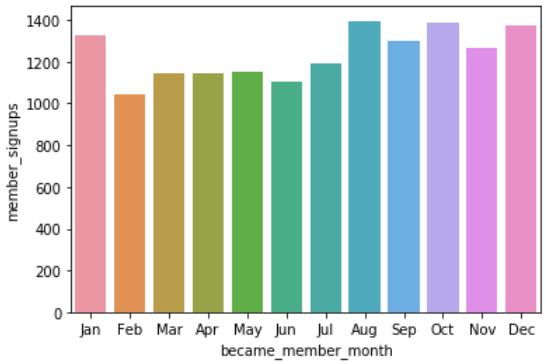
```
In [42]: profile['became_member_month'] = profile['became_member_on'].apply(lambda x:
                                                                           datetime.strptime(x, '%m/%d/%Y').strftime('%b')
                                                                          )

         profile['became_member_year'] = profile['became_member_on'].apply(lambda x:
                                                                           datetime.strptime(x, '%m/%d/%Y').strftime('%Y')
                                                                          )
```

```
In [43]: profile.head(10)
```

Out[43]:

	gender	age	id	became_member_on	income	became_member_month	became_member_year
1	F	55	0610b486422d4921ae7d2bf64640c50b	07/15/2017	112000.0	Jul	2017
3	F	75	78afa995795e4d85b5d9ceeca43f5fef	05/09/2017	100000.0	May	2017
5	M	68	e2127556f4f64592b11af22de27a7932	04/26/2018	70000.0	Apr	2018
8	M	65	389bc3fa690240e798340f5a15918d5c	02/09/2018	53000.0	Feb	2018
12	M	58	2eeac8d8feae4a8cad5a6af0499a211d	11/11/2017	51000.0	Nov	2017
13	F	61	aa4862eba776480b8bb9c68455b8c2e1	09/11/2017	57000.0	Sep	2017
14	M	26	e12aeaf2d47d42479ea1c4ac3d8286c6	02/13/2014	46000.0	Feb	2014
15	F	62	31dda685af34476cad5bc968bdb01c53	02/11/2016	71000.0	Feb	2016
16	M	49	62cf5e10845442329191fc246e7bcea3	11/13/2014	52000.0	Nov	2014
18	M	57	6445de3b47274c759400cd68131d91b4	12/31/2017	42000.0	Dec	2017

```
In [44]: signups = profile.groupby(['became_member_month']).size().to_frame('member_signups').reset_index()

In [45]: month_order = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
         sns.barplot(data=signups, x='became_member_month', y='member_signups', order=month_order)

Out[45]: <matplotlib.axes._subplots.AxesSubplot at 0x1a24a76450>
```



It looks like membership signups pick up in late summer/fall and are higher throughout the winter.

Finally, we explore the transcript dataset. This contains information about the actual transactions and offer interactions that occurred by users.

```
In [46]: transcript.shape

Out[46]: (306534, 4)

In [47]: transcript.head(10)
```

Out[47]:

	person	event	value	time
0	78afa995795e4d85b5d9ceeca43f5fef	offer received	{'offer id': '9b98b8c7a33c4b65b9aebfe6a799e6d9'}	0
1	a03223e636434f42ac4c3df47e8bac43	offer received	{'offer id': '0b1e1539f2cc45b7b9fa7c272da2e1d7'}	0
2	e2127556f4f64592b11af22de27a7932	offer received	{'offer id': '2906b810c7d4411798c6938adc9daaa5'}	0
3	8ec6ce2a7e7949b1bf142def7d0e0586	offer received	{'offer id': 'fafdcd668e3743c1bb461111dcafc2a4'}	0
4	68617ca6246f4fbc85e91a2a49552598	offer received	{'offer id': '4d5c57ea9a6940dd891ad53e9dbe8da0'}	0
5	389bc3fa690240e798340f5a15918d5c	offer received	{'offer id': 'f19421c1d4aa40978ebb69ca19b0e20d'}	0
6	c4863c7985cf408faee930f111475da3	offer received	{'offer id': '2298d6c36e964ae4a3e7e9706d1fb8c2'}	0
7	2eeac8d8feae4a8cad5a6af0499a211d	offer received	{'offer id': '3f207df678b143eea3cee63160fa8bed'}	0
8	aa4862eba776480b8bb9c68455b8c2e1	offer received	{'offer id': '0b1e1539f2cc45b7b9fa7c272da2e1d7'}	0
9	31dda685af34476cad5bc968bdb01c53	offer received	{'offer id': '0b1e1539f2cc45b7b9fa7c272da2e1d7'}	0

Viewing the data, we can see a couple of things right away. The ‘person’ column will allow us to join this dataframe with our profile dataframe. The ‘value’ column contains data about the offer_id, which will allow us to join in the portfolio dataframe as well, but we have to process that column first. We can also see what events occurred, and the relative time at which they occurred.

```
In [48]: transcript.info()
```

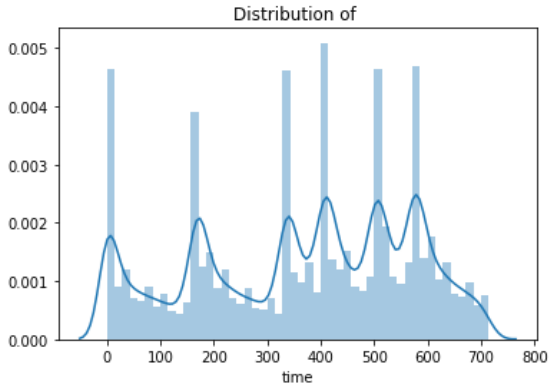
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 306534 entries, 0 to 306533
Data columns (total 4 columns):
 #   Column  Non-Null Count   Dtype 
---  ------  --------------   ----- 
 0   person  306534 non-null  object
 1   event   306534 non-null  object
 2   value   306534 non-null  object
 3   time    306534 non-null  int64 
dtypes: int64(1), object(3)
memory usage: 9.4+ MB
```

```
In [49]: transcript.isnull().sum()
```

```
Out[49]: person    0
         event     0
         value     0
         time      0
         dtype: int64
```

There are no nulls in the data set, so we don't have to worry about cleaning them.

There is only one numerical feature in the transcript data set, 'time'. Let's check the distribution of values to see if there are any outliers.



No major outliers, but there is a hint of periodicity in the ‘time’ data. This may be useful later on.

We want to do some initial cleaning of this data set before joining it to the other two for feature engineering. Namely, the "value" column contains a dictionary that contains information we want to link the transcript data to our other data sources. We want to take the dictionary values and convert them to their own columns for easier cleaning.

First, let’s split the dictionary values into separate columns using the json_normalize method in pandas. This will split dictionary items into columns for each unique key, with their values being converted into column values. Viewing the resulting temporary dataframe, we can see that the ‘values’ column contained 4 unique keys: amount, reward, and two kinds of offer ID.

In [52]:
```
# view the 'value' dataset
transcript['value']
```

Out[52]:
```
0         {'offer id': '9b98b8c7a33c4b65b9aebfe6a799e6d9'}
1         {'offer id': '0b1e1539f2cc45b7b9fa7c272da2e1d7'}
2         {'offer id': '2906b810c7d4411798c6938adc9daaa5'}
3         {'offer id': 'fafdcd668e3743c1bb461111dcafc2a4'}
4         {'offer id': '4d5c57ea9a6940dd891ad53e9dbe8da0'}
                              ...
306529                    {'amount': 1.5899999999999999}
306530                                {'amount': 9.53}
306531                                {'amount': 3.61}
306532                    {'amount': 3.5300000000000002}
306533                                {'amount': 4.05}
Name: value, Length: 306534, dtype: object
```

In [53]:
```
df = pd.json_normalize(transcript['value'])
```

In [54]:
```
df.head()
```

Out[54]:

	offer id	amount	offer_id	reward
0	9b98b8c7a33c4b65b9aebfe6a799e6d9	NaN	NaN	NaN
1	0b1e1539f2cc45b7b9fa7c272da2e1d7	NaN	NaN	NaN
2	2906b810c7d4411798c6938adc9daaa5	NaN	NaN	NaN
3	fafdcd668e3743c1bb461111dcafc2a4	NaN	NaN	NaN
4	4d5c57ea9a6940dd891ad53e9dbe8da0	NaN	NaN	NaN

It is likely that the two offer ID columns, ‘offer id’ and ‘offer_id’, refer to the same thing. We can combine these into a single column and see if that takes care of some of the nulls.

In [55]:
```
df['offer_id_new'] = np.where(df['offer id'].isnull() \
                              & df['offer_id'].notnull(), df['offer_id'],df['offer id'])
```

In [56]:
```
# drop old id columns and rename the clean one
df.drop(['offer id', 'offer_id'], axis=1, inplace=True)
df.rename(columns={'offer_id_new': 'offer_id'}, inplace=True)
```

In [57]:
```
# merge the new value df with the original transcript data
transcript = transcript.merge(df, how='inner', left_index=True, right_index=True)
```

In [58]:
```
transcript.drop(columns='value', inplace=True)
```

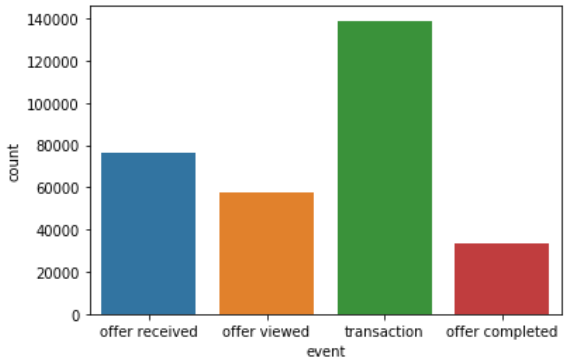
In [59]:
```
# replace null values with 0
transcript['amount'].fillna(0, inplace=True)
transcript['reward'].fillna(0, inplace=True)
```

In [60]:
```
# check data is clean
transcript.head()
```

Out[60]:

	person	event	time	amount	reward	offer_id
0	78afa995795e4d85b5d9ceeca43f5fef	offer received	0	0.0	0.0	9b98b8c7a33c4b65b9aebfe6a799e6d9
1	a03223e636434f42ac4c3df47e8bac43	offer received	0	0.0	0.0	0b1e1539f2cc45b7b9fa7c272da2e1d7
2	e2127556f4f64592b11af22de27a7932	offer received	0	0.0	0.0	2906b810c7d4411798c6938adc9daaa5
3	8ec6ce2a7e7949b1bf142def7d0e0586	offer received	0	0.0	0.0	fafdcd668e3743c1bb461111dcafc2a4
4	68617ca6246f4fbc85e91a2a49552598	offer received	0	0.0	0.0	4d5c57ea9a6940dd891ad53e9dbe8da0

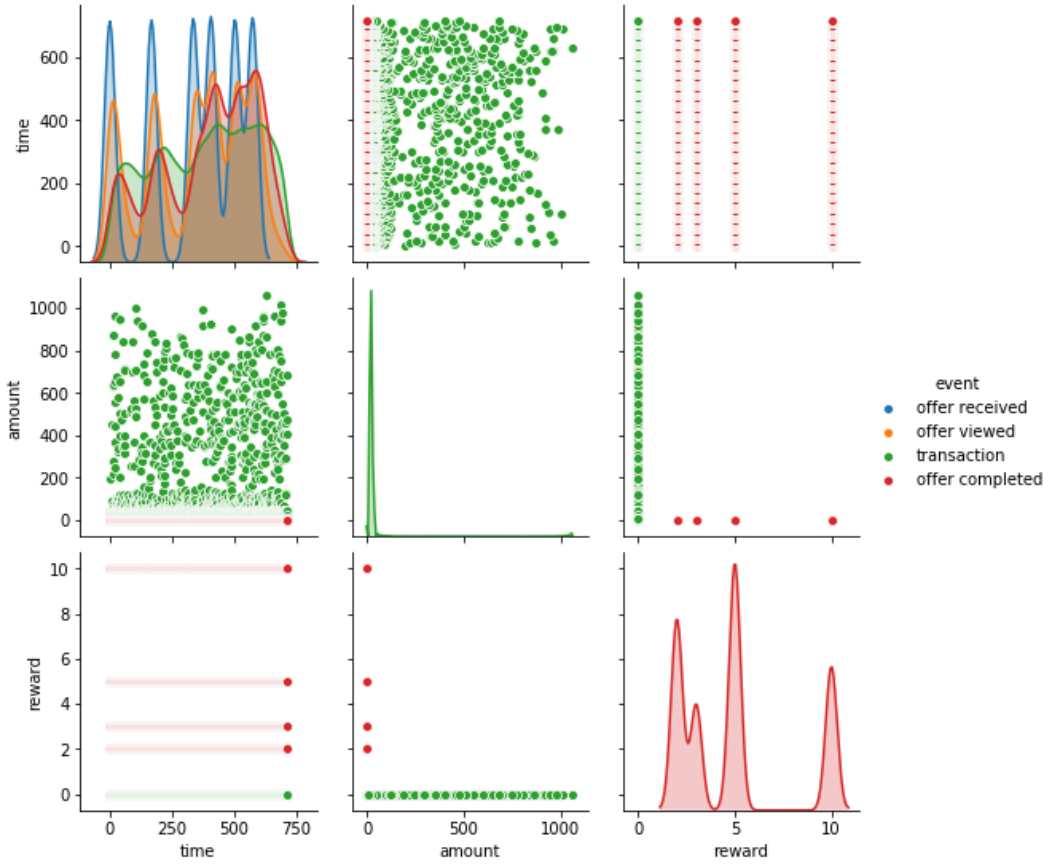
Now we can do a bit more visualization of the transcript data set. A simple countplot shows that the majority of the events are transactions.



We use a pairplot to get a better sense of some of the relationships that may occur between various fields. Here, we can see that the majority of the 'amount' values correspond to transactions and the 'reward' values correspond to offers completed

```
In [62]: import warnings
         warnings.filterwarnings('ignore')
         sns.pairplot(transcript, hue='event')
```

```
Out[62]: <seaborn.axisgrid.PairGrid at 0x1a26237610>
```



Now we will join the profile, portfolio, and transcript datasets to explore in more detail the relationships between these datasets.

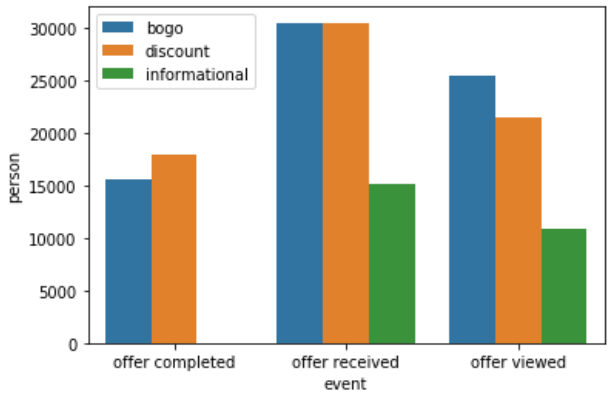
The transcript dataset contains both the customer ID ('person') and the offer ID ('offer ID'). We will use the person field to join transcript to profile, and the offer ID field to join transcript to portfolio, creating a new dataframe called 'data.' First, we rename columns in profile and portfolio to match the corresponding field names in transcript, for easier joining. We also take care of the duplicate 'reward' columns in transcript and portfolio by renaming them, since they refer to slightly different rewards (offered vs received).

```
In [63]: # rename ID columns in profile and portfolio for easier joining
         profile.rename(columns={'id':'person'}, inplace=True)
         portfolio.rename(columns={'id':'offer_id'}, inplace=True)

In [64]: transcript.rename(columns={'reward':'received_reward'}, inplace=True)
         portfolio.rename(columns={'reward':'offer_reward'}, inplace=True)

In [65]: data = pd.merge(transcript, profile, how='left', on='person')
         data = pd.merge(data, portfolio, how='left', on='offer_id')
```

Exploratory Data Analysis

Now that we have our merged data set, let's do some additional EDA. We have explored each of our data sources separately, but we haven't yet seen how all of the data comes together to reveal potentially significant relationships between customer and offer.

We start by grouping and aggregating the data to identify simple relationships, like how many offers were viewed or completed by type?



Discount offers do a little better at converting than BOGO. Makes intuitive sense, as a person might not necessarily need 2 coffees if they weren't going to even buy one in the first place, but a person might buy one coffee for a discount.

```
In [69]: data.groupby(['offer_type']).agg({'received_reward':['sum', 'mean', 'min', 'max']})
```

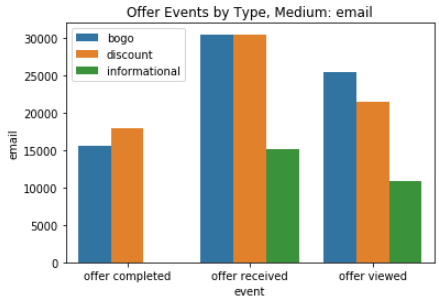
Out[69]:

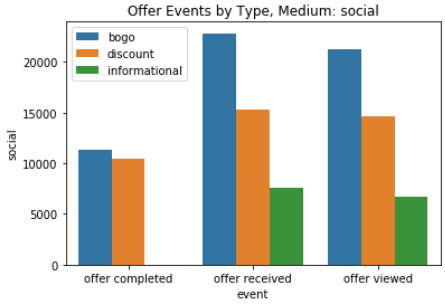
	received_reward			
	sum	mean	min	max
offer_type				
bogo	113440.0	1.583981	0.0	10.0
discount	51236.0	0.733011	0.0	5.0
informational	0.0	0.000000	0.0	0.0

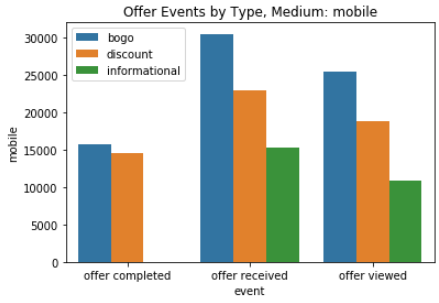
However, BOGO rewards are worth more money generally, although they convert less frequently.

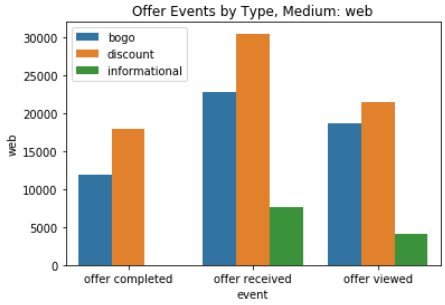
```
In [71]: media_groups = data.groupby(['offer_type','event']).agg({'web':'sum',
                                                                  'mobile':'sum',
                                                                  'social':'sum',
                                                                  'email':'sum'}).reset_index()
```

```
In [73]: for i in channels_list:
             sns.barplot(data=media_groups, x='event', y=i, hue='of
             plt.legend(loc='upper left')
             plt.title(f'Offer Events by Type, Medium: {i}')
             plt.show()
```









We see that discount offers are more popular on the web or over email, while BOGO offers are more prominent on social media and mobile. Given this split, perhaps there are some age differences in offer types, with potentially younger users preferring mobile and social media, whereas older users may use email and web more often.

In [74]:
```
customer_groups = data.groupby(['offer_type','event']).agg({'age':'median' ,
                                                            'income':'median'
                                                           }).reset_index()
```

In [75]:
```
customer_groups
```

Out[75]:

	offer_type	event	age	income
0	bogo	offer completed	57.0	70000.0
1	bogo	offer received	55.0	64000.0
2	bogo	offer viewed	55.0	64000.0
3	discount	offer completed	56.0	68000.0
4	discount	offer received	55.0	64000.0
5	discount	offer viewed	56.0	65000.0
6	informational	offer received	55.0	63000.0
7	informational	offer viewed	55.0	64000.0

In reality, there are not a ton of differences in offer completions by type when broken down by age and even income. In general, higher incomes translate to more completions and a slightly higher likelihood of viewing an offer that is received.

Data Preprocessing and Feature Engineering

Now that we've explored the data and it is mostly clean, we begin to preprocess our data for modeling.

Our data can generally be divided into 3 groups:

- People who receive and view an offer and effectively complete it ('offer completed')
- People who receive/view an offer but do not complete it ('offer received', 'offer viewed')
- People who complete a transaction irrespective of whether they received or viewed an offer ('transaction')

To begin, we should further clean our data set to separate out customers who completed a purchase that was not tied to a promotional offer, as they can inflate our dataset with false negatives -- i.e., there is no way to know if these users would have completed an offer if one were received, and so we have incomplete information about these samples with respect to the target variable ('offer completed')

By sorting the data by person and time, we can see that each stage in the offer process exists as its own row. There is a separate row for offer received, viewed, whether a transaction was completed after viewing an offer, and whether the offer itself was completed at the same time. Currently, the only way to tie these events to a single sequence of events is to look at the 'time' column along with the 'offer_id' column:

```
In [79]: # sort data by person, in chronological order of events
         data.sort_values(['person', 'time'], inplace=True)

In [80]: # get the event immediately preceding the current one, grouped by user
         person_groups = data.groupby('person')['event'].shift(1).to_frame()

In [81]: data['prior_event'] = person_groups['event']

In [82]: #view the data to make sure it is correct
         data[['person','event', 'prior_event', 'time']].head(50)
```

Out[82]:

	person	event	prior_event	time
55972	0009655768c64bdeb2e877511632db8f	offer received	NaN	168
77705	0009655768c64bdeb2e877511632db8f	offer viewed	offer received	192
89291	0009655768c64bdeb2e877511632db8f	transaction	offer viewed	228
113605	0009655768c64bdeb2e877511632db8f	offer received	transaction	336
139992	0009655768c64bdeb2e877511632db8f	offer viewed	offer received	372
153401	0009655768c64bdeb2e877511632db8f	offer received	offer viewed	408
168413	0009655768c64bdeb2e877511632db8f	transaction	offer received	414

We created a new helper column to view the event that immediately preceded the event of the current row. So if a transaction was associated with an offer, the 'prior_event' would be 'offer viewed' or 'offer received', while the event would be 'transaction'. If a transaction occurred independent of an offer, the prior event would be 'transaction' or 'offer completed' (from the previous offer) or NaN (if transaction is the first event recorded for a user). Now we can use this information to filter out transactions that occur without offers.

We likely only need one row per unique person/offer combo; otherwise there will be a lot of duplicate information in our features that could bias the model in some way. We ultimately want to filter our data set to identify the terminal end state of the offers -- if an offer was completed, we want to keep those rows as our target variable would == 1. Additionally, we will count informational offers that result in transactions as positives as well, as there is no 'offer' to complete but it would appear that the informational offer is still successful. For non-completed offers, we'd want to take the row with the maximum time value to indicate the last event associated with that offer for a user.

```
In [90]: data['max_time'] = data.groupby(['person', 'offer_id'])['time'].transform('max')

In [91]: data['target'] = np.where(data['event']=='offer completed', 1,0)

In [92]: data.loc[(data.event=='transaction') & (data.offer_type=='informational'), 'target'] = 1
```

We are getting closer to a state where we can begin modeling but are not quite there yet. There are some inconsistencies in the data, where occasionally an offer is completed but its 'terminal' state (max time) is the viewing of that same offer or the transaction associated with the offer. So there is still some cleaning to do around our positive cases, although we seem to have successfully removed the duplicate rows for our negative cases.

In preparing our data for modeling, we must also address the issue of data leakage; features that provide information or 'hints' about the target variable will compromise our model. We must remove features that reveal what the target is before we can train our model. We will also remove any

extraneous helper columns we created that do not add any additional predictive value. Below is a function that does all of this initial data preparation for training the model:

In [197]:
```
# roll all the above into a single function to reuse
def prep_data(data):
    X = data.copy()
    X = X[(X['target']==1) | (X['time']==X['max_time'])]

    remove_dupes = X.groupby(['person', 'offer_id'])['target'].idxmax().to_frame('idx').set_index('idx')
    X = X[X.index.isin(remove_dupes.index)]
    X.drop(columns=['event',
                    'amount',
                    'received_reward',
                    'prior_event',
                    'max_time',
                    'channels',
                    'became_member_on'], inplace=True, errors='ignore')

    return X
```

In checking over our prepared dataset X, we find that we have over 8000 rows with customer data missing. We already had a lot of missing age and gender data, and likely when we did the left join of the profile dataframe to the transcript dataframe, it is possible that the transcript contained customers we did not have data on.

We have 2 options here. We can drop the null values from our data set, or we can try and impute. Seeing as there are 6 columns we would have to perform imputation on, there is a higher likelihood that this will introduce too much error into our training data than if we were to only impute on a single column.

In [142]:
```
X['target'].value_counts()
```

Out[142]:
```
1    38791
0    26230
Name: target, dtype: int64
```

On the other hand, most of these nulls are target == 0, so dropping them could introduce some bias into our dataset. It's possible these nulls are not random and could have some influence over the target variable.

First, let's try filling in the missing values with either the most frequent value (for categorical data) or the median value (for continuous data). If the model does not perform well, we will try a version with the nulls dropped.

In [105]:
```
# modes for missing categorical data
gender_mode = X['gender'].mode()[0]
became_member_month_mode = X['became_member_month'].mode()[0]
became_member_year_mode = X['became_member_year'].mode()[0]

# medians for missing continuous data
age_median = X['age'].median()
income_median = X['income'].median()
```

In [180]:
```
X['gender'].fillna(gender_mode, inplace=True)
X['became_member_month'].fillna(became_member_month_mode, inplace=True)
X['became_member_year'].fillna(became_member_year_mode, inplace=True)
X['age'].fillna(age_median, inplace=True)
X['income'].fillna(income_median, inplace=True)
```

After some dummy encoding of categorical variables, and splitting off our 'target' column into its own vector y, we are finally ready to begin experimenting with a model.

Benchmark model

The simplest place to start would be a binary classifier to predict whether an offer will "convert" into a completion or a transaction. A "successful" offer, resulting in a sale/completion, will be class 1 while all other offers will be class 0.

We will first conform that this problem is solvable as a binary classification problem. We will fit a basic logistic regression and see how the model performs.

```
In [305]: lr = LogisticRegression()
          lr.fit(X_train, y_train)

          pred = lr.predict(X_test)
          print(confusion_matrix(y_test, pred))
          print(classification_report(y_test, pred))
```

```
[[3097 4703]
 [2095 9612]]
              precision    recall  f1-score   support

           0       0.60      0.40      0.48      7800
           1       0.67      0.82      0.74     11707

    accuracy                           0.65     19507
   macro avg       0.63      0.61      0.61     19507
weighted avg       0.64      0.65      0.63     19507
```

```
In [116]: print('Log loss: ',log_loss(y_test, pred))
          print('Accuracy score: ', accuracy_score(y_test, pred))
```

```
Log loss:  10.699794505100353
Accuracy score:  0.690213769416107
```

We can see that the model is trainable and that a binary classification approach is at least practical, even if this initial benchmark model does not perform great. Specifically, the model does not do a great job at predicting negatives (i.e., which samples will NOT result in an offer completion), although it does a pretty good job at predicting positives.

Next, we can try several classification approaches, including Random Forest and Decision Tree Classifier, along with another Logistic Regression where we tune the hyperparameters. To expedite the search for the best hyperparameters, we will use a Grid Search with cross-validation, which will test several sets of parameters on the estimator and return the best one.

We will evaluate these models' performance using the Receiver Operator Characteristic area-under-the-curve, which plots the rate of the true positives to the rate of false positives predicted by the classifier. We will also look at several metrics, such as the precision and recall (f1 score), log-loss or cross entropy, and since this data set is not overly imbalanced, accuracy.

Below are functions that will efficiently train these models using grid search with cross-validation, select the model with the best-performing parameters, and evaluate the model accoring to the chosen metrics.

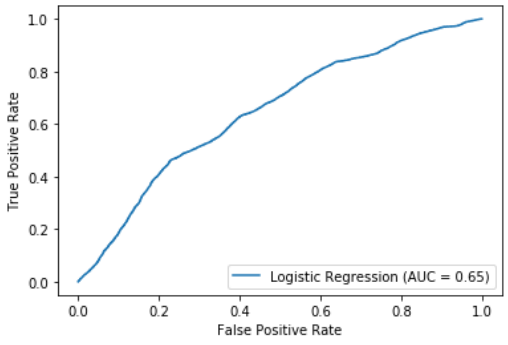
In [118]:
```
def train(estimator, X, y,  params=None, cv=5, scoring_metric='roc_auc'):
    """Fit a model using grid search with cross validation.
        Returns the model and parameters with the best performance.
    """
    model = GridSearchCV(estimator,
                         param_grid=params,
                         scoring=scoring_metric)
    model.fit(X, y)

    print(model.best_params_)
    return model.best_estimator_
```

In [119]:
```
def evaluate(model, X, y, name=None):
    """Evaluates model performance and prints a report of metrics and plots"""

    pred = model.predict(X)
    print('Confusion Matrix: \n', confusion_matrix(y, pred))
    print(classification_report(y, pred))

    print('Accuracy score: ',accuracy_score(y, pred))
    print('Log Loss: ', log_loss(y, pred))

    plot_roc_curve(model, X, y, name=name)
```

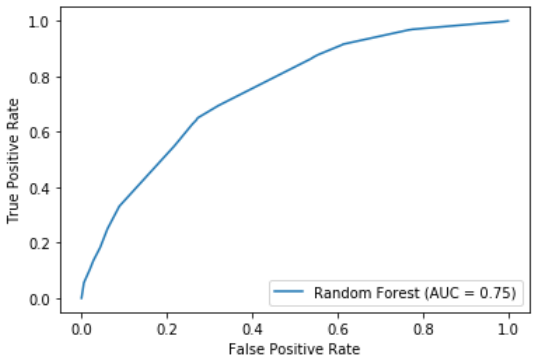
Evaluation

After performing Grid Search on our 3 algorithms and selecting the best of each model, we must now compare these models to the benchmark and to one another. Using our evaluate() method, we view the confusion matrix to see how each model classified the samples in the test set, the classification report to get the precision, recall, and F1 score, the overall accuracy, and the log loss. We also plot the ROC curve and view the AUC metric. Taken together, all of these metrics will give us a sense of how well each model performs.

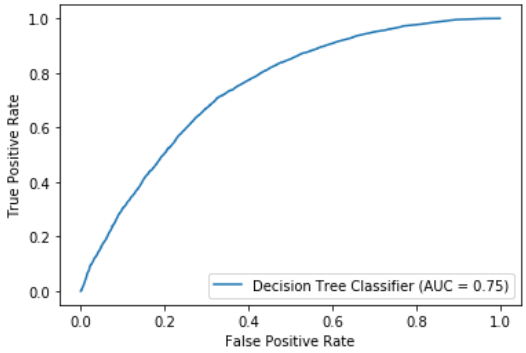
```
Logistic Regression Model Performance:
Confusion Matrix:
 [[4756 3044]
 [4525 7182]]
              precision    recall  f1-score   support

           0       0.51      0.61      0.56      7800
           1       0.70      0.61      0.65     11707

    accuracy                           0.61     19507
   macro avg       0.61      0.61      0.61     19507
weighted avg       0.63      0.61      0.62     19507

Accuracy score:  0.6119854411236992
Log Loss:  13.401672861416746
```



```
Random Forest Model Performance:
Confusion Matrix:
 [[5678 2122]
 [4103 7604]]
              precision    recall  f1-score   support

           0       0.58      0.73      0.65      7800
           1       0.78      0.65      0.71     11707

    accuracy                           0.68     19507
   macro avg       0.68      0.69      0.68     19507
weighted avg       0.70      0.68      0.68     19507

Accuracy score:  0.6808837853078382
Log Loss:  11.021970564621206
```



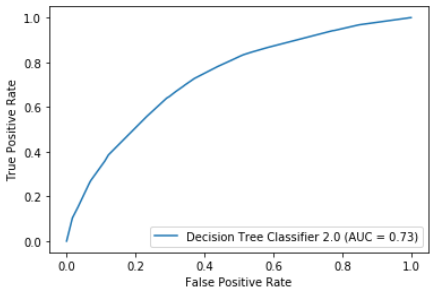

```
Decision Tree Classifier Model Performance:
Confusion Matrix:
 [[5283 2517]
 [3484 8223]]
              precision    recall  f1-score   support

           0       0.60      0.68      0.64      7800
           1       0.77      0.70      0.73     11707

    accuracy                           0.69     19507
   macro avg       0.68      0.69      0.69     19507
weighted avg       0.70      0.69      0.69     19507

Accuracy score:  0.6923668426718614
Log Loss:  10.625376005156651
```



It is clear that our Decision Tree Classifier performs the best, in terms of precision/recall/f1 score, accuracy, and log-loss, and is equal to the RandomForestClassifier in terms of ROC AUC. It also performs better than the benchmark Logistic Regression we ran earlier. We might try to improve performance by engineering some additional features, but first, let's check and see if removing the nulls from the original data set results in a better model than when we imputed the most frequent or median values.

After creating a new version of the feature set X with the null rows dropped, we fit another Decision Tree Classifier to compare it to the previous one.

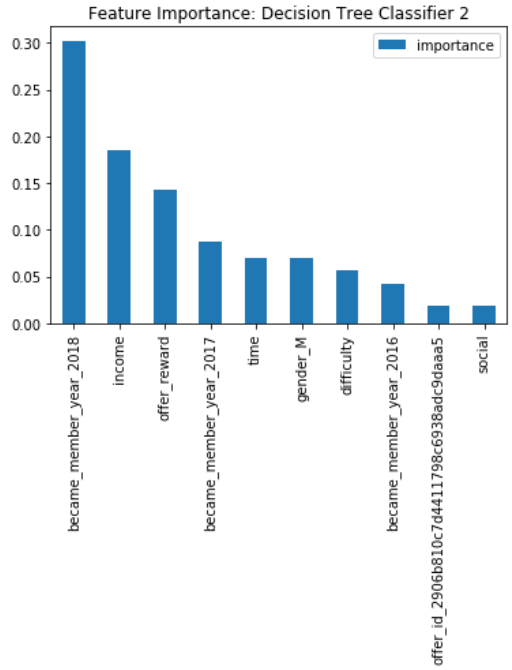
```
In [156]: dt2 = DecisionTreeClassifier(class_weight='balanced', max_depth= 5)
          dt2.fit(X_train2, y_train2)

Out[156]: DecisionTreeClassifier(ccp_alpha=0.0, class_weight='balanced', criterion='gini',
                                 max_depth=5, max_features=None, max_leaf_nodes=None,
                                 min_impurity_decrease=0.0, min_impurity_split=None,
                                 min_samples_leaf=1, min_samples_split=2,
                                 min_weight_fraction_leaf=0.0, presort='deprecated',
                                 random_state=None, splitter='best')

In [157]: print('Decision Tree Classifier performance, nulls removed:')
          evaluate(dt2, X_test2, y_test2, name='Decision Tree Classifier 2.0')

          Decision Tree Classifier performance, nulls removed:
          Confusion Matrix:
           [[3914 2041]
           [3346 7725]]
                        precision    recall  f1-score   support

                     0       0.54      0.66      0.59      5955
                     1       0.79      0.70      0.74     11071

              accuracy                           0.68     17026
             macro avg       0.67      0.68      0.67     17026
          weighted avg       0.70      0.68      0.69     17026

          Accuracy score:  0.6836015505697169
          Log Loss:  10.928111148698395
```



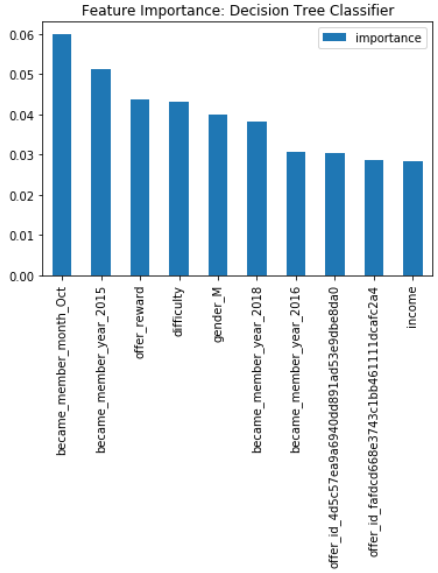
We can see that removing the null values results in a slightly worse performance for the Decision Tree Classifier. Another option is to drop these features altogether. First, we should see how important they are to the decision tree classifier models. To do this, we can use the feature_importance_ attribute of the DecisionTreeClassifer object in sklearn.

```
In [194]: pd.DataFrame(dt2.feature_importances_,
                       index = X2.columns,
                       columns=['importance']).sort_values('importance',ascending=False) \
                       .nlargest(10, 'importance').plot.bar(title='Feature Importance: Decision Tree Classifier 2')
```

```
Out[194]: <matplotlib.axes._subplots.AxesSubplot at 0x1a2d9b1610>
```



And comparing the original decision tree model:



It looks like income and how long users have been a member are important features, but age and gender less so. We tried a version where we drop the entire columns that have all these null values, but it performed the worst of all!

Model Refinement

We will continue with our original processed dataset, X, where we imputed mode values for gender, year, etc. and median values for missing age and income. To see if we can further improve model performance, we will engineer some additional features that could provide information about what kinds of offers appeal to which users.

To start, perhaps there is a correlation between how many offers a user receives and the likelihood they will redeem one.

In [228]:
```
# How many offers does a person receive?
cumulative_offers = data[data['event']=='offer received'].groupby(['person', 'offer_id']).count()['event'] \
.groupby(level=0).cumsum().reset_index()
```

In [229]:
```
cumulative_offers.rename(columns={'event':'cumulative_offers'}, inplace=True)
```

In [265]:
```
# need to recreate data set X because it is currently one-hot encoded
X = prep_data(data)
```

In [266]:
```
X['gender'].fillna(gender_mode, inplace=True)
X['became_member_month'].fillna(became_member_month_mode, inplace=True)
X['became_member_year'].fillna(became_member_year_mode, inplace=True)
X['age'].fillna(age_median, inplace=True)
X['income'].fillna(income_median, inplace=True)
```

In [230]:
```
cumulative_offers
```

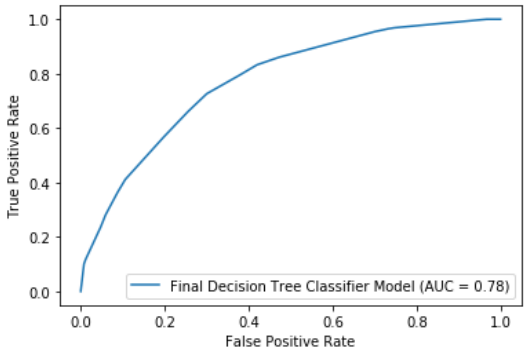
Out[230]:

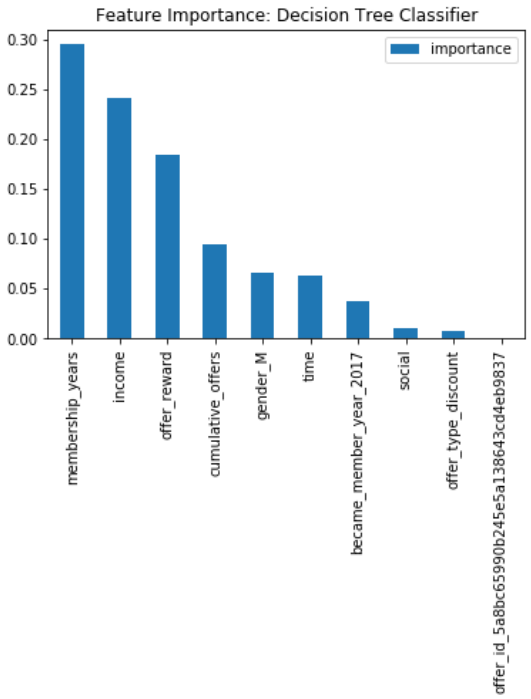
	person	offer_id	cumulative_offers
0	0009655768c64bdeb2e877511632db8f	2906b810c7d4411798c6938adc9daaa5	1
1	0009655768c64bdeb2e877511632db8f	3f207df678b143eea3cee63160fa8bed	2
2	0009655768c64bdeb2e877511632db8f	5a8bc65990b245e5a138643cd4eb9837	3
3	0009655768c64bdeb2e877511632db8f	f19421c1d4aa40978ebb69ca19b0e20d	4
4	0009655768c64bdeb2e877511632db8f	fafdcd668e3743c1bb461111dcafc2a4	5
...	...	...	...

The 'became_member_year' columns also seem to be important features. Perhaps there is additional predictive information in how long a user has been a member. We can engineer a new feature, "membership_years", to quantify membership time more linearly than the categorical variable. Looking at the max date, it looks like the most recent year is 2018, so we count backwards from there.

Now we have two new features that should provide more information about what types of offers are most successful, engineered from looking at the feature importances of previous model experiments. Now we are ready to train our final model and assess performance. The results of the final trained model are summarized below:

```
Confusion Matrix:
 [[5458 2342]
 [3203 8504]]
              precision    recall  f1-score   support

           0       0.63      0.70      0.66      7800
           1       0.78      0.73      0.75     11707

    accuracy                           0.72     19507
   macro avg       0.71      0.71      0.71     19507
weighted avg       0.72      0.72      0.72     19507

Accuracy score:  0.71574306659148
Log Loss:  9.817982660940238
```





We can see that this model is the best one yet! Additionally, the new features we created in the refinement stage seem to help.

Conclusions and Interpretation

Our final model has shown an improvement in nearly all evaluation metrics as compared to both the original benchmark model and our previous best model (Decision Tree Classifer). The results of these models are summarized below.

	AUC	Accuracy	Log Loss
Benchmark (Logistic Regression)	0.73	0.69	10.7
Decision Tree Classifier (Initial)	0.75	0.69	10.6
Decision Tree Classifier (Final)	0.78	0.72	9.8

We can see that our refined Decision Tree Classifier exceeds the AUC of the benchmark by .05 and the original Decision Tree Classifier by .03. The accuracy metric improves over both prior models by 3%. Additionally, the log loss has decreased to 9.8 from around 10.6. This is a marked improvement with the addition of only two new features.

Looking at the feature importance chart above, we can see that in particular, the new "membership_years" feature we developed made a large difference in predictive power, being the most important feature in the new model. The "cumulative_offers" feature is the 4th most important feature in the new model, so overall this was effort well-spent.

Lastly, we can use the model interpretation library eli5 to dive more deeply into the model features and explore how the decision tree is making predictions.

In [331]:
```
explanation = explain_weights.explain_decision_tree(dt_final, feature_names=X.columns.tolist())
html_rep = format_as_html(explanation)
display(HTML(data=html_rep))
```

Explained as: decision tree

Decision tree feature importances; values are numbers 0 <= x <= 1;
all values sum to 1.

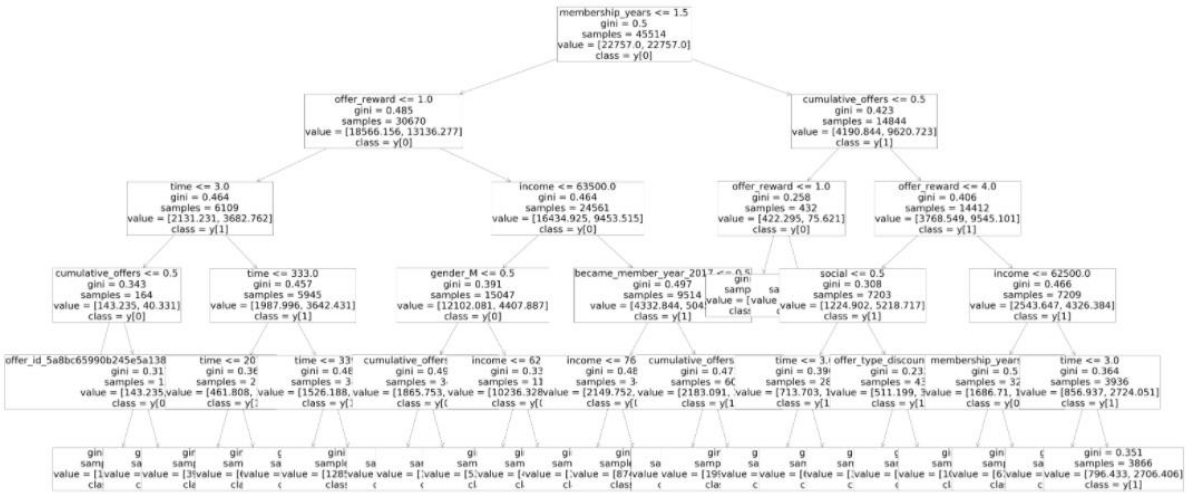
Weight	Feature
0.2946	membership_years
0.2408	income
0.1845	offer_reward
0.0945	cumulative_offers
0.0660	gender_M
0.0638	time
0.0373	became_member_year_2017
0.0106	social
0.0073	offer_type_discount
0.0007	offer_id_5a8bc65990b245e5a138643cd4eb9837

We can see that the feature importances found by eli5 match closely what is found in the DecisionTree().featureimportances attribute that we plotted above. Here we get a little more detail as to how much weight is assigned to each feature and what values are most important in the decision tree.

We can see in both feature importance charts that the most effective offer is a discount, and the most effective medium for offers is social (rather than email, mobile, or web). This is information perhaps Starbucks would find valuable in targeting future offers.

The model also thinks that male users are more likely to complete an offer than female users. While males did complete more offers than females overall, we know from our EDA that males outnumber females, and that they received and viewed more offers cumulatively. On the other hand, females completed a higher proportion of the offers they received, which the model does not seem to pick up on.

Finally, we can view the Decision Tree itself, using the plot_tree() method in sklearn.tree. This provides the final piece of the puzzle in interpreting the model. Like in the eli5 rendering above, we can see the decision splits for each feature. Now, however, we know how each split is classified.



Looking at the first node, membership_years <= 1.5 years is the first decision split. Users with less than 1.5 years as a member are less likely to complete an offer if the offer reward is less than one dollar. (offer_reward <= 1.0 yields a class of 0).

We can also see the role that income plays. Here the split is 63500 dollars, below which a user is less likely to complete an offer unless the reward amount is higher.

In completing this model and interpreting the results, we now have a lot of valuable information for Starbucks in what kinds of offers are most successful (discount), what kinds of customers to target (Male members of at least 1.5 years with income greater than 63500), and by what medium (social media). It will be interesting to see how targeting promotional offers using this model improves sales and customer engagement.