Machine Learning Group Project
Techscape - E-commerce

Joao Morais Costa (20211005), Qi Shi (20210981)

Alberto Parenti (20211304), Gaurav Luitel (20210979)

Master in Data Science and Advanced Analytics (NOVA IMS)

December 25, 2021

Content

1. Introduction

The following project aims to analyse and build a prediction model based on the data of a Portuguese online company “Tech-Scape”, which was founded in early 2020. The company was largely affected by the recent Coronavirus pandemic during its beginning phase, leading to a decrease in revenues, and gradually reactivating their business by April 2020. With the main goal of increasing their sales, the company was interested in analysing the online behaviour of their customers, by predicting which ones are more likely to buy their products depending on their online actions. To fulfil this goal, data from the company’s customer action and purchases from 2020, was analysed and several predictive models were built and tested. The following report aims to portray this process and its results.

2. Exploratory Data Analysis

The project’s data is composed by 2 datasets: “*train.csv*” and “test.csv”, only the first one was used for building the model. Therefore, no further mention of “*test.csv*” set will be done in the rest of this document. The dataset used to train and validate the model was created based on “*train.csv*”. This dataset consists of 17 features, including one target feature and 16 predictive features, with 9999 instances. The size of the dataset was checked using the pandas *.shape* method.

2.1. Missing data

By using the method *info()*, no missing data was found on the set.

2.2. Data types

By combining the methods *head()* and *info()*, we got a first preview of the set and its data types. Date is a particular type of data, and it’s not easy to use it directly in the analysis. So, in order to use it, we extracted from “*Date*”, the season (Trimester), month, week and day information. Regarding the year, there was only one year, so we decided not to extract it as a new feature.

Since date is a special type of data type, it is not easy to use in further analysis. From “Date”, we extracted the season (trimester), month, week and day information from the date (regarding year, there is only one year, so we will discard it). Based on data type, we divided further the data into numeric variables and categorical variables later. We dropped “*Access_ID*” as it provided no useful information.

2.3. Categorical Features

In our dataset, based on the characteristics of the features, we chose "*OS*", "*Browser*", "*Country*", "*Type_of_Traffic*", "*Type_of_Visitor*", "*Month*", "*Day*", "*Season*" and "*Week*" as the categorical features. Boxplots and *value_counts()* were used to show the distributions. From the figure (Appendix A, B), we can see that the distribution of some features, such as "*Browser*", was very imbalanced with some of the categories having very low frequencies. These categories should be grouped into other categories, however this couldn’t be done since there was no more description and/or information available from the various classifications. In "*Country*" and "*Week*", the distributions proportions of

the targets in different categories are very consistent, which indicates that these features may not provide us effective information for prediction. The solving process of these problems is showed in further processing.

2.4. Numerical Features

The numerical features were explored by graphical analysis using histograms (Appendix C). We noticed that all the numerical features have a right-skewed distribution, with most of the values surrounding zero.

2.5. Target Feature ("Buy")

Our target feature "*Buy*", was a binary numerical feature, in which 1 represents "*buying"* and 0 represents "*not buying"*. This feature is imbalanced, with 84.48% of *1* and 15.52% of *0*.

3. Data Pre-processing

3.1. Feature Grouping

In order to create new features and possibly combine information from the original ones, we decided to group some features that could be relevant. Our approach was to combine either by ratio and/or by common characteristics. In this step, six new features were created (Appendix D).

3.2. Feature transformation and scaling

In order to find the optimal combination for our dataset, different technics were tried. Since, many models have algorithms based on distance or weight calculation, to prevent the difference in weights between features, all the numerical features needed to be on a comparable scale. This is particularly important for models such as Logistic Regression, Support Vector Machine, Neural Network and K-nearest neighbours classifier. Because all the numerical features showed a right skewness, which is not ideal when performing further outlier detection and model fitting, our first step was to make them more like a Gaussian distribution, which may improve the model performance. By applying *Power Transformer* (Appendix E), using the *Yeo-Johnson* method, we transformed the distribution of the numerical features and then normalized them using the *MinMaxScaler* method [1, 2].

3.3. *Encoding Categorical Features*

During the machine learning (ML) process, most models can only perform calculations using numerical features. Therefore, it is a common strategy to apply coding on categorical features and use them as numerical. For instance, *Ordinal Encoder* is easy to apply to categories that have levels. It names different categories with sequential integers. However, this is not friendly to the categorical features without levels, since it will increase the distance or weight, between some categories, during the model calculation. One way to avoid this problem is by using the *One-hot Encoder*. This method creates a sparse matrix, which maps each category into a vector containing only 0 and 1, and therefore, balancing the distance between the categories. The combination of *Ordinal Encoder* and *One-hot Encoder* is a common strategy in ML. However, after some testing and research (not shown

on the code), we didn´t thought this was a good strategy for our project. An important step in ML is to reduce the number of features. However, *One-hot Encoder* will generate a very large sparse matrix for those categorical features containing more categories, which greatly increases the number of features. Consequently, this seriously increases the calculation and the model's complexity. Another problem with this method is when the numerical features are normalized (scaled from 0 to 1), the dummy variables generated only contain 0 and 1, which means only the maximum and minimum values would be calculated. It potentially increases the distance and weight of this feature. Therefore, we used different coding methods to process categorical variables.

In our project, we tested and implemented *Ordinal* and *Target Encoder.* For tree-based models, the data was encoded by *Ordinal Encoder*. For non-tree-based models we used *Target Encoder* (Appendix F), which is a supervised coding method that can effectively extract information through the target. This may cause the risk of model overfitting, which can be avoided by using cross-validation. There are no less than 15 methods for coding categorical features, including *Leave-One-Out, WOE Encoder*, etc., which could be useful encoders as well. [3] However, there is no perfect or one only strategy that fits all for coding categorical features, since each method has its advantages and disadvantages. In our project, it was impossible for us to try all the coding methods one by one. In our choice for *Target Encode,* we hoped to get more information directly from the target and as well because it has a package called *category_encoders* that can be easy and directly implemented. We recognize that if we had the opportunity, it would be beneficial to try and research more on other encoders in further studies.

3.4. *Outlier detection and handling*

Outliers can cause some models, such as Logistic Regression, to have data bias, which is "*a type of error in which certain elements of a dataset are more heavily weighted or more represented than others*" [4]. Therefore, it is very important to remove outliers. Also is important to mention is the fact that outliers in lower dimensions may not be considered as outliers anymore in higher dimensions. So, it is a good practice to use more than one method to detect outliers. After the feature transformation, scaling and categorical encoding, all metric features were subjected to an outlier analysis. Firstly, we did visualization using boxplots, and then implemented different outlier detection methods: the inter-quartile range (IQR) proximity rule, Isolation Forest and Local Outlier Factor (LOF). During visualization (Appendix G), some data points were observed far from the whiskers of the boxplots in several features, mainly "*FAQ_Pages*", "*FAQ_Duration*", "*GoogleAnalytics_Page_Value*" and "*FAQ Duration/Page*".

Each outlier detection method was applied independently, with nearly 35,2% of the original data being considered as outliers, even when filtering with an IQR range of 4. Isolation Forest (Appendix I) (with contamination argument set as default – "Auto") detected nearly 34,9% of outliers in the dataset. Since in LOF (Appendix H) we can define the parameter "contamination" (the proportion of outliers we want to be defined), it is not a good method to use independently. In order to detect outliers

more robustly, we combined all three methods together. By intersecting the results of these methods, still around 7,2% of the original data that was considered as outliers. This proportion was a little high to be considered as outliers. So, the data detected was not removed from the original dataset but rather identified and stored as different labelled dataset, to be used further during model implementation, according to the chosen algorithm (namely the ones sensitive to outliers).

4. Feature Selection

Feature selection, as one of the core parts of a ML process, aims to select the relevant subset of features for the construction of the ML model. Even if more features can provide more information, a model with reduced features is simpler and easier to interpret, less prone to errors during implementation, requires less training time and usually has less noise on the dataset. This is important since ML models can learn from noise which can lead to overfitting and reduced ability to generalize in other datasets [1].

In this step, we combined several methods for the feature selection, in order to have a more robust decision on the final set of features to implement in our models. The performance of the model is completely dependent on what data we "feed" it and different models require different data pre-processing processes. So, in our project, all feature selection methods used were coded in order to find two feature sets: one for tree-based models and another for non-tree-based models Before applying the wrapped and embedded methods for feature selection, the non-tree-based models needed to perform data transformation, normalization and categorical coding after splitting the training set and test set. While the tree model didn't require any special processing. Generally, more stable results could be achieved by using cross-validation during the wrapper and embedded methods, defined by the parameter *"cv",* but due to some technical problems (mentioned later in Section 6), we used the training and validation set split only once. In our feature selection workflow, first we used filter methods, which rely fundamentally on the characteristics of the data and do not involve any ML algorithm. Then, we used a wrapper method by applying selection models in order to find the optimal feature set. Finally, two embedded methods were used.

4.1. Filter

In this method, features are selected based on their scores in various statistical tests for their correlation with the taget variable.

4.1.1. Variance Evaluation

The highest variance was observed among "*GoogleAnalytics_PageValue*", "*FAQ_Pages*", "*FAQ_Duration*", and "*FAQ_Duration/Page*" with values around 0.15. The lowest variance was observed in *"OS*", *"Browser"*, *"Week"* and *"Country".* However, all variables are greater than zero. So, in this step we decided not to delete any features.

4.1.2. Chi-Square (for non-metric features)

Our non-metric features were "*Day*", "*Week*", "*Browser*", "*OS*", "*Type_of_Traffic*", "*Type_of_Visitor*", "*Month*", and "*Season*". The features to be removed are determined by statistical methods. By defining the hypothesis α value of 0.05, the features "*Week*" and "*Country*" with p-value greater than 0.05 were removed.

4.1.3. Correlation coefficient (for metric features)

We used Pearson's correlation factor for computing the correlation matrix and used a heatmap to visualize the results (Appendix J). As part of our exploration, we also used Spearman's correlation factor, which provided us with very similar results (not showed in the code). Therefore, we decided to keep the Pearson's correlation method for analysis purposes, since it can also provide us with a stricter linear correlation. The main goal of this step was to reduce the number of redundant features and keep the most relevant ones.

Regarding the correlation with the target variable, the highest coefficient was 0.61 for "*GoogleAnalytics_PageValue*". With the majority of the other features having coefficients between 0.1 and 0.25 with the target variable. By establishing a threshold 0.85 (minimum) and keeping the original features as much as possible, "*AccountMng_Pages*", "*FAQ_Pages*", "*Product_Pages*", "*GoogleAnalytics_BounceRate*", "*Product_Duration/Page*", "*AccountMng_Duration/Page*", "*Page_all*" and "*Duration_all*" were removed with this method, due to redundancy principle. All together 11 features were filtered out and the remaining 13 were subjected to wrapper and embedded methods.

4.2. Wrapper

In this second step of the workflow, the feature selection was performed using a specific sequential ML algorithm. The chosen algorithm on this step was Recursive Feature Elimination (RFE) using logistic regression and Decision Tree, as the classifiers to select the optimal set of features. In order to apply RFE, the original dataset was split (by applying *sklearn train_test_split)* using stratification with a test size of 0.2. Then several pre-processing steps were applied or not (based on model type): *PowerTransformer, MinMaxScaler* and categorical features' encoding.

Two different subsets of features were selected accordingly with the algorithm implemented. Using logistic regression as the classifier and f1 as the best score, the optimum number of features were calculated and then 8 top-ranked features were selected. When using decision tree as the classifier in RFE, the optimum number of features was 7 and, then these 7 top-ranked features were the selected one's. Both lists of selected features with these methods were independently stored, in order to be used in different final outputs as feature lists.

4.3. Embedded

In order to make our decision more robust, we decided to include two embedded methods in the feature selection process: the Lasso Regression and the Feature Importance from Random Forest

algorithm. Both methods use the same pre-processed data as described in section 4.2. By applying a threshold of 0.07 on the ranking outputted by Lasso Regression, the top 8 features were selected. Using the feature importance from Random Forest, with a threshold of 0.03, it resulted in the selection of top 9 features. Both lists of features were stored separated in order to be used in further different filtering combinations.

4.4. Preparing final features by filtering combinations

The selection process leads us to two different subsets of features that resulted from different methods: the features selected using "linear" methods (RFE Logistic and Lasso Regression) and the ones selected using "Tree" methods (RFE Decision Tree and Feature importance with Random Forest). These two groups of features were posteriorly separated in metric and non-metric features, in order to facilitate the data processing during model implementation steps.
The final features prepared for building the models were:

- Feature_selected_linear = ["*Type of Visitor*", *"GoogleAnalytics_ExitRate*", "*GoogleAnalytics_PageValue*", "*Month*", "*Browser*", "*Type_of_Traffic"*]
- Feature_tree = ["*GoogleAnalytics_PageValue*","*GoogleAnalytics_ExitRate*", "*Product_Duration*", "*AccountMng_Duration*", "*Rate_ratio*", "*Month*", "*Type_of_Traffic"*]

5. Model selection

5.1. Model selection method

We processed different training sets for tree based and non-tree-based models. For tree-based models there is no requirement for any pre-processing steps like transformation and normalization. So, the numerical and categorical features can be used for models directly after numerical encoding of the categorical ones. For non-tree-based models, a different dataset processing was implemented. For these, we performed power transformation, normalization and target encoding. If we had applied these processes before the split of the training and validation set, we would have got information in advance from the validation set, which may had led to data leakage. To prevent this, all data transformation, normalization, target encoding, and even further imbalance data handling with SMOTE-Tomek (Appendix K), had to be performed after splitting the training and validation set.

Because the *Repeated Stratified K Fold* function couldn't automatically do the data processing we wanted, after splitting the data, several functions were manually coded for testing the model performance in order to facilitate the workflow. In fact, there was a better way to apply all data pre-processing, by using the *pipeline* function, which can integrate pre-processing after data splitting automatically in grid search and final model testing. However, we faced a technical problem, since one of the encoders that we used, *Target Encoder* is not from the *sklearn's* package, and therefore it was not easy and simple to directly apply it in the *pipeline function*.

Target encoder uses the pre-converted string data types to encode the categorial features, which

is not easy to implement when we just want to encode some of the features inside the pipeline and not all. In order to overcome this issue, we decided to manually code functions to create our data workflow.

Fitting a model is a straightforward process, however, choosing the right one is a challenging step of any ML problem. In our project, 12 different models were tested, including: K-nears neighbours (KNN), Logistic Regression (LR), Naive Bayes classifier, Decision Tree classifier, Neural Networks (NN), Support Vector Machine (SVM) and 5 ensemble learning models: Random Forest classifier, Adaboost classifier, Stacking method, Voting method (Appendix L) and Extra Trees classifier (Appendix M).

During the selection step, for ensuring the model stability during the performance testing of all models, we used repeated stratified K-fold cross-validation. The K number was chosen based on the method $K \approx \log(n)$ [5]. In order to reduce the computational costs, we decided to use only 5-fold cross-validation, once during the hyperparameter tuning. But in the final model test, we still repeated the 3 times (by set the number of repeat as 3) the 5-fold cross-validation process. The results were collected after all the cross validations, the average and standard deviation of accuracy and F1 score was taken as the final metrics to evaluate the model performance.

Regarding the hyperparameter tuning, a combination of manual and grid search was performed for tree-based models. However, for NN, SVM and Adaboost classifier only the manual hyperparameter tuning was performed. For the stacking method, only very simple parameters were used for base models. For the voting method, only soft voting was used. The main reason for using manual search was the technical problem mentioned before. We can't use *pipeline* with *Target Encoder* and cross-validation in *GridSearchCV*. This issue was not presented for the tree-based models, since their training data didn't need target encoding. Even without grid search, the local optimal solution can still be achieved by this greedy algorithm-like manual search method. It is worth mentioned that manual search can as well greatly reduce the computational costs.

Due to fact that some models, especially the non-tree-based models, show sensitivity to outliers and imbalance data, the performance of the models was tested using 4 different data pre-processing pipelines: a normal data, a data without outliers, a normal data with target balancing using SMOTE-Tomek and a data without outliers with target balancing using SMOTE-Tomek. In addition, it's worth mention that SMOTE-Tomek should be also implemented after splitting the training and validation sets.

5.2. Model selection results

In general, the highest F1 score that most of the models achieved was around 0.66, with an accuracy for the training and validation of 87%, which suggests no obvious overfitting.

KNN model shown an accuracy for the training set of 91% and for the test 88% with an F1 score of 0.59, which indicates that KNN is overfitting on this dataset. In KNN, not only is the model seems to be overfitting, but due to target imbalance, the model seems also to be predicting more accurately

the negative values rather than the positive values. The tree base models are usually a good model for classification. So, we also tried the Extra Trees model. Compared with the Random Forest model, it uses the whole original sample and chooses the split randomly [6]. So, we hoped that this algorithm might perform better than the Random Forest model. However, its performance in our data was not ideal. The Extra Trees classifier score near 100% for accuracy on the training set, with an F1 score of 0.64, showing stronger evidence of overfitting

Also, the LR model seems to be affected by the same phenomena. Due to its accuracy exceeding 88% but the F1 score remained 0.62, LR could predict more accurately the negative target values. So, we used SMOTE-Tomek to balance the target. Interestingly, the LR's F1 score improved significantly after balancing the target data, which was not so obvious for other models. This shows that LR may be a potentially good model after balancing the data. Since we had many models that could achieve an F1 score of 0.66, we wanted to try if both the Voting and Stacking methods could achieve a higher score, since theoretically, it was likely to get better results by combining the predictions of these basic models. Unfortunately, the final F1 results haven't improved after these last steps.

After removing the outliers, the accuracy of some models improved slightly. When we evaluate a model based on the accuracy score, it is a good strategy to remove the outliers. However, in these models the F1 score decreased slightly, suggesting that the predictive ability for positive targets had decreased. This may be because "outliers" can contain more predictive information for positive target values. Therefore, could be a good strategy to keep these "outliers" in further predictions.

The F1 score of the different models could only reach 0.66 no matter what model it was, indicating that the prediction is more determined by the distribution of features rather than by model types. In our feature selection process, "*GoogleAnalytics_PageValue*" got much higher scores than other features, which shows that "*GoogleAnalytics_PageValue*" probably played an absolute dominant role during the prediction. Perhaps with further feature engineering, a higher F1 score might be achieved, however despite our efforts during this step, we were unable to extract more useful information.

6. Prediction

Finally, we selected several models with F1 scores around 0.66 for the final prediction. First, we use the entire *train.csv* as a training set for data transformation, normalization and categorical encoding, to then fit the model to the data. Also, the same data pre-processing was performed on the *test.csv* and finally, the models were used to predict and upload the result on Kaggle. From this final model predictions, we chose Support Vector Machine (SVM) for linear kernel as our final elected model. The choice was based in its overall performance, that was better the other models, high mean for F1 score with the smallest standard deviation. As well, the score it returns from Kaggle was very similar, and therefore consistent with the score we tested on the script, which suggest that is a stable model choice.

References

[1] S. Galli, "Feature engineering for machine learning," *https://www.udemy.com/course/feature-engineering-for-machine-learning/*.

[2] S. lakshmanan, "How, when, and why should you normalize/ standardize/ rescale your data?," *https://towardsai.net/p/data-science/how-when-and-why-should-you-normalize-standardize-rescale-your-data-3f083def38ff,* 2019.

[3] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *the Journal of machine Learning research,* vol. 12, pp. 2825-2830, 2011.

[4] AI DATA "Seven types of data bias in machine learning", 2021 https://www.telusinternational.com/articles/7-types-of-data-bias-in-machine-learning

[5] Y. Jung, "Multiple predicting K-fold cross-validation for model selection," *Journal of Nonparametric Statistics,* vol. 30, no. 1, pp. 197-215, 2018.

[6] P. Aznar, "What is the difference between Extra Trees and Random Forest?," *https://quantdare.com/what-is-the-difference-between-extra-trees-and-random-forest/,* 2020.

[7] J. D. Becher, P. Berkhin, and E. Freeman, "Automating exploratory data analysis for efficient data mining," in *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2000, pp. 424-429.

[8] Svideloc, "Target encoding vs. one-hot encoding with simple examples," *https://medium.com/analytics-vidhya/target-encoding-vs-one-hot-encoding-with-simple-examples-276a7e7b3e64*, 2020

[9] A. Mahbubul, "Anomaly detection with Local Outlier Factor (LOF)", *https://towardsdatascience.com/anomaly-detection-with-local-outlier-factor-lof-d91e41df10f2*, 2020

[10] P. Kumar, "Understanding LOF (Local Outlier Factor) – perspective for implementation", *https://medium.com/@pramodch/understanding-lof-local-outlier-factor-for-implementation-1f6d4ff13ab9*, 2020

[11] Sklearn Documentation for Isolation Forest Algorithm. *https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html*

[12] E. Lewinson, "Outlier detection and isolation forest", *https://towardsdatascience.com/outlier-detection-with-isolation-forest-3d190448d45e* ,2018

[13] R. A. A. Viadinugroho, "Imbalanced Classification in Python: SMOTE-Tomek Links Method" *https://towardsdatascience.com/imbalanced-classification-in-python-smote-tomek-links-method-6e48dfe69bbc*, 2018

[14] M. Zeng, B. Zou, F. Wei, X. Liu, and L. Wang, "Effective prediction of three common diseases by combining SMOTE with Tomek links technique for imbalanced medical data" in *2016 IEEE*

International Conference of Online Analysis and Computing Science (ICOACS), 2016: IEEE, pp. 225-228.

[15] J. Brownelee, "How to develop voting ensembles with python," *https://machinelearningmastery.com/voting-ensembles-with-python/,* 2020.

[16] D. Tunnicliffe, "Extra trees, please. venturing into the machine lerning realm of decision trees and random forests," *https://towardsdatascience.com/extra-trees-please-cec916e24827*, 2021

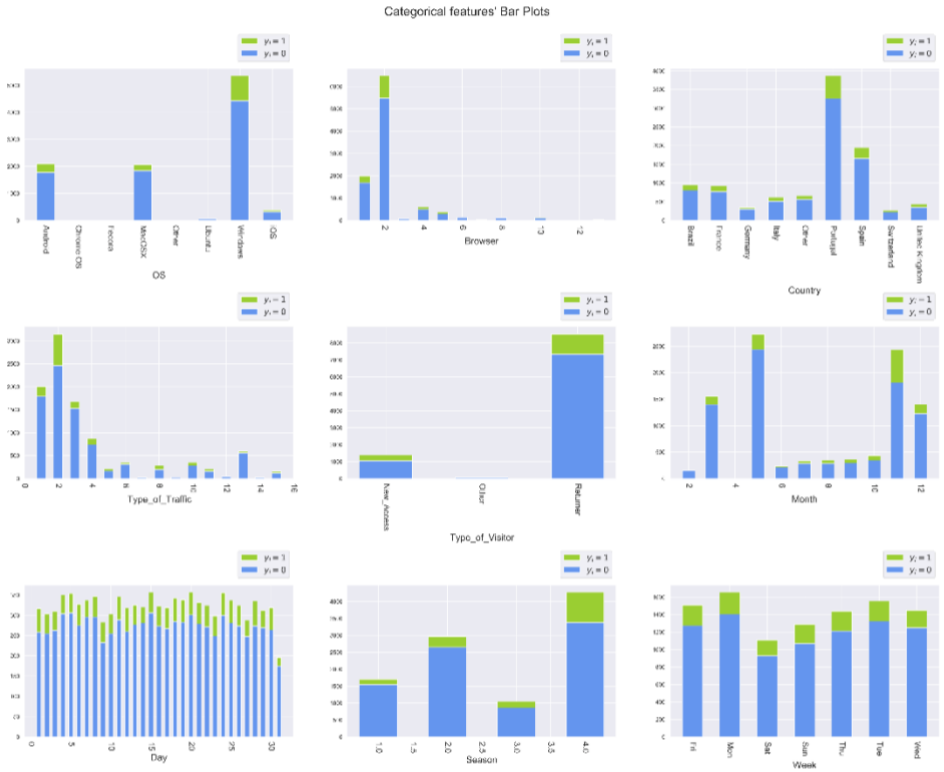
[17] Sklearn Documentation for Extra Trees Classifier. *https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html*

[18] J. Brownelee, "How to develope an extra trees ensemble with python," *https://machinelearningmastery.com/extra-trees-ensemble-with-python/,* 2021.

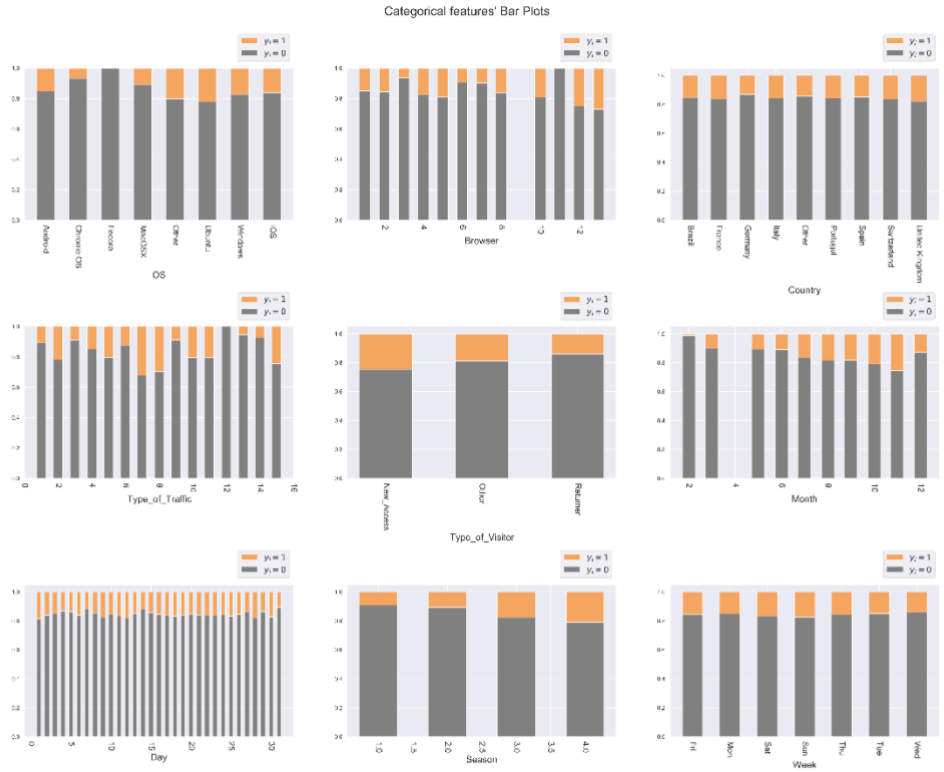
[19] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning,* vol. 63, no. 1, pp. 3-42, 2006.

Appendix

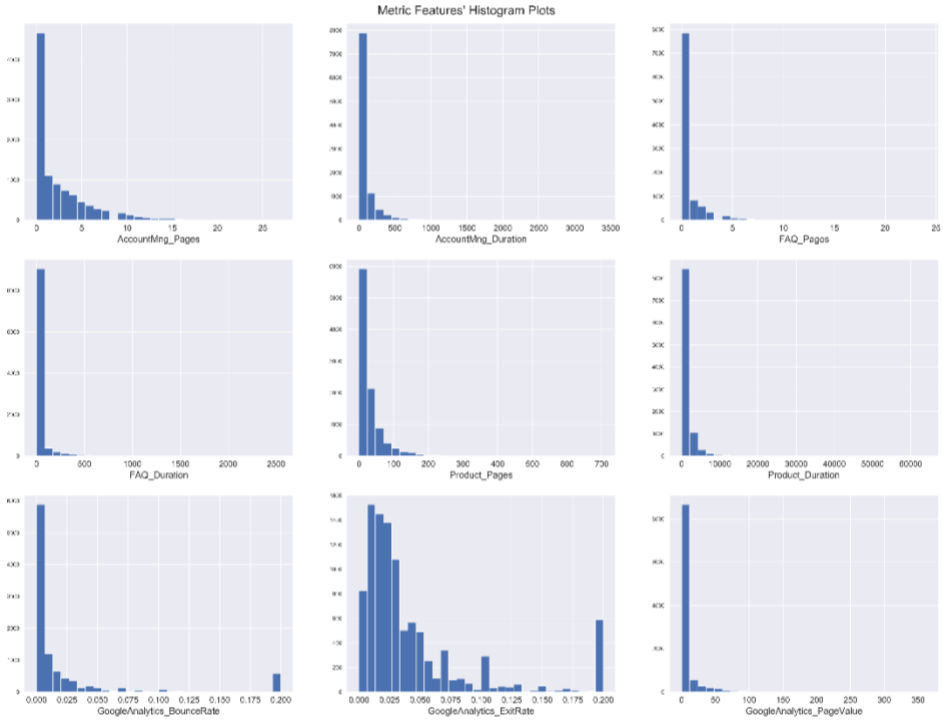
A. Categorical features Bar Plot (Frequency)



B. Categorical features Bar Plot (Proportion)



C. *Distribution of Metric Features*

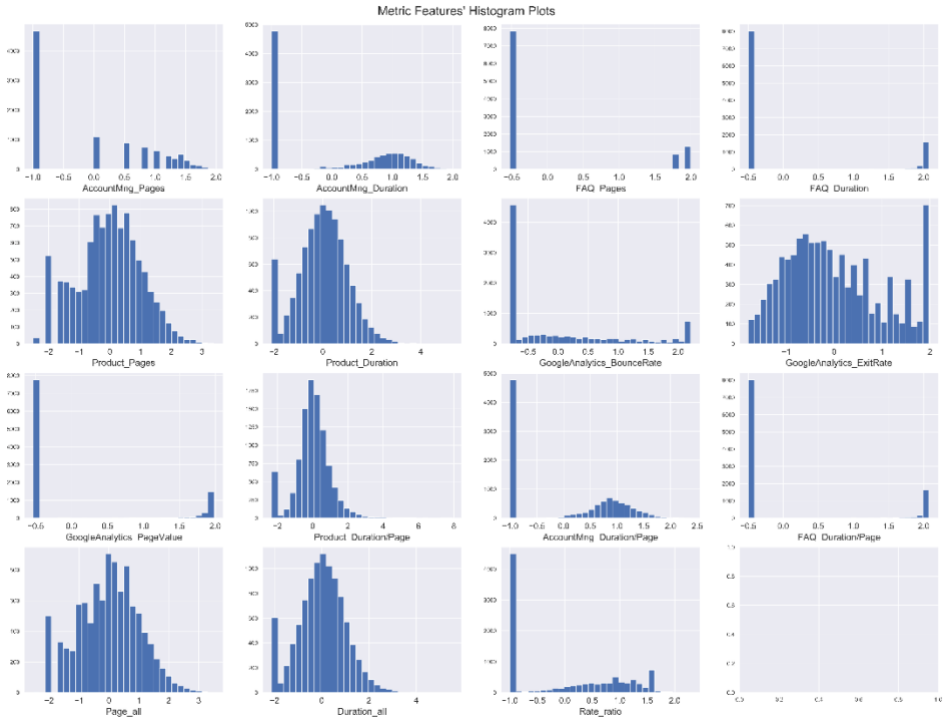


D. *Feature Grouping*

Table 1: Groping Features to extract other useful features

New features	Method	Description
Product_Duration/Page	Product_Duration/ Product_Page	Average duration on each product page
AccountMng_Duration/Page	AccountMng_Duration/ AccountMng_Page	Average duration on each management page
FAQ_Duration/Page	FAQ _Duration/ FAQ _Page	Average duration on each FAQ page
Page_all	Product_Page+ AccountMng _Page+ FAQ _Page	Total number of pages for product, management and FAQ pages
Duration_all	Product_ Duration + AccountMng _ Duration + FAQ _ Duration	Total number of duration for product, management and FAQ duration
Rate_ratio	GoogleAnalytics BounceRate/ GoogleAnalytics_ExitRate	The ratio of two GoogleAbalytics Rates

E. Distribution of Metric Features after the application of Power Transformer

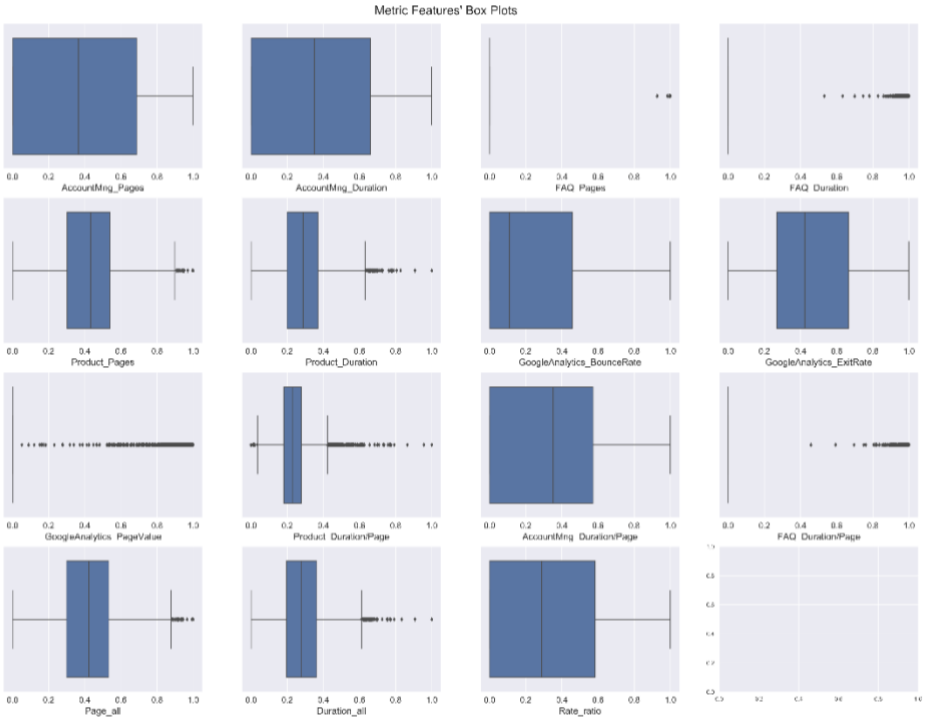


Power transformer is a technique that uses a power function to make the distribution of a random variable Gaussian or more-Gaussian like. A log transform is a typical example of this kind of transformation. The family of this transformer has two different approaches until now, one using *Box-Cox Transform* which strictly takes positive input whereas the other, *Yeo-Johnson* accepts both positive and negative inputs [3]. We came up on using this technique in this project since the numerical features in our dataset were significantly skewed to right and this transformation worked fine to change the shape of the distribution as shown above.

F. Target Encoder

Target Encoder is a technique used for transforming categorical features into numerical features. It works by replacing the categorical feature with a blend of posterior probability of the target given a particular categorical value and the prior probability of the target over all the training data [7]. In a simplistic way, the frequency of each label in a categorical feature is calculated and then it is used to obtain the probability of the target variable occurring given each specific label of the categorical variable. This leads to a numerical representation of the label corresponding to the probability value of each label in the categorical variable [8].

G. Box-Plot for Outlier Visualization



H. Local Outlier Factor (LOF)

The Local Outlier Factor is a density-based-unsupervised algorithm for outlier data handling. In summary, it works by comparing the density of a given point to its neighbours and hence determining whether that data point can be considered normal or as an outlier.

Its main assumption is that data points that exist in low-density areas are considered outliers. [9] By comparing the local density around a data point and the density of its K-nearest neighbours, LOF generates a score, that helps its interpretation. If the LOF of a point is X, that means that in average the density of its K-neighbors is X times greater than its local density. In a basic description, the algorithm follows these steps [10]:

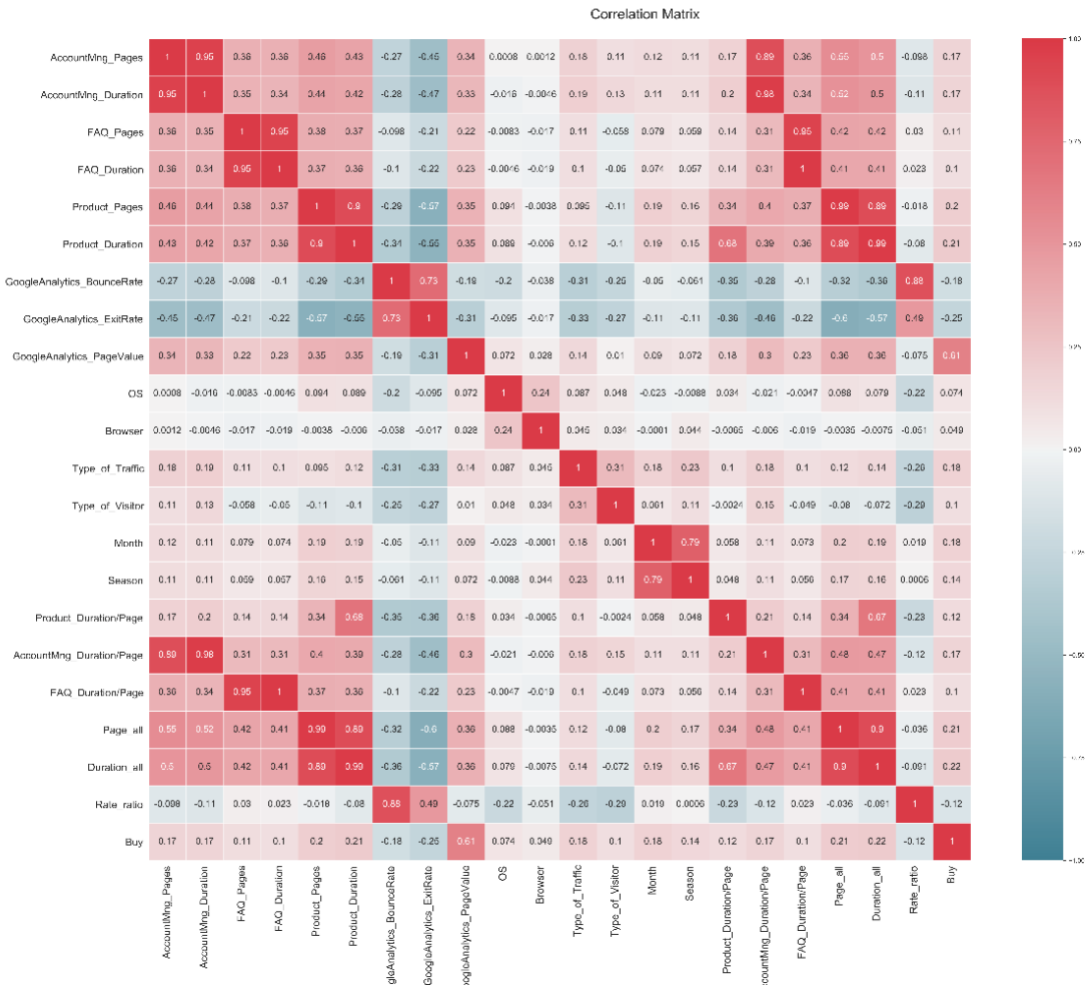
1. The K-nearest neighbors are calculated for each data point;
2. Using the previous K-N neigbours, the local density for each data point is estimate by computing the LRD (local reachability density), which represents the maximum distance from a particular point to its neighbor point;
3. Comparing the LRD of a data point with the LRD's of its K-neighbors, returns the LOF score.

Regarding the Score results, *"as a rule of thumb, a normal data point has a LOF score between 1 and 1.5*" whereas outliers have an higher value. "*The higher the LOF the more likely is to be an outlier"*. [9]

I. Isolation Forest (Outlier detection algorithm)

The Isolation Forest is an algorithm that aims to isolate observations of the dataset by selecting a random feature and then choosing a split value between the maximum and the minimum of the selected one [11]. The assumption is that the amount of splitting's needed to isolate a sample is equal to the length of the path from the root node to the terminating node [11]. It also assumes that since outliers are more rare and less frequent that the rest of the observations on the dataset, they would be easier to isolate, for example, closer to the root node. By opposite, the normal observations would require more partitions to be identified rather than an outlier data point. [12]

J. Pearson's Correlation Matrix



K. SMOTE-Tomek

It a method that handles imbalance data by combining two different techniques: an oversampling technique called Synthetic Minority Oversampling Technique (SMOTE) and a data cleaning technique called Tomek-links. SMOTE's goal is to generate samples by finding the K-nearest neighbors (usually by using the Euclidean distance), which allows the new samples to be different from the original minority class. Tomek-links which is a modification from condensed nearest neighbours under-

sampling technique, helps at identifying the samples from the majority class that are close to the minority class and then removes these data (the called *Tomek-links*) [13]. The combination of both shows superior results that when just SMOTE is applied in categorical data [14], since by combining the *Tomek-links*, the identification of the boundaries between both classes (majority and minority) is better defined. The algorithm follows the:

1. Choose random data from the minority class
2. Calculate the distance between the random data and its K-nearest neighbors
3. Multiply the difference with a random number between 0 and 1, then add the result to the minority class as a synthetic sample
4. Repeat steps 2 and 3 until the desired proportion of minority class is met (The end of SMOTE)
5. If the random data's nearest neighbor is the data from the minority class (i.e. created the Tomek-link), then remove the Tomek Link.

L. Voting Classifier

A voting classifier is an ensemble ML model that takes a combination of several other model's prediction to aggregate the final performance. It is not an actual classifier, but a wrapper for a set of different classifiers. There are two ways to use the model in classification, by hard or soft voting. The hard voting is based on the majority class label, whereas the soft voting makes predictions on the weighted average probabilities on each sample [15].

M. Extra Trees Classifier

The Extra-trees classifier (or Extremely Randomized Trees) is an ensemble algorithm very similar to the Random Forest, which uses a collection of decision trees in order to predict a class label or a data point. [16] According to *sklearn*, Extra Trees classifier is a meta estimator which fits a randomized number of decision trees in different sub-samples of the dataset. By calculating the average, it returns an improved predictive accuracy. [17] The decision trees used are let grown unpruned and the final prediction come from the majority voting (in a classification problem) or by using the average of the predictions of the trees (in a regression problem). It shows some similarity with Random Forest, namely that both only use a random subset of the features each time in each tree [18]. However, it differs from the later since instead of using a subsample with replacement from the dataset it uses the whole dataset. And instead of using a greedy approach to find the best possible split, it randomly chooses the split point in each decision tree: [18, 19]

- The number of decision tree to ensemble.
- The number of input features to randomly select and consider for each split point.
- The minimum number of samples required in a node to create a new split point.

N. Model Results

Overall performance of the model after Repeated Stratified K-fold for 5 splits and 3 repetition.

Table 2: KNN Classifier

Dataset	train	sd_train	val	sd_val	F1	sd_F1
Normal	0.914975	0.001745	0.881222	0.004630	0.592200	0.017130
Without outliers	0.920929	0.001491	0.890271	0.005054	0.575649	0.018547
SMOTE	0.930625	0.001543	0.834084	0.006972	0.595767	0.011442
SMOTE without outliers	0.934511	0.002058	0.838265	0.007756	0.571091	0.016943

Table 3: Logistic Regression model

Dataset	train	sd_train	val	sd_val	F1	sd_F1
Normal	0.887930	0.001545	0.886389	0.006001	0.620402	0.018946
Without outliers	0.897267	0.001774	0.896275	0.006117	0.614470	0.022010
SMOTE	0.857263	0.002740	0.866021	0.007290	0.655249	0.016134
SMOTE without outliers	0.855248	0.003577	0.873221	0.007601	0.641306	0.014544

Table 4: Naive Bayes Classifier

Dataset	train	sd_train	val	sd_val	F1	sd_F1
Normal	0.878463	0.002093	0.877855	0.006359	0.664179	0.013999
Without outliers	0.886089	0.002420	0.885159	0.005683	0.655069	0.013413
SMOTE	0.839932	0.006344	0.829183	0.009724	0.599372	0.015207
SMOTE without outliers	0.835455	0.004267	0.826828	0.006847	0.569434	0.013021

Table 5: Decision Tree Classifier

Dataset	train	sd_train	val	sd_val	F1	sd_F1
Normal	0.874379	0.873354	0.660858	0.001632	0.007335	0.015635

Table 6: Random Forest Classifier

Dataset	train	sd_train	val	sd_val	F1	sd_F1
Normal	0.874379	0.873354	0.660858	0.001632	0.007335	0.015635

Table 7: Neural Network Classifier

Dataset	train	sd_train	val	sd_val	F1	sd_F1
Normal	0.885172	0.001039	0.883922	0.004762	0.654788	0.011765
Without outliers	0.892737	0.001049	0.892738	0.006637	0.658511	0.016808
SMOTE	0.853064	0.003230	0.872054	0.004623	0.660751	0.010958
SMOTE without outliers	0.852926	0.003276	0.886553	0.004012	0.659180	0.008445

Table 8: AdaBoost Classifier

Dataset	train	sd_train	val	sd_val	F1	sd_F1
Normal	0.875763	0.002762	0.873354	0.005634	0.657622	0.011067
Without outliers	0.887251	0.001412	0.886482	0.005071	0.658432	0.010878
SMOTE	0.854660	0.003280	0.871921	0.004662	0.661243	0.010416
SMOTE without outliers	0.854046	0.002977	0.886553	0.004286	0.659854	0.008779

Table 9: SVM Classifier for parameters (linear)

Dataset	train	sd_train	val	sd_val	F1	sd_F1
Normal	0.872787	0.001647	0.872787	0.006590	0.661960	0.014086
Without outliers	0.886768	0.001730	0.886768	0.006922	0.659669	0.017792
SMOTE	0.853536	0.003446	0.872721	0.008470	0.661916	0.018795
SMOTE without outliers	0.852751	0.004026	0.886768	0.004400	0.659590	0.010464

Table 10: SVM Classifier for parameters(rbf)

Dataset	train	sd_train	val	sd_val	F1	sd_F1
Normal	0.873846	0.001562	0.873854	0.006711	0.663348	0.012969
Without outliers	0.886768	0.001777	0.886768	0.007110	0.659769	0.016059
SMOTE	0.853105	0.003469	0.871888	0.008475	0.660453	0.018764
SMOTE without outliers	0.852687	0.004026	0.886661	0.004446	0.659380	0.010548

Table 11: Stacking Method

Dataset	train	sd_train	val	sd_val	F1	sd_F1
Normal	0.880421	0.001203	0.880588	0.005345	0.659191	0.012979
Without outliers	0.890289	0.001814	0.889986	0.008463	0.655721	0.019332
SMOTE	0.853434	0.002476	0.872455	0.006707	0.661299	0.015198
SMOTE without outliers	0.852115	0.003456	0.886696	0.005419	0.659484	0.013129

Table 12: Voting Medthod

Dataset	train	sd_train	val	sd_val	F1	sd_F1
Normal	0.887530	0.001135	0.886323	0.006037	0.654564	0.013621
Without outliers	0.894819	0.001233	0.894453	0.005168	0.654098	0.013678
SMOTE	0.856990	0.002828	0.868288	0.007742	0.657120	0.016158
SMOTE without outliers	0.855691	0.003388	0.876724	0.007073	0.646316	0.014280

Table 13: Extra Trees Classifier

Dataset	train	sd_train	val	sd_val	F1	sd_F1
Normal	0.9998	0.89919	0.643783	0.000061	0.003723	0.015501