Starbucks Capstone Challenge

Machine Learning Engineer Nanodegree

Soodi Milanlouei

Contents

1 Definition

1.1 Project Overview

In today's world, marketing is an indispensable component of each and every industry. Marketing particularly assists businesses to increase their brand awareness in the consumer market, which eventually will help them to boost their chances of growth. According to Wall Street Journal, companies are allocating as high as 24% of their budget towards marketing every year, with an average of 11%. While allocations within the marketing budget can vary from company to company, it generally includes advertisement and discounts. With such a significant amount of budget being spent on marketing, companies are seeking intelligent ways to spend their money optimally. Many companies nowadays have analytics departments that advise them on identifying potential areas to target. The is where Machine Learning, and Data Science in general, starts to play a crucial role. There are various studies exploring the application of machine learning in marketing optimizations [1, 2, 3]. Machine Learning methods have enabled companies to intelligently decide on how and whom to target with their marketing activities to improve their revenue. Take offering discounts as an example. Companies use discount offers as incentives to persuade potential customers to purchase their product(s). For obvious reasons, they cannot offer discounts to all of their potential customers as it will adversely affect the revenue. Instead, they will offer it only to a proportion of the market. Deciding on whom to offer the discount to is a subject that data scientists and machine learning engineers have investigated thoroughly in recent years [4, 5, 6], identifying customer segmentations who will react favorably to marketing interventions and incentives.

1.2 Problem Statement

In the Starbucks Capstone Challenge of Udacity's Machine Learning Engineer Nanodegree, we are given a number of simulated datasets which emulate customer behavior on the Starbucks rewards mobile app. This app is mainly used for sending either informational messages or promotional offers. A customer might be targeted by (1) informational advertisement, (2) discount offer, or (3) buy one get one free (BOGO) offer. The data provided includes the attributes of all offers available, the demographics of each customer, and the features of each transaction made. While it is not possible to send all the offers to all customers, the goal of this project is to extract insights from the data provided and identify customer segmentation and particular offers that they react to better. Additionally, the aim is to design a new recommendation system which specifies which offer (if any) should be given to an individual customer.

In order to address the questions laid out above, we will rely on three main approaches:

- Exploratory Data Analysis
- Predictive Modeling
- Uplift Modeling

Exploratory Data Analysis: This approach will help us to summarize the main characteristics pertinent to our data visually detect patterns in customer behavior and identify particular demographics who react favorably to the offers that the mobile app is providing. This will be our initial investigation towards the data to gather insights for the modeling work.

Predictive Modeling: the initial hypothesis is that customer characteristics and offer attributes are associated with the likelihood of successfully completing an offer. We will build a model to find these potential associations. It is worth noting that while this model can provide insights about the purchasing likelihood, it cannot be directly used as a recommendation model.

Uplift Modeling: One of the goals of this project is to design a recommendation system which can assist the mobile app in intelligently sending offers to customers. The logic here is each offer acts as an intervention and an offer should be sent to a customer whose likelihood of purchasing increases if he/she receives the offer. Uplift models are mainly designed to find the right treatments for the right customers and we will use this approach in this project.

1.3 Metrics

In the predictive modeling, we will build a classification model to estimate the likelihood of successfully completing an offer given the customers' and offers' attributes. We will assess the performance of this model using accuracy, precision, recall, and F1 score. Below, we show the definition of each metric:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{1}$$

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

$$F1\ Score = \frac{2 * Recall * Precision}{Recall + Precision} \tag{4}$$

where TP is the number of True Positives, FP is for False Positives, TN stands for True Negatives, and FN is for False Negatives.

In the uplift modeling, we usually use the predicted uplift scores to rank customers in descending order to find out how many extra customers we would attract, if we used the uplift model to send our offers. Here, we calculate a metric called cumulative gain:

$$(\frac{Y^T}{N^T} - \frac{Y^C}{N^C})(N^T + N^C), \tag{5}$$

where C stands for control (the group of customers who have not received an offer) and T stands for treatment (customers who received an offer/treatment. Y is the number of customers who made a purchase in each bucket of uplift scores and N is the total number of customers in each bucket. We will use this metric to evaluate the goodness of our uplift models.

2 Analysis

2.1 Data Exploration

There are three datasets provided for this project. Each of these tables can help us understand how and in what way each offer is effective in persuading a customer:

- **portfolio.json** - containing offer ids and meta data about each offer (duration, type, etc.). Coming from three main families of promotions (informational, discount, BOGO), there are 10 unique promotions available in this table. There is no monetary incentive associated with informational advertisements. In the discounts offer, the customer will receive some discounts if they spend a particular amount. Lastly, in the BOGO offer, if a customer purchases a product, he/she will receive a free one as well.
- **profile.json** - demographic data for each customer. In this table, some of the values are missing which needs to be addressed before any modeling work.
- **transcript.json** - records for transactions, offers received, offers viewed, and offers completed.

The schema and explanation of each variable in the files are as following:

portfolio.json

- id (string) - offer id
- offer_type (string) - type of offer ie BOGO, discount, informational
- difficulty (int) - minimum required spend to complete an offer
- reward (int) - reward given for completing an offer
- duration (int) - time for offer to be open, in days
- channels (list of strings)

profile.json

- age (int) - age of the customer
- became_member_on (int) - date when customer created an app account
- gender (str) - gender of the customer (note some entries contain 'O' for other rather than M or F)
- id (str) - customer id
- income (float) - customer's income

transcript.json

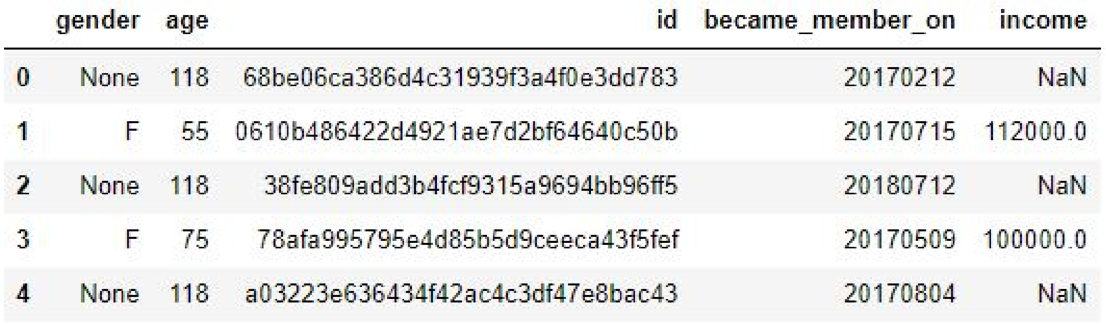
- event (str) - record description (ie transaction, offer received, offer viewed, etc.)
- person (str) - customer id
- time (int) - time in hours since start of test. The data begins at time t=0
- value - (dict of strings) - either an offer id or transaction amount depending on the record

It is important to note that each offer has a validity period before the offer expires, and this includes informational offers. In this case, we will assume that if an informational offer has X days of validity, the customer is feeling the influence of the offer for X days after receiving the advertisement. Another characteristic of the data is that a customer may complete an offer without viewing it. For instance, if a customer receives a BOGO offer with a validity of 7 days, he/she will get a free product at the time of purchase during that week without even seeing the offer. In this case, the offer is completed, but the credit does not go to the offer.

2.2 Exploratory Visualization

In order to fulfill the goals of this project, we start with Exploratory Data Analysis (EDA), investigating the datasets provided. To do so, we created the Processing module which includes the classes and functions that we need. We initiate the DataPrep class which reads all three tables, including portfolio, profile, and transcript. First, let's see how each table looks like.

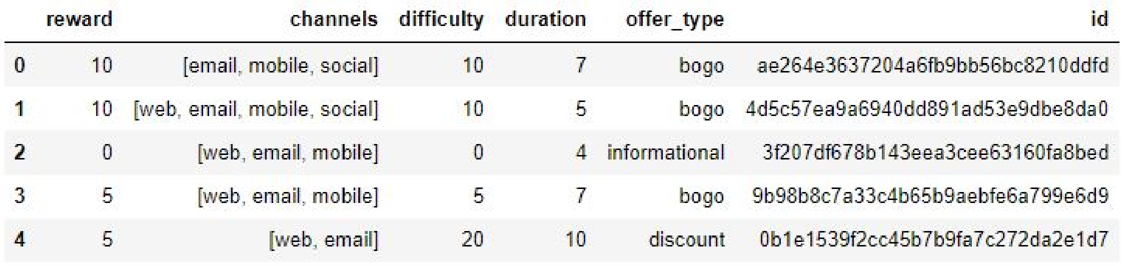
```
data_prep = Processing.DataPrep()
data_prep.profile.head()
```



	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

Figure 1

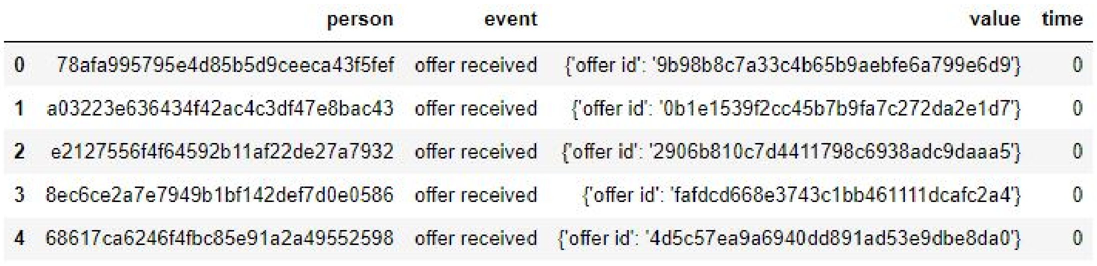
```
data_prep.portfolio.head()
```



	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

Figure 2

```
data_prep.transcript.head()
```



	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

Figure 3

We initiate the DataPrep class which reads all three tables, including portfolio, profile, and transcript. Next, we use the portfolio_prep function to prepare portfolio data for EDA. This function does the following:

1. transform the channel column and create dummy variables upon,
2. rename some of the variables,
3. rename offer IDs for better readability The new format will be OfferType_Duration_Reward.

profile_prep function is design to prepare the profile data by executing the following:

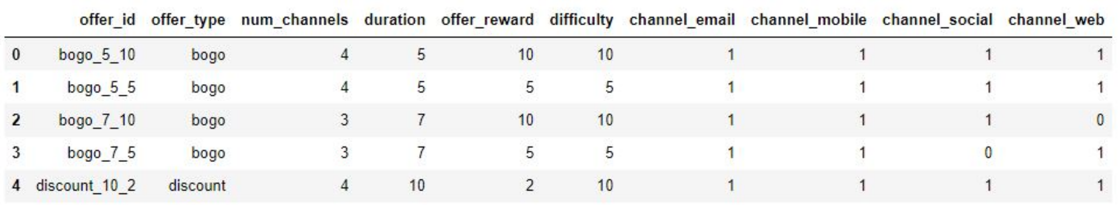
1. rename user IDs for better readability,
2. find the number of days of membership for each user,
3. find the year of membership,
4. find the month of membership,
5. replace age of 118 with NaN,
6. replace None gender with NaN,
7. drop rows which have missing values for all user attributes.

transcript_prep function will manipulate the transcript table by running the steps below:

1. transform the value column and create new columns containing the offer ID and transaction amount if applicable,
2. rename events for better readability,
3. rename offer IDs for better readability, consistent with the portfolio table,
4. rename user IDs for better readability, consistent with profile table.

Now that we have the portfolio data ready, we use the describe_portfolio function in the DataDescription class to explore various features of this table. Below, you can see the results of initiating this class and running the function.

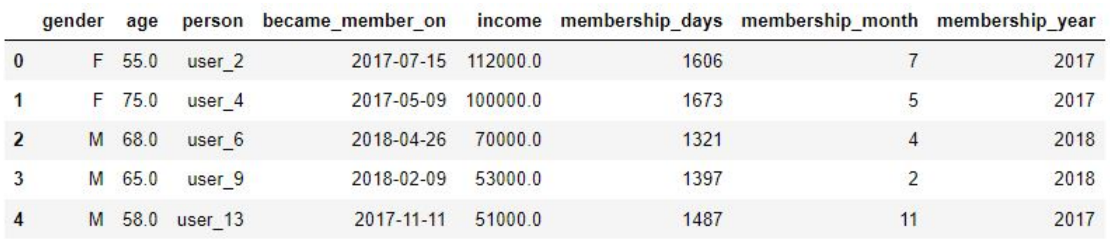
```
portfolio = data_prep.portfolio_prep()
portfolio.head()
```



	offer_id	offer_type	num_channels	duration	offer_reward	difficulty	channel_email	channel_mobile	channel_social	channel_web
0	bogo_5_10	bogo	4	5	10	10	1	1	1	1
1	bogo_5_5	bogo	4	5	5	5	1	1	1	1
2	bogo_7_10	bogo	3	7	10	10	1	1	1	0
3	bogo_7_5	bogo	3	7	5	5	1	1	0	1
4	discount_10_2	discount	4	10	2	10	1	1	1	1

Figure 4

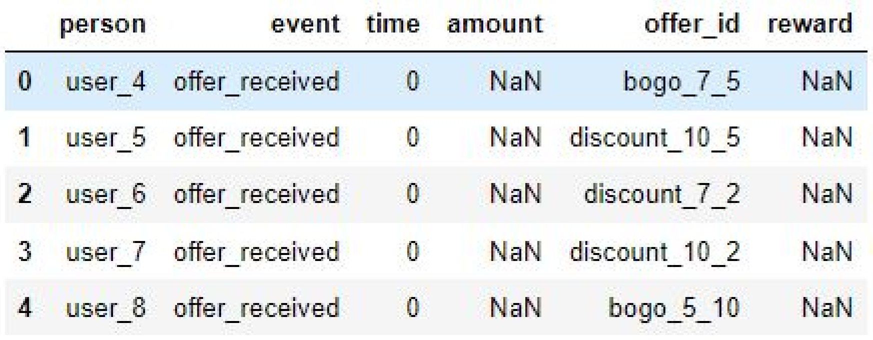
```
profile = data_prep.profile_prep()
profile.head()
```



	gender	age	person	became_member_on	income	membership_days	membership_month	membership_year
0	F	55.0	user_2	2017-07-15	112000.0	1606	7	2017
1	F	75.0	user_4	2017-05-09	100000.0	1673	5	2017
2	M	68.0	user_6	2018-04-26	70000.0	1321	4	2018
3	M	65.0	user_9	2018-02-09	53000.0	1397	2	2018
4	M	58.0	user_13	2017-11-11	51000.0	1487	11	2017

Figure 5

```
transcript = data_prep.transcript_prep()
transcript.head()
```



	person	event	time	amount	offer_id	reward
0	user_4	offer_received	0	NaN	bogo_7_5	NaN
1	user_5	offer_received	0	NaN	discount_10_5	NaN
2	user_6	offer_received	0	NaN	discount_7_2	NaN
3	user_7	offer_received	0	NaN	discount_10_2	NaN
4	user_8	offer_received	0	NaN	bogo_5_10	NaN

Figure 6

Now that we have the portfolio data ready, we use the describe_portfolio function in the DataDescription class to explore various features of this table. Below, you can see the results of initiating this class and running the function.

```
data_description = Processing.DataDescription(portfolio, profile, transcript)
data_description.describe_portfolio()
```

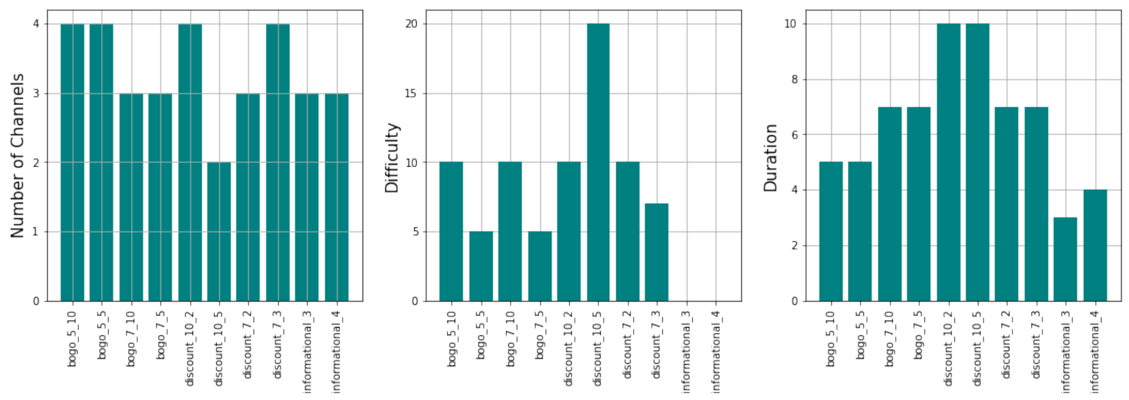


Figure 7

Above, we can see in the first plot that there are at most four channels through which offers are sent to users. While bogo_5_10, bogo_5_5, discount_10_5, and discount_7_3 are sent using four channels, the discount_10_5 is only sent via two channels. Not all the offers have the same level of difficulty (middle plot). There is no difficulty pertinent to informational offer and discount_10_5 has the highest difficulty. Moreover, the validity period varies from offer to offer. Informational offers have the lowest validity duration and discount_10_2 and discount_10_5 have the highest.

Now, we move on to the profile table and use the describe_profile function to explore this data. In the first plot below, we can see the number of users registers in each month cumulatively. We can see that in the last two quarters of the year, we usually see more users registered on the app with January following the same pattern, while the second quarter sees the least number of new registrations. In the second plot, we look at the same information by year. It is apparent that through years the app was able to attract more and more users with one caveat that the 2018 data goes only until July and we do not have the full data for that year. We can also see that the majority of the app users are male with 57.23%.

Next, we look at the distribution of the continuous variables. In plot 4, we depict the distribution of age, which has an average of 54 and standard deviation of 17. Income is quite normally distributed with an average of 65,405. The distribution of membership days is multi-modal, with a median of 1,580.

In the last row of plots, we look at the same distribution by separately for each gender. Female users have a higher age average compared to their male counterparts (57.5 vs 52.1). That's also the case for income, where female users have an average of 71,306 and male users' average is 61,195. In terms of membership days, female users have a median of 1,622, while for male users the median is 1,565.

```
data_description.describe_profile()
```

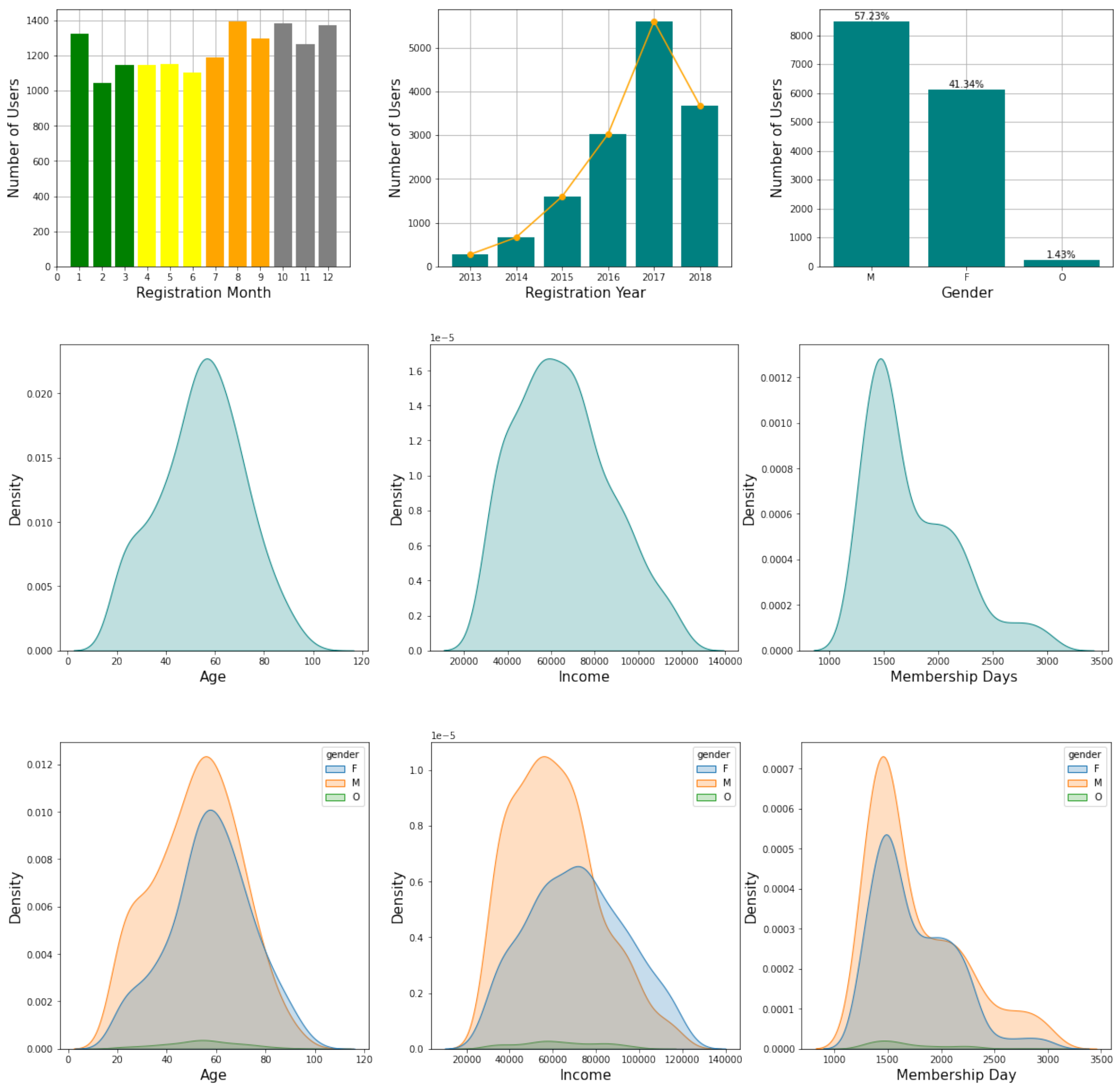


Figure 8

Exploring the transcript table, in the first plot, we can see the distribution of the number of received offers per user. Around 65% of the users received 4 or 5 offers. Moreover, 18% of users received 6 offers which is the maximum number of offers sent to a user. Looking at the distribution of the number of viewed offers, it is almost normally distributed with almost half of the customer viewing 3 or 4 offers. Lastly, in the third plot, we observe that all offer types are sent out evenly.

```
data_description.describe_transcript()
```

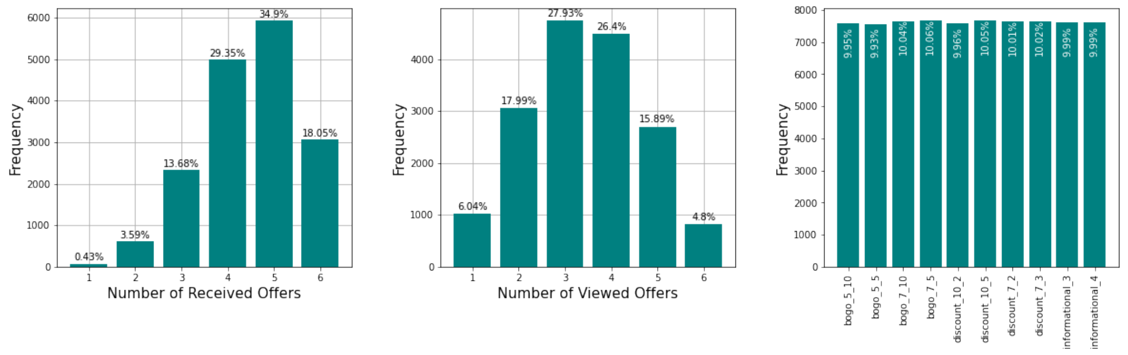


Figure 9

In the next step, we aim to merge all three table to create a unified view. To do so, we utilize the DataMerge class. There are four main functions in the class which will help us merging the data. Here, we explain how the data merge process is executed. We first start with the transcript table and create four subset tables by filtering on the event columns. Consequently, we will have a table for received offers, viewed offers, transactions, and completed offers. The first two tables are joined using person and offer_id columns. Then we add in the transactions using the person column. Finally, we merge the completed offers table using person and offer_id column again. At this stage, there are rows that are redundantly created and we need to remove them. In order to do this, we define a set of logic to filter out redundant rows. A row is valid if (1) view time and complete time are null, or (2) view time is larger than receive time and complete time is larger than receive time, or (3) view time is larger than receive time and and complete time is larger than receive time, or (4) view time is larger than receive time and the complete time is null.

After removing redundant rows, we aim to find offers that have been either tried or been successful. To achieve this, we again define a set of logics as following:

(i) If viewed time is null:

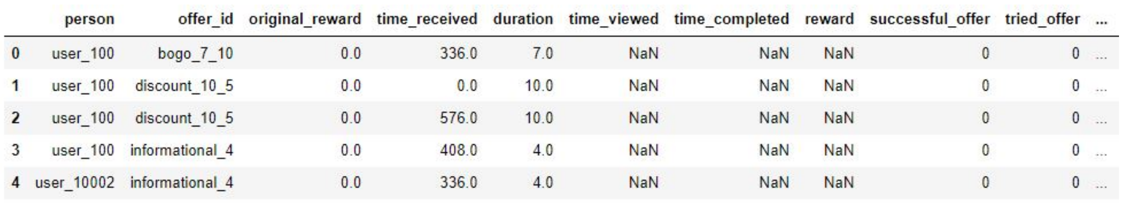
- The offer is unsuccessful.
- If the offer is informational, and there is a transaction between the receive time and offer expiry time, the offer is assumed to be tried.
- If the offer is not informational, and complete time is between receive time and expiry time, the offer is deemed to be tried.

(ii) If the viewed times is not null:

- If offer is informational and transaction time is null, the offer is unsuccessful.
- if there is a transaction between the receive time and expiry time, the offer has been tried. If the transaction has been happened after viewing the offer and before the expiry time, the offer is successful.
- if the offer is not informational, complete time is larger than view time and less than expiry time, the offer has been successful.
- if an offer was successful, it is assumed to be tried but not necessarily the other way around.

Next, we append portfolio and profile tables to this newly created table using the append_other_data function and save the final table using the save_data function.

```python
data_merge = Processing.DataMerge(portfolio,profile,transcript)
merged_data = data_merge.data_merge()
merged_data = data_merge.remove_redundant_data()
merged_data = data_merge.find_success_tried_offers()
merged_data = data_merge.append_other_data()
data_merge.save_data()
merged_data.head()
```



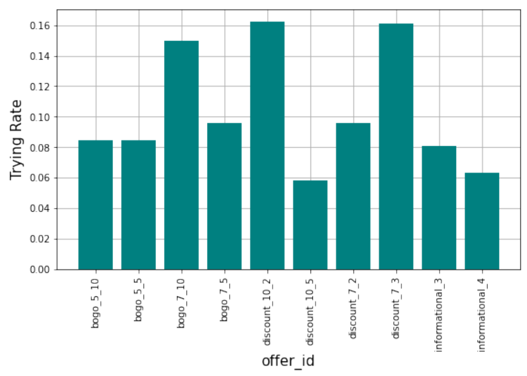
	person	offer_id	original_reward	time_received	duration	time_viewed	time_completed	reward	successful_offer	tried_offer	...
0	user_100	bogo_7_10	0.0	336.0	7.0	NaN	NaN	NaN	0	0	...
1	user_100	discount_10_5	0.0	0.0	10.0	NaN	NaN	NaN	0	0	...
2	user_100	discount_10_5	0.0	576.0	10.0	NaN	NaN	NaN	0	0	...
3	user_100	informational_4	0.0	408.0	4.0	NaN	NaN	NaN	0	0	...
4	user_10002	informational_4	0.0	336.0	4.0	NaN	NaN	NaN	0	0	...

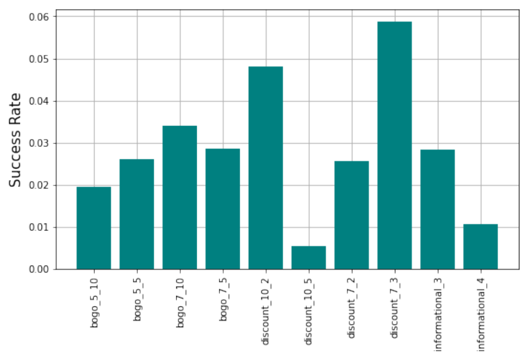
Figure 10

Now that we have a definition for tried and successful offers, we can look at the trying and success rates for each offer. Looking at the plot below on the left in the first row, bogo_7_10, discount_10_2, and discount_7_3 have high trying rate, while discount_10_5 and informational_4 have low trying rate. The plot on the right tells us that discount_10_2 and discount_7_3 also have high success rate; on the other hand, informational_4 and discount_10_5 have low success rate.

We can also look at these rates by some of the descriptive variables that we have. Plots in the second row demonstrate that female customers tend to try and complete an offer successfully more often than male customers. To investigate the association between age and trying and success rates, we can bin the age variable into five buckets including very young, young, middle age, old, and very low. Interestingly, the first bucket (very young customers) exhibits the highest trying rate, but the lowest success rate. Following the same approach for income, we group customers into five groups (very low, low, medium, high, and very high). Going higher into the income spectrum, customers tend to have lower trying rate but higher success rate.

```python
Processing.offer_performance(merged_data, "offer_id")
Processing.offer_performance(merged_data, "gender")
Processing.offer_performance(merged_data, "age")
Processing.offer_performance(merged_data, "income")
```





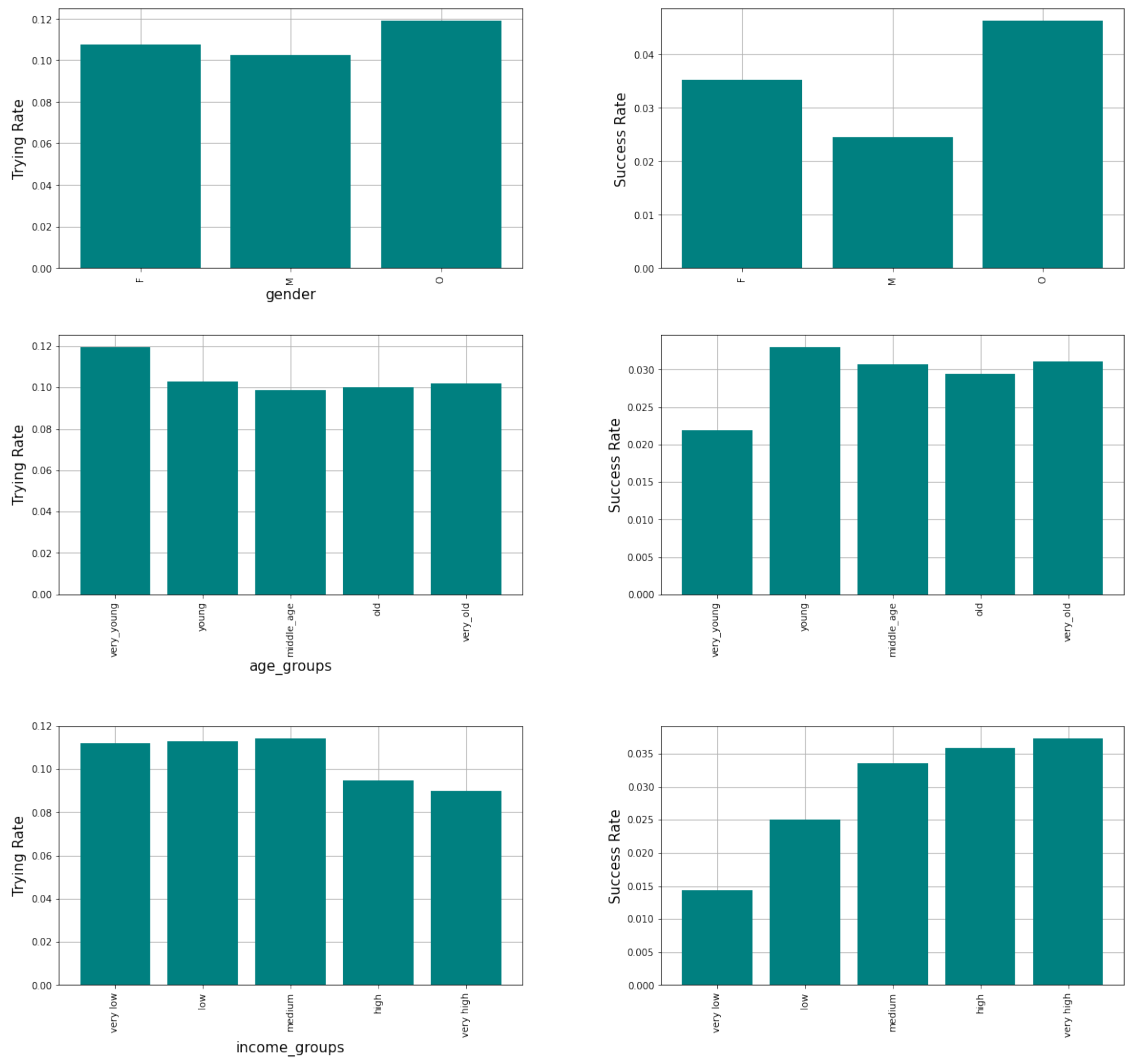


Figure 11

This EDA has helped us clean, manipulate, and merge the tables available; moreover, we extracted insights about the properties of each variable. We also specified whether an offer has been tried or been successful. The final table created in this script can be used to achieve the other two goals that we have defined in the project. The immediate next step from here is to use the knowledge gathered in the EDA step to build a predictive model, predicting whether an offer sent to a customer will be successful or not.

2.3 Algorithms and Techniques

2.3.1 Predictive Modeling

Gradient Boosting Machines or GBM models are a family of powerful machine learning algorithms that have been successfully leveraged and implemented in various domains. They are highly customizable to the particular needs of the application, like being learned with respect to different loss functions. Boosting is generally a technique for transforming weak learners into strong learners. Gradient boosting mainly consists of three components: (1) a loss function to be optimized which depends on the type of problem being solved, (2) a weak learner to make predictions which is a decision tree in this case, and (3) an additive model to add weak learners to minimize the loss function (i.e. follow the gradient) [7]. To build such a model, we use the LightGBM package which is a gradient boosting framework that uses tree-based learning algorithms [8].

2.3.2 Uplift Modeling

To explain the logic behind uplift modeling, let's assume customer A's likelihood of purchasing without any offer is 0.7, and if we send him an offer, it will increase to 0.8. On the other hand, customer B will buy our product with the probability of 0.3 and in case of receiving an offer, it will go up to 0.6. While the offer increases customer A's purchasing likelihood by 0.1, this difference for customer B is 0.3. It is evident that customer B is a better candidate to receive our offer. Interestingly, if were to only use a predictive model and utilize the scores to decide which customers to target, we would select customer A. This whole framework is called Uplift Modeling (also known as incremental modeling) which aims to find the change in the likelihood of an event. Uplift modeling is particularly useful in marketing campaigns, as it helps marketing teams to measure the effectiveness of their tactics and isolate their influence. The fundamental issue in uplift modeling is that we cannot measure the effect of an intervention on one individual customer, simply because a customer can either receive or not receive an offer. In this case, what we can do is estimate the uplift based on groups of similar customers. In particular, we classify all customers into four main groups:

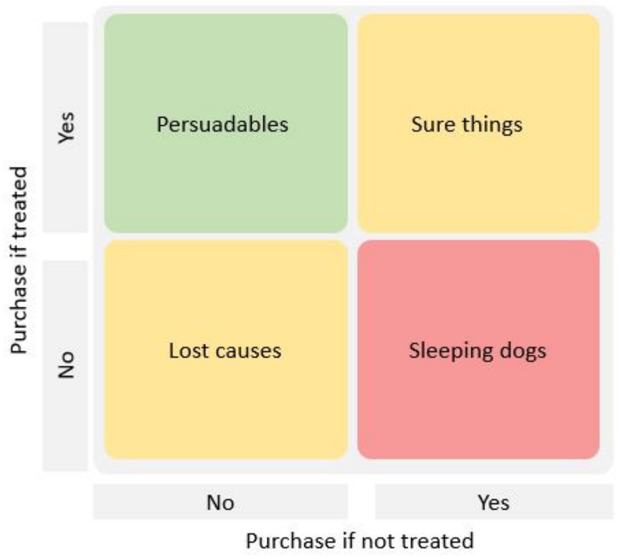


Figure 12

Persuadables are customers who we need to identify and target. These are customers who will purchase our product only if targeted by our treatment/offer. Sure things are will buy our product even if they do not receive any offer. Lost causes are customers are who are not going to purchase even if there are treated. These two groups have zero uplift. Treating these customers would be synonymous with wasting marketing money. The last group is called sleeping dogs. These are customers whose likelihood of purchasing will decrease if treated. It is important to identify these customers as well, since we do not want to lose customers with our marketing activities.

There are three main approaches towards uplift modeling. (1) tree-based algorithms which model uplift directly [9, 10], (2) meta-learners which model uplift indirectly [11, 12, 13] (3) and class variable transformation which makes uplift predictable by transforming the response variable [14]. Uplift models are usually applied on the data collected throughout AB testing where the entire population is randomly divided into two groups: control group where there is no treatment applied and test group where all customers will be treated by our intervention. In our problem here, we don't have a proper control group to compare each treatment two. All customers are treated by at least one type of offer. However, we can make some assumptions here. We can assume that our control group consists of customers who have received informational offer. Accordingly, we can assume that other types of offers are our treatments. When there is more than one treatment, uplift modeling can become quite complicated. To keep things simple, we leverage the S-learner approach.

S-learner algorithm estimates the response variable using all independent variables including the treatment indicator using a single model. The estimation has two steps. First, we build a predictive model associating all variables with the response variable. Then we we predict the response variable once assuming that the treatment indicator is equal to control and then assuming it is equal to treatment (while all other variables are kept fixed). The difference between these two values will be the estimated uplift. These steps can be shown mathematically:

Step 1: estimate the average outcome $\mu(x) = E[Y|X = x, T = t]$ using a machine learning algorithm.

Step 2: define uplift as: $\widehat{\tau}(x) = \widehat{\mu}(x, T = 1) - \widehat{\mu}(x, T = 0)$

2.4 Benchmark

In this project, for the predictive model we will build a simple logistic regression model and use it as the benchmark model. We will compare the accuracy, precision, recall and the F1 score of the GBM model with the logistic regression. For the uplift model, We aim to find a model and recommendation system which can provide more value compared to randomly sending out offers to customers. Moreover, we will compare it with the existing strategy which is encapsulated in the data. Here, we can assume that the offers are sent to customers based on a particular strategy which have already indicated which customers will successfully complete an offer. To do so, we utilize the cumulative gain metric that we defined in the Section 1.3.

3 Methodology

3.1 Data Preprocessing

In the previous step, we ran EDA to understand the datasets and combine them while specifying which offer was successful. Here, we hypothesize that customer characteristics and offer attributes are associated with the likelihood of an offer becoming successful. To examine this hypothesis, we will build predictive models to find these potential associations. It is worth noting that while this model can provide insights about the purchasing likelihood, it cannot be directly used as a recommendation model. To begin, we build a logistic model and use it as our baseline. The DataPrep class in the Modeling_Helper package can help us to to read the data, specify features, and create dummy variables upon categorical features. The continuous variables are:

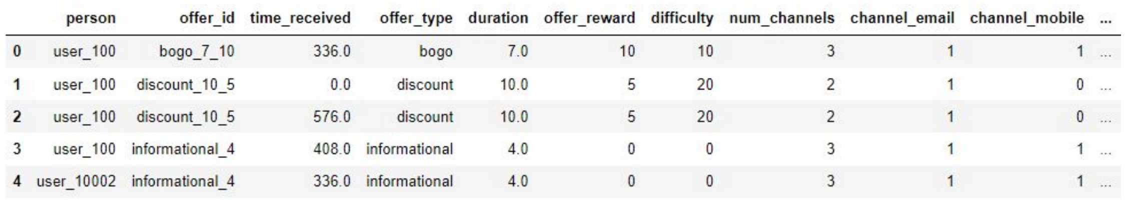
- age
- income
- membership_days
- duration
- difficulty
- num_channels
- channel_mobile (binary)
- channel_social (binary)
- channel_web (binary)

and categorical variables are:

- offer_id
- offer_type
- offer_reward
- gender
- membership_month
- membership_year

3.2 Predictive Modeling: Benchmark Model

```
data_prep = Modeling_Helper.DataPrep()
data_prep.prep_data_logistic()
modeling_data = data_prep.modeling_data
features = data_prep.features
y_var = data_prep.y_var
modeling_data.head()
```



	person	offer_id	time_received	offer_type	duration	offer_reward	difficulty	num_channels	channel_email	channel_mobile	...
0	user_100	bogo_7_10	336.0	bogo	7.0	10	10	3	1	1	...
1	user_100	discount_10_5	0.0	discount	10.0	5	20	2	1	0	...
2	user_100	discount_10_5	576.0	discount	10.0	5	20	2	1	0	...
3	user_100	informational_4	408.0	informational	4.0	0	0	3	1	1	...
4	user_10002	informational_4	336.0	informational	4.0	0	0	3	1	1	...

Figure 13

We split the data into train and test using the DataSplit class in the Modeling_Helper package. We make sure that one person ID does not appear in both train and test set. Before building any model, we scale our variables and look at the correlation heatmap to see whether there are any highly correlation variables that we can remove. Looking at the heatmap below, we decide to keep all the variables that we have. However, the channel_email variables is dropped as it is equal to 1 for all offer IDs.

```python
data_split = Modeling_Helper.DataSplit('person', modeling_data, y_var, 0.8, 2021)
train_df, test_df = data_split.split_data()
scaler = MinMaxScaler(feature_range = (0,1))

scaler.fit(train_df.loc[:, features])
train_df.loc[: , features] = scaler.transform(train_df.loc[: , features])
test_df.loc[: , features] = scaler.transform(test_df.loc[: , features])

cor_df = Processing.correlation_map(train_df, data_prep.features)
```

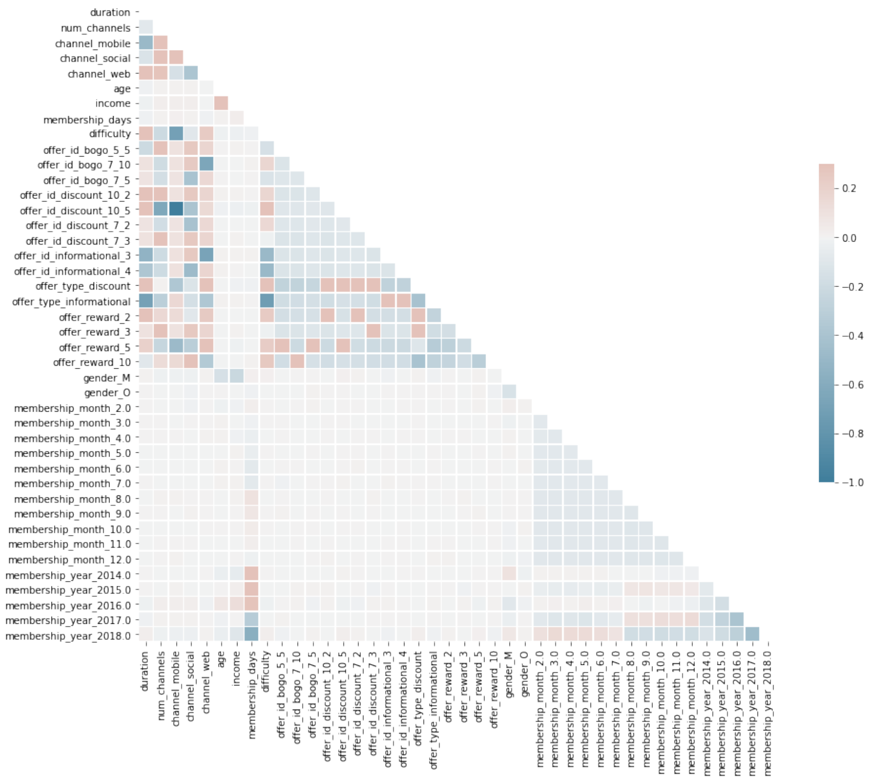


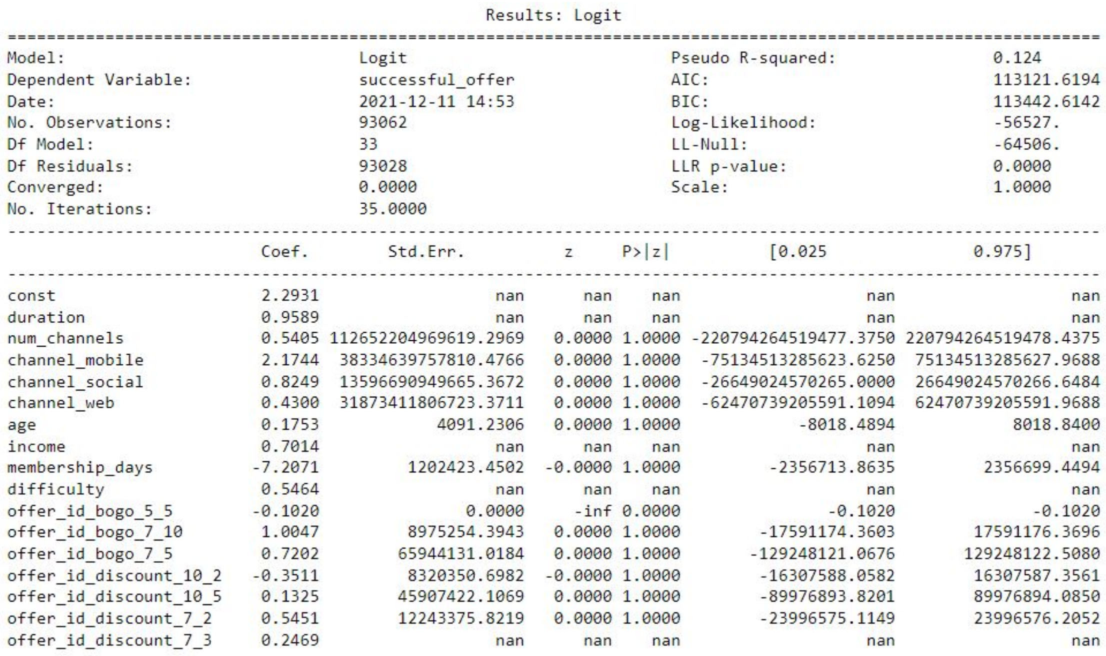
Figure 14

Another characteristic of of our data is that it is highly imbalanced. The abundance of examples from the unsuccessful offers can swamp the successful ones.This means that if we train a model on such data, the algorithm will focus on learning the characteristics of the majority class only, neglecting the examples from the minority one whose predictions are more valuable. There are various ways to tackle this issue, with the most simple one being over-sampling the minority class which we use for our first model.

```python
sm = imblearn.over_sampling.RandomOverSampler(random_state=2021)
X_train_res, y_train_res = sm.fit_resample(train_df[features], train_df[y_var].
    ravel())
X_train_res[y_var] = y_train_res

print("After OverSampling, the shape of train_X: {}".format(X_train_res.shape))
print("After OverSampling, the shape of train_y: {} \n".format(y_train_res.shape))

print("After OverSampling, counts of label '1': {}".format(sum(y_train_res==1)))
print("After OverSampling, counts of label '0': {}".format(sum(y_train_res==0)))
```

```
After OverSampling, the shape of train_X: (93062, 43)
After OverSampling, the shape of train_y: (93062,)

After OverSampling, counts of label '1': 46531
After OverSampling, counts of label '0': 46531
```

```python
X_train_res = sm.add_constant(X_train_res, has_constant='add')
logit_model=sm.Logit(X_train_res[y_var], X_train_res[['const']+features])
logit_model=logit_model.fit()
print(logit_model.summary2())
```



```
                                                 Results: Logit
=================================================================================================================
Model:                            Logit                             Pseudo R-squared:              0.124
Dependent Variable:               successful_offer                  AIC:                           113121.6194
Date:                             2021-12-11 14:53                  BIC:                           113442.6142
No. Observations:                 93062                             Log-Likelihood:                -56527.
Df Model:                         33                                LL-Null:                       -64506.
Df Residuals:                     93028                             LLR p-value:                   0.0000
Converged:                        0.0000                            Scale:                         1.0000
No. Iterations:                   35.0000
-----------------------------------------------------------------------------------------------------------------
                          Coef.          Std.Err.           z      P>|z|          [0.025                0.975]
-----------------------------------------------------------------------------------------------------------------
const                     2.2931                 nan       nan     nan                 nan                   nan
duration                  0.9589                 nan       nan     nan                 nan                   nan
num_channels              0.5405 112652204969619.2969    0.0000 1.0000 -220794264519477.3750 220794264519478.4375
channel_mobile            2.1744  38334639757810.4766    0.0000 1.0000  -75134513285623.6250  75134513285627.9688
channel_social            0.8249  13596690949665.3672    0.0000 1.0000  -26649024570265.0000  26649024570266.6484
channel_web               0.4300  31873411806723.3711    0.0000 1.0000  -62470739205591.1094  62470739205591.9688
age                       0.1753            4091.2306    0.0000 1.0000            -8018.4894            8018.8400
income                    0.7014                  nan       nan    nan                   nan                  nan
membership_days          -7.2071         1202423.4502   -0.0000 1.0000         -2356713.8635         2356699.4494
difficulty                0.5464                  nan       nan    nan                   nan                  nan
offer_id_bogo_5_5        -0.1020               0.0000      -inf 0.0000               -0.1020              -0.1020
offer_id_bogo_7_10        1.0047         8975254.3943    0.0000 1.0000         -17591174.3603        17591176.3696
offer_id_bogo_7_5         0.7202        65944131.0184    0.0000 1.0000        -129248121.0676       129248122.5080
offer_id_discount_10_2   -0.3511         8320350.6982   -0.0000 1.0000         -16307588.0582        16307587.3561
offer_id_discount_10_5    0.1325        45907422.1069    0.0000 1.0000         -89976893.8201        89976894.0850
offer_id_discount_7_2     0.5451        12243375.8219    0.0000 1.0000         -23996575.1149        23996576.2052
offer_id_discount_7_3     0.2469                  nan       nan    nan                   nan                  nan
```

Figure 15

Building our logistic model, we can use the PerformanceAnalysis function (in Modeling_Helper) to analyze the performance. As shown below, the model results in an accuracy of 0.67, precision of 0.65, recall of 0.71, and F1 score of 0.68 on the train set (Figure 16) and on the test set these numbers are 0.63, 0.05, 0.67, and 0.1 (Figure 17). The train set's AUC is 0.73 and for test set, it is 0.69. It is apparent that there is a significant level of over-fitting, meaning that our model learned too much from the train set so that it is not able to generalize over the unseen data (test set).

```python
perf_analysis = Modeling_Helper.PerformanceAnalysis(logit_model, X_train_res, ["
    const"]+features, y_var, "Train Set", prob = False)
perf_analysis.perf_analysis()

test_df = sm.add_constant(test_df, has_constant="add")
perf_analysis = Modeling_Helper.PerformanceAnalysis(logit_model, test_df, ["const"
    ]+features, y_var, "Test Set", prob = False)
perf_analysis.perf_analysis()
```

```
************** Performance: Train Set ****************
```

```
Accuracy: 0.67
Precision: 0.65
Recall: 0.71
F1: 0.68
```

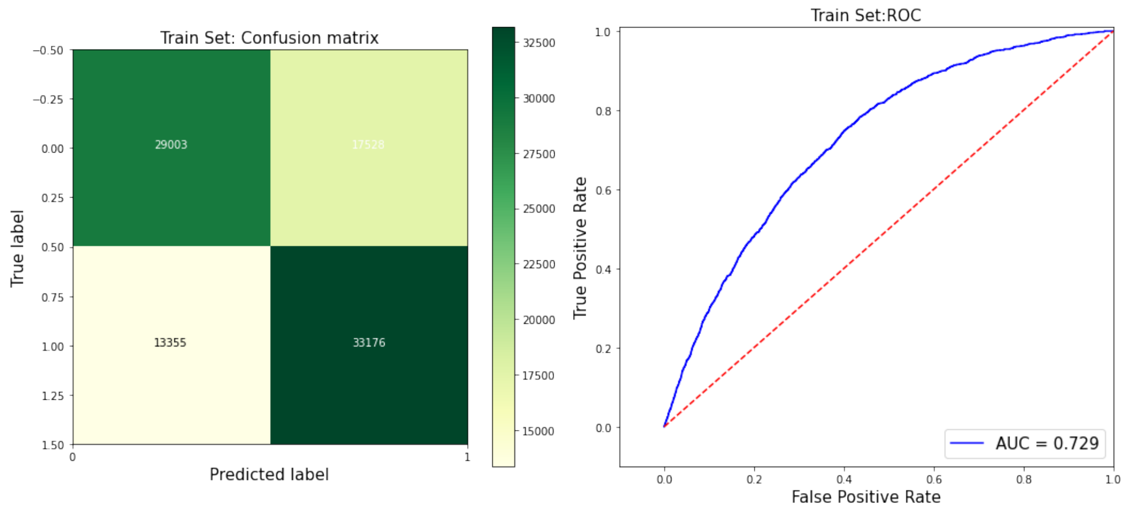


Figure 16

```
************** Performance: Test Set ****************

Accuracy: 0.63
Precision: 0.05
Recall: 0.67
F1: 0.1
```

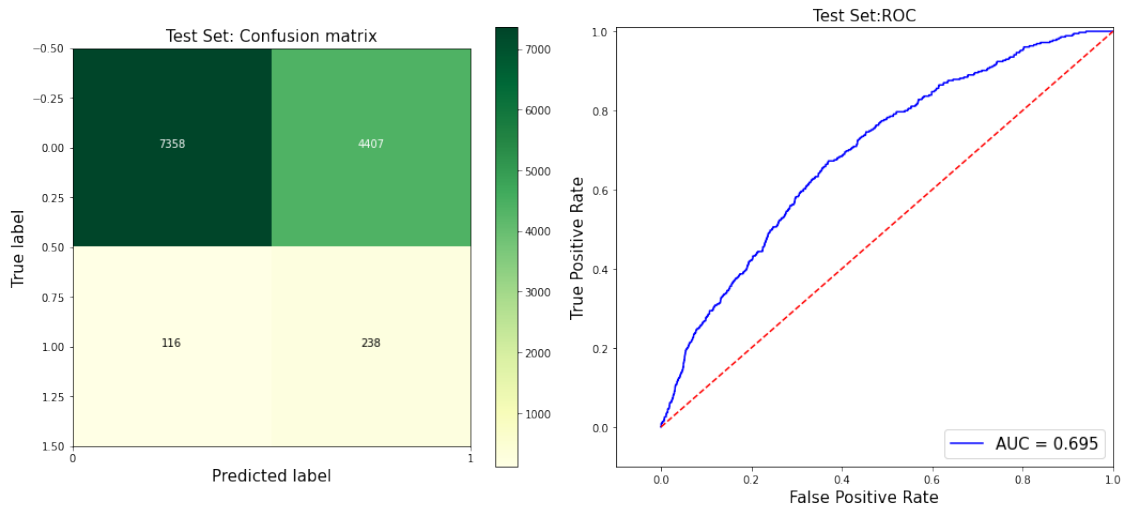


Figure 17

3.3 Predictive Modeling: Gradient Boosting Machines (GBM)

3.3.1 Implementation

Having the logistic model as the baseline, now the question is whether we can train a better model. We can use the DataPrep class in Modeling_Helper to prepare the data and this time we use the prep_data_gbm function. One nice capability of the LightGBM is that it can handle categorical variables internally and we do not need to transform them into binary variables. The prep_data_gbm function takes care of this and changes the type of categorical columns to category. Similar to before, we use the DataSplit class to split the data into train and test. We would not tweak the parameters for the first iteration of the model; however, there are two which are essential. First, we need to specify the objective. objective specifies the learning task and the corresponding learning objective. this parameter is defaulted to regression but since our response variable is whether an offer was successful or not, we need to change it to binary. In building the logistic regression model, we mentioned that our data is imbalanced and we iterated the challenges it may impose the model with. We used over-sampling to remedy this issue. With out GBM model, we are going to follow a similar path but differently. Instead of over-sampling our data, we leverage the scale_pos_weight parameter which is the ratio of number of negative class to the positive class and can be used to scale the gradient for the positive class. this parameter ensures that errors made by the model during training on the positive class are scaled and it persuades to correct for them more often. As a result, the model will make better predictions on the positive class. Looking at the train set, this value is equal to 33.4.

```
data_prep = Modeling_Helper.DataPrep()
data_prep.prep_data_gbm()
modeling_data = data_prep.modeling_data
features = data_prep.features
cat_vars = data_prep.cat_vars
y_var = data_prep.y_var
```

```
data_split = Modeling_Helper.DataSplit("person", modeling_data, y_var, 0.8, 2021)
train_df, test_df = data_split.split_data()
scale_pos_weight_val = (len(train_df)-sum(train_df[y_var]))/sum(train_df[y_var])
print(scale_pos_weight_val)
```

```
train_data = lgb.Dataset(train_df[features], label=train_df[y_var],free_raw_data=
    False)
test_data = lgb.Dataset(test_df[features], label=test_df[y_var], free_raw_data=
    False)
```

```
params = {"num_threads":-1,
          "seed":2021,
          "verbose": -1,
          "objective": "binary",
          "scale_pos_weight":scale_pos_weight_val
         }

gbm_model = lgb.train(params,
            train_data,
            feature_name=features,
            categorical_feature=cat_vars,
            verbose_eval=-1,
                 )
```

Building our first GBM model, We can achieve an accuracy of 0.77, precision of 0.11, recall of 0.95, and F1 score of 0.19 on the train set (Figure 18) and on the test set these numbers are 0.76, 0.06, 0.5, and 0.1. The train set's AUC is 0.92 and for test set, it is 0.69 (Figure 19).

```
perf_analysis = Modeling_Helper.PerformanceAnalysis(gbm_model, train_df, features,
    y_var, "Train Set", prob = False)
perf_analysis.perf_analysis()
perf_analysis = Modeling_Helper.PerformanceAnalysis(gbm_model, test_df, features,
   y_var, "Test Set", prob = False)
perf_analysis.perf_analysis()
```

```
************** Performance: Train Set ****************

Accuracy: 0.77
Precision: 0.11
Recall: 0.95
F1: 0.19
```

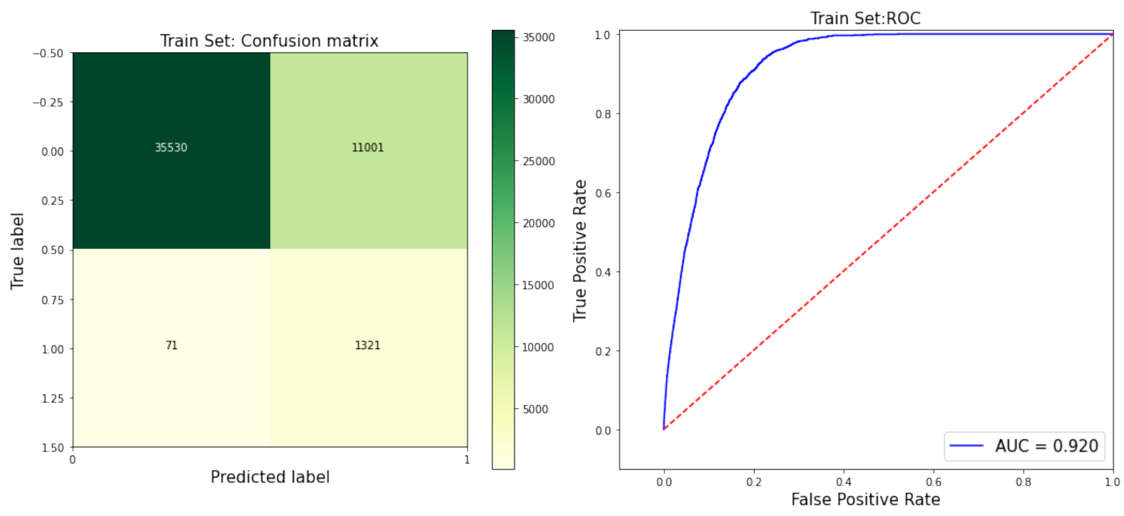


Figure 18

```
************** Performance: Test Set ****************

Accuracy: 0.76
Precision: 0.06
Recall: 0.5
F1: 0.11
```

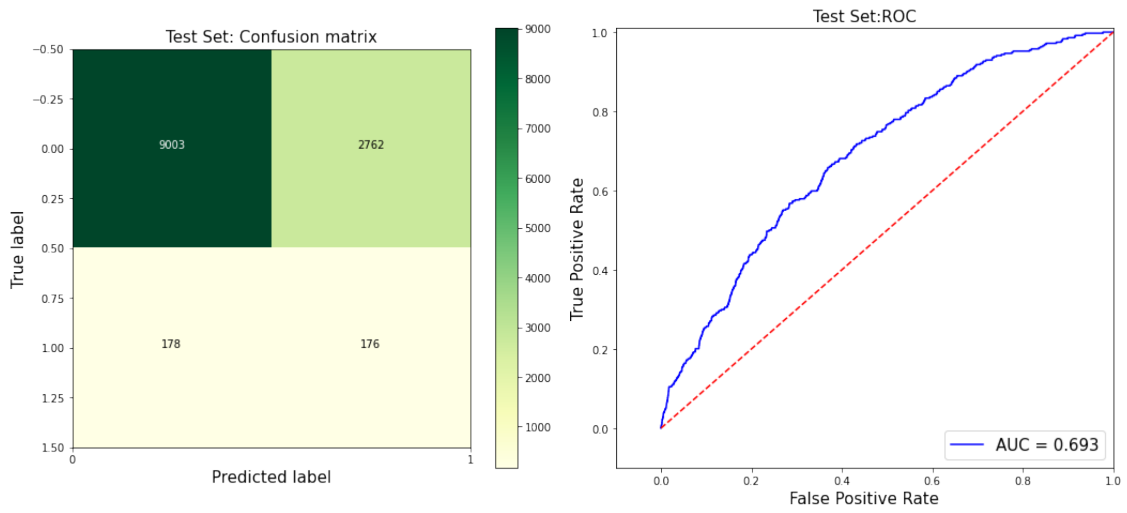


Figure 19

GBM models by themselves are not interpretable, at least not easily and GBMs are not alone in that. Many complex models that can achieve the highest accuracy for large modern datasets can be quite challenging to interpret. Lundberg and Lee however responded to that challenge and proposed the SHAP (SHapley Additive exPlanations) values that can show how much each independent variable contributes to the response variable [15]. Leveraging SHAP values not only we can find the importance of each variable but also the positive/negative relationships that each variable has in regard to the response variable. These values can be found using the Python module shap. We use this package, to see how each variable in our dataset contributes into an offer becoming successful. Below, you can see our SHAP plot using the train set. Now, let's try and understand this plot. The dots your are seeing in this plot are pertinent to each observation in the train set. Variables are ranked based on the their importance. While for continuous variables we have a gradient of colors from blue to red indicating the feature value, categorical variables are naturally gray. The horizontal location specifies whether the effect of that value results in higher or lower predictions. Let's use income as an example. It seems that overall higher income is associated with higher likelihood of an offer becoming successful.

```
explainer = shap.TreeExplainer(gbm_model)
shap_values = explainer.shap_values(train_df[features])

shap.summary_plot(shap_values[1], train_df[features], feature_names = features,
    max_display = 30)
```

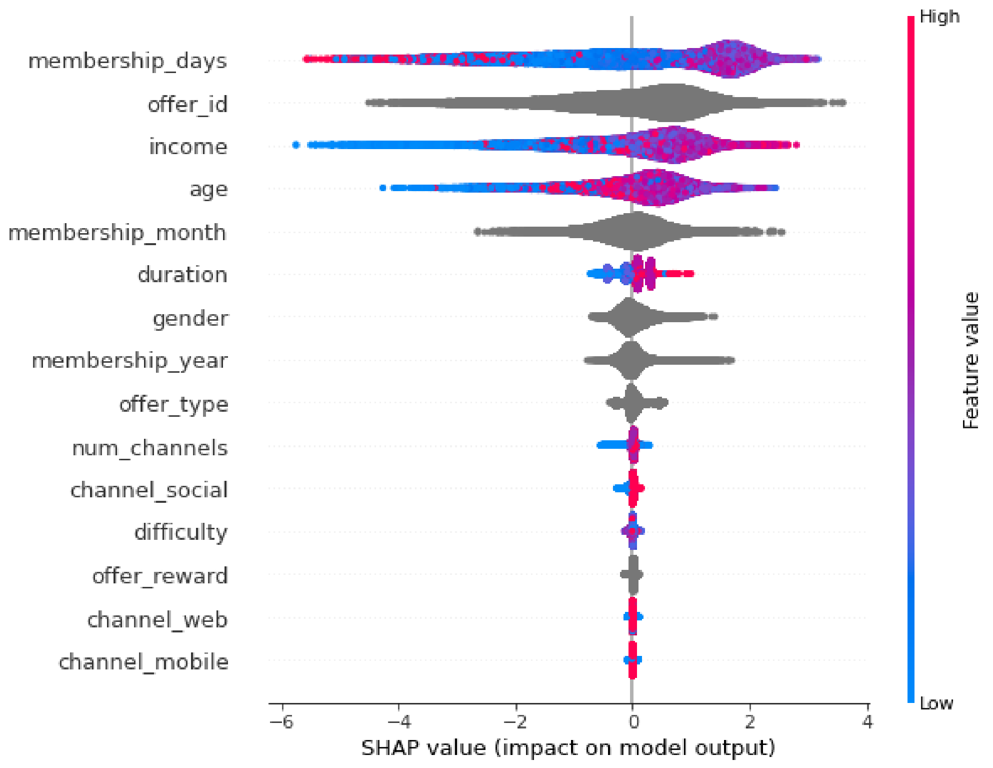


Figure 20

3.3.2 Refinement

Now, we try to improve our initial GBM model following three approaches:

- More feature engineering
- Hard encoding interactions
- Hyper-parameter tuning

3.3.2.1 Feature Engineering

One additional feature that we can add to the data is the cumulative number of offers that each user has received at each time. Our hypothesis is that the more that we target the users (sending more offers), users' likelihood of successfully completing an offer will increase. Another feature that we can tweak is the membership month. So far, we treated this variable as categorical, but is it really? The hour of day and the month of year are indeed cyclical features and they need to be transformed into a representation that can properly preserve information such as the 12th month and the first month are close to each other. A common method to encode cyclical features is to transform them into two dimensions using sine and cosine functions, as done below (Learn More [16]). Now we can include these two newly created continuous features in our model and maintain the cyclical nature of the month data.

```
modeling_data = modeling_data.sort_values(by = ["person", "time_received"]).
    reset_index(drop = True)
modeling_data["num_offers"] = modeling_data.groupby("person")["time_received"].
    transform(lambda group: group.astype("category").cat.codes + 1)
```

```
modeling_data["membership_month_sin"] = np.sin(2 * np.pi * modeling_data["
    membership_month"].astype(int) / 12)
modeling_data["membership_month_cos"] = np.cos(2 * np.pi * modeling_data["
    membership_month"].astype(int) / 12)
```

3.3.2.2 Hard encoding interactions

Tree-based method intrinsically take into account the interactions between independent variables. Having said that, we can hard-encode some of the interactions to make it easier for the model to pick them up. Looking at SHAP values (Figure 20), we found that membership days and income are among the most important variable in the model. We hypothesize that that there is an interactive effect among these two. We also include the interaction terms between income and the number of channels and between age and income.

```
modeling_data["membership_days_X_income"] = modeling_data["membership_days"]*
    modeling_data["income"]
modeling_data["num_channels_X_income"] = modeling_data["num_channels"]*
    modeling_data["income"]
modeling_data["age_X_income"] = modeling_data["age"]*modeling_data["income"]
```

```
cont_vars = [
    "duration",
    "num_channels",
    "channel_email",
    "channel_mobile",
    "channel_social",
    "channel_web",
    "age",
    "income",
    "membership_days",
    "num_offers",
    "membership_days_X_income",
    "num_channels_X_income",
    "age_X_income",
    "membership_month_sin",
    "membership_month_cos",
    "difficulty",

]

cat_vars = [
    "offer_id",
    "offer_type",
    "offer_reward",
    "gender",
    "membership_year"
]

features = cont_vars+cat_vars

y_var = "successful_offer"

for i in cat_vars:
    modeling_data.loc[:, i] = modeling_data[i].astype("category")
```

```
data_split = Modeling_Helper.DataSplit("person", modeling_data, y_var, 0.8, 2021)
train_df, test_df = data_split.split_data()
```

```
scale_pos_weight_val = (len(train_df)-sum(train_df[y_var]))/sum(train_df[y_var])
```

3.3.2.3 Hyper-parameter Tuning

As we mentioned before, GBMs are highly customizable and there are many hyper-parameters that we can tune to achieve a better performance. Among many hyper-parameters, we will focus on the following ones:

- **learning_rate**: this can slow down or speed up the learning. Smaller values can decrease the influence of each individual tree and leaves space for future trees to improve the model.
- **n_estimators**: this determines the number of boosting iterations. The more trees the more accurate our model can be but at the same time, it will increase the training time and there will be a higher chance of over-fitting.
- **max_depth**: this controls the maximum depth of each trained tree. Smaller values can improve the training speed and address potential over-fitting.
- **num_leaves**: this will control the complexity of the model. It specifies the maximum number of leaves each weak learner has. Large values will improve the accuracy of the model but also can the chance of over-fitting.
- **colsample_bytree**: this specifies the fraction of features to consider at each iteration. It can be used to speed up training and to deal with over-fitting.
- **reg_alpha**: this applies L1 regularization and can address over-fitting.
- **reg_lambda**: this applies L2 regularization and can address over-fitting.
- **min_child_samples**: this determines the minimal number of data in one leaf and can be used to deal with over-fitting.
- **subsample**: this specifies the percentage of observations (which are selected randomly) used in each iteration. Smaller values can improve the training speed since. It can also improve generalization over unseen data.

Now that we specified our hyper-parameters of interest, the question is how to tune them. The important note here is that we should not use our test set to do so as it will cause data leakage. We can either create a separate validation set or we can leverage cross-validation. Since we don't have a very large dataset to split it into three sets, we rely on cross-validation. There are various methods to find the best hyper-parameters including grid search and random search. However, here we utilize a specific method called Bayesian Optimization. Bayesian Optimization is a probabilistic approach to find the minimum of any function that returns a particular metric, particularly effective in hyper-parameter tuning for complex machine learning models. It utilizes the Bayes theorem by setting a prior over the objective function and find the posterior function by combining it with evidence coming from the data. Unlike grid and random search methods, this approach keeps track of previous results and uses them to create a model associating hyper-parameters to a probability. This technique will move our prior understanding of the hyper-parameters and reshape them according to the structure of the dataset in hand ([17]). To implement this method, we use the Optuna module and create our own class called BayesianOpt (van be found in the Modeling_Helper). Below, we summarize the steps that we took to create this class:

- In the init function, specify the train set, features, response variable and the value for scale_pos_weight.
- In the call function, specify the hyper-parameters to tune, and the range to search within.
- Build the GBM model with the found hyper-parameters.
- Use cross-validation to find the score associated with the hyper-parameters.
- Return the best score.

We specify our score of interest to be the F1 score, meaning that the Bayesian optimization will try to find the best set of hyper-parameters to maximize the F1 score coming from the cross-validation on train set. We run our optimization for 100 iteration and will check the progress.

```
sampler = TPESampler(seed=2021)
study = optuna.create_study(sampler = sampler, direction="maximize")
study.optimize(Modeling_Helper.BayesianOpt(train_df, features, y_var,
    scale_pos_weight_val), n_trials=100)
```

```
[I 2021-11-29 15:36:48,448] Trial 0 finished with value: 0.11309781035250034 and␣
↪parameters: {'learning_rate': 0.0377391257196892, 'num_leaves': 45,␣
↪'colsample_bytree': 0.311157725382719, 'reg_alpha': 0.06980640530295552,␣
↪'reg_lambda': 4.915069954353206, 'max_depth': 5, 'min_child_samples': 148,␣
↪'subsample': 0.8023403430208751, 'n_estimators': 1493}. Best is trial 0 with␣
↪value: 0.11309781035250034.


...

[I 2021-11-29 16:34:10,349] Trial 99 finished with value: 0.13699639978045827 and␣
↪parameters: {'learning_rate': 0.002705308206219089, 'num_leaves': 39,␣
↪'colsample_bytree': 0.6026949384914291, 'reg_alpha': 0.06340267353850104,␣
↪'reg_lambda': 0.5855842656520097, 'max_depth': 20, 'min_child_samples': 457,␣
↪'subsample': 0.2032396945240998, 'n_estimators': 547}. Best is trial 61 with␣
↪value: 0.14505585908392027.
```

In the plot below, you can see the cumulative best F1 score in each iteration. We started from an F1 score of around 0.11 and we were able to improve it to 0.145 after 100 iterations. The best hyper-parameters found are as following:

- learning_rate: 0.00165
- n_estimators: 660
- max_depth: 25
- num_leaves: 26
- colsample_bytree: 0.555
- reg_alpha: 0.115
- reg_lambda: 1.403
- min_child_samples: 470

- subsample: 0.967

```
op_tuna_cummax = np.array(study.trials_dataframe()["value"].cummax())
print(f"Optuna best score = {op_tuna_cummax[-1]:.4f}")
fig = plt.figure(figsize = (10, 7))
plt.plot(op_tuna_cummax, "b", label="Bayesian Opt - Optuna")
plt.xlabel("Iteration")
plt.ylabel("F1 Score")
plt.title("Value of the cumulative best score");
plt.legend()
```

```
Optuna best score = 0.1451
```

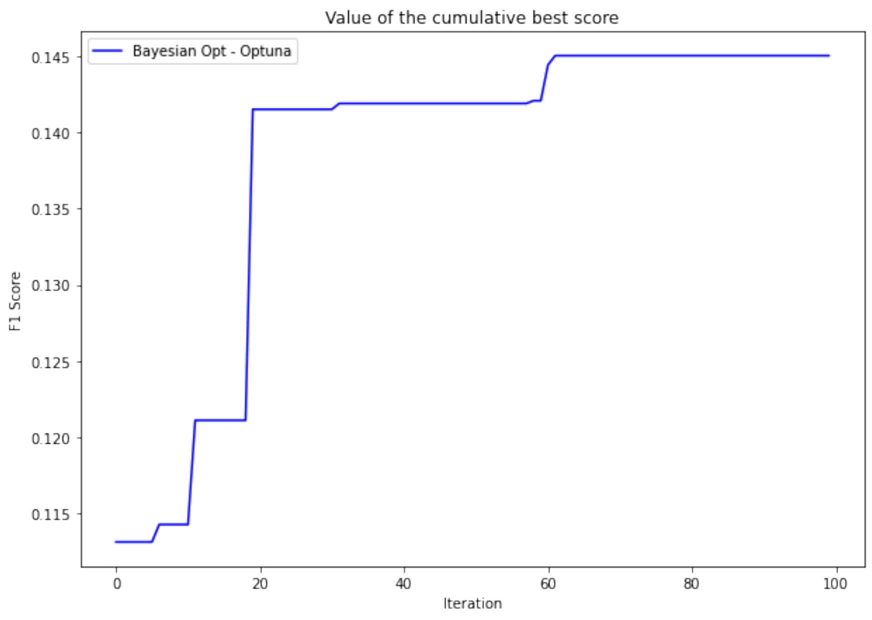


Figure 21

3.3.3 GBM: Final Model

Now, we take the best hyper-parameters found from the Bayesian Optimization and build our final model and assess the performance on both train and test sets. As shown below, this model achieved an accuracy of 0.92, precision of 0.11, recall of 0.24, and F1 score of 0.15 on the train set and on the test set these numbers are 0.92, 0.11, 0.24, and 0.15. The train set's AUC is 0.8 and for test set, it is 0.75.

```
best_params = study.best_params
best_params
```

```
{'learning_rate': 0.001651540420257919,
 'num_leaves': 26,
 'colsample_bytree': 0.5546771722017886,
 'reg_alpha': 0.11511983596488812,
 'reg_lambda': 1.402881028322646,
 'max_depth': 25,
          'min_child_samples': 470,
          'subsample': 0.966533554147176,
          'n_estimators': 660}
```

```
train_data = lgb.Dataset(train_df[features], label=train_df[y_var],free_raw_data=
    False)
test_data = lgb.Dataset(test_df[features], label=test_df[y_var], free_raw_data=
    False)
```

```
params = {"objective": "binary", "metric": ["binary_logloss", "binary_error"], "
    num_threads":-1, "seed":2021, "verbose": -1,

            "learning_rate":best_params["learning_rate"],
            "num_leaves":best_params["num_leaves"],
            "colsample_bytree": best_params["colsample_bytree"],
            "reg_alpha":best_params["reg_alpha"],
            "reg_lambda": best_params["reg_lambda"],
            "max_depth": best_params["max_depth"],
            "min_child_samples": best_params["min_child_samples"],
            "subsample": best_params["subsample"],
            "n_estimators": best_params["n_estimators"],

            "scale_pos_weight":scale_pos_weight_val
         }

gbm_model = lgb.train(params,
            train_data,
            feature_name=features,
            categorical_feature=cat_vars,
            verbose_eval=-1,
                     )
```

```
perf_analysis = Modeling_Helper.PerformanceAnalysis(gbm_model, train_df, features,
     y_var, 'Train Set', prob = False)
perf_analysis.perf_analysis()
perf_analysis = Modeling_Helper.PerformanceAnalysis(gbm_model, test_df, features,
    y_var, 'Test Set', prob = False)
perf_analysis.perf_analysis()
```

```
************** Performance: Train Set ****************

Accuracy: 0.92
Precision: 0.11
Recall: 0.24
F1: 0.15
```

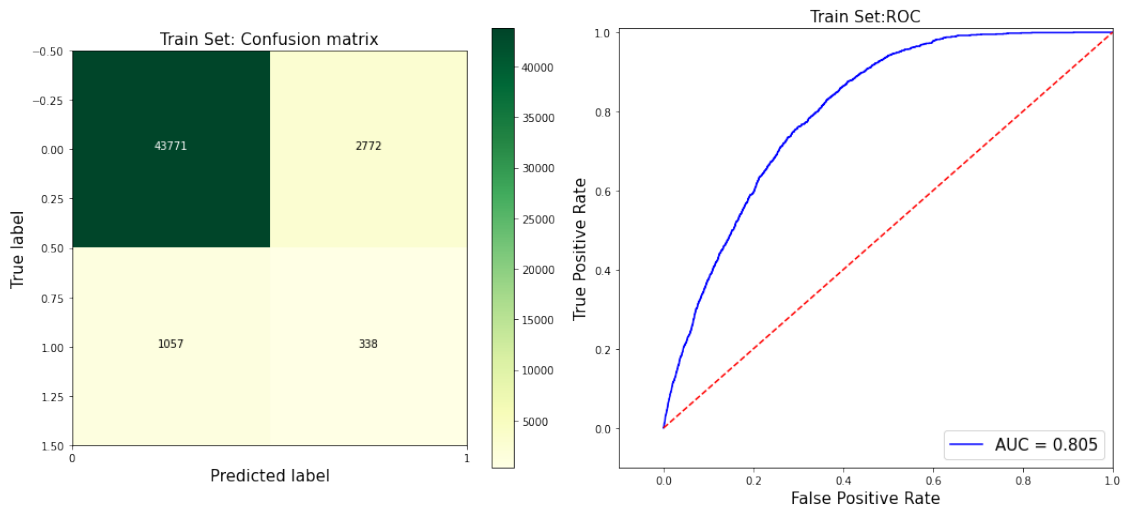


Figure 22

```
************** Performance: Test Set ****************

Accuracy: 0.92
Precision: 0.11
Recall: 0.24
F1: 0.15
```

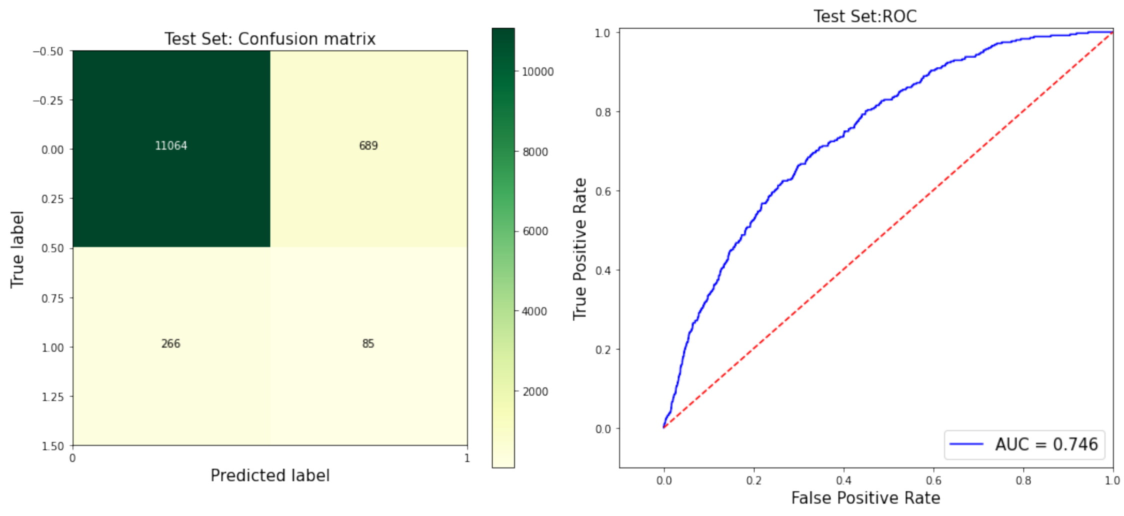


Figure 23

Let's see how much we were able to improve our baseline model (logistic regression). We were able to improve our metric of interest (F1 score) by 50%, along with the AUC, accuracy, precision, and recall. Another interesting achievement here is that we were able to address a lot of the over-fitting that we observed in our logistic model and now the performance metrics in our train and test set are close to each other.

```
performance_dict = {"logistic":{"auc": 0.695, "accuracy": 0.63, "precision": 0.05,
    "recall": 0.67, "f1_score": 0.1}, "gbm": {"auc": 0.746, "accuracy": 0.92, "
    precision": 0.11, "recall": 0.24, "f1_score": 0.15}}

Modeling_Helper.performance_comparison(performance_dict)
```

```
Improvement in AUC:  7.34 %
Improvement in Accuracy:  46.03 %
Improvement in Precision:  120.0 %
Improvement in Recall:  -64.18 %
Improvement in F1 Score:  50.0 %
```

Taking another look at the SHAP values, we observe that some of the hard-encoded interaction terms that we defined are among the most important features. We save this model so can be used later on for scoring purposes.

```
explainer = shap.TreeExplainer(gbm_model)
shap_values = explainer.shap_values(train_df[features])

shap.summary_plot(shap_values[1], train_df[features], feature_names = features,
    max_display = 30)
```

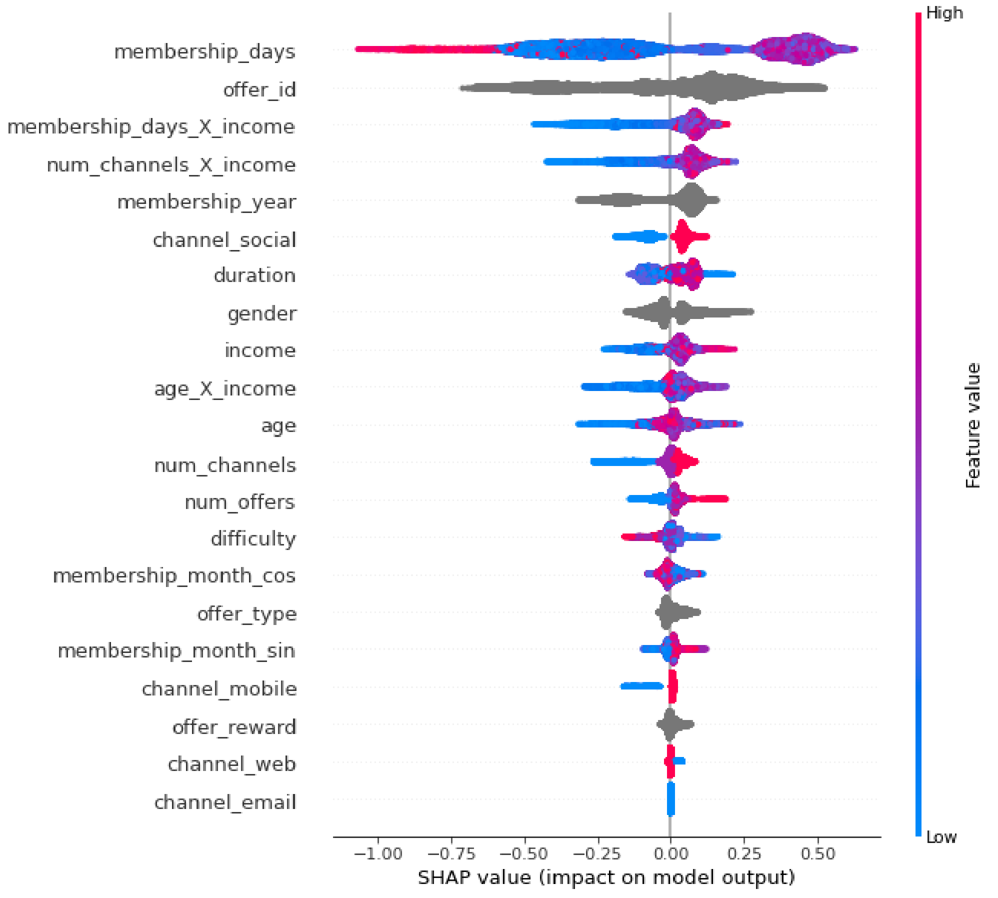


Figure 24

```python
joblib.dump(gbm_model, "gbm_model.pkl")
```

3.4 Uplift Modeling

We mentioned that we build an S-Learner as our uplift model. Luckily, we have already done step 1 of the S-Learner algorithm in the previous step the where we built our predictive model. Let's see whether we can use the same model to estimate uplift and accordingly build a recommendation engine. We now prepare the data so we can use the GBM model that we already saved, split the data into train and test as before and read the model object.

```python
data_prep = Modeling_Helper.DataPrep()
data_prep.prep_data_gbm()
modeling_data = data_prep.modeling_data
y_var = data_prep.y_var
modeling_data = modeling_data.sort_values(by = ["person", "time_received"]).
    reset_index(drop = True)
modeling_data["num_offers"] = modeling_data.groupby("person")["time_received"].
    transform(lambda group: group.astype("category").cat.codes + 1)
modeling_data["membership_month_sin"] = np.sin(2 * np.pi * modeling_data["
    membership_month"].astype(int) / 12)
modeling_data["membership_month_cos"] = np.cos(2 * np.pi * modeling_data["
    membership_month"].astype(int) / 12)
modeling_data["membership_days_X_income"] = modeling_data["membership_days"]*
    modeling_data["income"]
modeling_data["num_channels_X_income"] = modeling_data["num_channels"]*
    modeling_data["income"]
modeling_data["age_X_income"] = modeling_data["age"]*modeling_data["income"]
```

```python
cont_vars = [
    "duration",
    "num_channels",
    "channel_email",
    "channel_mobile",
    "channel_social",
    "channel_web",
    "age",
    "income",
    "membership_days",
    "num_offers",
    "membership_days_X_income",
    "num_channels_X_income",
    "age_X_income",
    "membership_month_sin",
    "membership_month_cos",
    "difficulty",

]

cat_vars = [
    "offer_id",
    "offer_type",
    "offer_reward",
    "gender",
    "membership_year"
]
features = cont_vars+cat_vars

```

```python
y_var = "successful_offer"

for i in cat_vars:
    modeling_data.loc[:, i] = modeling_data[i].astype("category")

```

```python
gbm_model = joblib.load("./model/gbm_model.pkl")
```

```python
data_split = Modeling_Helper.DataSplit("person", modeling_data, y_var, 0.8, 2021)
train_df, test_df = data_split.split_data()
```

3.4.1 Predict Uplift: S-Learner Model

In the Modeling_Helper module, we created a class called FindUplift which we use the predict the uplift for each customer. This class has two main components:

1. Find the uplift:

   1.1. Predict the outcome if the offer was informational. Since we have two informational offers, we make two predictions and use the maximum probability. This will be our prediction if the customer was assigned to control.

   1.2. For all other offer types, similarly predict the outcome.

   1.3. Find the uplift of each offer type by subtracting probabilities found in the previous step from the probabilities in the first step.

   1.4. Find the maximum uplift across offer types. This will be our final best uplift likely to be achieved from treating a customer.

   1.5. Find the offer type which is associated with the maximum uplift.

2. Create the uplift curve:

   2.1. Rank the customers by their predicted uplift (x-axis). Find the cumulative number of successful offers in the treatment group (scaled by the cumulative treatment size) minus the cumulative number of successful offers in the control group (scaled by the cumulative control size).

   2.2. Redo the previous step but this time randomly order the customers. This will find the uplift curve in case of randomly sending offers to customers.

   2.3. Calculate the area under the uplift curve for the model and for the random assignment.

First, we predict uplift in our train set. In the uplift plot below, the blue line shows the cumulative gain if we were using this model's recommendations to send out offers and the orange line shows the same thing but in case of randomly sending out offers. We can see that the our model outperforms the random assignment. Putting into numbers, the AUUC of the model is 545, while for the random assignment it is only equal to 288 (95% relative improvement).

```python
uplift_train = Modeling_Helper.FindUplift(gbm_model, train_df)
uplift_train.find_uplift()
uplift_train.calculate_auuc()
```

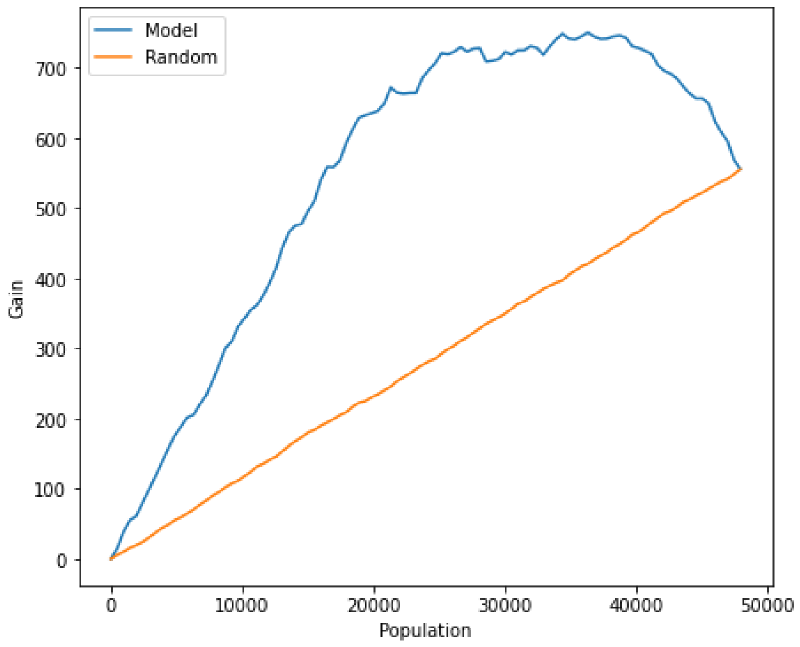


Figure 25

```
Model AUUC: 544.98    Random AUUC: 278.94
```

Let's see how our model performs on the test set. As shown below, our model achieves a better AUUC (152) compared to the random assignment (96), by providing 58% relative improvement.

```
uplift_test = Modeling_Helper.FindUplift(gbm_model, test_df)
uplift_test.find_uplift()
uplift_test.calculate_auuc()
```

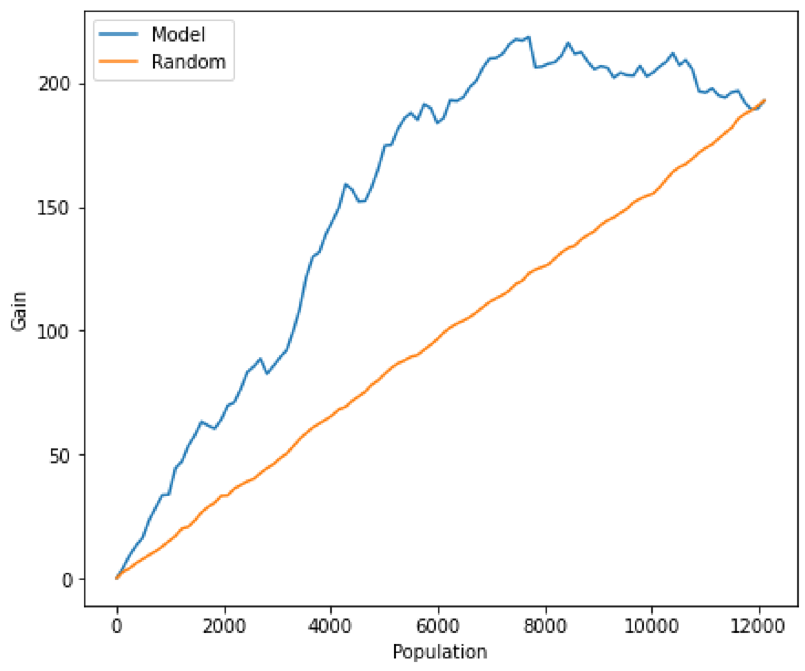


Figure 26

```
Model AUUC: 152.47    Random AUUC: 96.27
```

3.4.2 Predict Uplift: Baseline

While we showed that our model outperforms a random assignment, the question remaining is that whether it also performs better than the strategy upon which the offers have been sent out in the first place. To make this comparison, we use the find_baseline_uplift function in the FindUplift class and similar to before calculate the AUUC. In the plots below, we can observe that the baseline strategy performs better than random assignment in both train and test set. However, it seems that the baseline strategy cannot outperform our model's recommendations. On the train set, our model results in 17% relative improvement in AUUC, and on the test set this number is equal to 12%.

```
uplift_train.find_baseline_uplift()
uplift_train.calculate_auuc(original_uplift = True)
```

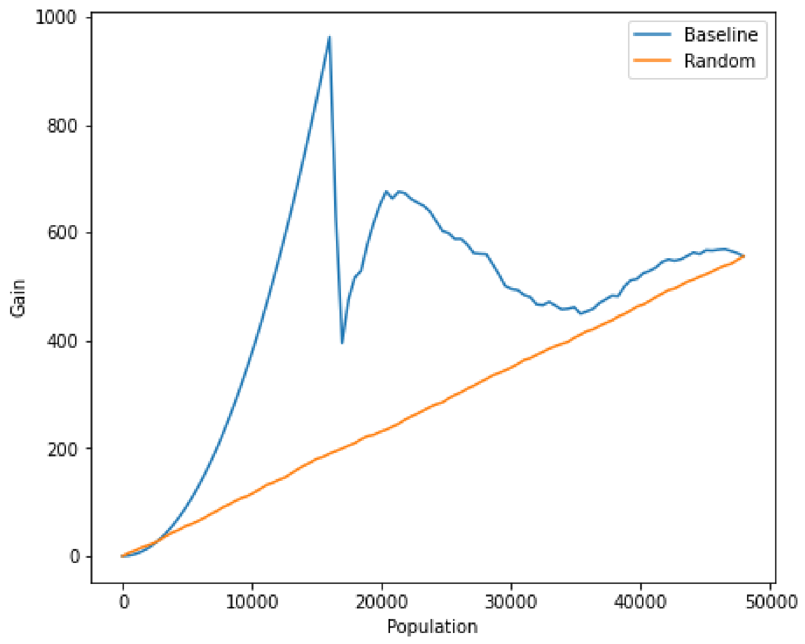


Figure 27

```
Baseline AUUC: 469.94    Random AUUC: 278.94
```

```
uplift_test.find_baseline_uplift()
uplift_test.calculate_auuc(original_uplift = True)
```

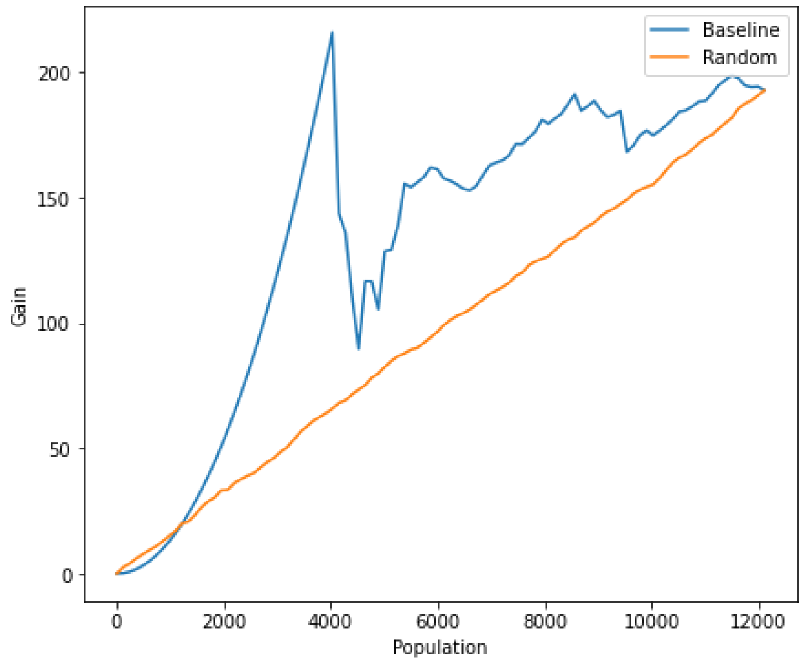


Figure 28

```
Baseline AUUC: 135.81    Random AUUC: 96.27
```

4 Results

4.1 Model Evaluation and Validation

The hyper-parameters of our GBM model were found using Bayesian optimization and the process is explained in detail in Section 3.3.2.3. Throughout our implementation, we evaluated our model as well. Here, we summarize the main points on model evaluation and validation. Starting with the GBM model that we used to predict the likelihood of an offer being completed successfully, we showed that our model outperforms the baseline model that we build. We showed that with our refined GBM model We have improved our metric of interest (F1 score) by 50%, along with the AUC, accuracy, and precision.

Model robustness is an important topic which is sometimes ignored which results in models failing to generate any positive impact. Unrobust models are mainly fitting the training data and not the underlying process, a.k.a over-fitting. To the test the robustness of our predictive model, we compared the performance metrics in the train set with the metrics in the test set. We showed that we indeed addressed the over-fitting that we observed in our logistic model and with the GBM model the performance metrics in our train and test set are close to each other (Section 3.3.3).

We later re-purposed our predictive model into an S-Learner uplift model and evaluated it by utilizing the cumulative gain metric. We showed that our model outperforms both the random assignment (58% relative increase) and existing assignment (12% relative increase).

4.2 Justification

Our predictive model yielded in a very high accuracy (0.92). However, this does not necessarily mean that we have a good classifier in hand. It is because that we are dealing with an imbalanced dataset and even a naive model predicting class zero for all observation can achieve a high accuracy. The AUC of our model is also relatively high (0.75), but even AUC has its own disadvantages. Mainly, it does not take into account the predicted probabilities but only the ranking of the predictions. Two models can have similar AUCs but the ability of separating observations into class zero and one can be quite different in each model [18]. In case of imbalanced datasets, AUC becomes less informative. Having said that, we decided to find the model with provides the best F1 Score. While the F1 score in the GBM model is higher than the baseline model, its absolute values still seems to be low (0.15). While we know that the higher the F1 Score the better, its absolute value depends on the amount of signal and information present in our data. In Section 2.1, we saw that particularly for the users' attributes, we have limited information (only age, gender, income, and membership date).

5 Conclusion

In this project, we used the synthetic data from Starbucks rewards mobile app to understand how customers are reacting to different types of offers that are sent to them. In the first step, we investigated the data visually to understand the underlying distributions in the dataset. We devised a definition for our response variable which was a binary indicator of whether an offer sent to customer has been successful or not. Next, we built a GBM model to predict whether an offer would be successfully completed or not. We evaluated the performance of our model by comparing it with our benchmark model which was a logistic regression. Finally, we re-purposed the GBM model into an uplift model which can act as a recommendation system. The uplift model can determine which offer is the best to send to a particular customer. Using the AUUC metric, we showed that our recommendation engine outperforms both the random assignment and the existing strategy depicted within the data. Below, we enumerate the main challenges faced throughout this project and list some of the areas for improvement.

5.1 Reflection

The main challenge in this project was the data preparation and processing step, since there was not unique identifier for each offer following its course from the time it is sent out until the expiry date. We had to manipulate the datasets from various perspective to document the course of each offer sent. Another challenge that we faced was in defining the response variable. Some offers are documented as completed, but it doesn't necessarily mean that a particular offer was successfully completed. We had to define numerous logical checks to properly define our response variable.

In building our uplift model, we encountered another challenge since there was no clear control (untreated) group in our dataset. Accordingly, we had to make some assumptions and use the customers who received informational offers as our control population.

5.2 Improvement

While we attempted to address questions raised in our problem statement using various approaches, there is always room for improvement. Below, we list some of the potential improvements that can add value to our analysis:

- Make use of the information potentially buried in the transaction amount. This can be an analysis by its own where one can build a model to predict that in case of an offer becoming successful, how much a customer may spend. This can provide useful insights for the stakeholders.
- Our variable space in the project was limited to a handful features, we ran feature engineering to create new potentially useful variables. The feature engineering piece though can be improved. For instance, one can use the SHAP module to find significant interactions among the variables and hard-encode more interaction terms.
- In our predictive model, while we mainly focused on building a GBM model, there are many other machine learning models that one can investigate including Random Forests, XgBoosts, Support Vector Machines and Artificial Neural Networks.
- In our uplift modeling, we re-purposed our predictive model to estimate uplifts. There are however numerous other algorithms that one can investigate. Beside other meta-learners such as T-Learners and X-Learners, there are tree-based methods such as Causal Random Forests which model the uplift directly.