# Starbucks Capstone Project

## Overview

The project has its beginnings in the availability of data extracted from the Starbucks application that contains both demographic data, such as age, gender, income of customers and also transactional data relating to offers received, offers viewed, offers accepted.

With this data the objective is to use some machine learning model that is able to generate predictions about which offers are
generate predictions about which offers are accepted by a given customer (the actual data has been anonymized to avoid disclosure of personal information).
The data presented indicates that in the past the customer may have received one or more offers and it is desired to classify (i.e. compute the probability) that an offer received will result in a change in the order (acceptance of the offer) based on the characteristics of the customer in the provided dataset.

This problem has already been addressed numerous times and there are award-winning papers for such analysis, such as:

**Predicting Customer Churn: Extreme Gradient Boosting with Temporal Data First-place Entry for Customer Churn Challenge in WSDM Cup 2018 Bryan Gregory**
*(https://arxiv.org/pdf/1802.03396v1.pdf)*

Following the guidelines in the paper, we decided to create an XGBoost model to evaluate the predictive ability of the data provided.

In this report I will detail the 5 main stages of development which can be seen in code in the Jupyter Notebook:

-Step1: Data inspection, Data preprocessing, Exploratory Data Analisys and Data Storage on s3
-Step2: Baseline model training
-Step3: XGBoost: Hyperparameter Optimization and Training
-Step4: Model deployment: Endpoint Creation
-Step5: Model Inference: Invoking the endpoint to perform predictions

# Step 1: Data inspection, Data preprocessing, Data cleaning, Exploratory Data Analisys and Data Storage

## Data inspection

The data comes in three  json files that are located into the data directory of this project. After reading the files into pandas Dataframe I've inspected a bit each file.

Starbucks provides a description of the files I copy below:

The data is contained in three files:

- portfolio.json - containing offer ids and meta data about each offer (duration, type, etc.)
- profile.json - demographic data for each customer
- transcript.json - records for transactions, offers received, offers viewed, and offers completed

Here is the schema and explanation of each variable in the files:
**profile.json**
Rewards program users (17000 users x 5 fields)

- 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

**portfolio.json**
Offers sent during 30-day test period (10 offers x 6 fields)

- 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)

**transcript.json**
Event log (306648 events x 4 fields)

- 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

All three files have some features that relate one another.
transcript.json is the main file is the largest one that contains the main time series of transactions , and contains the "person" feature, that relates this file with the profile.json (that contains all the customer demographic data and a customer id). So we can combine those files in just one with as many rows as trasactions in the transcript.json. portfolio.json has a feature called id, which is the offer_id (as per Starbuks guide), and relates with the transcript.json "value" feature. The "value" feature is actually a dictionary that contains several features more, as "offer_id", "offer id" (a typo in the app?), and some more.

Using pandas we read these files into DataFrames and using pandas capbiliies we were able to create just one large DataFrame called "data".

We also look for outliers into the individual files, and actually we found some (in the “age” feature, for example), but we did not remove outliers since we observe several times that a lot of values of the big “data” DataFrame were missing.

## Data Preprocessing - Data Cleaning

So the first approach, after unifying the data, was to count the missing values of each feature and decide to eliminate the entire row from the data or, in some cases where the feature was almost empty, get rid of the whole feature.

We also removed all the ocurrencies of offers that were just transactions, because they do not afford any useful information for the problem.

Using these techniques we were able to kept a dataFrame with 167581 rows out of the 306534 rows we had intially, but with a dataFrame completely filled of data.
We were able to keep such a large percentage of the initial data since we used not only the offer_id contained in the transaction file, but also the ‘offer_id’ and ‘offer id’ columns from the dictioanary contained in the ‘value’ column.

## Exploratory Data Analysis

Then we proceed to perform the EDA of the data DataFrame, by ploting several histograms and we noticed that just by deleting rows with missing values or whole feature columns with too many nulls the outliers had disappeared.
We computed the data correlation of the DataFrame and we found that one of the features was actually 100% correlated with another one, so it was the same data and we drop that feature too.
Finally we check the gender to notice that the gender class was highly inbalanced, and to deal with that, and as a suggestion of the mentor, we changed the metric from Accuracy to f1-score.

Finally we had to split in 80/20% fashion the data to create a train.csv and test.csv data files with clean data.
We copied those files of data already processed into the data_processed directory of s3 bucket to be consumed later for model traininig and inference.

# Step2: Baseline model training

It is always convenient to use a simple baseline model in order to be able to compare with more sophisticated models later. The exercise of applying a baseline model is inexpensive in terms of time and resources. It is important to know that the baseline model can give bad results and not indicate that the data set has no predictive power.

Often a bad result in the baseline model indicates a problem in the data, wich let me to the point of revisiting the data, but no problems were found.

Also maybe that it has been mishandled, that there are missing data, or that any interpolations have been made incorrectly. Or perhaps the model requires normalization of the data if it has very different values.

Those facts were taken into account too.

It can also happen that the model is not able to model those data. In our case that seems to be the problem.

As we mentioned in the proposal we created a multiclass (Since we have 10 classes in our target variable, which is the offer_id) LogisticRegression model from sklearn as baseline and we trained it.
We then computed predictions and apply as metrics both the f1-score and the ROC-AUC-score.

Roc-auc is bad when is about 0.5 and good if it is near 1.

F1-score is good approaching 1 and bad approaching 0.

We notice that both were very low, so very poor predictive capability was found for the baseline:

roc-auc-score: 0.5134227452928022

f1-score: 0.04717431528542856

It is always useful to compare with another model to see how it behaves with respect to the first one so we proceed to the XGBoost.
The baseline results gives us a lot of space to improve using more powerfull models applied to the same data.

# Step3: XGBoost: Hyperparameter Optimization and Training

As a main modelization for the Starbucks dataset we choose to use XGBoost. The first time I've heard about this model was in a review some years ago that told that XGBoost was the most succesfully applied model to win Kaggle competitions.

In short, XGBoost is a gradient boosting decision tree implementation designed for speed and performance.

In long: https://medium.com/@pushkarmandot/how-exactly-xgboost-works-a320d9b8aeef

We used the scikit learn module XGBoost to create a tuner and provide a parameter range to accomplish a Hyperparameter optimization:

```
hyperparameter_ranges = {
    "max_depth": IntegerParameter(3, 10),
    "eta": ContinuousParameter(0.1, 0.8),
    "num_round" : CategoricalParameter([10, 30, 50, 100])
}
```

With these parameter range we obtained a set of best hyperparameters that we used then to compute the training of the model to be deployed.

{'max_depth': 4, 'eta': 0.3214635921477187, 'num_round': 30}

And these parameters performed very well, giving a f1-score of 1.
This number seems to be too good to be true, and the reason can be found by analyzing the data. We have in total around 170k rows of data, but those 170k transactions correspond to only 17k people. So the customers repeat and repeat with a mean of 10 times. So it is not surpising that similar transactions are located both in the train set as in the test set. So this surprinsing results give me the impression that the data that Starbucks possesses provides a good understanding of the preferences of each of its customers.

Once we got those hyperparameters we computed again the traingin process but without tuning the hyperparameters and once the model was trained we save the model to an s3 location.
We created an *xgb_estimator* from sagemaker.xgboost.estimator module and then deploy this object for the sake of creation of an endpoint.

# Step4: Model deployment: Endpoint Creation

Once we have the xgb_estimator object created and trained, we can easyly create an endpoint at sagemaker that can be accessed from our Jupyter Notebook in SageMaker, or, eventually, from a Lambda function to expose this predictor as a service.

The code to create the endpoint can be found in the notebook:

*predictor=xgb_estimator.deploy(instance_type="ml.m5.xlarge", initial_instance_count=1, endpoint_name=endpoint_name)*

The endpoint name can be chosen at this point by defining it in the *endpoint_name* variable, but usually the date-time is used in the name to be able to discern between serveral deployed endpoints.

It is pretty straightforward  to create one and the only consideration to be taken into account is that while the endpoint exists, it has resources allocated to it, and therefore it causes costs.
So only when in production one has to have an endpoint deployed.

Also the cost of the endpoint varies with the instance type chosen, in our case we used an *ml.m5.xlarge*, but that choosing should be based in both cost and necessity of performance.

# Step5: Model Inference: Invoking the endpoint to perform predictions

Finally we decides to test the endpoint invocation, which is as simple as executing this line:

```
def predict_from_csv(payload):
    print('predicted target: ',int(predictor.predict(payload).decode("utf-8")[1]))
```

In short predictor.predict is enought to get a prediction given a payload that is submitted to the endpoint.
We choose to create a function with this predictor to use it repeatedly in a loop to test some predicitions on the testing set. The results were excelent, as was expected due to the high f1-score obtained and can be seen below:

```
original test.csv row: ['2', '6', '2', '10', '7', '1', '2', '71', '20180418', '86000.0', '2']
payload to endpoint:  2,6,2,10,7,1,2,71,20180418,86000.0
expected target: 2
predicted target:  2
original test.csv row: ['2', '336', '2', '10', '10', '1', '0', '48', '20160926', '90000.0', '9']
payload to endpoint:  2,336,2,10,10,1,0,48,20160926,90000.0
expected target: 9
predicted target:  9
original test.csv row: ['0', '678', '2', '10', '7', '1', '0', '63', '20151024', '79000.0', '2']
payload to endpoint:  0,678,2,10,7,1,0,63,20151024,79000.0
expected target: 2
predicted target:  2
original test.csv row: ['2', '288', '0', '0', '4', '2', '1', '74', '20170827', '41000.0', '3']
payload to endpoint:  2,288,0,0,4,2,1,74,20170827,41000.0
expected target: 3
predicted target:  3
original test.csv row: ['0', '672', '5', '5', '5', '0', '1', '66', '20170305', '76000.0', '8']
payload to endpoint:  0,672,5,5,5,0,1,66,20170305,76000.0
expected target: 8
predicted target:  8
```

# Final words

The success in the results of the f1-score of 1 surprised me a lot as I mentioned above. But giving a second thought a very plausible reason can be found by analyzing the data.

With an approximate total of 170k rows of data, and  17k differnent customers, on average each customer appears 10 times in the file. The possible outcomes of the offer_id are just 10, so it is very likely that each customer case in the test set is present also in the traingin set. So this surprinsing results give me the impression that the data that Starbucks possesses provides a very good understanding of the preferences of each of its customers.