Starbucks Capstone Challenge – Project Report

1. Definition of the problem

Starbucks is one of the world's largest coffee-shop chains with ca. 35,000 locations in 84 countries around the globe[1]. Having started as a small coffee and tea selling shop back in 1971[2] it is now a market leader and a beloved neighborhood place for thousands of people. Therefore, it is possible to collect and analyze a huge amount of data to further develop and improve the company.

For Starbucks it would be a good possibility and well invested money to create ML-models for their customer behavior analysis. For myself it is not only an exciting and challenging area to improve my ML-skills but also a warm memory of my very first employer back in 2011. I think it's very symbolic that I'm starting my new career path in ML by learning from the data provided from the company that gave me my first job.

For this project Starbucks has provided simulated data on their mobile app usage. The data is simplified to only one product to make it more accessible, but it should be representative for the customer behavior in general. The data consists of three json-files which are:

- *profile.json*: Rewards program users (17,000 users x 5 fields)
    - gender: (categorical) M, F, O, or null
    - age: (numeric) missing value encoded as 118
    - id: (string/hash)
    - became_member_on: (date) format YYYYMMDD
    - income: (numeric)
- *portfolio.json:* Offers sent during 30-day test period (10 offers x 6 fields)
    - reward: (numeric) money awarded for the amount spent
    - channels: (list) web, email, mobile, social
    - difficulty: (numeric) money required to be spent to receive reward
    - duration: (numeric) time for offer to be open, in days
    - offer_type: (string) bogo, discount, informational
    - id: (string/hash)
- *transcript.json:* Event log (306,648 events x 4 fields)
    - person: (string/hash)
    - event: (string) offer received, offer viewed, transaction, offer completed
    - value: (dictionary) different values depending on event type:

    - offer id: (string/hash) not associated with any "transaction"
    - amount: (numeric) money spent in "transaction"
    - reward: (numeric) money gained from "offer completed"
- time: (numeric) hours after start of test

With this data provided one could try various ML-solutions, such as **clustering** for the groups of most active/most money spending/most offer-responsive customers or **classification** whether a customer would complete an offer or not or prediction via **regression** of how many offers a person would complete depending on their age, income, gender, time spent as a member and an amount of offers received. Despite the great interest on all of the named topics I have decided to choose the last (but not least!) option.

Since it is a regression task the best choices for evaluation metrics would be the **R**oot **M**ean **S**quared **E**rror (**RMSE**)[5] and the Coefficient of Determination[6] R^2. The goal of the project is to now build a regressor model which should be able to predict how many offers a customer would complete aka which offer a customer would be most attracted to.

2. Analysis of the data

As usual, the very first step after uploading the data is the **E**xploratory **D**ata **A**nalysis (EDA). As already mentioned above, there are three datasets provided: *portfolio, profile* and *transcript*:

```
portfolio
```

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

```
profile
```

	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
...	...	...	...	...	...
16995	F	45	6d5f3a774f3d4714ab0c092238f3a1d7	20180604	54000.0
16996	M	61	2cb4f97358b841b9a9773a7aa05a9d77	20180713	72000.0
16997	M	49	01d26f638c274aa0b965d24cefe3183f	20170126	73000.0
16998	F	83	9dc1421481194dcd9400aec7c9ae6366	20160307	50000.0
16999	F	62	e4052622e5ba45a8b96b59aba68cf068	20170722	82000.0

17000 rows × 5 columns

```
transcript
```

	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
...	...	...	...	...
306529	b3a1272bc9904337b331bf348c3e8c17	transaction	{'amount': 1.5899999999999999}	714
306530	68213b08d99a4ae1b0dcb72aebd9aa35	transaction	{'amount': 9.53}	714
306531	a00058cf10334a308c68e7631c529907	transaction	{'amount': 3.61}	714
306532	76ddbd6576844afe811f1a3c0fbb5bec	transaction	{'amount': 3.5300000000000002}	714
306533	c02b10e8752c4d8e9b73f918558531f7	transaction	{'amount': 4.05}	714

306534 rows × 4 columns

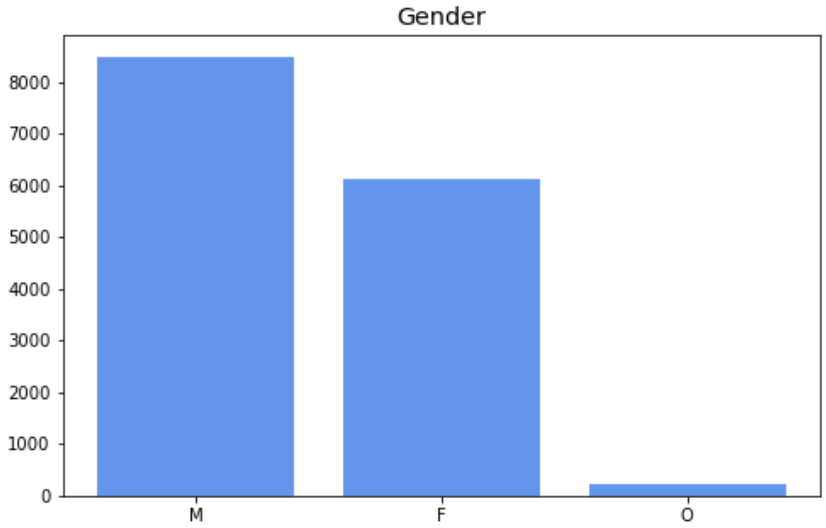
Only *profile*-table had missing values; 2,175 NaNs in columns *gender* and *income* out of 17,000 rows. Because the missing values were simultaneously in one row at a time (if *gender* was missing, *income* was missing too and vice versa) it was easy to drop these rows. The resulting *profile* table then had 14,825 rows. After completing this step, I had a look on the value counts and the descriptions of tables:

- Gender

```
profile['gender'].value_counts()
```

```
M    8484
F    6129
O     212
Name: gender, dtype: int64
```

Slightly more men then women.



- Age

```
profile['age'].value_counts()[0:20]
```

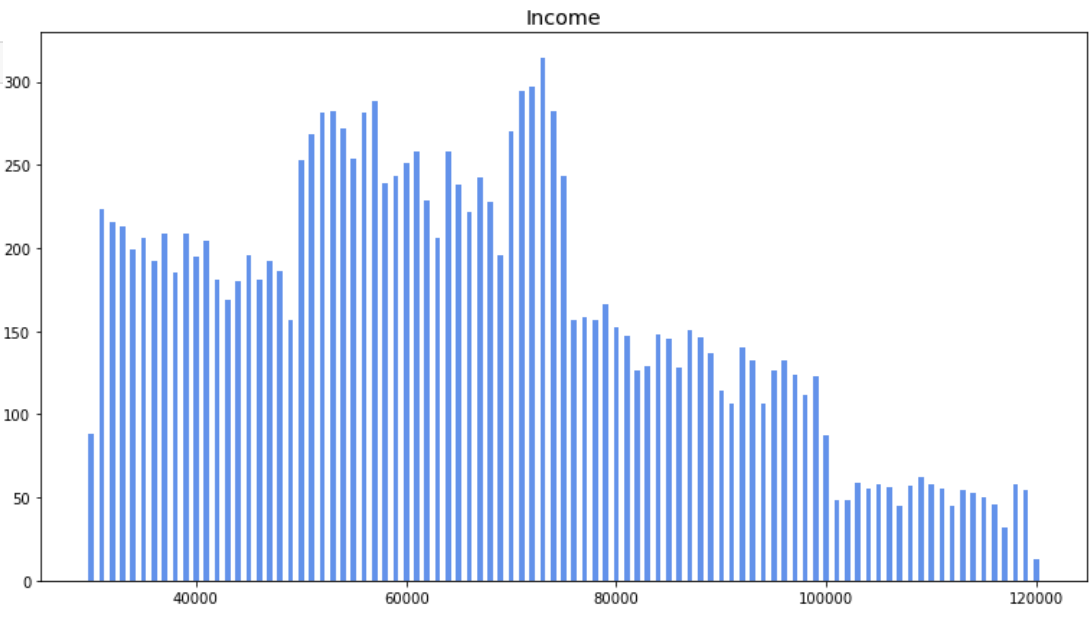
```
58    408
53    372
51    363
59    359
54    359
57    353
52    351
55    350
56    342
63    338
60    333
49    321
62    318
67    317
64    311
61    309
48    288
66    284
50    284
65    280
Name: age, dtype: int64
```

Seems like a lot of registered users are in their 50s and 60s. Interesting.

- Income

```
profile['income'].value_counts()
```

```
73000.0     314
72000.0     297
71000.0     294
57000.0     288
53000.0     282
           ...
116000.0     46
107000.0     45
112000.0     45
117000.0     32
120000.0     13
Name: income, Length: 91, dtype: int64
```



I have also deleted the ids from NaN-containing rows out of *transcript* and named the resulting table *transcript_cut*. Then I wanted to know how many customers received, viewed and completed an offer. It turned out that 14,820 out of 14,825 customers actually received at least one offer, 14,675 then viewed it and only 11,916 customers completed an offer. These numbers, however, have to be viewed with caution: It was not always a straight path received-viewed-completed. Sometimes customers completed offers without even receiving an offer or without viewing it. It was clear to me that the data needs to be processed further to adequately reflect the customers' behavior. In addition to that, one must almost always perform feature engineering to create new relevant information out of an existing one. This will be described in the section "3. Implementation".

Due to a relatively small size of the dataset it won't necessarily be the best idea to use Neuronal Networks as the first choice. It could, however, be a good idea to try a pre-trained NN or to use an AutoGluon[3] to find the best model across the hundreds of models. Both approaches, though, require larger computational resources and more time. For the purpose of this project I have therefore decided to choose a simple enough, yet representative model: Gradient Boosting Regressor from scikit-learn[4]. In order to have some reference and to be able to compare results I've had a look on the performance from other ensemble methods and from a base model for the task, linear regression. This will be discussed in the "4. Results" section.

Taking the size of the data into consideration and also the timeframe given to complete the project, which accidentally lies exactly in my university exams phase, I set the goal not to create a perfect, almost flawless model with the best scores possible but to build a representative model and to test the hypothesis, that is, if the solution approach I've chosen is applicable for the data provided and for the results wanted. If the relatively good performance could be achieved even on a simple model it would mean that the potential of this solution could be further improved with more time and with, probably, more complex models.

3. Implementation

The first part of this section will be covering feature engineering.

It has come to my attention that the column *time* in the *transcript*-table has an entity of hours. The column *duration* of the table *portfolio*, on the other hand,

is measured in days. To be able to better compare these two columns I created a new column *time in days* for the *transcript*-table by simply dividing by 24 and rounding up. For instance, 13 hours would be considered as one day.

As mentioned above, I needed to develop an approach to differentiate between transactions made by a customer in order to analyze the possible influence of offers. In order to that I derived three tables from the *transcript*-table: *received*, *viewed* and *completed* tables. It is easily made by projecting the *event*-column of the *transcript*-table to the corresponding values "offer received", "offer viewed" and "offer completed". For instance:

```
received = transcript_cut.loc[transcript_cut["event"]=="offer received"]
received
```

	person	event	value	time	time in days
0	78afa995795e4d85b5d9ceeca43f5fef	offer received	{'offer id': '9b98b8c7a33c4b65b9aebfe6a799e6d9'}	0	0
1	e2127556f4f64592b11af22de27a7932	offer received	{'offer id': '2906b810c7d4411798c6938adc9daaa5'}	0	0
2	389bc3fa690240e798340f5a15918d5c	offer received	{'offer id': 'f19421c1d4aa40978ebb69ca19b0e20d'}	0	0
3	2eeac8d8feae4a8cad5a6af0499a211d	offer received	{'offer id': '3f207df678b143eea3cee63160fa8bed'}	0	0
4	aa4862eba776480b8bb9c68455b8c2e1	offer received	{'offer id': '0b1e1539f2cc45b7b9fa7c272da2e1d7'}	0	0
...	...	...	...	...	...
229121	d087c473b4d247ccb0abfef59ba12b0e	offer received	{'offer id': 'ae264e3637204a6fb9bb56bc8210ddfd'}	576	24
229122	cb23b66c56f64b109d673d5e56574529	offer received	{'offer id': '2906b810c7d4411798c6938adc9daaa5'}	576	24
229123	6d5f3a774f3d4714ab0c092238f3a1d7	offer received	{'offer id': '2298d6c36e964ae4a3e7e9706d1fb8c2'}	576	24
229124	9dc1421481194dcd9400aec7c9ae6366	offer received	{'offer id': 'ae264e3637204a6fb9bb56bc8210ddfd'}	576	24
229125	e4052622e5ba45a8b96b59aba68cf068	offer received	{'offer id': '3f207df678b143eea3cee63160fa8bed'}	576	24

66501 rows × 5 columns

After doing that I thought of a way of differentiating the transactions. I reached the following conclusion: In general, we are only interested in finding out if a person viewed an offer and completed it because of that. I assumed that if a person viewed an offer and then completed the same offer within its duration, then it could happen due to the offer seen. I am aware that it is merely an assumption, but we'll have to go with it. So, the algorithm should do the following:

- Filter the *viewed* and *completed* tables for every *id* in *profile*, which means every unique customer;
- For every row in this *viewed* selection save the offer id, its duration in days, its index from 0 to 9 and the current day;
- Have a look if this id is also present in this *completed* selection. If yes, calculate how many days lie between viewed and completed. If this

number fits within the offer duration, count plus one for the offer index in the separate table *offers_completed.*

During the implementation of the described algorithm I've noticed that the offers 2 and 7, both informational offers, do have a duration, but neither of them is present in the *completed* table. In other words, it is impossible to say if a person completed an informational offer or not. So, I needed a workaround here. I've made a second assumption: If a customer received an informational offer and then completed **any** transaction within the offer duration, then it **might be** that the informational offer had an influence on that. This assumption is very watery, but there was no other way to try and find a correlation between informationals offers and transactions completed.

The code for the resulting algorithm can be seen here:

```
for person in offers_completed['id']:
    viewed_part = viewed.loc[viewed["person"]==person]
    completed_part = completed.loc[completed["person"]==person]
    transcript_cut_part = transcript_cut.loc[transcript_cut["person"]==person]

    for l in range(viewed_part.shape[0]):
        id_value = viewed_part['value'].iloc[l]
        offer_id = id_value["offer id"]
        days_to_complete = how_many_days(offer_id)
        viewed_time = viewed_part['time in days'].iloc[l]
        column = offers_id_dict[offer_id]

        if column == 2 or column == 7:
            index_list = list(transcript_cut_part.index)
            where_to_start = index_list.index(viewed_part['index'].iloc[l])
            for i in range(where_to_start+1, transcript_cut_part.shape[0]):
                if transcript_cut_part["event"].iloc[i]=="transaction":
                    difference = transcript_cut_part["time in days"].iloc[i]-viewed_time
                    if difference <= days_to_complete and difference >=0:
                        offers_completed[str(column)].loc[offers_completed['id']==person] +=1

        else:
            for i in range(completed_part.shape[0]):
                if completed_part['value'].iloc[i]["offer_id"] == offer_id:
                    difference = completed_part['time in days'].iloc[i]-viewed_time
                    if difference <= days_to_complete and difference >=0:
                        offers_completed[str(column)].loc[offers_completed['id']==person] +=1
```

When the loop terminated, the resulting table *offers_completed* had the following columns: id and 0 to 9, which describes the amount of each offer completed for each unique customer:

```
offers_completed
```

	id	0	1	2	3	4	5	6	7	8	9
0	0610b486422d4921ae7d2bf64640c50b	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	78afa995795e4d85b5d9ceeca43f5fef	1.0	0.0	0.0	3.0	0.0	0.0	0.0	2.0	0.0	0.0
2	e2127556f4f64592b11af22de27a7932	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0
3	389bc3fa690240e798340f5a15918d5c	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	2.0	1.0
4	2eeac8d8feae4a8cad5a6af0499a211d	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...
14820	6d5f3a774f3d4714ab0c092238f3a1d7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
14821	2cb4f97358b841b9a9773a7aa05a9d77	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
14822	01d26f638c274aa0b965d24cefe3183f	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.0	0.0	0.0
14823	9dc1421481194dcd9400aec7c9ae6366	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
14824	e4052622e5ba45a8b96b59aba68cf068	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0

14825 rows × 11 columns

The columns 0 to 9 are the target values for the regressor.

After creating targets there were still be a couple of features to engineer and a couple of features to modify:

The values of *gender* were recoded to categorical values 0, 1, 2 from M, F, O.

The column *became_member_on* was turned to type "datetime". Later on, it turned out that Gradient Boosting does not accept this type and I had to change it to numerical.

Other than this I created ten more columns named *received_n* with the 0 to 9 instead of n that represented the number of received offer each.

The new table named *data* contained all values from *offers_completed* plus ten new features. After completing this I had to scale all features so that they would have a standard distribution. This was done to the columns *age, income, became_member_on* and all ten engineered *received_n* features.

Finally, the *data* was separated into *X* and *y* tables with *X* having *age, income, gender, became_member_on* and all ten engineered *received_n* features and *y* having only the target 0 to 9 colums.

The *X* and *y* were then split into *X_train, X_test, y_train, y_test* with the test split of 10% and a random state 13. These are the training and testing data.

After completing data wrangling I could create and fit the selected estimator.

Fortunately, scikit-learn has already done the whole implementation for us. The only difference to a normal regressor in my case was that I wanted to have a multiple regressor, which means an output for all ten offer types together. That's why I used the Multi Output Regressor[7] from scikit-learn together with the Gradient Boosting regressor.

In order to have an overview of an output I initialized the model with default parameters:

```
gbr = GradientBoostingRegressor(random_state=13)

regressor = MultiOutputRegressor(gbr)
regressor.fit(X_train, y_train)
```

```
MultiOutputRegressor(estimator=GradientBoostingRegressor(random_state=13))
```

As the multioutput regressor would give a multiple output (obviously), the evaluation metric must be implemented accordingly. The metrics provided by scikit-learn.metrics[8] can be easily used for such implementation:

```
def multi_rmse(y_true, y_pred):
    predictions_T = np.transpose(y_pred)
    scores = []
    for column in y_true.columns:
        rmse = mean_squared_error(y_true[column], predictions_T[int(column)], squared=False)
        scores.append(rmse)
    return scores


def multi_r2(y_true, y_pred):
    predictions_T = np.transpose(y_pred)
    scores = []
    for column in y_true.columns:
        r2 = r2_score(y_true[column], predictions_T[int(column)])
        scores.append(r2)
    return scores
```

The predictions are a numpy-matrix, which means that the first array has the predictions for the first row of the *X_test* data. In order to compare across offers, not customers, I had to transpose the predictions. The results differ depending on a way of evaluation (row-vise or offer-vise), which will be discussed in the sections "4. Results" and "5. Conclusion".

4. Results

The results from the first estimator were as follows:

- Offer-wise (a score over one offer each for all customers, 10 outputs, then averaged):

    - RMSE: 0.4850205358376778

    - R^2: 0.3351661600912227

-Row-wise (a score over one customer each for ten offers, 14,823 outputs, then averaged):

    -RMSE: 0.4253019489330003

    -R^2 0.11526161734322998

One can see that the metrics are not outstanding and, more interesting, vary a lot between these two approaches. For the further evaluation I will use the first approach, offer-wise, to compare the model performance with the other estimators. Because both metrics for the first model with default parameters probably weren't the best, I performed Grid Search for the following hyperparameters:

    - learning_rate = [0.1, 0.01, 0.05, 0.2, 0.21, 0.22, 0.25, 0.3]

    - n_estimators = [50, 100, 300, 500, 1000]

    - max_depth = [1,2,3,4,5,8]

    - min_samples_split = [2,3,4]

The best hyperparameters turned out to be:

- learning_rate=0.2,
- n_ estimators =50,
- max_depth=4,
- min_samples_split=4

resulting with the:

    - RMSE: 0.48264993736504785,

    - R^2: 0.3415363107771194,

which is a minor improvement to the first estimator.

Then I compared the performance of the model fit on the best hyperparameters with the benchmark models for the task, such as Linear Regression[9], AdaBoost Regressor[10] and Random Forest Regressor[11]:

Linear Regression

```
linear_regression = LinearRegression()

linear_regressor = MultiOutputRegressor(linear_regression)
linear_regressor.fit(X_train, y_train)

predictions = linear_regressor.predict(X_test)
preds_rounded = predictions.round()
rmse = multi_rmse(y_test, preds_rounded)
avg_rmse = avg_score(rmse)
print("RMSE:",avg_rmse)

r2 = multi_r2(y_test, preds_rounded)
avg_r2 = avg_score(r2)
print("R2:", avg_r2)
```

```
RMSE: 0.5102244767533842
R2: 0.2623074934143525
```

AdaBoost

```
ada = AdaBoostRegressor(random_state=13, learning_rate=0.2, n_estimators=50)

regressor = MultiOutputRegressor(ada)
regressor.fit(X_train, y_train)

predictions = regressor.predict(X_test)
preds_rounded = predictions.round()
rmse = multi_rmse(y_test, preds_rounded)
avg_rmse = avg_score(rmse)
print("RMSE:",avg_rmse)

r2 = multi_r2(y_test, preds_rounded)
avg_r2 = avg_score(r2)
print("R2:", avg_r2)
```

```
RMSE: 0.5165034343325131
R2: 0.25237353766643433
```

Random Forest

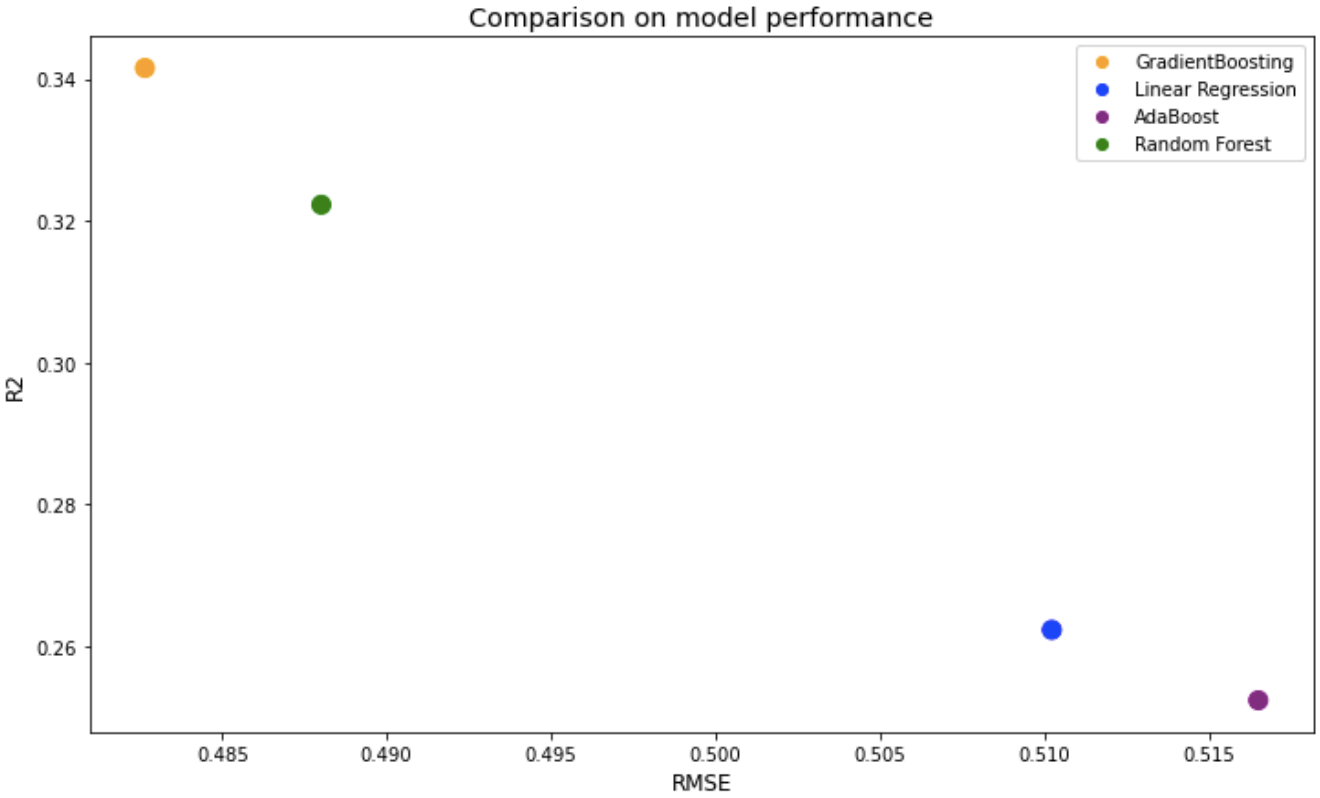
```
forest = RandomForestRegressor(random_state=13, max_features=10,n_estimators=50,
                               max_depth=4, min_samples_split=4)

regressor = MultiOutputRegressor(forest)
regressor.fit(X_train, y_train)

predictions = regressor.predict(X_test)
preds_rounded = predictions.round()
rmse = multi_rmse(y_test, preds_rounded)
avg_rmse = avg_score(rmse)
print("RMSE:",avg_rmse)
r2 = multi_r2(y_test, preds_rounded)
avg_r2 = avg_score(r2)
print("R2:", avg_r2)
```

```
RMSE: 0.4880133594894239
R2: 0.3222507322159441
```

It is clear that all models used for the comparison are similar with their predictions and performance; the three reference models, however, are slightly worse than the Gradient Boosting:



5. Conclusion

The first issue I'd like to address in the conclusion is the difference between two evaluations' approaches. Whereas the RMSE-scores lie relatively close to each other and could be considered tolerable for the task, the R^2 -scores are dramatically different in the first evaluation: 0.335 against 0.115. While 0.335 is still relatively acceptable, the 0.115 is not good enough. These differences are a result of the completely different number of predictions which then are averaged: 10 and 14,823, respectively. This means that the predictions for a certain offer are, in average, tolerable, while the predictions for a single customer could be anything between 1 and 0, or even worse. This makes the model not accurate enough and one cannot trust its decisions in order to predict in which number a particular person would respond for a particular number of offers.

Just to have a glance on how the same model would perform as a classifier, I slightly adjusted the target data to be binary (zeros are still zeros and everything else turned to one). I then fit the Gradient Boosting Classifier[12] to these data and implemented three metrics to evaluate a multiple output of the classifier:

```
gbc = GradientBoostingClassifier(random_state=13)
multi_gbc = MultiOutputClassifier(gbc)
multi_gbc.fit(X_train, y_cls_train)
```

```
MultiOutputClassifier(estimator=GradientBoostingClassifier(random_state=13))
```

```
def multi_accuracy(y_true, y_pred):
    predictions_T = np.transpose(y_pred)
    scores = []
    for column in y_true.columns:
        accuracy = accuracy_score(y_true[column], predictions_T[int(column)])
        scores.append(accuracy)
    return scores

def multi_f1(y_true, y_pred):
    predictions_T = np.transpose(y_pred)
    scores = []
    for column in y_true.columns:
        f1 = f1_score(y_true[column], predictions_T[int(column)])
        scores.append(f1)
    return scores

def multi_roc_auc(y_true, y_pred):
    predictions_T = np.transpose(y_pred)
    scores = []
    for column in y_true.columns:
        ra = roc_auc_score(y_true[column], predictions_T[int(column)])
        scores.append(ra)
    return scores
```

```
predictions = multi_gbc.predict(X_test)

accuracy = multi_accuracy(y_cls_test, predictions)
avg_acc = avg_score(accuracy)
avg_acc
```

```
0.8849629130141604
```

```
f1 = multi_f1(y_cls_test, predictions)
avg_f1 = avg_score(f1)
avg_f1
```

```
0.5872345998105015
```

```
roc_auc = multi_roc_auc(y_cls_test, predictions)
avg_roc_auc = avg_score(roc_auc)
avg_roc_auc
```

```
0.7554652747439754
```

As we can see here, the accuracy score[13] is quite good at 0.885, although the f1 score[14] could definitely be better. Simultaneously the area under the ROC-curve[15] has a value of 0.755, which means that the model does predict and does not just guess. Nevertheless, there is room for improvement here as well.

This small supplemental experiment shows that the data indeed has the potential to be used for a development of a solid ML-model for classification

or regression tasks. In my particular case I would suggest spending more time of searching for a better regression model, for instance using AutoGluon for Tabular Prediction. The larger amount of training data would also, most likely, result in a better model. I would recommend trying these two approaches for further improvement.

6. References

1. Starbucks World Map – Starbucks Stores in the World, accessed on 10 February 2023, <https://www.mappr.co/worlds-best-starbucks/>

2. About Us: Starbucks Coffee Company, accessed on 10 February 2023, < https://www.starbucks.com/about-us/>

3 AutoGluon: AutoML for Text, Image, Time Series, and Tabular Data, accessed on 10 February 2023, <https://auto.gluon.ai/stable/index.html>

4. sklearn.ensemble.GradientBoostingRegressor - scikit-learn 1.2.1 documentation, accessed on 10 February 2023, <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html#sklearn.ensemble.GradientBoostingRegressor>

5. Root-mean-square deviation – Wikipedia, accessed on 10 February 2023, <https://en.wikipedia.org/wiki/Root-mean-square_deviation>

6. Coefficient of determination – Wikipedia, accessed on 10 February 2023, <https://en.wikipedia.org/wiki/Coefficient_of_determination>

7. sklearn.multioutput.MultiOutputRegressor - scikit-learn 1.2.1 documentation, accessed on 10 February 2023, <https://scikit-learn.org/stable/modules/generated/sklearn.multioutput.MultiOutputRegressor.html>

8. API Reference - scikit-learn 1.2.1 documentation, accessed on 10 February 2023, <https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics>

9. sklearn.linear_model.LinearRegression - scikit-learn 1.2.1 documentation, accessed on 10 February 2023, <https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html>

10. sklearn.ensemble.AdaBoostRegressor - scikit-learn 1.2.1 documentation, accessed on 10 February 2023, <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostRegressor.html#sklearn.ensemble.AdaBoostRegressor>

11. sklearn.ensemble.RandomForestRegressor - scikit-learn 1.2.1 documentation, accessed on 10 February 2023, <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRRandomFo.html#sklearn.ensemble.RandomForestRegressor>

12. sklearn.ensemble.GradientBoostingClassifier - scikit-learn 1.2.1 documentation, accessed on 10 February 2023, <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier>

13. Classification: Accuracy | Machine Learning | Google Developers, accessed on 10 February 2023, <https://developers.google.com/machine-learning/crash-course/classification/accuracy?hl=en>

14. F-score - Wikipedia, accessed on 10 February 2023, <https://en.wikipedia.org/wiki/F-score>

15. Classification: ROC Curve and AUC | Machine Learning | Google Developers, accessed on 10 February 2023, <https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc?hl=en>