Learning to Rank Hotels

Alec Hon
abh466@nyu.edu
New York University

Alex Dong
awd275@nyu.edu
New York University

Eric He
eh1885@stern.nyu.edu
New York University

Jesse Swanson
js11133@nyu.edu
New York University

Abstract

We benchmark learning-to-rank (LTR) and recommender systems (RecSys) models on an e-commerce dataset provided by Rocketmiles: a hotel booking platform. In contrast to other RecSys datasets, the Rocketmiles dataset contains all search results per user search as well as an array of contextual features pertaining to queries, users, and items. From the LTR space, we implement standard pointwise classification models using boosting trees, as well as the Rocketmiles default ranker which follows the LambdaMART learning methodology. For RecSys models, we implement the standard alternating least squares (ALS) method as well as two deep learning approaches: Mult-VAE and Hotel2Vec. We find that for this particular dataset, standard classification methods which leveraged contextual features were able to achieve the highest performance, while RecSys methods which only considered the hotel and user were unable to achieve good performance. Our code can be found at https://github.com/EricHe98/sad_final_project.

1 Introduction

Recommender systems and information retrievals traditionally use linear latent factor models such as matrix factorization. Deep learning methods have been highly successful for other high dimensional problems, such as natural language processing and computer vision yet, as Dacrema, Cremonesi, and Jannach [4] show, many recent deep learning methods fail to outperform well-optimized simple baselines. In this paper, we compare deep learning methods to Rocketmiles' LambdaMART [2] production model. We implement Mult-VAE [13], a variational autoencoder for collaborative filtering, and learned hotel embeddings based on Word2Vec [14]. These two methods are implemented for a proprietary dataset provided by Rocketmiles.

Rocketmiles is a company that sells hotel bookings online. Users search for hotels based on standard search criteria on the Rocketmiles website, receive a recommended list of hotels, and shop for hotel bookings. Rocketmiles stores the interaction data, contextual information, and the original search query. Currently, the recommendation model used in production is LambdaMART.

We first provide an overview of the dataset and an overview of the relevant evaluation metrics. We then provide an overview of the LambdaMART and Mult-VAE models as well as the hotel embedding generation method (Hotel2Vec). Lastly, we provide the results of our implementations.

2 Related Work

LambdaMART, the current production ranker at Rocketmiles, is a gradient boosting tree with a customized loss function [2]. It builds off a rich history of ranking models from the information retrieval space, such as RankSVM [9] and LambdaRank [3].

Learned vector representations of words are critical to achieving state-of-the-art performance on natural language processing (NLP) tasks. Ideas from learning vector representations for words have successfully been applied to the E-Commerce space. Embedding techniques have been applied to learn representations of items [7] or ads [8] that were clicked or purchased . Airbnb has successfully integrated and deployed learned embeddings into their production search ranking RecSys [6]. The listing embeddings successfully encode location, price, listing type, architecture, and listing style.

Latent Matrix Factorization methods are commonplace in recommender systems, but are generally linear methods. Deep Learning models are more expressive models that can handle nonlinear functions, and within common deep learning architectures, two popular methods to learn latent factors are autoencoders and variational autoencoders. Liang et al.(2018) [13] successfully implemented a variational autoencoder with multinomial loss to obtain better test ranking metrics on the MovieLens and Netflix Prize Dataset when compared to linear baseline models such as Matrix Factorization and SLIM.

3 Data Overview

The dataset contains a year of search request data from the Rocketmiles hotel booking dataset. The first ten months from January to October comprise the training set, while November is used for the validation set and December is used for the test set. Relevant stats for the dataset splits are posted in Table 1.

Each search result is tagged with a label corresponding to a funnel stage:

- Label = 0: the user did not interact with the search result
- Label = 1: the user clicked on the search result, taking him/her to a page holding details about the hotel
- Label = 2: the user selects a specific room for the hotel, taking him/her to a payment submission page.
- Label = 3: the user attempts to book the hotel.

For classification models, we opt to binarize the labels so that any search result with an interaction (click, payment click, book attempt) is labelled 1.

We comment on some difficulties with modeling the dataset.

- The train, validation, and test sets follow fairly different data distributions since they are sampled from different sections of the year. For example, December booking activity contains a greater share of business travelers since leisure travelers have generally booked at other times in the year. However, this methodology is closest to the conditions in which ranking models are AB tested at the company, while also preventing leakage by ensuring test data was generated after validation data, and validation data was generated after training data.
- Anonymous users comprise just under half of the search requests, causing difficulties for latent factor models relying on user latent factors.
- Over two-thirds of the items do not have interactions in the training dataset, making it difficult for item factor models to handle recommendations without a cold-start strategy.
- For privacy reasons, only one year of one product line data could be released. The dataset contains approximately half a million user interactions and is low for many modeling methods. Moreover, the interactions per user follow a power-law distribution, meaning a few users have outsized contributions towards interaction modeling.
- Although many contextual features are provided, feature selection is required to determine the highest value features.
- The dataset is large (9 GB) and requires downsampling for prototyping.

4 Evaluation

Our target performance metric is NDCG@10. We acknowledge NDCG can be difficult to interpret since an "expected" NDCG of a random shuffle is affected by the following factors:

1. Holding a label rate constant, more results in a query lowers the expected NDCG
2. A higher of quantity positive labels in a query raises the expected NDCG
3. Higher label values increases the expected NDCG as DCG scores scale exponentially with the value of the labels (e.g. a label of 3 has 2^3 more weight than a label of 0)
4. Increasing k for NDCG@k raises the expected NDCG.

To account for this, we computed a robust set of baseline NDCG performances which we list below:

1. PopularitySort: one of the features in the dataset is the regional share of bookings of a hotel at that current point in time. Ranking by this feature results in a baseline popularity ranker. For models which cannot predict on all users or items, we fill with the popularity score, making sure that the null predictions are ranked below the results which do have predictions. We use PopularitySort as a simple baseline model which we aim to outperform.
2. RandomSort: we generate a random ranking across the results by assigning randomly drawn scores to results. Though the NDCG of a random sort can be analytically computed given a dataset, the formula is not simple, so we take the average NDCG across 100 random shufflings. We use RandomSort as the lower bound of our performance.
3. DefaultSort: This is the actual order of items that appears on the Rocketmiles website. This order was generated by different production models at different times in the dataset.
4. CheatingSort: we train a LambdaMART model on the test set to see what performance this model can get through memorization. We use CheatingSort as an expectation of the upper bound of our performance.

5 Models

5.1 Baseline Models

The information retrieval space classifies ranking models into three variants based on their loss function: **pointwise** models consider a search result on its own, **pairwise** models compare two results in a search request to generate loss, and **listwise** models consider metrics computed across the entire list of search results in their loss function [12].

Based on this framework, a binary classification model which simply attempts to test if a search result is interacted with or not can be considered a pointwise ranking model. We train logistic regression using the sklearn library and boosting classification trees using the xgboost library.

For each model class, we train one model on all *standard* features and another model on the *core* features which were empirically shown to be important for the performance of the baseline NDCG ranker. There are 47 standard features, and 7 of them are classed as core features.

Table 1. Dataset characteristics

Feature	Total	Train	Val	Test
Distinct users	51010	47767	5287	3450
Distinct hotels	175811	168829	84665	76836
Number of search requests	343868	323910	12648	7310
Number of search results	43944992	41040549	1835914	1068529
Clicked search results	317879	307445	7824	2610
Booked search results	116896	103264	8386	5246
Number of search requests made by anonymous users	143855	140426	2136	1293
Number of items with interactions in the dataset	69937	55592	9138	5207

Boosting models are trained for 100 trees, but to see the model learning curve, for each model class we also train a 10-tree variant.

5.2 LambdaMART

LambdaMART is a gradient boosting tree using a listwise loss function. For a given search query q, list of n results returned by that query $\{r_i\}_{i=1}^{n}$, and corresponding labels $l(r_i)$, we write $f_k(r_i)$ as the current score produced by our gradient boosting tree with k trees. Then the loss λ of the result r_i by the current model f_k can be written as

$$\lambda(r_i) = \Big(\sum_{j \neq i} \mathbf{1}[l(r_i) > l(r_j)] - \sigma(f_k(r_i) - f_k(r_j)) \Big) |\Delta\text{NDCG}|_{(i,j)}$$

where σ is the sigmoid function, $\mathbf{1}[l(r_i) > l(r_j)]$ is the indicator function for whether r_i has a higher label than r_j, and $|\Delta\text{NDCG}|$ is the change in the NDCG of the ranking from swapping r_i with r_j in a ranking ordered by the model scores $\{f_k(r_i)\}_{i=1}^{n}$.

The $\sigma(f_k(r_i) - f_k(r_j))$ term is the difference between the predicted and true probabilities that r_i has a higher label than r_j, and can be interpreted as the gradient of the cross-entropy loss in classifying whether r_i has a higher label than r_j. With only this loss term and not the $|\Delta\text{NDCG}|$ term, the model would be a pairwise ranking model, since it uses all pairs of results to construct the loss.

The $|\Delta\text{NDCG}|_{(i,j)}$ term weights each (i, j) pair by the impact the pair has on the NDCG of the list. This is done to focus the model on the results at the top of the list, which have a bigger impact on both the user experience.

Using the kth tree to predict the gradient of the previous model f_{k-1} sequentially builds us a boosting model $f_1, f_2, ..., f_k$.

5.3 Mult-VAE

5.3.1 Overview.

Traditional matrix factorization models determine latent factors in a reduced shared dimension between items and users. However, these methods are inherently linear, and cannot capture more complex calculations in determining the ideal latent state. In deep learning, architectures that are used to determine latent factors are Autoencoders and Variational Autoencoders. These models are are two-stage architectures where the input is encoded into a smaller latent dimension, then decoded to determine whether the reconstruction of the encoding is close to the original input. However, unlike traditional matrix factorization methods, these deep learning methods are able to capture nonlinear elements, and thus can perhaps learn more about latent features given a interactions matrix.

We focus on Variational Autoencoders which uses techniques from Variational Inference during training to encode inputs into a latent space distribution rather than a latent space encoding. In the case of Liang et al., the input is a user u's click history x_u, which we encode into a K-dimensional latent distribution and then sample a latent representation z_u from a standard Gaussian prior. We then decode z_u via a a nonlinear function f_θ and softmax function to produce a probability distribution $\pi(z_u)$ over all $\mathbb{I}$ items from which the click history x_u is to have been drawn from, where x_u is assumed to have been sampled from a multinomial distribution, thus the proposed name of "Mult-VAE". For our nonlinear f_θ, we utilize a standard multilayer perceptron, with the architecture described in the following section.

5.3.2 Architecture and Loss Function.

We designed the our Mult-VAE to be very similar to Liang et al.'s Mult-VAE architecture. The input to the encoder was a user's click history $x_u \in \mathbb{R}^{\mathbb{I}}$. The model had one hidden layer of dimension 600 and latent layer of dimension 200. After sampling from the latent dimension, we decode the latent dimension with a hidden layer of size 600 then to return back to the item dimension. Dropout of 0.5 was applied at the input layer, and all nonlinearity activation functions used the Tanh function. The output of the encoder-decoder architecture was softmaxed and multiplied by the number of total positive interactions in the input, so that the input and the output could be reasonably compared.

For our loss function, we used a standard variational autoencoder loss with annealing, as per Liang et al.[13]. This consists of binary cross entropy loss on each entry in the output $\hat{x_u}$, summed with a KL-divergence from the decoded distribution onto the encoded distribution. We also include an annealing term β which scales the KL-divergence term. The annealing term linearly scales from 0 to 1 over a limited range of epochs. As such, our loss function is

$$\ell(x_u, \hat{x_u}) = BCE(x_u, \hat{x}_u)) + \beta\, KL(\text{Enc}(x_u)||\text{Dec}(z_u))$$

For our annealing term β, we tried multiple schedules, which are described in Table 5 on the following page.

In addition, we also incorporated an early stopping mechanism, which was not implemented in the original Mult-VAE paper. During the training process, we evaluated the validation loss over the validation set at the end of every epoch. If the validation loss failed to decrease for 5 epochs, model training would be stopped.

5.4 Hotel Embeddings

To generate hotel embeddings, we draw heavily on ideas from Word2Vec [14]. In our hotel LTR application, user sessions correspond to sentences in Word2Vec. A user session is defined as a series of uninterrupted interactions with the Rocketmiles website. Given a user session S_i with n interactions, let S_i be the set of interactions such that $S_i = \{c_1, c_2, ..., c_n\}$. Our aim is to learn an embedding for each listing $c_j = \mathbf{v_j}$ where $\mathbf{v_j} \in \mathbb{R}^{32}$.

To learn the hotel listing embedding, we iterate over each item in the user session and generate a context window of 5 items preceding the central item and 5 items after the central item. Each item in this surrounding context is stored as a positive pair with the central item. Context size is a potential hyperparameter but we follow [6] and use 5 as the context size since this generally captures item similarity [11].

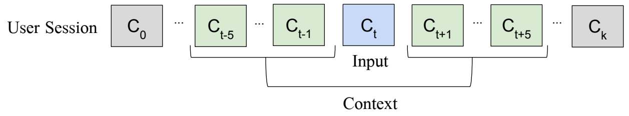

Figure 1. A user session S_i with a series of interactions user interactions. The input item C_t is paired with each of the items in the contextual window $[C_{t-5}, C_{t+5}]$ to generate positive pair samples.

To generate negative contrastive pairs, we uniformly sample from unclicked items in the set of session search results and uniformly sample from items outside of the set of session search results. Let n_{in} be the number of search results sampled from unclicked results returned to the user and n_{out} be the number of sampled hotels outside of the user session. To determine the effect of sampling negative pairs from different distributions, we generate hotel embeddings with $n_{in} = \{0, 5, 10\}$ and $n_{out} = \{0, 5\}$.

The generated positive and negative samples were used to train a lookup table of embeddings. Based on the successful application of item embeddings by Airbnb [6], we train a hotel embedding vector $\mathbf{v_i} \in \mathbb{R}^{32}$. Given a sampled pair of items $p_i = (p_{i,1}, p_{i,2})$ with label y and prediction $\hat{y}$, the loss function is

$$\ell(y, \hat{y}) = BCE(y, \hat{y})$$

where $\hat{y} = \sigma(p_{i,1} \cdot p_{i,2})$ and $y \in \{0, 1\}$ where $y = 0$ is a negative label and $y = 1$ is a positive label. We used an Adagrad optimizer as it is well-suited for sparse data [5] such as word or hotel embeddings in $\mathbb{R}^{32}$. The model was trained for a maximum of 20 epochs with patience= 3 and learning rate $\in \{0.1, 1, 10\}$ based on the guidelines to train good embeddings in [10]. We included these learned hotel embeddings as features to the original LambdaMART model to observe the impact of these learned features.

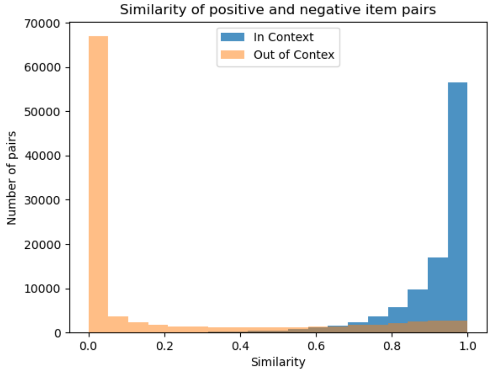

Figure 2. 100k positive (in context) and negative (out of context) pairs were sampled and compared using the similarity given by $\sigma(p_{i,1} \cdot p_{i,2})$. The in context and out of context pairs are highly differentiated due to the item embedding training procedure.

6 Results

All methods were implemented on NYU Prince and Greene High Performance Computing Clusters, utilizing the same train/val/test splits. Models were chosen based on the best validation NDCG, then evaluated on the test set. Table 2 is a summary of our results utilizing our baseline models and on the test set.

We also conducted ablation studies. One such study was training models to use different feature sets, which are shown in Table 3. Table 4 contains results with tree-based models using 10 trees instead of the default 100.

The Rocketmiles production ranker greatly downsampled the negative samples in the training set, with only around 8 million search results instead of 44 million. We were able to preserve the majority of the performance, with an NDCG of 0.354.

Table 2. Summary of Test NDCGs

Model	Test NDCG
CheatingSort	0.466
XGBoost Classifier	0.393
LambdaMART	0.369
LambdaMART w/ Hotel2Vec	0.357
Mult-VAE	0.315
DefaultSort	0.311
PopularitySort	0.269
ALS	0.233
Logistic Regression	0.137
RandomSort	0.0814

Table 3. The Delta column shows the the change in NDCG when training the model with a core subset of features.

Model	Test NDCG	Delta
LambdaMART	0.371	+0.002
Logistic classifier	0.361	+0.224
XGBoost classifier	0.393	+0.014

Table 4. Training on 10 trees instead of 100

Model	Test NDCG	Delta
LambdaMART	0.336	-0.033
XGBoost classifier	0.332	-0.047

6.1 Mult-VAE Experiments and Results

To determine the best possible configuration for the Mult-VAE architecture, we experimented with on a few parameters, each described in Table 5. Our Optimizer was Adam, with learning rates mentioned in the table as well. For early stopping, we stopped if the validation loss failed to decrease for five epochs in a row.

While we had implemented early stopping originally, we decided to not utilize it for one run, as we were not necessarily trying to optimize on the Mult-VAE loss Function, but rather trying to obtain the best validation NDCG. As models that stopped later tended to have higher NDCG, we had implemented a model to run much longer than any of our previous models, and obtained our best validation NDCG of .3431. Figure 3 shows the validation NDCG per Epoch for our final model.

For our best Mult-VAE model, we obtained a testing NDCG of 0.3155. The decreased NDCG is most likely due to the fact that Mult-VAE is not well setup for the cold-start problem, as it assumes a user history, so as a result, for new users, we implemented the popularity baseline to generate predictions. Anonymous users aside, 15% of the users in the testing set were first time documented users.

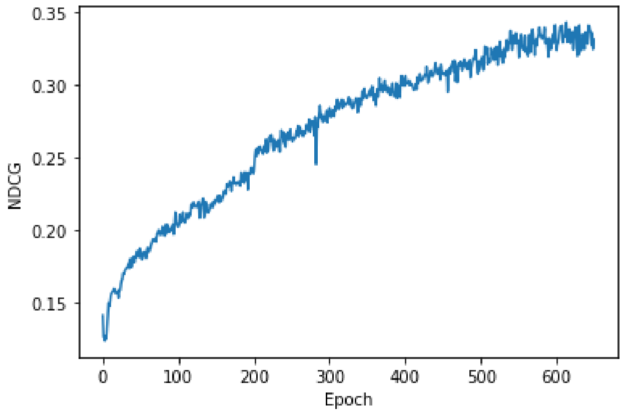

Figure 3. Validation NDCG over Epoch

6.2 Hotel Embeddings

To determine the effect of sampling negative pairs from different distributions, we generate hotel embeddings with $n_{in} = \{0, 5, 10\}$ and $n_{out} = \{0, 5\}$. The results for these experiments are summarized in 6.

7 Discussion

Gradient boosting classification models obtained the best performance on NDCG. This actually matches performance in Rocketmiles-side tests run across products a year ago. However, upstream processing requires the predicted probabilities to be pairwise calibrated, i.e. of the results $\{r\}_1$ which have a probability prediction of 30% and the results $\{r\}_2$ which have a probability prediction of 10%, then the results in $\{r\}_1$ should be booked three times as much as the results in $\{r\}_2$. In this case, LambdaMART which explicitly calibrates pairwise preference probabilities was superior to the classification models.

For this particular dataset, latent factor methods generally did not perform as well as feature-based methods. The user-item interaction matrix is extremely sparse, with most users not having more than one interaction and most items not having any bookings. This makes it very difficult for latent factor methods to learn good hidden representations. With many more interactions, it's possible that latent factor methods might eventually exceed performance of feature-based methods. However, for the hotel ranking problem it is natural to be in a "continuous cold-start" state [1]; even if a user has many bookings, it is difficult to guess whether a user is looking to book resorts for vacation or hotels near city centers for business conventions even when they come back.

Table 5. Mult-VAE Experiment Descriptions

Annealing	Early Stop	Learning Rate	Hidden Dim	Latent Dim	Num Hidden Layers	Epochs	Best Val NDCG
None	Yes	0.001	600	200	1	52	0.1808
0.02	Yes	1e-3	600	200	1	48	0.1888
0.02	Yes	1e-3	600	200	2	11	0.1477
0.02	Yes	1e-3	800	400	2	21	0.1323
0.02	Yes	1e-4	600	200	1	18	0.2135
0.02	Yes	1e-5	600	200	1	179	0.2067
0.005	Yes	1e-3	600	200	1	75	0.2188
0.005	Yes	1e-4	600	200	1	75	0.2171
0.005	No	1e-3	600	200	1	650	0.3436

Notes: If the number of hidden layers is more than one, we simply use the same amount of hidden nodes for the other hidden layers. Each model took about six minutes per epoch.

Table 6. Summary of validation NDCGs for different positive and negative sampling methods. n_{in} is the number of negative contrastive samples in the user session and n_out is the number of negative contrastive samples from outside the user session. Different negative sampling methods appeared to have little impact to the performance of the model, likely due to more informative contextual features.

n_{in}	n_{out}	NDCG
5	0	0.356
10	0	0.356
20	0	0.356
0	5	0.356
5	5	0.356

7.1 Importance of features

Though latent factor models underperformed compared to feature-based models, using too many features also underperformed. On an absolute basis, the data only covered 1 year of data, and only had around half a million interactions. As a result, even a simple logistic regression was able to achieve very good performance.

In previous experiments by Rocketmiles, they found that including query-standardized features such as the *srq_price_zscore*, defined by $z = \frac{\text{price} - \mu_q}{\sigma_q}$, where μ_q is the average price of results in the query and σ_q is the standard deviation of the price of results in the query, led to similar performance for pointwise classification models and listwise ranking models. The XBGoost classification tree achieved the top performance.

Though using some of the provided features were critical, many features were deemed counterproductive and noisy, as dropping them improved performance for all feature-based models. The embeddings features added by the Mult-VAE and Hotel2Vec models also failed to improve model performance. It is possible that since the supplied the dataset is only a small part of the full data, many contextual features were ultimately more informative to account for seasonal and regional trends.

7.2 Future Work

Much of the work done for this project was proof-of-concept, and can be redeployed on the full Rocketmiles dataset and infrastructure. From an engineering perspective, we were able to test out MLFlow, a common framework for training and evaluating machine learning models. From the data science side, we were able to build a robust set of baseline models, including latent factor models which we did not test before.

We plan to retest the models on the full Rocketmiles dataset which is over an order of magnitude larger, as well as possibly using the Hotel2Vec embeddings approach to power hotel recommendations given a clicked hotel (rather than ranking all hotels for a given search query).

Some work was also done to model inverse propensity scores for each ranking position, but because it used data not shared with the entire team and because we were unable to complete it, we left it out of this paper. The work, however, continues.

8 Conclusion

Our goal was to outperform Rocketmiles' the production model. We implemented several baseline methods as benchmarks for comparison to developed models. We (re)implemented two methods, Mult-VAE and generating hotel embeddings using Hotel2vec, and tested a few combinations of hyperparameters. We found that Mult-VAE performs surprisingly well, given that it is a collaborative filtering method, and does not use any context information, while using hotel embeddings ended up decreasing LambdaMART's performance. We also note an interesting phenomenon where the validation NDCG for Mult-VAE increases as the loss function plateaus.

9 Attribution

Eric prepared the dataset for the group, constructing the baseline models (XGBClassifier, logistic regression, ALS), running LambdaMART, and implementing an evaluation script used for every model. Alex and Alec contributed to the Mult-VAE portion of the analysis, which included preprocessing the data to work with PyTorch, reimplementing the model in PyTorch, and generating the predictions. Jesse trained and evaluated the Hotel2Vec embeddings with PyTorch, Spark, and LambdaMART.

References

[1] Lucas Bernardi et al. *The Continuous Cold Start Problem in e-Commerce Recommender Systems*. 2015. arXiv: 1508.01177 [cs.IR].

[2] Chris J.C. Burges. *From RankNet to LambdaRank to LambdaMART: An Overview*. Tech. rep. 2010.

[3] Quoc Viet Le Chris Burges Robert Ragno. *Learning to Rank with Nonsmooth Cost Functions*. 2006.

[4] Maurizio Ferrari Dacrema, Paolo Cremonesi, and Dietmar Jannach. "Are We Really Making Much Progress? A Worrying Analysis of Recent Neural Recommendation Approaches". In: *Proceedings of the 13th ACM Conference on Recommender Systems*. RecSys '19. Copenhagen, Denmark: ACM, 2019, pp. 101–109. ISBN: 978-1-4503-6243-6. DOI: 10.1145/3298689.3347058. URL: http://doi.acm.org/10.1145/3298689.3347058.

[5] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159. URL: http://jmlr.org/papers/v12/duchi11a.html.

[6] Mihajlo Grbovic. *Listing Embeddings in Search Ranking*. May 2018. URL: https://medium.com/airbnb-engineering/listing-embeddings-for-similar-listing-recommendations-and-real-time-personalization-in-search-601172f7603e.

[7] Mihajlo Grbovic et al. "E-commerce in your inbox: Product recommendations at scale". In: *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 2015, pp. 1809–1818.

[8] Mihajlo Grbovic et al. "Scalable semantic matching of queries to ads in sponsored search advertising". In: *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*. 2016, pp. 375–384.

[9] Thorsten Joachims. *Optimizing Search Engines using Clickthrough Data*. 2002.

[10] S. Lai et al. "How to Generate a Good Word Embedding". In: *IEEE Intelligent Systems* 31.6 (2016), pp. 5–14. DOI: 10.1109/MIS.2016.45.

[11] Omer Levy and Yoav Goldberg. "Dependency-Based Word Embeddings". In: *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Baltimore, Maryland: Association for Computational Linguistics, June 2014, pp. 302–308. DOI: 10.3115/v1/P14-2050. URL: https://www.aclweb.org/anthology/P14-2050.

[12] Hang Li. *A Short Introduction to Learning to Rank*. 2011.

[13] Dawen Liang et al. *Variational Autoencoders for Collaborative Filtering*. 2018. arXiv: 1802.05814 [stat.ML].

[14] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].