Predicting the click behaviour of users in a hotel ranking system

Ioannis Gatopoulos[2654227] Filip Knyszewski[2656880]

Philipp Ollendorff[2655580]

September 1, 2019

1 Introduction

Recommender systems are powerful algorithms that appeared with the rise of the internet and helped build some of the biggest companies in the world today. Tourism, more specifically hotel bookings, was one of the industries that was the most disrupted by this new technology due to its natural fit with recommender systems.
In this report we are given the task of predicting hotels most likely to be clicked in a search query. Such a system could greatly help the improvement of the recommender system and consequently increase user satisfaction and engagement. To achieve this, a large dataset is provided with information about the query, hotels that were clicked on and all kinds of related data. Most likely, a big part of the provided data is redundant or not necessarily useful so a major part of this assignment will be the preprocessing of said data, feature engineering and extraction, or in short, exploratory data analysis (EDA).

2 Business Understanding

The task at hand was inspired on the ICDM 2013 competition organized by Expedia the world's largest online travel agency. The competition was named 'Personalize Expedia Hotel Searches' and the main focus was the hotel sort order. The final winner was a participant Owen which achieved an impressive score of 0.53984, closely followed by 'Jun Wang' with 0.53839.

Owen's winning strategy had a large focus on the preprocessing and featuring engineering of the data. Some of the methods he adopted included replacing missing values with large negative values to allow models to learn dependencies from them, bounding numerical values and down sampling negative instances for creating a more balanced dataset.

From a feature engineering perspective Owen put a big emphasis on creating new features like 'price_diff_from_recent' (difference between hotel price and

recent price) and price_order (order of the price within same srch_id. Since his models required numerical features only a target variable average method was employed.

Owens winning model ended being an ensemble of gradient boosting machines with an NDCG loss function.

3 Data Understanding

3.1 The dataset

Before starting the preprocessing it is wise to take some time and get familiar with the dataset at hand. This way we can develop some intuition for the type of preprocessing strategies that we will need and become more acquainted with the data.

Rows	4,958,347
Columns	57
Unique searches	199,795
Countries	210
Hotels	129,113
Searches that lead to a booking in %	69.27
Total NaN values in %	42.36

Table 1: Dataset basic statistics

On first inspection, we already notice the large size of the dataset and a high number of missing values. This is a first indication of a challenging dataset.

3.2 Feature correlation

An important part of the initial analysis of a dataset is to check the correlations between each feature and the target feature. In the case of our dataset we just used the feature 'booking_bool' as the target. In Figure 1 we can observe all the positive and negative correlations with absolute value above 0.01.

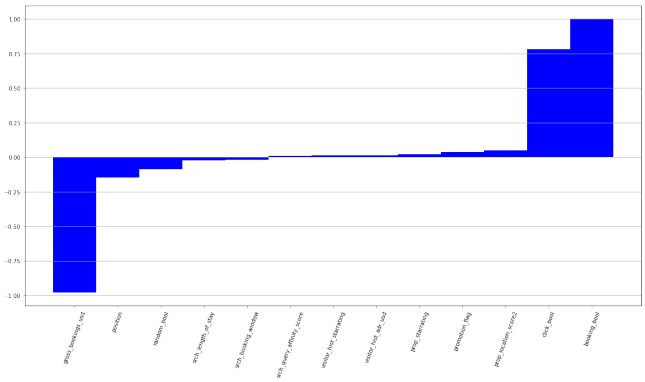

Fig. 1: Highest absolute correlations with feature 'booking_bool'

Some interesting conclusions can be drawn from this. In general, most individual features do not show a high degree of correlation with the 'booking_bool' feature, suggesting that we are better of generating our own features by combining the ones from the dataset. We see a large negative correlation of feature 'gross_booking_usd' which is rather expected since this corresponds to the price level of the property. The more expensive it is, the less likely that it is booked (in general). We can also see that clicking is rather strongly correlated booking, as expected since booking always requires clicking (but not the other way around). Unfortunately, both of these strong correlations cannot be used for training directly, because they are not present in the test set. One of the biases that deserves extra focus is the position bias, typical of ranking systems like this one. In Figure 2 this can be seen very clearly: there's a big discrepancy between the amount of bookings done of properties that were presented in he first position of the ranking and of those lower down. One may argue that this is because the ranking system is good and pushes to first position the better 'deals' but by looking at how massive the difference is between first and second place, compared to second and third (for example) it quickly becomes apparent that it is not the only reason. Additionally, the visualizations in 2 show a strong price discrimination and a tendency to book higher rated hotels. Surprisingly, it is not the 5 or 4 star hotels that are clicked the most. This indicates, that customers want not just value, but rather good value for money.

4 Data Preparation

4.1 Feature generation

Despite the large amount of features that are part of the dataset we observed that none of them will directly correlate with our target. Besides this, as seen in Figure 3, many of them are also filled with NaNs, making them even less useful. So we have set out to generate new features, using the ones that are already

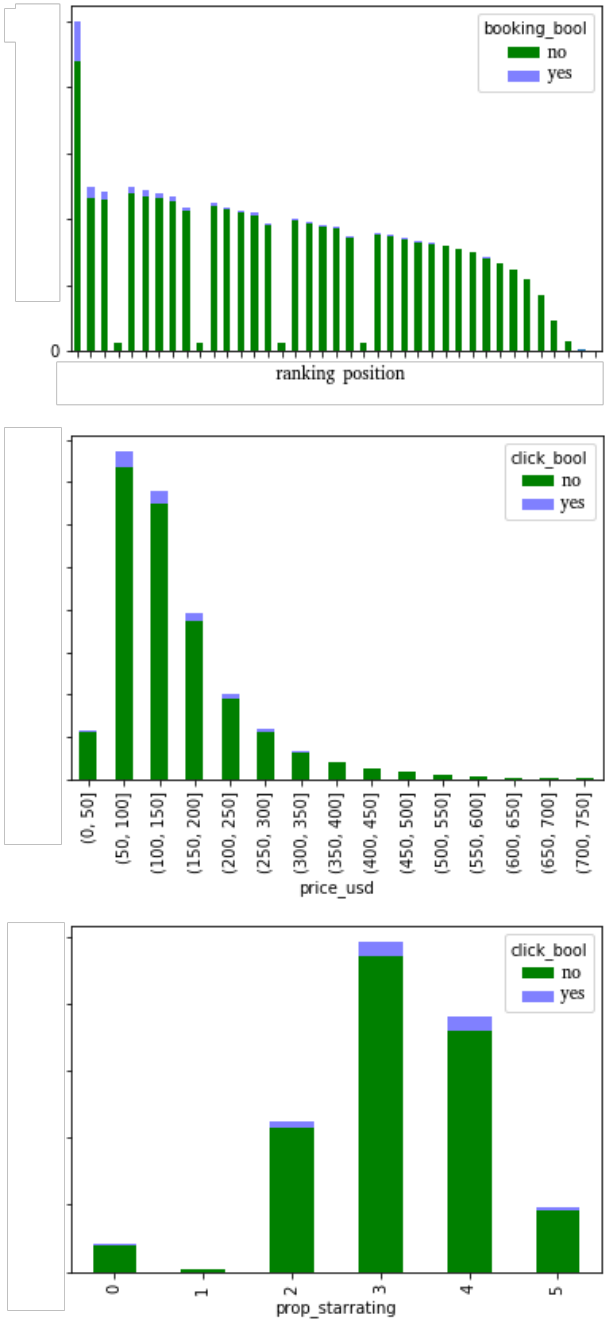

Fig. 2: From Top to Bottom: Amount of bookings per position on the whole dataset, Clicks by price range, Clicks by star rating

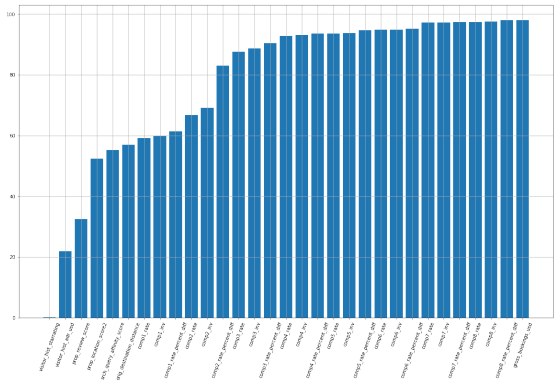

Fig. 3: NaN values % per feature

given. An easy way of seeing how this works is to think of a date feature which on its own may be almost meaningless, but when tweaked can generate a feature like weekday. By introducing new similarities between two dates, weekdays can have more of an impact in the performance of a model than a date alone. In a dataset as rich as ours, there's a lot of opportunity for feature generation and so the following features were created. `weekday`, `month` and `total search count` are three trivial features, first two capturing temporal dependencies more closely, while the latter captures the preferences of different groups e.g. preferences of couples (2) or families (4) etc.

Moreover, we tried to take into account and model the user's preferences with the present hotel. To put it into practice, we calculated the scores `price score`, `star score` and `star2 score` in the following way; we assumed that every user has a Gaussian distribution around these features, where the mean, and in that case also the maximum value, is the provided mean value, while the standard deviation is calculated with respect to all the search ids. The output is a score, from 0 to 1 (closer to the mean, closer to score 1), which shows how satisfied the user would be with the standards of the presented hotel, according to his history. Even though there was a correlation with the target value, the problem is that we have this information only for a very small amount of search queries and it will not be very useful in general. However, it will be very useful for those that do.

Finally, we wanted to create a feature that captures the value of a hotel per search. Since the user is exposed only to the hotels listed as a result of the search query, our hypothesis is that the majority of them will want to book the hotel which has a fair price, rating and location in comparison to the others. Thus, we fist create 5 new features (difference on the hotel prices, on both star and location ratings per search) which are essentially a ratio of the e.g. price of the hotel to the median of all the hotel in this particular search. We add these features together using weighted sums, with weights proportional to the correlations with the target. Together, they make the feature called `hotel worth`, a value between 0

and 1, where 1 corresponds to a superior hotel while 0 has the least worth to this user. The latter had 0.13 correlation with the target value, which is a huge leap from the original features. The most important benefit of creating this feature is that it will improve the model's capacity to compare hotels with each other.

4.2 Missing values

As mentioned in the related works section, the given dataset is imperfect and one of its most prevalent shortcomings is the large amount of missing values.

In general, there are three ways to deal with this: Delete just the data point (row) in which the missing value appeared, delete the entire column in which it appeared, or replace the value with an estimate.

The first strategy is not feasible, as it will effectively delete our entire data set. The second is useful and has been implemented as a simple baseline. The third option is a little more complex:

Replacing missing values is a daunting task: One could for example take the minimum, average, median or maximum of either the column or just the current search to replace them. In addition, we could give missing values a unique value that purposely lies outside the common range of values for this feature. This will help the model identify dependencies between missing values and the target and eventually learn from the lack of data as well.

For this reason, we also implemented this method and substitute NaN values across all features for large negative integers. This allows the data to still be fed to most models while isolating those occurrences enough for the model to be able to learn from it.

4.3 Outlier detection

Numerical columns in large datasets often have values underlying some distribution. However, human errors and unrepresentative data can lead to values too far outside of this distribution. We call these values outliers. Any model's accuracy can be highly skewed due to these values, which makes it imperative to deal with them somehow.

We did this using the **Tukey rule** (also known as Tukey's fences [1]). It relies on the interquartile range (IQR), which is the distance between the values of the first quartile and the third quartile of the Data. The Tukey rule finds appropriate boundaries per feature f to determine outliers in a data set. The boundaries are as follows:

$$[Q1_f - 1.5 \cdot IQR_f, Q3_f + 1.5 \cdot IQR_f]$$

Anything outside these values is considered an outlier and is discarded by deleting the entire respective row. We chose to delete, because even a single outlier indicates a lack of trust in this data point. However, if the respective data point has been clicked by the user, we interpret this hotel as simply exceptionally good, so we keep it regardless of any outliers.

4.4 Balancing the dataset

Large datasets are often imbalanced with regard to the target variable. In our context, being imbalanced means there are many more datapoints with e.g. zero clicks than with one click.

This imbalance can bias our model towards the majority class. It will not properly learn the dependencies between features and the (desired) minority class. To counteract this phenomenon, we randomly delete unclicked searches until the ratio is exactly 1:1.

4.5 Normalizing features

Some Machine Learning algorithms perform better when their input has been normalized. Hence, to build a good ranking model, corresponding features should be normalized in an appropriate manner. For example, hotel prices can vary across cities. Customers might expect vastly different standards for 100 euro per night in London compared to e.g. Maastricht.

Initially, we implemented normalization using the MinMaxScaler, which adds comparability between two values by squeezing the entire column range between 0 and 1. It uses the following formula:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

However, we also tried different normalization techniques and found that using the monotonic function log, boosted our performance even further by making our features have monotonic utility with respect to the target. This kind of normalization is enabled in all our generated features (i.e. `hotel worth` or `diff price per search`).

5 Modeling and Evaluation

5.1 The target

In this assignment there is no clear target for each datapoint like in a classification task. Most machine learning algorithms and methods require some kind of target value to train the model so for this assignment this value was manufactured. Since we are trying to model the probability of a certain hotel to be booked when presented in a ranking we make the assumption that most of the booking probability comes from the quality of the hotel but that part of it comes from an inherent position bias that is characteristic of online rankings. The target is then built as follows:

- 80% of the target value is built based on user feedback: If it was booked, clicked or ignored we assign 0.8, 0.4 and 0.0 points respectively.

– The other 20% are based on the position of the property within the ranking. If the property was clicked on, we would like to have a larger target value if it is lower in the ranking (since the positional bias works against it). This is achieved by applying the log function on the position, which also nicely models a reduction in the influence of the position as the ranking becomes lower (we want the difference between 1st and 2nd place to be greater than 35th and 36th place, for example). For the case of properties that were not clicked on, naturally we want them to have a lower target and so we apply the inverse function $\frac{1}{x}$ on their position to determine the position bias score. The reason for this is a strong belief in the accuracy of Expedia's ranking model. There must be a reason why some unclicked search results were higher up in the ranking than others, and so we would like to preserve said ranking.

5.2 Evaluation metric

Recommender systems evaluate rankings of the 'best' hotels that are unbiased from factors like positions on a web page. A good evaluation metric for such a system is the Normalized Discounted Cumulative Gain (NDCG), which we are being tested on as well. The NDCG@5 per search query s is given by:

$$nDCG_s = \frac{DCG_s}{IDCG_s} \tag{1}$$

$$DCG_s = \sum_{i=1}^{5} \frac{rel_i}{\log_2(i+1)} \tag{2}$$

where IDCG is the ideal discounted cumulative gain and DCG the actual one. In practice the IDCG is simply computed by sorting based on real target values, while the DCG is sorted on predictions.

The final evaluation score will then be the mean NDCG over all search queries.

5.3 Cross validation and splitting the Data

Due to the constraint of a maximum of 2 submissions per day on Kaggle, we decided to implement a local validation procedure. This includes an implementation of the NDCG metric as well as splitting the training set into training and validation parts. We decided to use cross validation to have a more solid local procedure and increase the overall generalization capabilities of our model.

The split cuts all unique search ID's randomly into two parts, with 95% of all searches going into the training set and the remaining 5% into the validation set.

Cross validation repeats the entire process of splitting the training set, fitting a model, predicting the respective validation set, and returning its NDCG. Due to randomness in the split and model, we can average out a lucky run and therefore be more confident in the accuracy of our prediction.

5.4 Models

The main models used in this assignment belong to the family of booster models. Boosting, in its essence, is a method of transforming a number of weak learners into a stronger one. By weak learners we mean models that do slightly better than random guessing and by strong we mean those that clearly out perform random guessing.

Random Forest A random forest is, in its essence, a number of if-else statements organized in a cascading graph structure. Despite being fairly simple, these models perform quite well in a wide range of problems and are often used as benchmark for more advanced modelling techniques.

XGBoost One of the models used in the task was XGBoost (Extreme Gradient boosting). This a typical algorithm which used for supervised learning problems and which achieves a respectable amount of success if a number of different Kaggle competitions. It iteratively adds weak learners to the ensemble model which fit the most 'problematic' data i.e the data with the biggest error. The thing that makes XGBoost perform so well is a couple of tricks that are used with this rather trivial method: clever penalisation of trees, proportional shrinking of leaf nodes, newton boosting and randomisation parameters.

AdaBoost AdaBoost (Adaptive Boosting) is another boosting algorithm that works in a similar fashion to XGBoost. It samples from the training data and classfies it using a base learner (which can be almost any learner). The misclassified points are then given higher weights for the next training data sampling which makes it more probably that they are chosen, and a new learner is used for the newly sampled data. The final model is the ensemble of all the learners used throughout this iterative process.

6 Analysis of results

After all the feature engineering and data manipulation the data was fed to each of the models mentioned before. Naturally, this was an iterative process and so there was a lot of back and forth between evaluating the models and going back to feature engineering to try and improve the score.

Some of the issues we encountered are mentioned in our process report. In the following, we will focus on the working models.

The nDCG of each model is shown in table 2

The Random Forest score was achieved using all our generated features and the manufactured target value, replacing missing values with unique ones per feature and balancing the data set. No normalization was used, as Random Forests are indifferent towards this step.

Model	**nDCG@5**
Random forest	0.314
XGBoost	0.331
Adaboost	0.348
Fine-tuned XGBoost	**0.356**

Table 2: Top nDCG for all models

Both the XGBoost and AdaBoost use the same preprocessing steps as before, but we added normalization this time.

Finally, hyperparameter tuning led to the best performing model of XGBoost with a final score of **0.356**. Most notably a learning rate of 0.001 a max depth of 20 and 51 estimators led to this improvement. We found these values using grid-search.

Surprisingly, the outlier detection did not work as expected. We were unable to improve the score of any of our models by applying the Tukey Rule as outlined in Chapter 4.3. Replacing the missing values gave us a small edge over simply removing their columns entirely. Balancing was always favorable and as a side-effect reduced training time significantly.

7 Future Work

There are several ways we could potentially increase this score further, that were not explored during this project due to time constraints:

First and foremost, a closer investigation on how to estimate missing values can be conducted. Since there are more than 40% of all values missing, there is large potential to augment the dataset and gather deeper insights with more accurate predictions. This type of work needs to be done on a feature by feature basis, as estimates naturally vary among different attributes.

Second, different kinds of models may be introduced such as neural networks or pure ranking algorithms such as LambdaMART.

8 Learnings

During this project, we learned how to tackle a dataset that does not only consist of more than 40% missing values, but also lacks any feature with a high correlation towards the target. For these reasons, we had to spend a large amount of time engineering new features and finding ways to deal with missing data. This challenged our team on both a creative and analytical level. On a more practical level we deepened our knowledge in Pandas and Python in general, statistics fundamentals and collaborative coding best practices.

References

1. Tukey, John W.: Exploratory Data Analysis. AddisonWesley Publishing Company Reading (1977)