Data Mining Assignment 2 - Group 77

J. de Deijn (2516200), L. Scholte (2500114), and A. Kahali (2542764)

VU Amsterdam, De Boelelaan 1105,
1081 HV Amsterdam, The Netherlands, May 29, 2016

1 Introduction

The objective of this assignment is to predict what hotel an individual is most likely to book and create a ranking based on these predictions to present to the user. It is based on the Kaggle competition 'Personalize Expedia Hotel Searches' that was hosted from September 3, 2013 to November 4, 2013. We begin this report by discussing the approaches used by the top participants of the Kaggle competition. We will then perform an explanatory data analysis (EDA) to get more insight into the dataset itself. Next, we combine our insights from the EDA and the knowledge obtained from the approaches in the Kaggle competition to prepare the data and perform feature engineering. Finally, we discuss different modelling and evaluation techniques, explain the implementation procedure, and provide the results.

2 Business Understanding

It is well known that the online hotel booking business is very competitive. Therefore, one way for Expedia to obtain a competitive advantage would be an accurate method for ranking the hotel suggestions at each search query. Accurate here means that highly ranked items have a high chance of being clicked or even booked. From this perspective, Kaggle hosted a contest Personalize Expedia Hotel Searches during the International Conference on Data Mining (ICDM) 2013. In this section we discuss key methodological insights obtained from the Personalize Expedia Hotel Searches - ICDM 2013 Forum on Kaggle with a special interest in those obtained from the top contestants.

Notable insight from the forums include: (i) some hotel prices are extremely large and these should be treated as outliers, (ii) missing variables could either be meaningful missing values or meaningless missing values, (iii) it might be useful to randomly split data based on `srch_id` or on `booking_bool` so that you end up with an equal amount of booked and unbooked data. Finally, Adam Woznica (a moderator on the forum) stresses the presence of a position bias (i.e. the fact that some hotels are more likely to be clicked or booked since they were assigned a higher rank, which affects the performance of the position benchmark on both random and non-random impressions). Consequently, one must assume that the order of hotels within a certain `srch_id` in the test data is also the order that is displayed on the Expedia web-page itself.

Commendo (Michael Jahrer and Andreas Toescher) obtained the highest score on the leader-board (but was not the winner because they failed to comply with all the competition requirements), states that the best single model is the LambdaMART, which is a learning to rank algorithm based on Multiple

Additive Regression Tree (MART). They find this by comparing, amongst others, the following models: Stochastic Gradient Descent (SGD), Neural Network, and Gradient Boosted Decision Trees. For their final solution they use a linear blender for 24 predictors in combination with the Automatic Parameter Tuner (APT1) search algorithm. They use only 10% as validation set. Moreover, the Commendo teams stresses that time and price features do not have explanatory power, but instead extra performance can be achieved by introducing statistical features.

Owen Zhang, who takes the second place on the leader-board, used an ensemble of 26 Gradient Boosting Machines (GBM) and evaluated the ranking using the Normalized Discounted Cumulative Gain (NDCG) loss function. He defined position, price and location desirability as the most important features. He takes the following approach. Firstly, for the data-preprocessing, Zhang takes three steps: (i) imputing missing values with a negative value, (ii) bounding numerical values, (iii) down sampling negative instances for faster learning (and predictive performance). Secondly, he defines five groups of features: (i) the original features, (ii) numerical features averaged over `srch_id`, `prop_id`, and `destination_id`, (iii) composite features, (iv) categorical features converted into numerical features, (v) the estimated position which is the average of the position of the same hotel in same destination in the previous and next search and a numerical feature based on `prop_id`, `destination_id`, and `target_month`.

Jun Wang and Alexandros Kalousis, who take the third place in the competition, discuss that a linear model may not handle the categorical (discrete) features and hence take a nonlinear approach, LambdaMART. They fill the missing values of hotel description with the worst case scenario. This is because Wang and Kalousis assume that users do not like to click on or book hotels with missing values. They also set the missing values of competitor descriptions to zero since there is no significant difference between Expedia and competitors.

binghsu & MLRush & BrickMover Team, a collaboration of eight persons who take the fourth place, used models such as GBM and LambdaMART, but also the Deep Neural Network (DNN). However, they highlight that single models cannot get the best results and that ensemble techniques offer flexibility and diversity. Therefore they use the z-score technique, GBM ensemble, Deep Learning ensemble, and Listwise Ensemble. The latter leads to the final and best model, LambdaMART, which is robust in ranking hotels. The teams take the following approach in the preprocessing: (i) they deal with the missing data by taking the first quartile to represent it, (ii) they use only 10% of the data set, sampled by `srch_id`, (iii) they used balanced positive and negative data, (iv) they split data by `prop_country_id`. They find that the normalized features `price_usd`, `prop_location_score1` and `prop_location_score1` are good in their models.

3 Data understanding

The training and test set contain data of 199 795 respectively 199 549 searches on Expedia websites between November 2012 and June 2013, yielding a total of 4 958 347 respectively 4 959 183 search results. Every search result contains

information about the corresponding search, customer, property and possible competitors. All 54 original, non-transformed attributes of the training set[1] (excluding `date_time`) are summarized in Table 1. Notable observations are that very few instances are actually clicked or booked and that the main activity originates from one country (probably the US, where Expedia is founded). Moreover, only 12.8% of these customers search for properties abroad.

Among the integer type data there are hardly any missing values, 49.4% of the customers are couples without children and many search for one room, for one night and/or for the next day. Also, properties on average get better customer review scores than expected from their star rating. The attributes in the lower part of Table 1 often have many missing values and/or many (big) outliers, which are in most cases explicable (e.g. the outliers of `prop_log_historical_price` are mainly due to zero values when the property's stock was not traded in the last period), but not always: 0.3% of the instances have `comp_inv` = -1. This is strange, because 3.6% of these instances are booked, whereas the description implicitly suggests that this means Expedia does not even have availability. We will cover the treatment of these missing and outlier values in Section 4.

Table 1: The original, non-transformed training attributes summarized.

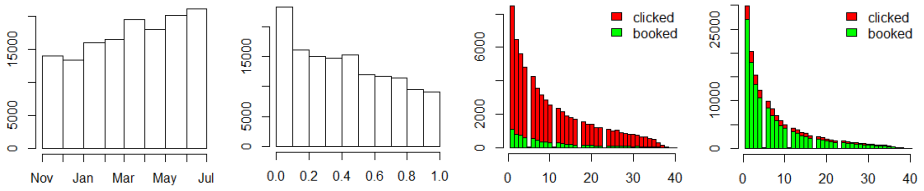
Booleans	%TRUE	%Miss.				
srch_saturday_night_bool	50.2%	0%	*Total # instances:*			
random_bool	29.6%	0%	4 958 347			
prop_brand_bool	63.5%	0%	*Note:*			
promotion_flag	21.6%	0%	* *only in training set*			
click_bool*	4.5%	0%	** *multiplied by 2*			
booking_bool*	2.8%	0%	*** *for all 8 competitors combined*			
			† *not within box-and-whisker range*			
Integers	Range	%Miss.	#Uniq.	IQ Range	Mean	Mode
srch_id	{1, . . . , 332 800}	0%	199 795	-	-	-
site_id	{1, . . . , 34}	0%	34	-	-	5 (61.0%)
visitor_location_country_id	{1, . . . , 231}	0%	210	-	-	219 (57.2%)
prop_country_id	{1, . . . , 230}	0%	172	-	-	219 (61.1%)
prop_id	{1, . . . , 140 800}	0%	129 113	-	-	-
srch_destination_id	{2, . . . , 28 420}	0%	18 127	-	-	-
position*	{1, . . . , 40}	0%	40	-	-	-
srch_length_of_stay	{1, . . . , 57}	0%	-	{1, 2, 3}	2.38	1 (43.2%)
srch_booking_window	{0, . . . , 492}	0%	-	{4, . . . , 48}	37.47	1 (7.6%)
srch_adults_count	{1, . . . , 9}	0%	-	{2}	1.97	2 (65.5%)
srch_children_count	{0, . . . , 9}	0%	-	{0}	0.35	0 (76.9%)
srch_room_count	{1, . . . , 8}	0%	-	{1}	1.11	1 (91.2%)
prop_starrating	{0, . . . , 5}	0%	-	{3, 4}	3.18	3 (39.3%)
prop_review_score**	{0, . . . , 10}	0.1%	-	{7, 8, 9}	7.56	8 (43.2%)
Floats	Range	%Miss.	#Outl.†	IQ Range	Mean	Median
visitor_hist_starrating	[1.41, 5]	94.9%	0	[2.92, 3.93]	3.37	3.45
visitor_hist_adr_usd	[0, 1958.7]	94.9%	11 364	[109.8, 213.5]	176.0	152.2
orig_destination_distance	[0.01, 11 670]	32.4%	417 607	[139.8, 1501]	1301.0	386.6
prop_location_score1	[0, 6.98]	0%	0	[1.79, 4.04]	2.87	2.77
prop_location_score2	[0, 1]	22.0%	271 139	[0.019, 0.1805]	0.1304	0.069
prop_log_historical_price	[1.61, 6.21]	0%	715 654	[4.69, 5.39]	5.044	4.91
price_usd	[0, 19 730 000]	0%	285 363	[85, 185]	254.2	122
srch_query_affinity_score	[−326.6, −2.494]	93.6%	12 299	[−30.77, −13.35]	−24.15	−20.45
gross_bookings_usd*	[0, 159 300]	97.2%	12 115	[124, 429.8]	386.3	218.4
comp_rate_percent_diff***	[2, 1 002 000]	92.6%	219 971	[7, 19]	38.2	11
Integer-valued factors	Range	%Miss.	%(-1)	%(0)	%(1)	
comp_rate***	{−1, 0, 1}	78.1%	2.5%	16.5%	2.9%	
comp_inv***	{−1, 0, 1}	76.8%	0.3%	21.6%	1.3%	

The characteristics of the attributes in the test set are very similar. Our job is to rank the properties in each user search of the test set, such that the property *most likely to be booked* is displayed first on the Expedia website. We could also choose to rank on probability of clicking, hoping that a click is often followed by a booking. However, we really need to focus on the probability of booking. The reason for this is that some attributes influence the conditional

[1] Descriptions of the attributes can be found on https://www.kaggle.com/c/expedia-personalized-sort/data

probability of a booking given a click (62.3% overall). In particular, this probability can decrease to below 40% in case of bad property reviews and increase when the property exceeds expectations (e.g. $\frac{778}{994} = 78.3\%$ for `prop_starrating` $= 2$, `prop_review_score` $= 8$ and `prop_brand_bool` and `promotion_flag` both `TRUE`).

Another interesting observation is that properties that are part of a major hotel chain require fewer promotions (17.9% versus 28.0%). Furthermore, Fig. 1a suggests there is an increasing trend in bookings. However, regarding `srch_booking_window` simultaneously, the pattern is more likely to be of seasonal nature. When the summer approaches, more and more people are searching for nice holiday trips. Next, Fig. 1b shows that customers are logically more interested in booking a relatively cheap property. In this figure, the cheapest (most expensive) hotel within a search has relative price 0 (1). Finally, Fig. 1c and 1d illustrate the so-called position bias: the higher a property's position the more often it is clicked, regardless of the order of the properties. However, when the properties are cleverly ordered, the fraction of clicks resulting in a booking is much higher than when they are randomly ordered. This illustrates the importance of our task.



(a) Bookings per month

(b) Bookings per price class within search

(c) Clicks/bookings per random position within search

(d) Clicks/bookings per non-random position within search

Fig. 1: Histograms of some of the most important findings

4 Data preparation

As we have seen in Table 1, the floats and competitor-related attributes contain many missing values and outliers. This section describes how we treated these values and which features we extracted from the data to build our models on.

A first observation to make is that fifteen of the original attributes are unique within each search. These attributes are therefore only helpful in transforming/creating other features; They do not have any predictive power on their own. As a result, we do not have to bother about the 29 missing values of `date_time`. Furthermore, the non-predictive location identity attributes can be used to estimate the missing values of `orig_destination_distance` in a proper way by taking medians of similar pairs of origin and destination.

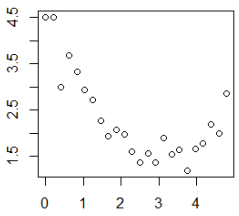
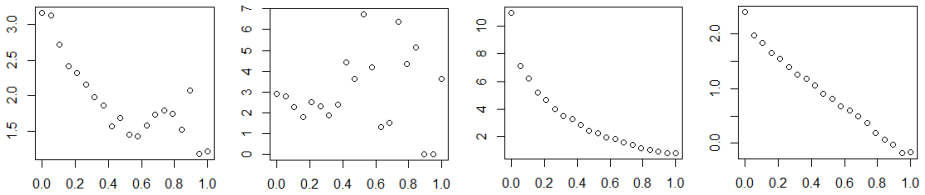


Fig. 2: Percentage of bookings per star rating difference class

Obvious new features to create from the visitors' historical prices and star ratings are their absolute differences with respect to these

of the properties. For example, `starrating_diff` is the absolute difference between `prop_starrating` and `visitor_hist_starrating`. For the latter one, we replaced missing values by the median value. Fig. 2 shows the resulting feature has a clear relation with the probability of booking. However, replacing missing values by the median does not work for `price_usd`, because these prices are highly varying due to different price conventions among countries and prices can be per night or per stay. We solve the issue on prices per night/stay partly by replacing the large outliers by the 99.5%-quantile ($770) and taking the log prices. We also tried to normalize the prices per `site_id`, because we expected that all prices on one site are either per night or per stay. Unfortunately this turned out not to be the case. However, normalizing the prices per country successfully solved the issue on different price conventions per country, as can be seen in Fig. 3a. After transforming `price_usd` in this way, we take the absolute difference with `visitor_hist_adr_usd` and call this feature `price_diff2`. Comparing this to Fig. 3b shows that its relation with bookings is much more obvious than for `price_diff`. For this one, we just took absolute differences of the logs of `price_usd` (with missing values simply replaced by the median) and `visitor_hist_adr_usd` as suggested on the Kaggle forum.



(a) (b) (c) (d)

Fig. 3: Percentage of bookings per normalized class of price difference (a,b) and median position (c). In (d), the log of (c) is shown.

The attributes that are missing in the test set obviously cannot be used in our model. Since we want yet to be able to use the valuable `position` information, we create new features containing for each property the median, mean and standard deviation of its normalized position within a search. Here we only take statistics over results from nonrandom searches, for reasons described earlier in Section 3. Properties occurring only in random searches are estimated at median/mean 0.5 and standard deviation 0.25, according to median values over all properties. Fig. 3c illustrates how valuable especially the median normalized position (called `med_relrank`) is, possibly even for linear models (see Fig. 3d).

As mentioned in Section 2 the three property scores are valuable features. However, both `prop_location_score2` and `srch_query_affinity_score` have many missing values, which we replace with their smallest score over all properties with similar destinations as suggested on the Kaggle forum. We tried to replace zero values for `prop_location_score1`, `prop_starrating` and the feature `prop_review_score` since these have specific meanings. For example, we supposed that no review is less worse than a bad review and accounted for this. However, it turned out the used algorithms were able to recognize these specific values and their consequences for the booking probability. Hence, we kept them

as zero values in our final feature set. Moreover, we replaced missing values for `prop_review_score` by zero's.

Since the many missing values of the competitor-related attributes cannot be estimated properly, we must create new features. We assume that missing competitive data means that there is no competition and create the following features: (1) The number of competitors with availability; (2) A boolean indicating whether Expedia has the lowest price available and; (3) The percentage difference (either positive or negative) between Expedia's price and the lowest competing price. We think that these three features cover the most interesting information contained in the 24 original attributes. We also create a feature called `comp_info` that combines all `comp_info` and `comp_rate_percentage_diff`. In particular, we multiply the rate by the percentage difference and take the maximum price difference from the eight competitors. If no information is available, we assign value 0 to `comp_info`.

The last features we added are, first, the log of the `price_usd`, i.e. `log_price`. This is because we reduce the effect of outliers and the high variation in the prices. Next, we calculate its difference w.r.t. `prop_log_historical_price`, indicating how much higher/lower the price is than the price for which the property was sold last. We call this difference the discount, i.e. `log_price_discount`. Missing values for discount are given the value 0 (no discount). Second, inspired from Fig. 1b, the relative price rank. This `rank_price` is defined as the rank of hotel i based on a price within a `srch_id`, where the lowest price receives rank 1. In the spirit of the Commendo team we also add the mean, median and standard deviation of the discount (`log_price_discount`) and price rank (`rank_price`) feature per hotel, indicated by `prop_id`. Finally, we added the empirical probability of a click or booking per `prop_id`, which is the mean of `click_bool` and `book_bool` per hotel. However in our initial feature evaluation phase these probability features seemed to result in unrealistically high NDCG values, thus we decided to omit the use of these features.

5 Ranking Algorithms

After some trial and error with various linear and non-linear algorithms, as we have seen used by the previous winners of this competition (in particular the success of LambdaMART), there were two algorithms that performed well in our initial trials. Both included in the `Gbm` package in R. These are the LambdaMART (pairwise distribution in `GBM`), and Gradient Boosted Logit Trees (bernoulli distribution in `GBM`).

5.1 LambdaMART

LambdaMART is a model used in Learning-To-Rank. We define the Learning-To-Rank problem as follows: we have queries $x_i \in \chi$ that contain a list of N candidate items $S_i = s_{i,1}, s_{i,2}, ..., s_{i,N}$. Then the task is to output a ranking of items S_i, in particular we want a permutation π_i of indices $\{\sigma(1), \sigma(2), ..., \sigma(N)\}$, which we must produce with some mapping $f(x_i, S_i)$. The LambdaMART uses gradient boosting methods for the optimization of some cost function such as the Mean average precision (MAP) or the Normalized discounted cumulative gain

(NDCG). As the name suggests the LambdaMART is nested in the Multiple Additive Regression Trees (MART) family, which is a method that builds a single model as a weighted sum of an ensemble of regression trees. The objective function is

$$f(x) = \arg\min_{g} \mathbb{E}[Loss(y, g(x))|x], \tag{1}$$

where $f(x) = \sum_{k=1}^{N} g_k(x)$ is approached as the combination of $g(x)$ trees (see the following `tutorial`). The optimization problem is tackled using the gradient. Because of non-differentiability at points in this loss-functions, LambdaMART employs a method from LambdaRank. In this method for item pairs (i, j), a value $\lambda_{i,j}$ acts as the necessary gradient where $\lambda_{i,j}$ can be seen as the driver that pushes items up or down in the ranking [3]. Then using gradient descent the algorithm generates the ensemble by adding trees at iteration j that form the best fit to the residuals of the ensemble at iteration $j-1$. This ensembling at iteration j is summarized by the equation:

$$V_j(x_i) = V_{j-1}(x_i) + \eta \sum_{k=1}^{D} \gamma_{kj} \mathbb{1}_{\{x_i \in R_{kj}\}}. \tag{2}$$

With fixed learning rate η (< 1), V_j the score at step j, R_{kj} is the tree node with training examples $\{x_i, r_i\}_{i=1}^{M}$, and γ_{kj} the weighting of the combination. For this model we mainly focus on finding a good combination of hyper-parameters D (tree depth, # of nodes), M (minimum number of observations per node), T (number of trees) and the learning rate η. To be able to train the LambdaMART we need to specify a response variable which we define as follows:

$$r_i = \min\{5, 5 * \texttt{bookin\_bool}_i + \texttt{click\_bool}_i\}. \tag{3}$$

Note that following the logic of Zhang we use the Normalized Discounted Cumulative Gain (NDCG) as the objective function in the optimization problem, and the `gbm.perf` function in the GBM package to estimate the optimal number of trees using an out-of-sample estimate with the validation set.

5.2 Gradient Boosted Logit Trees

The Gradient Boosted Logit Trees algorithm is also a member of the MART family (gradient boosted machines) [4]. It is based on the Bernoulli distribution and built with the principles of logistic regression. Thus, being a binary response model, we can readily use either `booking_bool` or `click_bool` as the response variable. In fact we experiment with both and find a good method to incorporate information from both models. The specific algorithm here is the AdaBoost algorithm [2] [1]. We tweak the same hyper-parameters as with the LambdaMART. After obtaining the ranking vectors from these models we also experiment with taking convex combinations of these vectors (model blending) as we have seen done by Commendo.

5.3 Evaluation

Following the team of binghsu we don't use the entire data for evaluating the discussed models, but we only use a sub-sample of 30% of the entire data for training, validation and testing. This is possible due to the sheer size of the available data, which in turn relieves us of the need for k-fold cross validation. We then split the data following the train, validate, and test scheme. Thus we split our training data into a set for training (65%), the cross validation set (10%) and finally a test set (15%) that we leave aside for evaluating the models we have optimized on the validation set. We also control for balancedness of the subsets. The proportion of clicks in the entire training set is 0.0447 and the proportion of bookings is 0.0279. By randomly sampling by `srch_id` we aim to adhere to these proportions. Indeed the for the 30% sub-sample we have proportions of 0.0446 and 0.0279. For the splits of this sub-sample in training, validation and test sets we have very similar results. As expected only the validation set has proportions that deviate slightly from these with the values being 0.0444 and 0.0283. As a metric to evaluate the prediction file we use the NDCG defines as follows: suppose i represents one of the `srch_id`'s and N_i the number of hotels at each query. Note that we have $N_i \leq 38$ we then have for score $\{v_{ij}\}_{j=1}^{38}$ (for hotel j):

$$NDCG^{(38)} = \frac{1}{N}\sum_{i=1}^{N}\frac{DCG_i^{(38)}}{IDCG_i^{(38)}}, \qquad DCG_i^{(38)} = \sum_{j=1}^{38}\frac{2^{v_{ij}}-1}{\log_2(j+1)}. \tag{4}$$

6 Results

Using the cross validation set we start experimenting with training the models with various feature combinations, where-after the best model is applied to the test set. We begin with training the LambdaMART model by adding features until we have used all of them to see how much they influence the results, where-after we find some optimal combination of hyper-parameters. Then, depending on the relative influence of the attributes with some optimal hyper-parameter setting, we remove the least significant attributes. After training the LambdaMART we use a similar strategy to train the Gradient Boosted Logit Trees model, in particular we, rather crudely, use the optimal hyper-parameter settings we found with the LambdaMART.

6.1 Features

Using the property related attributes that are readily available (numeric attributes without any missing data) with the default hyper-parameters $\eta = 0.05, T = 100, D = 5, M = 30$ we obtain an NDCG of 0.4402. If we use this same score vector that determines the ranking of each item per search ID, take a random permutation of it and compute the NDCG we get a value of 0.3464. Thus even with the minimal amount of attributes and no feature engineering whatsoever the LambdaMART model gives a higher NDCG than randomly ranking

properties. We call this model M_0. The next step is to add the property related attributes that contain the imputed missing values. We have a model now that includes

```
prop_location_score2,prop_location_score1,price_usd,
prop_starrating,prop_review_score, prop_log_historical_price,
srch_query_affinity_score,
```

which we call M_1. With this model we get an NDCG value of 0.4868 where `srch_query_affinity_score` has a relative importance of 0 so we might as well remove it but we leave it (knowing that this rel. influence changes through parameter tweaking). Next, we add `log_price`, `rank_price` and `log_price_discount` and obtain an NDCG of 0.4937 which we define to be M_2. By adding `comp_info` we in fact get a lower NDCG of 0.4930 (M_3). Model M_4 includes `price_diff` and `starrating_diff` with an NDCG value of 0.49461 and model M_5 includes all the statistical features with an NDCG of 0.4955. Lastly, in model M_6 we add the `price_diff2` and the statical features of `relrank` we get an NDCG of 0.4980.

6.2 Hyper-parameters

When we start tweaking the hyper-parameters we notice changes in the relative importance of the used features, thus before removing any of them we try to find some optimal parameter setting for the model. After doing a crude grid search (i.e. trying various combinations of parameters by hand because each model training iteration takes a lot of time) we find that $\eta = 0.1, T = 500, D = 10, M = 30$ gives the best NDCG of 0.5141 (we call this model M_7). Increasing T did not always result in a better score after $T = 500$, and the 0.05 increase in the learning rate (thus increasing the step size in each iteration) possibly allows for the avoidance of less-optimal local optima. With the default hyper-parameter settings we had 9 variables with a relative influence below 1% and we now have that 7, among which are the statistical features of `relrank` and `log_price`. We remove the insignificant features and make a selection of features that is both small enough to avoid over-fitting as well as potent enough to attain a high NDCG. With this model M_8 we obtain an NDCG of 0.5152 on the validation set which is now our best model. When we use this model with the test set we get a value of 0.5107. For the binary response models after some trial and error, using a similar set of features and the same hyper-parameter settings we obtain an NDCG of 0.4969 on the test set with the Gradient Boosted Logit Trees when we take `click_bool` as the response variable (M_9). When we let `booking_bool` be the response variable the NDCG becomes 0.50297 M_{10} on the test set.

6.3 Model Combinations

Following the winner of the competition we also experiment with combining models, in particular suppose we have vectors of scores $\{V_i\}_{i=1}^N$ generated by either one of the previous models. Then we take a convex combination of the

vectors $\hat{V} = \sum_{i=1}^{N} V_i w_i$, where $w_i \geq 0$ for all i and $\sum_i w_i = 1$. We start with combining the best Bernoulli response variable models. It would be natural to let $w_1 = \frac{1}{5}$ for the `click_bool` model (M_9) and $w_2 = \frac{4}{5}$ for the `booking_bool` model (M_{10}) (as mentioned in the data understanding section we should focus more on the probability of booking.). This combination gives an NDCG of 0.5038 (M_{11}) on the test set. Possibly more interesting is combining M_{11} with the best LambdaMART to get $w_1 V_{M_{11}} + w_2 V_{M_8}$. We experiment with various combinations for $w_1 \in \{0.1, 0.15, \ldots, 0.5\}$ and $w_2 \in \{0.5, 0.55, \ldots, 0.9\}$ on a separate validation set and find the best pair (w_1, w_2) to be $(0.3, 0.7)$ which is our final model M_{12} with an NDCG of 0.5111 on the test set. Following some advice on the Kaggle forums we also experiment with normalising the vectors $\{V_i\}_{i=1}^{N}$ using for instance the logistic function, but this only results in an increase in NDCG for the combination of binary response models (NDCG = 0.5040) and a decrease in the final combination (NDCG = 0.5108) so we stick with the final model M_{12}.

7 Conclusion

In this report we have given a summary of the steps we have taken and the methods we have employed in producing the final ranking of hotels in the 'Peronalize Expedia Hotel Searches' competition. We noticed how important feature engineering and selection turned out to be; Where some features were of high impact, others showed little significance and only rigorous experimentation could give us answers as which are the most useful. This was also because changing the hyper-parameters of the models also changed the way the algorithm rated the attributes in terms of relative influence, and hence we found attributes that served the other competitors at Kaggle very well did not the same for us. What was the same however was the use of LambdaMART, this model gave promising results from the very beginning and was the best (non-combined) model in the end. Model combining also increased the NDCG scores and there is still a lot space for experimentation. For instance an interesting idea would be to experiment with larger model combinations with a higher variety of model types. Also it could be possible to implement an efficient grid search algorithm to further tune the hyper-parameters. Further experimentation and research could alleviate the current discrepancy between our final score and the one of the top contenders in the Kaggle competition.

References

1. Freund, Y., Schapire, R.E.: A Decision-theoretic Generalization of On-line Learning and an Application to Boosting. In: Journal of Computer and System Sciences, vol. 55, pp. 119-139. (1997).
2. Minka, T. P. (2003). A comparison of numerical optimizers for logistic regression. Unpublished draft.
3. Burges, C. J. (2010). From ranknet to lambdarank to lambdamart: An overview. Learning, 11, 23-581.
4. Friedman, J.H.:. Greedy Function Approximation: A Gradient Boosting Machine. In: Annals of Statistics, vol. 29, pp. 1189-1232. (2001).