Assignment 2: Learning to Rank from a search history data set originated from the Expedia hotel booking platform.

Baptiste Avot[1][2637357], Marcin Michorzewski[2][2688837], and Hinrik Snær Gumundsson[3][12675326]

[1] Vrije Universiteit, Amsterdam, The Netherlands baptiste.avot@gmail.com
[2] Vrije Universiteit, Amsterdam, The Netherlands marcin.michorzewski@gmail.com
[3] University of Amsterdam, Amsterdam, The Netherlands Hinriksnaer@gmail.com

Abstract. In this report, we cover our approach to create a search engine that can effectively provide search results for hotels based on a user query using a provided dataset from Expedia. We cover how we effectively pre-process the dataset to fit a gradient boosted learning to rank LambdaRank model that will be used to generate the ranking. We will empirically show that the model reaches a satisfactory performance and is able to optimize for information retrieval metrics directly.

Keywords: Lambda Rank · Learning to Rank · Search Engine · Data Mining.

1 Introduction

Search engine optimization is a major concern in today's world and with recent advancements, users have high expectations towards search results. Users expect the results to be generated quickly while also providing highly relevant results. Most modern algorithms use a query-document combination to find the document that is the most relevant given the user's query. In order to do this, search engine corporations analyze carefully the search data that they have, the latter displays patterns that hint at the preferences of the users. In the case that we were assigned, we will be looking at Expedia's hotel search engine data in order to build a model that ranks the results in a most optimal way([5]).

Relevancy scores can be difficult to annotate and sometimes judges are hired to give relevancy scores to query-document pairs. The problem is that judges relevancy scores may not align with user preferences and annotating a large dataset can be expensive. Another method is to infer the relevancy scores based on user interaction with previously provided results. We will look at click logs and booking logs to infer the relavancy scores for the query-hotel pairs.

One of the challenges that the dataset introduces is its size and the amount missing values that it contains, this will be dealt with by performing an extensive amount of pre processing that will aim to emphasize the patterns mentioned before and clearing out any possible noise ([8]). After data processing is finished, we will use the offline learning to rank model LambdaRank ([3]) to learn to give relevancy scores to query-hotel pairs and evaluate the results.

2 The Data

The data set we were provided with contains roughly 5 million search queries from Expedia users. Those search queries display the following features (table 2.1).

2.1 Data overview

We want to infer the relevancy scores of a query-hotel pair based on *booking_bool* and *click_bool*. A query-hotel pair where the hotel was booked is given a relevancy score of 5, if the hotel is only clicked, then it gets a relevancy score of 1, other results will be given a relevancy score of 0. One idea was to merge both columns into a single *label* or *rank_score* column (which contains the relevancy score for the query-hotel pair). Some columns can be found in the test set but not in the training set and vice versa, these columns include *gross_booking_usd* and *position*, which appear in the training set. We decided to remove these columns because the information that they carry cannot be extracted from the test set. After these operations, there are 3 main classes of features:

Boolean features These features are always represented by 0 or 1. They are great input features, because all their information is bit-encoded. These features include: *prop_brand_bool*, *promotion_flag*, *random_bool*, *srch_saturday_night_bool* *comp_i_inv*. The latter may be treated as a numerical value due to proportionally high amount of missing values.

Categorical features These features are hard to encode for a neural network because they introduce a lot of new sparse features that can increase the risk of overfitting. These values include: *srch_id*, *prop_id*, *prop_country_id*, *srch_destination_id*, *visitor_country_id*, *site_id*, *date_id*. The first does not provide any information after we group all rows by *srch_id*. The second and third ones represent the property so they might be extremely useful for ranking. The information that they carry might be essential for scoring. The fourth, fifth and sixth one describe geolocations and location. All are useful, but are the same across one search so their impact on the property's score for a particular search might be similar, thus their impact can be deemed negligible. The last feature represents time. While it may carry a lot information, its retrieval might be incredibly hard on such a big dataset.

Numerical features The are all the features that are represented by numbers and have more than 2 possible values. They can all be treated as a part of total–order.

Search ID	Integer	ID of search that was carried out
Date and time point	Object	Time at which the search was done.
Site ID	Integer	Corresponds to the Expedia point of sale (ie. expedia.fr, expedia.com, expedia.co.uk...)
User's location country ID	Integer	ID of country where the user carried out the search
visitor_hist_starrating	Float	Mean star rating of the user's previous booked hotels.
visitor_hist_adr_usd	Float	Average price per night of hotels booked in the past (in US dollars).
prop_country_id	Integer	ID of the country where the hotel is located.
prop_id	Integer	ID of the hotel.
prop_starrating	Integer	Rating of the hotel (integer between 1 and 5).
prop_review_score	Float	Mean customer review score for the given hotel (on a scale from 0 to 5).
prop_brand_bool	Boolean	1 if the property is part of a chain 0 otherwise.
prop_loc_score (1 and 2)		Scores relating to the hotel's attractiveness.
prop_log_historical_price	Float	Logarithm of the hotel's mean price.
position	Integer	Hotel's position on the Expedia results page.
price_usd	Float	Displayed price for the hotel at the time of search.
promotion_flag	Boolean	1 if the hotel displayed a promotion at the time of search 0 otherwise.
gross_booking_usd	Float	Total value of the transaction (only if booking was made).
srch_destination_id	Integer	ID of the destination where the hotel search was performed.
srch_length_stay	Integer	Length of stay input by the user for the search.
srch_booking_window	Integer	Number of days between date of search and intended start of booking
srch_adults_count	Integer	Number of adults input by user during search.
srch_children_count	Integer	Number of children input by user during search.
srch_room_count	Integer	Number of rooms specified by user during the search
srch_saturday_night_bool	Boolean	1 if intended stay contains a saturday night, 0 if not.
srch_query_affinity_score	Float	Log of probability that hotel will be clicked.
origin_destination_distance	Float	Geographical distance between the user's location and the hotel's location
random_bool	Boolean	1 if the displayed order is random, 0 otherwise
comp_i_rate	Integer	(for $i = 1, 2, \cdots, 8$) 1 if Expedia has a lower price than competitor i, 0 if equal and -1 if greater
comp_i_rate_percent_diff	Float	Absolute relative difference in price between competitor i's price and Expedia's price
click_bool	Boolean	Target value: 0 if property was not clicked by the user, 1 if it was
booking_bool	Boolean	Target value: 0 if property was not booked by the user, 1 if it was

3 Preprocessing

We considered and combined several pre processing approaches in order to get a best performing model.

3.1 Dealing with missing values and feature reduction

First we need to analyse the amount of missing values in our dataset and assess how to effectively represent these missing values([9]). We plot percentage of missing values for every feature (that contains at least one missing value) in the provided training and test dataset. Lets denote important rows as these rows that represent a clicked or booked property. Important rows are minority (about 5%) in the training dataset.To ensure that missing values are not connected with important rows we also plot percentage of missing values per each feature among them. Results allow us to assume that missing values are similarly distributed among important rows as among entire dataset.

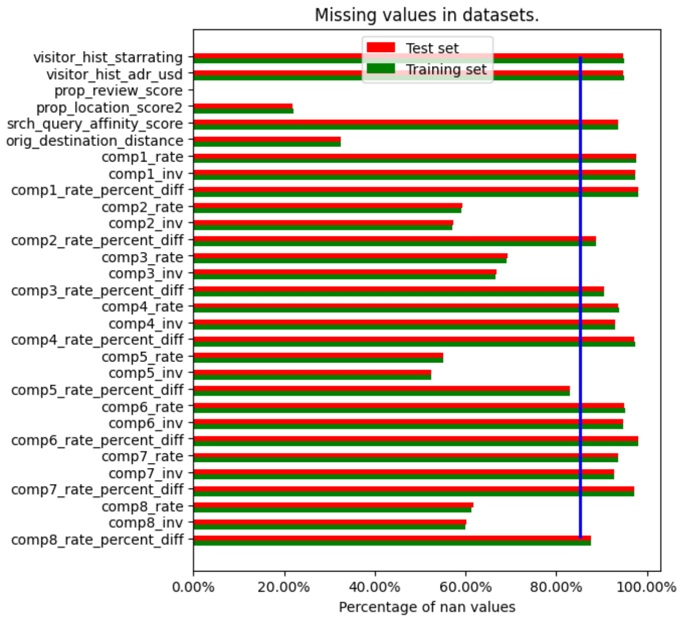

Fig. 1: Missing values seem to be similarly distributed among both test and training datasets.

A vertical bar is set to 85% and represents a threshold above which we considered a feature to be useless in some datasets. If in the test and training sets a feature has non NaN values in $< 15\%$ of the rows then it implies that the amount of information that we can acquire from the latter is extremely low. Thus we decided to delete the columns that met this criteria from some datasets.

The remaining missing values are imputed by a global mean or median computed across test and training dataset per feature. Medians were used when the

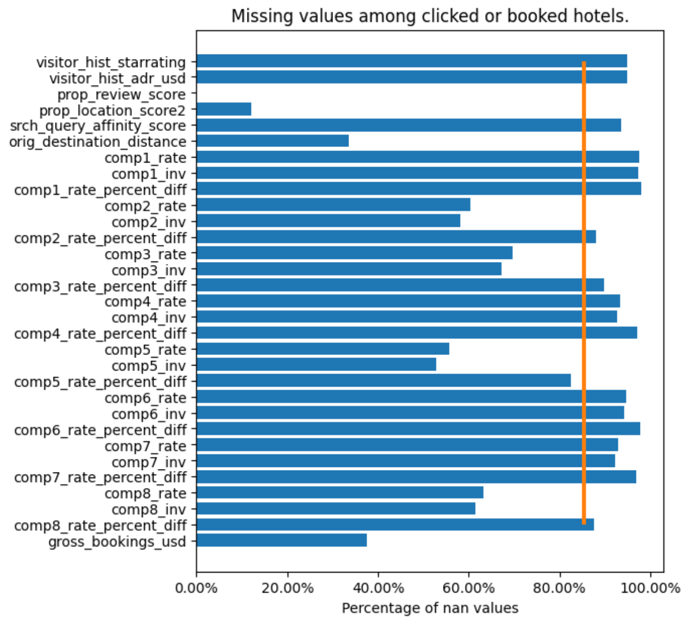

Fig. 2: Missing values seem to be similarly distributed among important rows too.

feature was represented by a boolean, integer or had less than 20 possible values in case of a float data type. In other cases means were used. We decided to use this technique due to its low computational cost and efficiency in such a large dataset.

3.2 Undersampling

Given the very large amount of data and the scale of the class imbalance (around 83k instances that were marked as booked versus millions that were not even clicked) that the training set displayed we considered undersampling in order to get a set that displayed a reduction in the imbalance (not a completely balanced one since that would not be representative of the initial dataset) since it was proven quite problematic in learning to rank studies([13]). Therefore we decided to use a Condensed Nearest Neighbor (CNN) Under Sampler ([10]) that works as follows:

1. The first item is placed in a set [1].
2. A second item's category is assigned by the Nearest Neighbor rule, if the category is correct it is placed in set [1] otherwise in set [2]
3. Step 2 is repeated for all the elements in the set
4. When one pass has been performed through the set, a loop is carried out through set [2] which performs the same operation as in step 3 until termination (set [2] is empty).
5. Contents of set [2] are discarded and set [1] represents the resampled set.

3.3 Feature Selection

Since we had a large number of features, we decided to combine two different feature selection methods: firstly, Mutual information and secondly, RFECV (Recursive Feature Elimination with Cross Validation).

Mutual Information Mutual Information is a feature selection model based on information theory (or entropy in other words)([2]). The entropy of a statistic X is given by:

$$H(X) = \sum_{i=1}^{N} p(x_i)log(p(x_i)) \tag{1}$$

The conditional entropy of X given Y is defined by:

$$H(X|Y) = \sum_{i=1}^{N}\sum_{j=1}^{M} p(x_i|y_j)log(p(x_i|y_j)) \tag{2}$$

Where X can take values X_i with i=1,...,N and Y can take values Y_j with j=1,...,M, $p(x_i|y_i)$ is the conditional probability of X_i (values that a specific feature can take) given Y_i (values that the target value can take). Finally, the mutual information is given by:

$$MI(X,Y) = H(Y) + H(X|Y) \tag{3}$$

The function compute Mutual information for all features, ranks them in descending order and selects the K top features (where K is input by the user).

Recursive Feature Elimination with cross validation Feature selection was done by performing Recursive Feature Elimination with Cross Validation (RFECV) since it was proved to work for training models that aimed at learning to rank ([1]). The name is quite self explanatory: this function takes a model, a training set, a cross validation strategy and a scoring function as inputs. It fits a model several times and removes features each time depending on the weights given to each on during the training of the latter. The crossing strategy that we used was a five folds one. After training the model at each iteration, predictions are made and a score (precision in our case) is computed. This allows the choice of an optimal amount of features for the training of the model.

3.4 Feature addition

The provided dataset is missing attributes for hotels that might provide better search results e.g. the amount of rooms a hotel room has. We are not able to acquire these possibly useful features but what we can do is improve the descriptiveness of the numerical features that we already have for a property. We decided to add additional features to provide more statistical measures for features that are present for properties in the provided dataset. These measures include *mean, median* and *standard deviation* (computed based on test and training dataset). All 3 of them give a good overview of the distributions, assuming the distribution of a given numerical feature per property is not dynamic over time. Knowing the information about the distribution for every feature might allow the model to achieve a better performance overall.

4 The Ranking Model

Our goal is to find a representation for a query-hotel combinations so that we acquire a numerical vector $x \in R^n$. This vector is then fed into ranking model $f : x \to R$ where the output is the score for our query-hotel pair. This ranking model f is optimized to score each query-hotel combination so that relevant hotels are scored higher.

We will be using an offline learning to rank model to compute the rankings for the query-hotel combinations. The literature covers a wide variety of offline learning to rank methods but we will choose a model that is able to optimize for information retrieval metrics such as nDCG ([12]). We chose to use LambdaRank because it fulfills this criteria ([4]).

4.1 LambdaRank

LambdaRank is based on the pairwise RankNet learning to rank algorithm ([3]). The model consists of multiple linear layers followed by a non-linear activation function excluding the final linear layer. The network computes $f : x \to R$ where the final output is the score of the query-hotel combination. The loss of the RankNet network is computed based on the output for two hotel pairs for a given query, and minimizes the number of incorrect inversions.

$$\lambda_{RankNet} = \sum_{d_i > d_j} log(1 + e^{-\gamma(s_i - s_j)}) \tag{4}$$

Where d_n is hotel n, γ is the hinge function and s_j is the predicted score for hotel n.

A common problem with offline learning to rank models such as RankNet is that sorting is non-differentiable. This makes optimization based on information retrieval metrics such as nDCG infeasable without extending the model further. LambdaRank manages to bypass this limitation by multiplying the loss by the nDCG score, effecting the gradient of the RankNet loss and allowing for optimization based on IR metrics such as ERR or nDCG. The final LamdaRank loss is defined by

$$\lambda_{LambdaRank} = \lambda_{RankNet} \cdot |\Delta NDCG| \tag{5}$$

4.2 Gradient Boosting

To improve the performance of the model, a Gradient Boosting Machine (GBM) will be used with LambdaRank as the objective function. The Gradient Boosting Machine is an ensemble model of decision trees that are trained in sequence. The algorithm will learn the decision trees by looking at the residuals error in each iteration.

5 Results

5.1 Training Setup

The lightgbm library ([11]) provided the gradient boosted LambdaRank model that was used during training. Default parameters of this model provided sufficient score of 0.38 although some hyper-parameter tuning was performed to find the best performing model.

The best performing dataset was the one that:

- Had imputed missing values by medians/means.
- Had removed frequently missing features.
- Contained additional columns for every numerical feature: mean, median and standard deviation per prop_id.

Each set of parameters was trained on a randomly sampled 70% of the training set. The remaining 30% of dataset was used for evaluation to oversee the model's performance. Every model was tested with *early stopping rounds* at 50 to prevent over-fitting. To ensure every model trains to the best of its capabilities *n_estimators* was set to 500.

5.2 Model evaluation and hyperparameter tuning

The best parameters were chosen based on an evaluation of 30% of the training data. The evaluation set was split into 5 different subsets to provide a more reliable score for the instance of the model. The evaluation method was the same as in the contest (i.e. nDCG@5). Although choosing optimal hyper-parameters might an extremely difficult problem ([7]), we used partial grid search to get an approximation of the best performing parameters. Grid search is widely used for hyper-parameter tuning ([6]) so it seemed like a suitable solution.

Learning rate and boosting type. The first tested parameters were: *learning rate* and *boosting type*. *Learning rate* values that are input for the grid search were presented in a set containing $\{0.06, 0.08, 0.1, 0.12, 0.14, 0.16, 0.18, 0.2\}$, where 0.1 is the default parameter. *Boosting type* possible parameters are $\{"gbdt", "goss"\}$ which stand for "Gradient Boosting Decision Tree" (default value) and "Gradient-based One-Side Sampling" respectively. The remaining values were not providing significant enough improvements to be included. The results are displayed in figure 3.

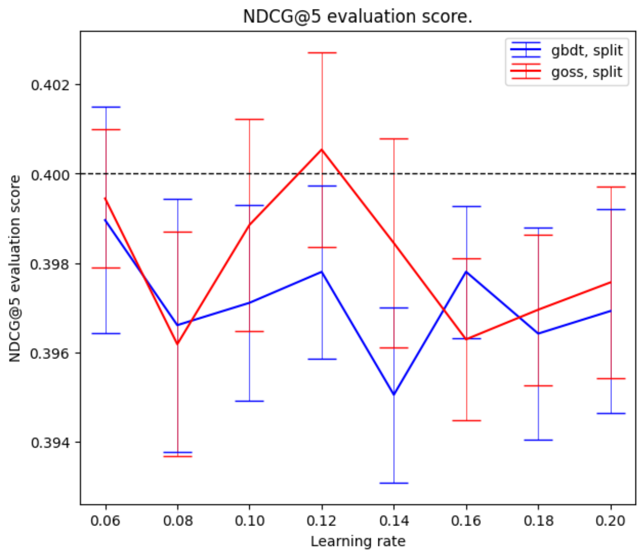

Fig. 3: Vertical bars represent the standard deviation, the points are computed as a mean score across all validation sets. The dashed line represents the acceptable threshold.

Alpha and lambda Second pair of parameters that were tuned were: *alpha* (L1 regularization term on the weights) and *lambda* (L2 regularization term on the weights). Both have default values of 0. Both were tested with a set containing $\{0, 1, 2, 5, 10\}$. The results can be seen in figure 4

Rest of parameters Grid search was performed on other parameters but did not provide significant improvement to the results so default values provided by the lightgbm library were used.

5.3 Final model

The hyper-parameters that were used in the best performing model were:

– $learning\_rate = 0.12$
– $boosting\_type = "goss"$
– $alpha = 5$
– $lambda = 2$
– $subsample = 0.85$
– $colsample\_bytree = 0.92$

The model was trained without early termination. Its performance can be found in table 1.

6 What we learned

We learned about the different aspects of big data preparation which is an essential part of Data Mining. These aspects range from how to come up with

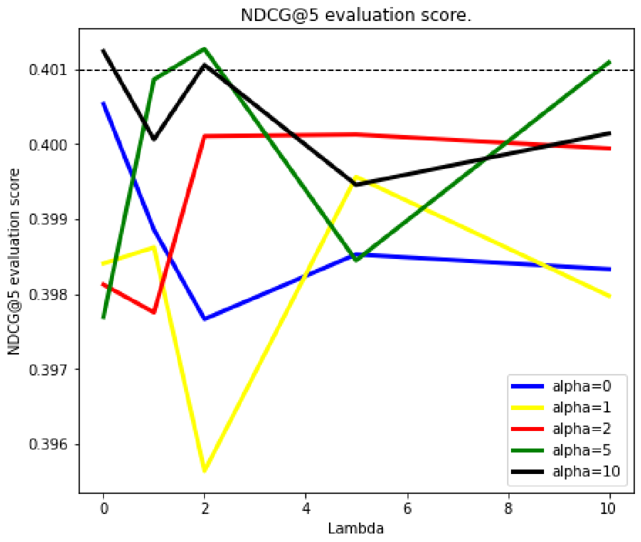

Fig. 4: Vertical bars representing standard deviation were omitted for better visibility, points are computed as a mean score across all evaluation sets. Dashed line represents acceptable threshold.

Validation set	1	2	3	4	5	Avg score
nDCG@5 score	0.407154	0.400613	0.403398	0.402656	0.401912	0.403147

Table 1: Results of the best performing model across all validation sets.

solutions when data is missing, bearing in mind the relevancy of the data that is used to replace empty gaps, to feature engineering that is used to keep the most relevant aspects of the data without losing too much of its value. It also became obvious that there are many methods to extract information, cleaning the data, rearranging it to achieve optimal performance during training, but those methods come with several advantages and disadvantages regarding suitability, computational power or processing time. We also realized how computationally expensive this science can be, since we ran into quite a lot of issues relating to a lack of computational power locally. This was bypassed by running the code on virtual machines. It was also challenging to deal with large class imbalance that usually results in biased models, which is why we considered undersampling or oversampling methods.

Furthermore we realized that large amounts of missing data is quite hard to deal with and just removing them takes away the 'spirit' of the data set and replacing them by constant values can result in heavy bias if not done correctly.

In the end we also learned that different implementations of the same method can have a significant impact on the overall score. This assignment gave us the opportunity to learn how to manage big datasets and complex models which are computationally expensive, in an efficient way.

7 Conclusion

We were provided with a query-hotel search dataset from Expedia and were tasked with creating a search engine that would provide satisfactory performance based on information retrieval metrics. We covered which pre-processing methods we hypothesised would be best suited for this task and how well they performed in practice. It can be added that the combinations of the pre processing methods could have been optimized in order to build a stronger and more suitable pipeline that might have resulted in overall better performances. It could also be discussed that we could have trained several learning to rank models in order to compare performances even though LambdaRank is kind of the current state of the art, but it might come out to be interesting so see the trade offs between time requirements and performances. We showed that Gradient boosted LambdaRank was well suited model for this task and produced optimal results which at time of writing stands at 7th place on Kaggle.

References

1. Bing Bai, Jason Weston, David Grangier, Ronan Collobert, Kunihiko Sadamasa, Yanjun Qi, Olivier Chapelle, and Kilian Weinberger. Learning to rank with (a lot of) word features. *Information retrieval*, 13(3):291–314, 2010.
2. Mohamed Bennasar, Yulia Hicks, and Rossitza Setchi. Feature selection using joint mutual information maximisation. *Expert Systems with Applications*, 42(22):8520 – 8532, 2015.
3. Chris J.C. Burges. From ranknet to lambdarank to lambdamart: An overview. Technical Report MSR-TR-2010-82, June 2010.
4. Christopher Burges, Krysta Svore, Paul Bennett, Andrzej Pastusiak, and Qiang Wu. Learning to rank using an ensemble of lambda-gradient models. In *Proceedings of the learning to rank Challenge*, pages 25–35, 2011.
5. Olivier Chapelle and Yi Chang. Yahoo! learning to rank challenge overview. In *Proceedings of the Learning to Rank Challenge*, pages 1–24, 2011.
6. Davide Chicco. Ten quick tips for machine learning in computational biology. *BioData Mining*, 10, 12 2017.
7. Marc Claesen and Bart De Moor. Hyperparameter search in machine learning. 02 2015.
8. Xiubo Geng, Tie-Yan Liu, Tao Qin, and Hang Li. Feature selection for ranking. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, page 407–414, New York, NY, USA, 2007. Association for Computing Machinery.
9. Jerzy W Grzymala-Busse and Ming Hu. A comparison of several approaches to missing attribute values in data mining. In *International Conference on Rough Sets and Current Trends in Computing*, pages 378–385. Springer, 2000.
10. P. Hart. The condensed nearest neighbor rule (corresp.). *IEEE Transactions on Information Theory*, 14(3):515–516, 1968.
11. Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in neural information processing systems*, pages 3146–3154, 2017.
12. Hamed Valizadegan, Rong Jin, Ruofei Zhang, and Jianchang Mao. Learning to rank by optimizing ndcg measure. In *Advances in neural information processing systems*, pages 1883–1891, 2009.
13. Suzan Verberne, H van Halteren, Stephan Raaijmakers, DL Theijssen, and LWJ Boves. Learning to rank qa data: Evaluating machine learning techniques for ranking answers to why-questions. 2009.