Assignment 2 Data Mining Techniques

Reinier van Elderen[1 [2619743]] and Caspar Hentenaar[1 [122090678]]

Vrije Universiteit, De Boelelaan 1105, 1081HV, Netherlands

This report reviews the process of creating a model for ranking Expedia hotel searches, based on the original Kaggle competition: "Personalize Expedia Hotel Searches - ICDM 2013". Expedia has compiled a dataset comprising user search query results, hotel characteristics, and competitor information. The objective of both the original competition and this assignment is to devise an algorithm that ranks hotel searches, thereby ensuring that the hotels most likely to be booked or clicked are displayed at the top. This report delves into the dataset, applies feature engineering to enhance the data, explains our model and evaluation methodology, and finally, discusses potential practical applications of our model. An examination of related work precedes the presentation of our solution.

1 Business understanding

This section examines the three highest-scoring solutions from the original competition. These winners showcased their approaches at the IEEE International Conference on Data Mining.

Despite their individual approaches, all winners of the Expedia Hotel Ranking Contest share common elements, the most notable being the use of tree-based modeling methods. The first-place winner, Zhang and Woznica (2013), utilized an ensemble of Gradient Boosting Machines (GBM), a decision tree-based learning paradigm, trained with a loss function incorporating the Normalized Discounted Cumulative Gain (NDCG). Wang and Kalousis (2013), the second-place winner, and Liu et al. (2013), the third-place finisher, both deployed LambdaMART, another tree-based learning model. LambdaMART combines two machine learning algorithms - LambdaRank and MART (Multiple Additive Regression Trees) - and is specifically designed for solving ranking problems. By adopting LambdaMART, both Wang and Liu were indirectly leveraging the power of gradient-boosting trees, akin to Zhang.

The preprocessing and feature engineering stages, however, revealed different strategies across the top three solutions. For feature engineering, Zhang employed features which average numerical features over searches, properties and destinations. Moreover he transformed categorical features to numerical features by averaging the target over the categories. Lastly he came up with features for estimating the position of a hotel in the previous and next searches, which also proved valuable. Missing values were imputed with a negative value. In contrast, Wang filled missing values with worst-case scenarios. Moreover, he implemented difference features to indicate a match or mismatch between the historical data and given data. He also constructed a proxy for hotel quality by estimating the probability that a hotel was clicked or booked, as we replicate in section 3.2. Bingshu took a unique approach with listwise features, focusing on rankings within a search instead of relying solely on numerical values.

The preprocessing and feature engineering stages, however, saw different strategies across the top three solutions. Zhang averaged numerical features across searches, properties, and destinations . He also transformed categorical features into numerical ones by averaging the target over the categories. Further, he created features to estimate the position of a hotel in previous and subsequent searches, proving valuable. Zhang imputed missing values with a negative value. Conversely, Wang filled missing values with worst-case scenarios, implemented difference features to denote a match or mismatch between the historical data and given data, and constructed a proxy for hotel quality by estimating the probability that a hotel was clicked or booked, as we replicate in section 3.2. Liu et al. (2013) took a unique approach by focusing on listwise features, which rely on rankings within a search, rather than solely on numerical values.

The specific combination of preprocessing methods and the choice of model were unique to each winner's approach. Zhang's utilization of the NDCG loss function in conjunction with a GBM ensemble appears to have optimized his model's performance for this specific problem. Liu et al. (2013), meanwhile, highlighted the importance of intra-search ranking. Despite each participant's distinctive strategy,

a combination of feature engineering and tree-based modeling approaches led to their success in the competition. Based on these insights, our study will mainly focus on tree-based methods and implement various features as outlined above. More specifically, we will employ the LambdaMART ranker as implemented in LightGBM. Moreover, we plan to utilize several features, including normalized features, intra-search ranking features, and desirability features, similar to the strategies employed by the aforementioned authors.

2 Data understanding

The dataset is divided into a training set and a test set, roughly split 50/50. The training dataset comprises 4,958,347 rows and 54 columns, with data spanning from November 1, 2012, to June 30, 2013. The test set consists of 4,959,183 rows and 50 columns. The discrepancy of four columns between the two arises from the target columns—click, booking, booking price, and position in the original Expedia search—not being included in the test set.

The dataset columns provide information about the visitor, such as the mean historical rating, mean historical price per night, and the country in which they are located. Furthermore, details about the property (hotel) are included, such as the ratings of the hotel and room, a location desirability score, and price. Lastly, data about the search and competitor pricing are incorporated, such as the number of people, length of stay, and the availability of the given hotel from the competitor.

Since the target is not provided for the test set, we will create a model validation set ourselves. For this purpose, we will use 15% of the training data.

Below, a table containing some descriptive statistics is provided for a subset of the training set columns. From this table, we can observe that the data encompasses 199,795 searches returning 129,113 different hotels. There seem to be outliers present in the `price_usd` column—the maximum price of 19.7 million dollars appears unusually high. However, this could be the result of a speculative search from someone curious about the cost of staying in a top-tier hotel for an extended period.

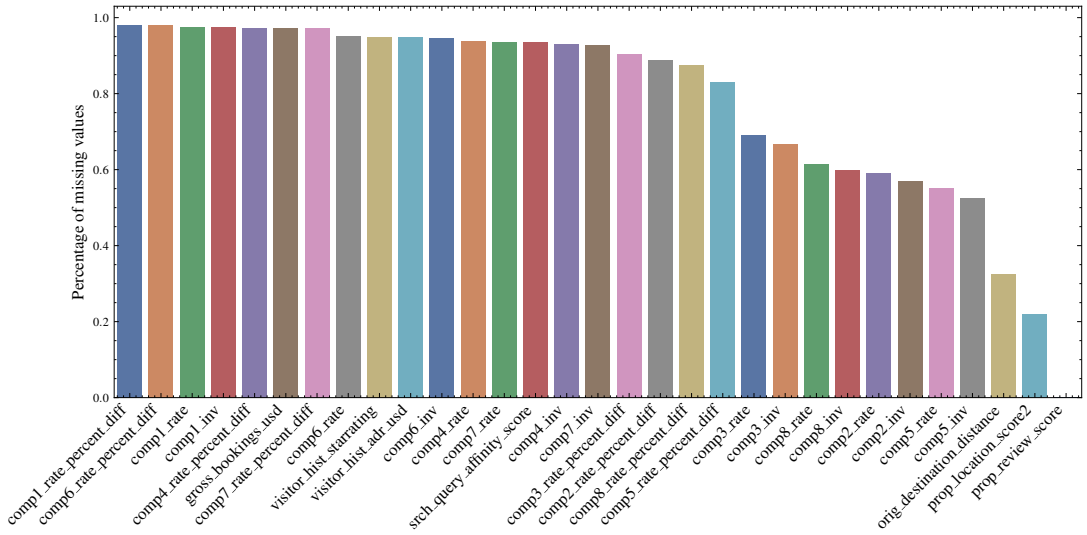
Table 1: Descriptive statistics by variable

	unique	*min*	*max*	*mean*	*null %*
srch_id	199795	1.00	332785.00	166366.56	0.00
site_id	34	1.00	34.00	9.95	0.00
visitor_location_country_id	210	1.00	231.00	175.34	0.00
visitor_hist_starrating	312	1.41	5.00	3.37	94.92
visitor_hist_adr_usd	7799	0.00	1958.70	176.02	94.90
prop_country_id	172	1.00	230.00	173.97	0.00
prop_id	129113	1.00	140821.00	70079.18	0.00
prop_starrating	6	0.00	5.00	3.18	0.00
prop_review_score	10	0.00	5.00	3.78	0.15
prop_brand_bool	2	0.00	1.00	0.63	0.00
prop_location_score1	337	0.00	6.98	2.87	0.00
prop_location_score2	9342	0.00	1.00	0.13	21.99
prop_log_historical_price	392	0.00	6.21	4.32	0.00
n price_usd	76465	0.00	19726328.00	254.21	0.00
srch_destination_id	18127	2.00	28416.00	14042.63	0.00
srch_length_of_stay	36	1.00	57.00	2.39	0.00
srch_booking_window	429	0.00	492.00	37.47	0.00
srch_adults_count	9	1.00	9.00	1.97	0.00
srch_children_count	10	0.00	9.00	0.35	0.00
srch_room_count	8	1.00	8.00	1.11	0.00
srch_saturday_night_bool	2	0.00	1.00	0.50	0.00
srch_query_affinity_score	199387	-326.57	-2.49	-24.15	93.60

Let us then consider the missing values in the dataset. Figure 1 displays the percentage of missing values in the data by column. The 30 columns with the most missing values are displayed. From the plot, it becomes apparent that the competitor and visitor history features are missing the most data.

Next, we address the missing values in the dataset. Figure 1 shows the percentage of missing values in the data by column, displaying all columns with missing values. The plot reveals that the competitor and visitor history features have the most missing data.

Fig. 1: Percentage of missing values by column



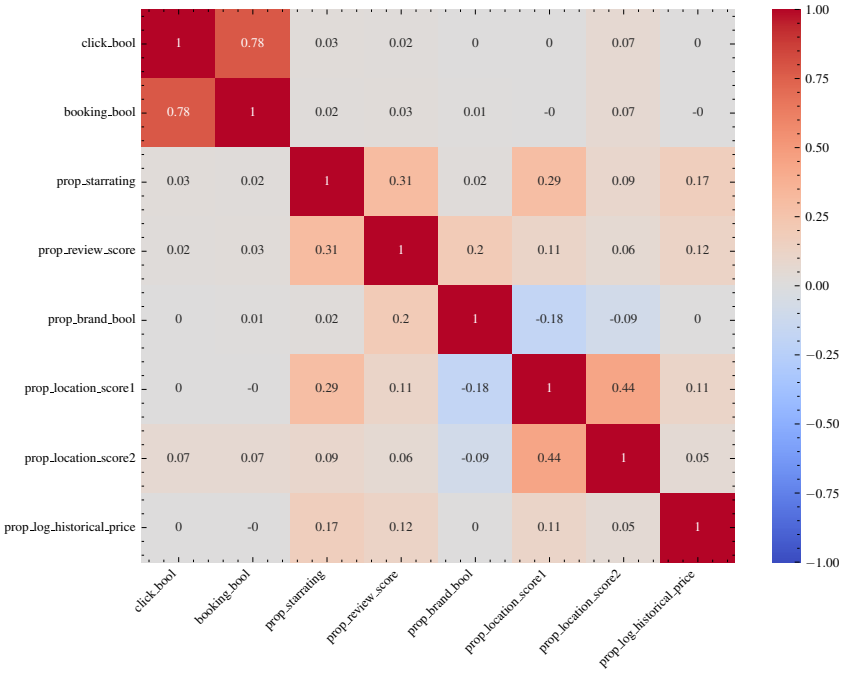
In addition to a substantial number of missing values, the dataset is quite imbalanced. Table 2 provides the relative count of target classes. Remarkably, positive training instances (search results that were clicked on or booked) comprise only 4.5% of all data. To address this imbalance, Zhang and Woznica (2013) downsampled negative instances (those not booked nor clicked) and reported improved training time and predictive performance. However, we were unable to replicate these results, as discussed in Section 4.

Table 2: Relative count by category

	Relative count
No booking or click	0.955
Booking	0.028
Click	0.017

Figure 6 investigates the correlation between property characteristics and the target variables `click_bool` and `booking_bool`, as well as the correlation among the property characteristics themselves. Likely due to the extensive size of the dataset, all displayed correlations are significantly different from 0, even though rounding to 2 digits might make some correlations appear to be 0. It is worth noting that `prop_location_score2`, `prop_starrating`, and `prop_review_score` all exhibit a positive correlation, as one might expect from these features. Interestingly, `location_score_1` appears to be nearly uncorrelated with both target variables. The same holds true for `prop_brand_bool` and `prop_log_historical_price`, features indicating whether a hotel is part of a brand and the logarithm of the historical prices for a given hotel, respectively.

Fig. 2: Correlation plot between target variables and hotel characteristics



3 Data preparation

3.1 Handling missing values

As observed in the preceding section, several features have missing values. We need to devise an appropriate method for imputing these values. Given that not every feature can be imputed in the same manner, we propose strategies tailored to each feature based on common sense, literature review, or trial and error.

One feature requiring special handling is `orig_destination_distance`, for which around 30% of all values are missing. We elected to impute missing values with the mean of all recorded values, as setting a distance measure to zero would be unrealistic.

For the `location_score_2` feature, we found that some searches had missing values, while different searches assigned a score to the same property. By filling these missing values with the mean score recorded for this property, we reduced the number of missing values from 1,090,348 to just 182,213. Although this seemed like a sensible solution, the model's generalization performance suffered. We initially decided to impute these missing values with -1, but upon testing, we discovered that validation performance improved when this value was changed to 0.

Finally, let's consider the data regarding competitor offerings. As seen from Figure 1, much of this data is missing. We experimented with different values to replace the missing data, including the mean, -1, and 0. The mean did not seem suitable for most columns due to the vast amount of missing information. After some trials, replacement with 0 yielded the best performance. For `compx_rate`, `compx_inv`, and `compx_rate_percent_diff`, we set the recorded value to 0 when values were missing. As outlined in section 4, we employ a tree-based ranking algorithm, which allows us to assign this symbolic value without implying a linear relationship, given that these boosting trees are nonlinear by nature.

3.2 Feature engineering

To extract additional insights from the dataset, we can introduce new features that may provide more interpretability or relevancy for our model. The models we consider do not natively support the `date_time`

feature as given. Therefore, we have generated five new features from the `date_time` feature: `hour`, `day`, `month`, `day_of_week`, and `is_weekend` (a boolean feature).

We then incorporated difference features. The dataset includes information about a user's historical star rating and historical prices paid. These can be compared to the current star rating and price of a property. This assumes a correlation between a user's previous and current behavior, i.e., users might favor properties within the same rating and price range as their past purchases. Importantly, we don't take the absolute difference here, as an increase or decrease in price might signify different things.

This comparison can also be applied to the `prop_log_historical_price` feature, which represents the historical price for staying at a property. If the price has changed from the previous price it sold for, this might indicate that a property has become more or less in demand.

We also added ranking features for six columns. This feature ranks the hotels in a search based on these columns. The columns we used were `price_usd`, `prop_starrating`, `prop_review_score`, `prop_location_score1`, `prop_location_score2`, and `usd_diff`.

For the property features, we also added monotonic features giving the absolute distance from the mean of properties booked and clicked.

$$\texttt{mono\_book} = abs(value - mean(values_{booked}))$$
$$\texttt{mono\_click} = abs(value - mean(values_{clicked}))$$

A significant number of the new features were normalization features. Here, we normalize the most important property features concerning different indicators, as done by the second-place solution from the original competition. This takes into account price differences between properties in different locations and seasons. The same six features as ranking are used here. The different indicators that are used are `srch_id`, `prop_country_id`, `srch_destination_id`, `srch_length_of_stay`, `srch_booking_window`, and `month`. For each indicator group, we normalize the respective feature values. This involves subtracting the mean of the feature values in a group from each individual feature value in that group and then dividing the result by the standard deviation of the group's feature values. We obtain

$$z_{ij} = \frac{x_{ij} - \mu_i}{\sigma_i}$$

where x_{ij} is the feature value for the j^{th} observation in the i^{th} group, μ_i is the mean of the i^{th} group, and σ_i is the standard deviation of the i^{th} group. The resulting z_{ij} is the standardised or normalised value for the j^{th} observation of the feature in the i^{th} group

We also adopt desirability features following the feature engineering process of Wang and Kalousis (2013). We use the observed probability of a hotel being booked or clicked on as a proxy for hotel desirability:

$$\texttt{desire\_booking} = \frac{booking(prop\_id)}{count(prop\_id)}$$
$$\texttt{desire\_click} = \frac{click(prop\_id)}{count(prop\_id)}$$

Initially, these features were constructed using the entire training set, yielding a series `(prop_id, desire_var_value)`, which was then merged with both the training and validation data. However, constructing these features using the training data worsened generalization results as the model quickly overfitted using these features. This makes sense as the target is clearly leaking into the training data this way. Therefore, we made a further split of the training data, one part to be used for training and the other solely used for constructing desirability features. Although it seemed wasteful to use a significant part of the data solely for the construction of 2 features, this procedure did increase generalization performance. After many runs of hyperparameter optimization, the optimal split used for these features was between 30% and 50%.

We also added a feature called `prop_id_count`. This feature was added using the rationale that for a property to be booked, it at least needs to be displayed. Therefore, the number of times a property is present in the dataset might be of predictive value.

Finally, we converted some of the features into categorical features. This is a special feature used with LightGBM that turns an integer-encoded feature into categories without the need for one-hot encoding. We did this for the features `hour`, `day`, `month`, `day_of_week`, `site_id`, `visitor_location_country_id`, `prop_country_id`, `prop_id`, and `srch_destination_id`. An overview of all features is provided in Table 3.

Table 3: Overview of all newly created features

Feature	Description
target	A combination of click_bool and booking_bool, where bookings are a five and clicks are a one
month	Month extracted from datetime
day	Day extracted from datetime
hour	Hour extracted from datetime
day_of_week	Day of week extracted from datetime
is_weekend	Boolean that indicates weekend extracted from datetime
norm_price_usd_srch_id	Hotel price grouped by search and normalised
norm_price_usd_prop_id	Hotel price grouped by property and normalised
norm_price_usd_prop_country_id	Hotel price grouped by country of property and normalised
norm_price_usd_srch_destination_id	Hotel price grouped by search destination and normalised
norm_price_usd_srch_length_of_stay	Hotel price grouped by search length of stay and normalised
norm_price_usd_srch_booking_window	Hotel price grouped by booking window and normalised
norm_price_usd_month	Hotel price grouped by month when the search was performed and normalised
norm_prop_starrating_srch_id .... norm_prop_location_score2_month	Same normalisation as with price for star rating, review score, location score 1 and location score 2
usd_diff	The difference between the historic price a customer has paid and the price of the property in the search
star_diff	The difference between the historic star rating the customer has booked and the star rating of the property in the search
log_price_diff	The difference between the historic price a property has sold for and the current price of the property
prop_id_count	The count of how often a property is shown in searches.
rank_usd_diff	A ranking of the difference in prices a customer historically paid and the property price for each search
rank_log_price_diff	A ranking of the difference in prices a property previously sold for and the current property price for each search
rank_price_usd	A ranking of the property prices for each search
rank_prop_starrating	A ranking of the property star ratings for each search
rank_prop_review_score	A ranking of the property review scores for each search
rank_prop_location_score1	A ranking of the properties first location score for each search
rank_prop_location_score2	A ranking of the properties second location score for each search
mono_book_price_usd	The absolute value of the price of a property subtracted by the average price of booked properties
mono_click_price_usd	The absolute value of the price of a property subtracted by the average price of clicked properties
mono_book_prop_starrating	The absolute value of the star rating of a property subtracted by the average star rating of booked properties
mono_click_prop_starrating	The absolute value of the star rating of a property subtracted by the average star rating of clicked properties
mono_book_prop_review_score	The absolute value of the review score of a property subtracted by the average review score of booked properties
mono_click_prop_review_score	The absolute value of the review score of a property subtracted by the average review score of clicked properties
mono_book_prop_location_score1	The absolute value of the first location score of a property subtracted by the average first location score of booked properties
mono_click_prop_location_score1	The absolute value of the first location score of a property subtracted by the average first location score of clicked properties
mono_book_prop_location_score2	The absolute value of the second location score of a property subtracted by the average second location score of booked properties
mono_click_prop_location_score2	The absolute value of the second location score of a property subtracted by the average second location score of clicked properties

4 Modelling setup

In this section, we set up suitable models to tackle the problem at hand. The performance of these models is evaluated using NDCG@5, where bookings and clicks on a hotel are assigned values of 5 and 1, respectively. Content-filtering systems are naturally suited to this problem as they make numerical predictions of the target based on the input features. The first step involves choosing an appropriate model. We construct three benchmarks using K-nearest neighbours (KNN), Random Forest (RF) regressor, and random ordering. We then explore the LambdaMART or LambdaRank model as implemented in LightGBM, which is a state-of-the-art model for learning-to-rank (LTR) tasks (Lyzhin et al., 2022). This model is based on a boosting trees algorithm, but the loss function is tailored to a specific Learning-to-Rank task. Accordingly, a pairwise, Bernoulli loss is considered as described by C. Burges et al. (2005). Given two samples i, j, where i should be ranked higher than j, the target Y_{ij} equals 1, while the converse Y_{ji} equals 0. These can be regarded as target probabilities. The goal of our model is to predict these probabilities, denoted by $\hat{Y_{ij}}$. The loss function then becomes the log-likelihood function on Bernoulli distributed variables:

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\left[Y_{ij}log\left(\hat{Y_{ij}}\right)+\left(1-Y_{ij}\right)log\left(1-\hat{Y_{ij}}\right)\right]$$

Minimization of this loss is carried out using a gradient boosting tree algorithm, where gradients are adjusted using the NDCG, as described by C. J. Burges (2010).

Model quality evaluation is performed on the aforementioned validation set, which comprises 15% of the training set data. Moreover, predictions of models that perform well in this validation step are evaluated on an unseen test set in the Kaggle competition. With respect to evaluation metrics, only NDCG@5 is considered. The competition hosts chose this measure to evaluate submissions, so our model aims to maximize NDCG@5. The LambdaMART model offers an added benefit of built-in feature importances, which enhances the model's explainability.

While we aim for a fair comparison, the LGBM ranker is much more efficient. It was not feasible to run the KNN and RF as implemented in scikit-learn (Pedregosa et al., 2011) on the full dataset. For the Random Forest model, runtime was roughly 4 minutes when training on 3% of the rows and taking a subset of the 25 most important features as indicated by the LGBM ranker's feature importances. Runtime for the KNN regressor was even longer. In contrast, the LGBM ranker could fit the entire training dataset with its 100 features in 4 minutes and 19 seconds. Consequently, we continue with 3% of the instances and the 25 columns for the RF and KNN regressors. Performance for all models is evaluated on the same validation set, consisting of 15% of the training instances.

We utilized Optuna for hyperparameter optimization. This library employs a TPE (Tree-structured Parzen Estimator) algorithm, as described in detail by Bergstra et al. (2011), allowing for efficient tuning of the hyperparameters in the selected models. The relevant domains and hyperparameters to be optimized are presented in Table 4 below.

Table 4: Optimization Parameter Ranges

Table 5: LGBM Parameter Ranges

Parameter	*Range*
n_estimators	250 - 900
num_leaves	10 - 100
max_depth	1 - 20
learning_rate	0.01 - 0.2
subsample	0.4 - 1
colsample_bytree	0.4 - 1
reg_alpha	0 - 0.2
reg_lambda	0 - 0.2
min_child_samples	5 - 100
min_child_weight	0.0001 - 0.1
val_size	0.3 - 0.8

Table 6: RF Parameter Ranges

Parameter	*Range*
n_estimators	100 - 1000
max_depth	10 - 30
min_samples_split	2 - 5
min_samples_leaf	1 - 5
max_features	auto, sqrt
bootstrap	True, False

Table 7: KNN Parameter Ranges

Parameter	*Range*
n_neighbors	1 - 30
weights	uniform, distance
p	1, 2
algorithm	auto, ball_tree, kd_tree, brute
leaf_size	1 - 50

Experiment tracking was handled using Weights and Biases (WB), a platform freely available for educational purposes. This tool was invaluable in logging validation performance during each training iteration, as well as recording final performance metrics on the validation set. It also logged performance on the training set, making it easy to spot models that were overfitting or underfitting.

In addition to performance metrics, WB monitored various system variables, such as CPU load and memory usage, and provided simple visualization tools for all these variables. A brief report of the experiments recorded using this tool can be accessed here.

5 Results

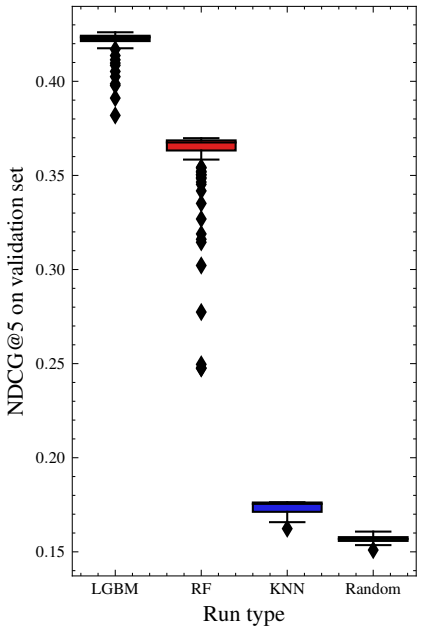
In Table 8, we present summary statistics for the NDCG@5 score from all runs during the hyperparameter optimization process. The "Max" column reflects the NDCG@5 score on the validation data using the best set of hyperparameters for each respective model. From this table, we observe that the KNN regressor appears to perform only slightly better than a random ordering. The Random Forest (RF) exhibits a significantly improved performance, although even its best run still underperforms the worst run of the LGBM Ranker. The LGBM Ranker demonstrates the highest performance, achieving an NDCG@5 score of 0.43 (rounded to two decimal places) on the validation set.

Table 8: Run Type Statistics

Run Type	*Max*	*Min*	*Mean*
LGBM Ranker	0.43	0.38	0.42
RF regressor	0.37	0.25	0.36
KNN regressor	0.18	0.16	0.17
Random	0.16	0.15	0.16

The NDCG@5 scores for all runs are also depicted in Figure 3. This visual representation echoes our earlier findings. It confirms that the KNN performs on par with the random ordering, while the RF and LGBM Ranker significantly outperform the other two. Among these, the LGBM Ranker outperforms the RF model.

Fig. 3: NDCG@5 of all runs by model



To further substantiate our findings, we carried out a paired T-test on the NDCG by `srch_id` to determine whether the LGBM Ranker's performance is statistically significantly better than the other models. The results are provided in Table 9. At any conventional confidence level, the LGBM Ranker surpasses all benchmarks.

Table 9: Paired T-test on NDCG@5 for LGBM ranker against benchmarks

Test against	*Test Statistic*	*p-value*
Random Forest Regressor	30.27	0.00
KNN Regressor	87.27	0.00
Random order	92.83	0.00

The optimal model specifications, which have been compared above, are reported in Table 10. At the time of writing (28th May), this winning model achieved a NDCG@5 of 0.40891 on the public leaderboard of the unseen test set. This resulted in the 9th place on the public leaderboard.

Table 10: Optimal Parameter Configurations

Table 11: LGBM Optimal Parameters

Parameter	*Optimal Value*
n_estimators	421
num_leaves	36
max_depth	9
learning_rate	0.075
subsample	0.441
colsample_bytree	0.445
reg_alpha	0.134
reg_lambda	0.135
min_child_samples	96
min_child_weight	0.065
val_size	0.315

Table 12: RF Optimal Parameters

Parameter	*Optimal Value*
n_estimators	853
max_depth	12
min_samples_split	2
min_samples_leaf	5
max_features	sqrt
bootstrap	True

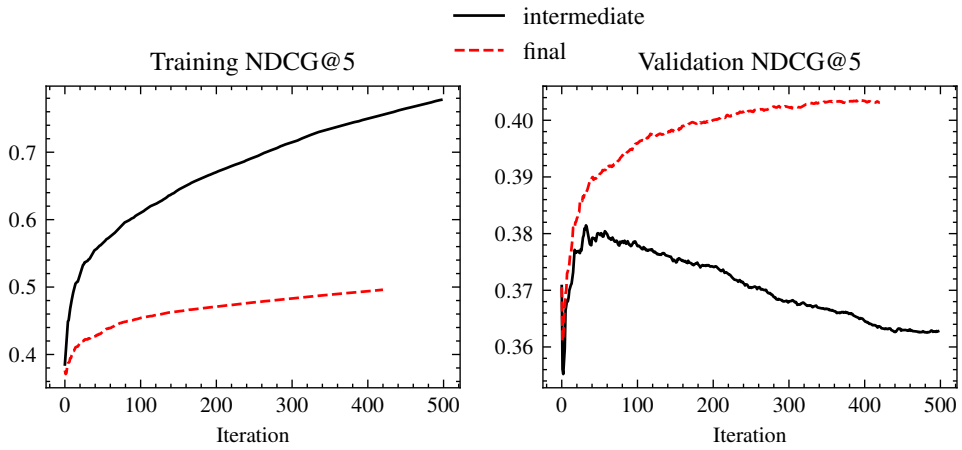
Table 13: KNN Optimal Parameters

Parameter	*Optimal Value*
n_neighbors	28
weights	distance
p	1
algorithm	ball_tree
leaf_size	43

LGBM ranker evaluation

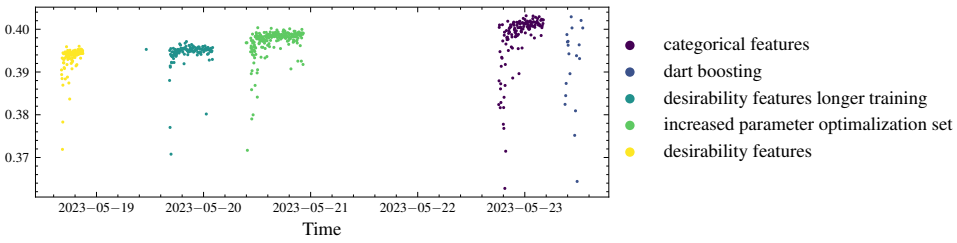
The training and validation curves for the winning run (`astral-tree-553`) and an intermediate run (`fragrant-blaze-487`) are plotted below using the metrics from WB. These metrics facilitate easy detection of overfitting or underfitting. For the intermediate run, the training NDCG continually increases to a much higher level compared to the final run. However, on the validation data, performance initially improves and then declines substantially for the intermediate run. This is a clear indication of overfitting; the model is learning the training set nearly perfectly, which negatively impacts its generalization performance. Conversely, for the winning model, both the training and validation NDCG increase consistently throughout their entire domain, and training halts once the increase in validation performance levels off. This is the desired outcome.

Fig. 4: Training and Validation NDCG@5 for final and intermediate run



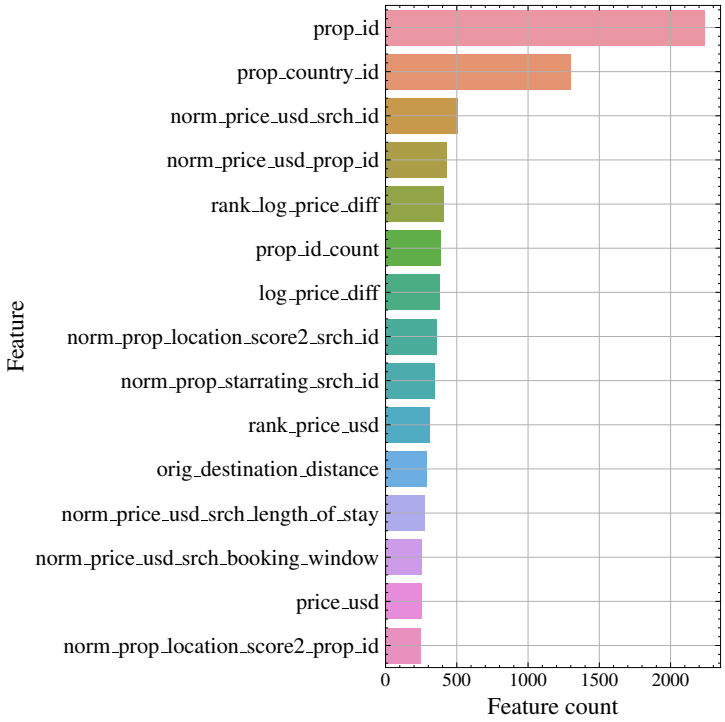
In Figure 5 below, we display all Optuna optimization runs for the LGBM ranker logged in WB. A notable increase can be seen when the parameter set to be optimized is expanded and features are encoded as categorical when applicable. The run where the boosting type was set to dart showed no performance improvement and required significantly more time.

Fig. 5: Validation performance (NDCG@5) over time for all runs



The LGBM ranker also offers the advantage of built-in feature importances. This is defined as the number of times a particular feature is used across all trees in the model. The following plot shows the 15 most important features as identified by the winning model.

Fig. 6: Validation performance (NDCG@5) over time for all runs



As can be seen from the figure, the most important features appear to be related to property identification. Following that, many price-related features are prominent in this top 15. Scores and ratings are of lesser importance. However, after the initial two features, feature importance decreases slowly and most features seem to be utilized in the final model. Furthermore, normalized and ranking features seem to enhance the predictive power of the model, typically ranking higher in terms of feature importance than their untransformed counterparts.

6 Deployment

The deployment of our model in a real-world scenario requires addressing several practical considerations, primarily due to the dynamic and large-scale nature of the data.

Hyperparameter tuning for our final model, although time-consuming (taking several hours), is manageable for a large company like Expedia, which would have superior computational resources to expedite this process. Considering the continuous change in data, regular updates to the model are necessary. However, given the time commitment associated with tuning and validating the model, these updates may be most efficiently implemented on a quarterly basis. Daily updates, while optimal, would primarily involve retraining the model with the most current data, a process that took roughly 4 minutes in our implementation.

Expedia may also face the challenge of integrating new properties into the recommendation system. As the model is trained on historical data, it will inherently favor properties with an established record. To ensure fair representation of new properties, a supplementary function could be incorporated to increase their visibility. This would also contribute valuable data for the future refinement of the model.

For scalable implementation, the model could be containerized and deployed as a microservice within Expedia's existing digital infrastructure. This approach would enable easy scalability to manage increased traffic by adding more instances of the service as required. The search engine of Expedia could then interact with this service, providing a list of hotels matching a user's search query, and in return, receive a ranked order of these hotels to present to the user.

Post-deployment, regular monitoring of the model's performance is crucial. Metrics such as the booking location within the list can provide valuable insights. If users are not predominantly booking the top-ranked suggestions, it may indicate that the model is not performing as expected, warranting further investigation and potential refinement.

7 Learnings

I, Reinier, have learned a lot during this assignment. I had little experience with feature engineering and machine learning models before this course. During the assignment, I learned what suitable methods are to create new features, such as looking at differences and normalising features by other features. Furthermore, I learned to work with model-tuning programs such as Optuna and Wandb. These programs were invaluable in creating our final model and getting insight into the process. I did not expect that tuning the hyperparameters of the model would have such a large impact as it did. The tuned model increased the score by almost 1 compared to a non-optimised version.
Caspar:
I learned about learning to rank problems, the current state-of-the-art models for LTR and how to implement those. For the first time I used Weights & Biases which greatly simplified the experiment tracking process. Because of this I no longer had to run experiments twice when outputs were lost, everything was saved in the cloud. Moreover, it was the first time I worked with a dataset containing multiple millions of rows. The challenges this posed were interesting and called for new solutions regarding data storage and loading.

References

Bergstra, James, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl (2011). "Algorithms for hyper-parameter optimization". In: *Advances in neural information processing systems* 24.

Burges, Chris, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender (2005). "Learning to rank using gradient descent". In: *Proceedings of the 22nd international conference on Machine learning*, pp. 89–96.

Burges, Christopher JC (2010). "From ranknet to lambdarank to lambdamart: An overview". In: *Learning* 11.23-581, p. 81.

Liu, Xudong, Bing Xu, Yuyu Zhang, Qiang Yan, Liang Pang, Qiang Li, Hanxiao Sun, and Bin Wang (2013). *Combination of Diverse Ranking Models for Personalized Expedia Hotel Searches*. arXiv: 1311.7679 [cs.LG].

Lyzhin, Ivan, Aleksei Ustimenko, Andrey Gulin, and Liudmila Prokhorenkova (2022). "Which Tricks are Important for Learning to Rank?" In: *arXiv preprint arXiv:2204.01500*.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

Wang, Jun and Alexandros Kalousis (Dec. 2013). *Personalized Expedia Hotel Searches – 2nd place*. Presentation at ICDM 2013 – Dallas.

Zhang, Owen and Adam Woznica (Dec. 2013). *Personalized Expedia Hotel Searches – 1st place*. Presentation at ICDM 2013 – Dallas. Compiled and presented by Adam Woznica, PhD.