Starbucks Rewards: Predicting Consumer Responses

Project Overview

This project is an experiment that determines how do we take this experimental data and discover what are the offers that excite people? So, the Capstone Project is about discovering what is the most valuable offer there is, not just for the customers as a whole but at an individual personal level.

Link to an academic paper where machine learning was applied to this type of problem: http://ceur-ws.org/Vol-3026/paper18.pdf

Problem Statement

To predict if someone will reply to an offer, transaction data and demographic information must be combined. GitHub
customers will also have access to the data and in the Data Sets section of the proposal, it is clearly stated that the repository and data sets are available.

Note that every offer will have an expiration date. For example, a buy one gets one free offer might be valid for only a certain number of days. You'll see in the data set that informational offers have a validity period even though these advertisements are providing information about a product; for example, if an informational offer has 10 days of validity, you can assume that after receiving the advertisement the customer will be aware of the offer for 10 days.

Metrics

This is a multi-class classification problem so the key metric we will use is f1-score. The simulating dataset only has one product, while Starbucks offers dozens of products. Therefore, this data set is a simplified version of the real Starbucks app.

Data Exploration

Dataset details and inputs

Starbucks rewards mobile app customers are simulated with this simulated data. An offer could be an advertisement for a drink, a discount, or a buy one get one free deal. Some customers may not be sent offers during certain days or weeks and not all customers will receive the same offer. That is the challenge to solve using this dataset, who gets what offer?

Also, this transactional data contains a record for each offer that a customer receives as well as a record for when a customer sees the offer. There are also records for when a customer completes the offer that is viewed.

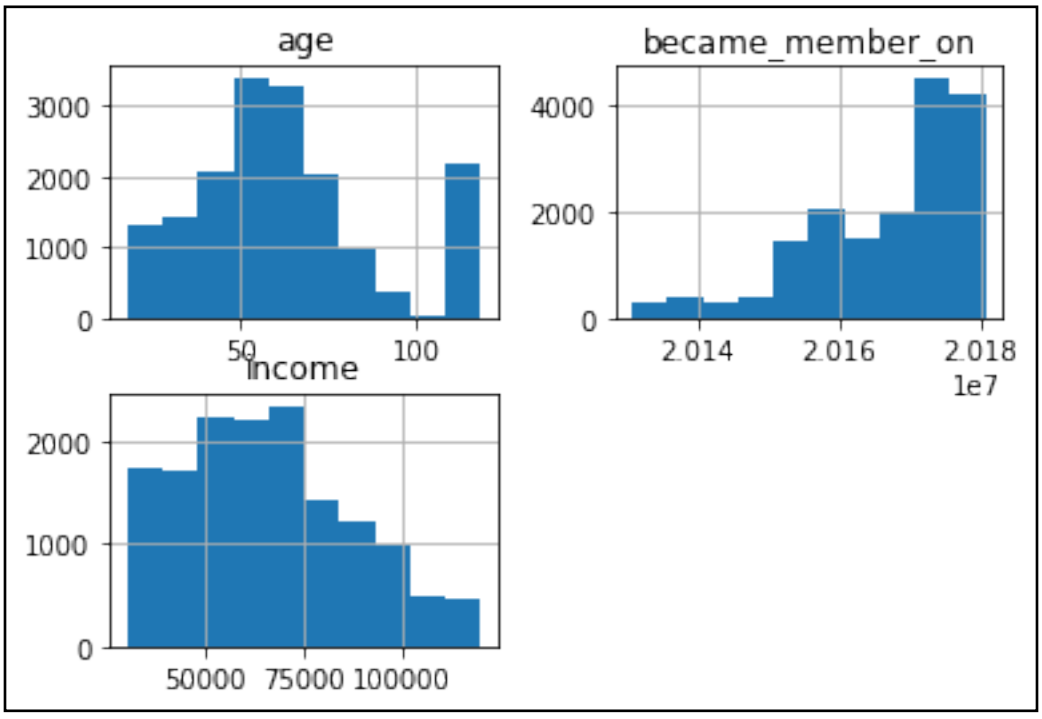
The data is in three files:

portfolio.json
* id (string) - offer id
* offer_type (string) - a type of offer ie BOGO, discount, informational
* difficulty (int) - the minimum required to spend to complete an offer
* reward (int) - the reward is given for completing an offer
* duration (int) - time for the 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

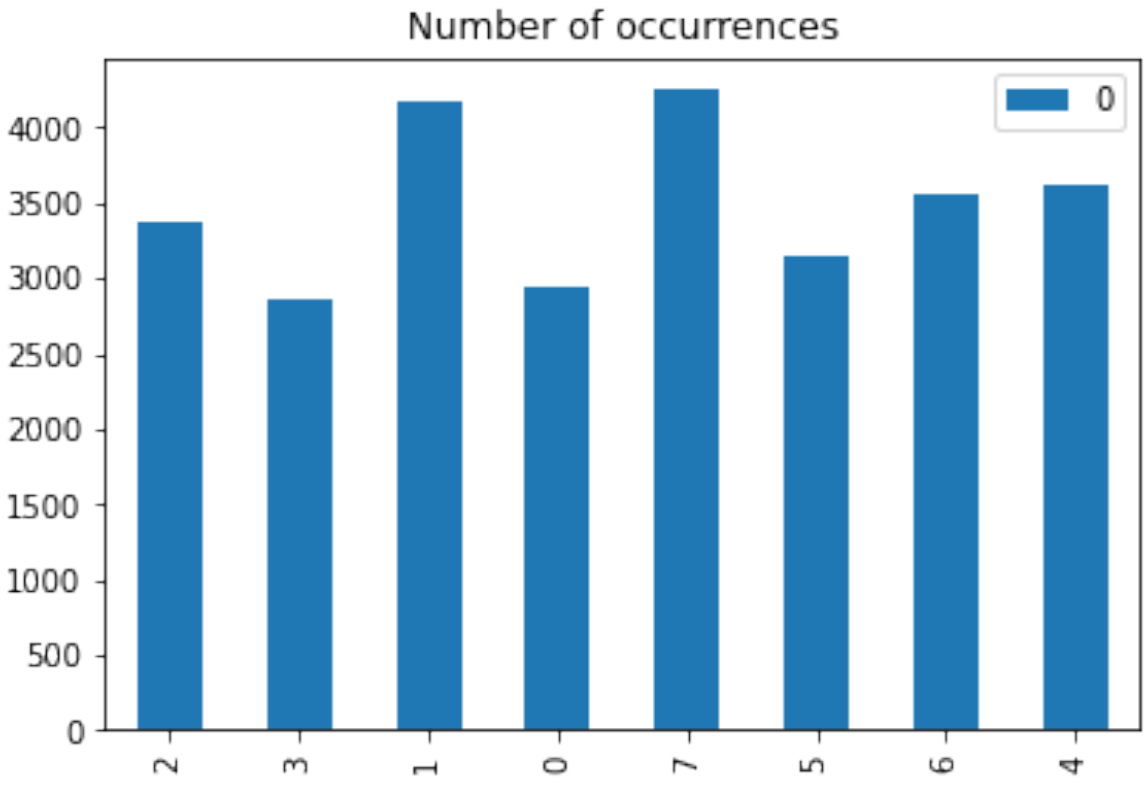
transcript.json
* event (str) - record description (ie transaction, offer received, offer viewed, etc.)
* person (str) - customer id
* time (int) - time in hours since the start of the test. The data begins at time t=0
* value - (dict of strings) - either an offer id or transaction amount depending on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



Exploratory Visualization

Notice we only have 8 offers from the previous 10 in the `portfolio.json` file. This is because two are merely informational

Class distributions of offers completed (Exploratory Visualization)



Algorithms and Techniques

Using Extreme Gradient Boosting, we will analyze the attributes of customers and/or offers to create customer classifications.

XGBoost or Extreme Gradient Boosting can be used for classification. It's referred to as an All-in-One algorithm.

There are many advantages of XGBoost, some of them are mentioned below:

- It is Highly Flexible.
- It supports regularization.
- It's designed to handle missing data with its built-in features.
- It works well in small, medium, and large datasets.

In this problem, we considered the hyperparameters' max depth, eta, and the number of rounds.

{'eta': '0.2102253387393228', 'max_depth': '4', 'num_round': '"50"'}

Benchmark

We will be using multiclass logistic regression against which we can benchmark.

Advantages

- In some instances, Logistic Regression delivers great training efficiency and is one of the easiest machine learning algorithms to implement. Due to these reasons, training a model with this algorithm does not require a lot of computing power.

- Based on these predicted parameters (trained weights), it can be inferred how important each feature is. It is also indicated whether the association is positive or negative. Therefore, logistic regression can be used to find out the relationships between the attributes.

- Unlike decision trees or support vector machines, this algorithm makes updating models easy. Stochastic gradient descent can be used to update models.

- In addition to classification results, Logistic Regression produces well-calibrated probabilities. Unlike models that only predict the final classification, Logistic Regression produces well-calibrated probabilities. The inference is made about which training examples are more correct for the formulated problem if one has a 95% probability and another has a 55% probability.

In our case, we defined the model below and obtained an f1-score below

LogisticRegression(max_iter=10000, multi_class='multinomial', solver='saga')

f1-score: 0.2048716783909761

Data Preprocessing

We will handle missing values in the **genders** column in the `profile.json` data set by deleting rows

Also will handle missing values in the **income** column in `data` by deleting rows.

Pros:
Complete removal of data with missing values results in a robust and highly accurate model
Deleting a particular row or a column with no specific information is better since it does not have a high weightage

Cons:
Loss of information and data

Works poorly if the percentage of missing values is high (say 30%), compared to the whole dataset

Note: We only consider customers who received offers.

1. Next, we split/explode a column of dictionaries into separate columns of the transcripts.json file
2. merge all three data sets
3. Creating additional features by changing DateTime into separate columns “year”, “month”, “day” respectively.
4. Drop the following columns
5. 'offer id’(We only consider customers who received offers) so we only look at ‘offer_id’ instead
6. 'event'(We only consider customers who received offers) so we remove this redundant column
7. ‘id’ we merged the three data sets so there is no need for this column.
8. ‘became_member_on’ we turned this column into three columns called month, year, and day.
9. ‘reward_y’, since we only consider offers received every one gets a reward. We are mostly concerned with which person completed which offer
10. 'reward_x', since we only consider offers received every one gets a reward. We are mostly concerned with which person completed which offer
11. ‘time’, this is for test purposes and does not give information about the customer
12. ‘offer_type' other columns would classify the offer type so can remove this column
13. Set target to be offer_id
14. Convert object type to avoid error
15. Then drop duplicates rows

Implementation

1. Upload to S3 bucket

2. Load test and training data for training and validation

```
def _get_train_data():
    train = pd.read_csv ('train.csv')
    return xgb.DMatrix(
        train.loc[:, train.columns != "target"], label=train["target"]
    )

def _get_test_data():
    test_data = pd.read_csv ('test.csv')
    df_test = xgb.DMatrix(
        test_data.loc[:, test_data.columns != "target"], label=test_data["target"]
    )
    return test_data, df_test
```

3. During training, we test the testing data which calculates the f1 score then save the model

```
def train():
    param = {"max_depth": 5, "eta": 0.1, "objective": "multi:softmax",
"num_class": 8}
    num_round = 50
    train_loader = _get_train_data()
    test_data, test_loader = _get_test_data()
    bst = xgb.train(param, train_loader, num_round)

    test(bst, test_loader, test_data)
    save_model(bst)


def test(model, test_loader, test_data):
    preds = model.predict(test_loader)
    score1=f1_score(test_data["target"], preds, average='weighted')
    print(f"f1-score:{score1}")
```

Refinement

If we drop all columns by ‘age’, ‘income’, ‘year’, ‘month’, and ‘day’ we get a relatively low f1 score so include other columns like duration and compare it to the benchmark we obtain an f1 score which is significantly higher.

Model Evaluation and Validation

We will use Hyperparameter Tuning since it finds the best version of a model by running many jobs that test a range of hyperparameters on your training and validation datasets

Justification

The final results compared to the benchmark result is higher by a large margin.

Therefore the final model and solution are significant enough to have adequately solved the problem.

{'MetricName': 'validation:f1', 'Value': 0.8800793290138245}

{'eta': '0.2102253387393228', 'max_depth': '4', 'num_round': '"50"'}