# Learnings from a Retail Recommendation System on Billions of Interactions at bol.com

Barrie Kersbergen Sebastian Schelter
*Ahold Delhaize Research & AIRLab, University of Amsterdam*
bkersbergen@bol.com s.schelter@uva.nl

***Abstract*—Recommender systems are ubiquitous in the modern internet, where they help users find items they might like. We discuss the design of a large-scale recommender system handling billions of interactions on a European e-commerce platform.**

**We present two studies on enhancing the predictive performance of this system with both algorithmic and systems-related approaches. First, we evaluate neural network-based approaches on proprietary data from our e-commerce platform, and confirm recent results outlining that the benefits of these methods with respect to predictive performance are limited, while they exhibit severe scalability bottlenecks. Next, we investigate the impact of a reduction of the response latency of our serving system, and conduct an A/B test on the live platform with more than 19 million user sessions, which confirms that the latency reduction of the recommender system correlates with a significant increase in business-relevant metrics. We discuss the implications of our findings with respect to real world recommendation systems and future research on scalable session-based recommendation.**

## I. Introduction

Today's internet users face an ever increasing amount of information.This situation has triggered the development of recommender systems: intelligent filters that learn about the users' preferences and suggest relevant information for them. With rapidly growing data sizes, the predictive performance, processing efficiency and scalability of machine learning-based recommendations systems and their underlying computations becomes a major concern.

In this paper, we describe the architecture of a real world recommender system ABO for bol.com, a large European e-commerce platform which handles billions of interactions on several dozen million items every day in Section II. The ABO ('Anderen bekeken ook', Dutch for 'others also viewed') recommendations are shown on the product detail page[1] to enable customers to discover other products that are relevant to them, such as different versions of the same product, similar products or products that are complementary to the displayed item. We describe the individual components of our system, which are backed by cloud infrastructure from the Google Cloud Platform such as BigTable and BigQuery. In addition, we detail our nearest-neighbor-based recommendation approach, we discuss how we conduct distributed offline model training, and how we efficiently serve the recommendations online with low latency.

A natural question when operating such a real world recommendation system is how to improve its predictive performance. In this work, we explore two directions for improvement, and present the results of two corresponding studies. First, we investigate the potential of *algorithmic improvements* in Section III. Neural networks have shown outstanding performance in computer vision [1] and natural language processing tasks [2], and we therefore evaluate recently proposed neural network-based approaches [3]–[6] for session-based recommendation on real data from our platform, based on an existing academic study [7]. We evaluate the predictive performance of these neural networks, as well as their deployability for production settings, in terms of training time, cost of hyperparameter search, prediction latency and scalability. Next, we study a *system-specific improvement*: we do not change the recommendation algorithm itself, but optimise our serving infrastructure to drastically reduce its response latency (Section IV). We describe how we control the insertion rate of bulk updates into the production database of our recommendation system, in order to adhere to a latency SLA (service-level agreement) of 50ms for recommendation responses. We run a large-scale online A/B test on 19 million user session to investigate the impact of this response latency reduction on the predictive performance of our recommender system. In summary, we provide the following contributions:

- We discuss the design of a large-scale recommender system handling billions of interactions on a European e-commerce platform (Section II).
- We present two studies on enhancing the predictive performance of this system: $(i)$ We evaluate recent neural network-based approaches on proprietary data from our e-commerce platform, and confirm recent results outlining that the benefits of these methods with respect to predictive performance are limited, while they exhibit severe scalability bottlenecks (Section III); $(ii)$ We optimise the response latency of our serving system, and conduct an A/B test on the live platform with more than 19 million user sessions, which confirms that the latency reduction correlates with a significant increase in metrics based on purchases and revenue (Section IV).
- We discuss the implications of our findings with respect to real world recommendation systems, as well as future research on session-based recommendation (Section VI)

[1] https://www.bol.com/nl/p/-/9200000104430048

## II. System Architecture

In this section we provide a high-level overview of the architecture of our recommendation system, as illustrated in Figure 1. We describe our recommendation approach (Section II-B), distributed model training (Section II-C), and how to serve low-latency recommendations in response to user requests (Section II-D).

### A. Overview

ABO is designed for distributed computation and implemented on top of Apache Spark [8]. All of our components are loosely coupled, so they can run independently from each other, while exchanging data via a distributed filesystem. Our architecture consists of a batch system that is responsible for executing data-intensive model training workloads offline on 18 billion user-item interactions which is 3.5TB in size. We periodically precompute the recommendations for every item in our catalog, typically once per day for all 70M products. The online serving system is responsible to serve recommendations online with low latency. The most important Big Query data source for the candidate algorithms is the click data containing the historical production interactions from customers, including clicks, purchases and shopping cart additions. This data source grows by about 30M records per day. We clean and filter this data, and join it with the catalog and offer data sources to determine active recommendable products. We describe the components of our system from Figure 1 in detail:

**Candidate generation**. We compute nine recommender models that each generate recommendations for products in the catalog: (1) *Related-Products* recommends products that where clicked after clicking items, (2) *Order-After-Click* recommends the products that where purchased after clicking an item, (3) *Purchased-Together* recommends products that are purchased together, (4) *Family* recommends products that only differ in one product attribute such as size or color from a recent item. (5) *Similar* recommends products that are similar in title and description using cosine similarity with tf-idf weighting. (6) *Trending* recommends products that are popular in the same category (7, 8, 9) *Creator*, *Brand* & *Publisher* recommend products from the same content creator, brand or publisher. Note that each model has it own MLlib pipeline and the parallel execution and machine resource allocation is automatically handled by Spark.

**Ensembling & business rules**. The ensembling component aggregates the outputs of all nine recommendation models to generate a final set of recommendations per item, and applies a set of manually defined business-specific filters to the recommendations. These filters remove potentially unwanted recommendations, such as combinations of adult and non-adult products.

**"Avalanche"**. This component handles the bulk update of the pre-computed recommendations into the Bigtable database of the prediction server, our online serving component. It is implemented on top of Apache Beam, and converts the

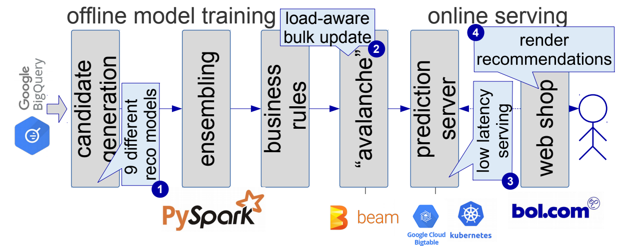


Fig. 1. Architecture overview of the offline training and online serving components of our retail recommendation system, based on Apache Spark and Apache Beam, and backed by infrastructure from the Google Cloud Platform.

recommendations to a Protobuf format with a fixed maximum amount of 21 recommended products to guarantee fast deserialization at serving time. The avalanche component will automatically adjust its insertion rate into BigTable to guarantee a low serving latency during the ingestion (see Section IV for further details).

**Prediction server**. The task of this component is to serve the recommendations to end users with low latency. We implement it in Java on top of the Spring framework running in a Kubernetes cluster. This component elastically scales its underlying cluster of machines using Kubernetes's Horizontal Pod Autoscaler, by automatically spawning or removing extra serving nodes based on the overall CPU load.

**Webshop**. This component communicates with the prediction server and renders the final web page served to customers, which also includes the product recommendations.

### B. Recommendation Approach

Next we describe the nearest neighbor-based algorithms underlying the first three approaches of our recommendation models, and subsequently discuss how to implement the model training in a scalable dataflow system.

**Item-based recommendation**. We leverage item-based collaborative filtering for recommendations [9], a simple but highly popular approach, deployed in many production settings [10]–[12]. This approach compares user interactions to find related items in the sense of "people who like this item also like these other items". The resulting pairs of cooccuring products are later on combined with the output from the other models and re-ranked. Our system bases its recommendations on so-called "implicit feedback data", e.g., count data that can easily be gathered by recording user actions such as clicks, shopping cart items or purchases.

**Collection and scoring of item cooccurrences**. We first clean the recorded interaction data in windows of 24 hours: we filter out data from users who visited an unusual amount of items during that time window. Next, we set the item cooccurrence count $c_{uijg}^{(t)}$ between item $i$ and item $j$ to one if we find a cooccurrence in a sliding window of 42 days for a user $u$ who interacted with these items in the given window $t$. Note that the count is 0 otherwise, and that we additionally record the type of interaction $g$ (e.g., click or purchase). Note that we

do not use personal identifiable information to determine the recommendations in order to respect the privacy of our users. Based on the collected cooccurrences, we compute the score $s_{ij}$ for all observed cooccurrences $c_{uijg}^{(t)}$ of an item pair $ij$ as follows. We sum up the observed cooccurrence counts across all users $u$, windows $t$ and interaction types $g$. We apply an interaction specific weight $w_g$ to each cooccurrence and decay the score with a function $\gamma(t)$ based on the amount of days passed since that interaction happened, to compute the final similarity score $s_{ij} = \sum_u \sum_t \sum_g \gamma(t)\ w_g\ c_{uijg}^{(t)}$. We filter out item pairs below a certain threshold to prevent recommending pairs with low user support.

**Ensembling**. We finally compute an ensemble to aggregate the item-to-item recommendations from our nine different models. Our models include collaborative filtering based approaches like the one described previously, but also content-based approaches that compute item similarity based on item metadata. The latter approach has the advantage to also provide recommendations for 'cold-start' items for which we have not seen interactions yet and which cannot be handled by out-of-the-box collaborative filtering algorithms therefore. The ensemble combines the algorithm-specific scores from all models with a weighted sum with manually tuned chosen weights. The final weights are based on $(i)$ the results of offline evaluation experiments, measuring the normalized discounted cumulative gain (NDCG) on held-out conversion, revenue and click data; $(ii)$ online A/B tests in the live system measuring several business metrics; and $(iii)$ qualitative offline evaluation by business experts.

### C. Distributed Model Training

Next, we describe how to compute our cooccurrence-based recommendations with Apache Spark [8]. The input for our model is clickstream data that spans several years of time, containing more than 18 billion user-item interactions at the time of writing. The dataset comprises 3.5 terabytes of data, and is stored in Google BigQuery (BQ). This data contains historical user actions from our webshop such as product views, additions to the shopping cart, and purchase events.

We process this data in parallel with Apache Spark. We model our computations based on the abstractions for feature transformations, models and evaluators of Spark MLLib [13], which improve the reusability and composibility of the computations. In accordance with our previously described recommendation approach, we partition the user-item interaction data by day (corresponding to the window $t$ from the previous section), and collect the item cooccurrences via a distributed self-join on the user id. The collection of the item cooccurrences for one day is independent of the collection of the item cooccurrences for other days which allows for massive parallelism in the computation. We store the respective cooccurrence counts per day in the distributed file system. The scoring of the cooccurrences is conducted by a second Spark job, which reads these cooccurrence counts, sums them up according to the scoring function and retains the top scored item pairs per item.

We execute the resulting computation in the Google cloud leveraging the Dataproc service[2]. The corresponding cluster is configured to autoscale up to 75 worker nodes of type `n1-highmem-8` each with a 500GB disk. We use dataproc image_version '1.4' which provides Apache Spark '2.4'. Executing the Spark jobs for the offline model training (all nine recommendations models, ensembling, and business rule filtering) takes approximately two hours.

### D. Online Serving

The concerns of generating and serving the recommendations are separated in our architecture. The prediction server component is responsible for serving the recommendations with low latency. We implement it as a Java Spring service with BigTable as database backend. Our service is designed to perform auto-scaling of its computational resources: it spawns or revokes additional machines based on the current CPU load, and is also responsible for managing the amount of BigTable machines. The service achieves this by monitoring the CPU load of the BT machines every minute, and adjusting the amount of machines correspondingly. In order to prevent stale data being served from BT, we apply a simple optimistic locking scheme: After data is written to BT, we update a counter in a BT table and set the timestamp to the start of the insertion time. Our service compares the timestamp of that counter value with the column value timestamp before serving the value for that key. If the counter timestamp value is greater than the timestamp of the column value the row is not returned. Our webshop performs approximately 1,500 requests per second to the serving component of which approximately 400 requests per second are going to the recommender system.

A natural question when operating such a real world recommendation system is how to improve its predictive performance. In the following sections, we explore two directions for improvement.

## III. Neural Networks versus Nearest Neighbor Methods for Session-Based Recommendation

In recent years, neural networks have drastically outperformed traditional ML models in various domains such as computer vision [1] and natural language processing [2]. It is therefore a natural question to explore the benefits of neural-based approaches in comparison to nearest neighbor techniques (such as our system) for our e-commerce scenario as well. In contrast to academic studies, we have to look at additional dimensions such as scalability and training cost as well, in order to judge whether it would make sense to deploy a neural-based approach.

The academic setup that is closest to our production use case is *session-based recommendation*, where the goal is to predict the next item (or the set of next items) that a user will interact with, given the current items of her session on ecommerce datasets. Interestingly, recent academic research [7], [14] indicates that neural-based approach do not outperform classical

[2]https://cloud.google.com/dataproc

nearest neighbor approaches in this scenario. We replicate a prominent study [7] on a sample of our production data to evaluate whether it may be beneficial to invest in neural approaches for our use case , and to investigate the scalability and training performance of current neural-based approaches for session-based recommendation on a large-scale ecommerce data. We include our approach as well, even though it has not been specifically designed and optimised for this particular academic task, but can be considered a special case of session-based recommendation as it only considers the most recent item in the session. Our main goal of this study is to confirm that the family of nearest neighbor-based algorithms provides state-of-the-art performance on e-commerce recommendation tasks.

### A. Data & Algorithms

**Dataset**. We use real world clickstream data from our platform for our study. We create five samples, each spanning 31 days from different times of the year, in order to minimise the impact of seasonal effects. We bin users by the amount of purchases that they made, and apply stratified sampling based on these bins to select a set of sessions that represents the activities of a wide range of our customers. We include around 1.2 million actions on 120 thousand items in each sample.

**Algorithms**. We include eight different recommendation algorithms in our evaluation, in accordance with [7]. Four of these employ neural network-based learning approaches for session-based recommendation: GRU4Rec [3], an RNN-based approach in combination with ranking loss functions [15] tailored to the session-based recommendation setting; NARM [5], an attention mechanism-based approach that aims to learn a user's sequential behavior in the current session; STAMP [6], an attention mechanism-based approach that aims to learn a user's sequential behavior in the current session by also learning the priority of the last item; and NEXTITNET [4], an approach employing convolutions to learn high-level representations of both short-and long-term item dependencies.

We additionally include the following four nearest-neighbor methods from [7] in our study: AR *(Association rules)*, an approach based on counting pairwise item cooccurrences in the observed sessions; SR *(Sequential rules)*, a frequent pattern mining approach based on sequential cooccurrences in the observed sessions, as well as S-KNN & VS-KNN *(Session k-Nearest Neighbor)*, two nearest-neighbor approaches based on session similarity, where the latter puts more emphasis on recent events in sessions.

**Experimental Setup**. We evaluate the predictive performance of all methods on our five data samples, based on the experimentation framework provided by Ludewig et al.[3] We use the first thirty days of each dataset for training and evaluate the predictions for the subsequent 31st day. Testing on a consecutive day also resembles our production setting where we re-train our model every day. For each session in the test data we replay one interaction after another. After

[3]https://rn5l.github.io/session-rec/index.html

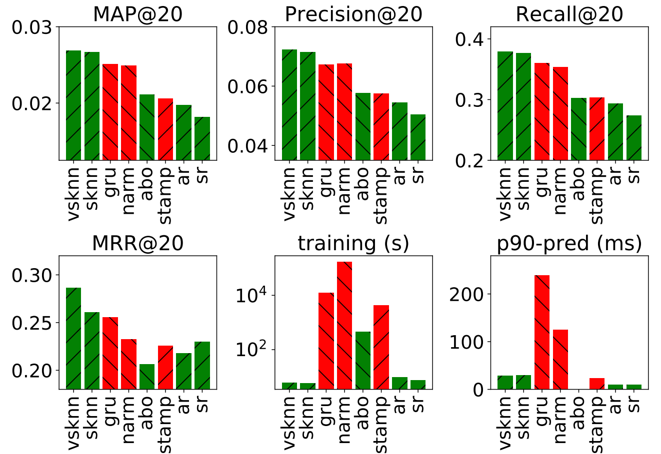


Fig. 2. Prediction quality, training time and the 90th percentile of the prediction time for session-based recommendation with the neural-based (red) and nearest-neighbor-based (green) recommendation approaches (including our approach ABO which was not designed for this task), averaged over five different data samples from our e-commerce platform.

each revealed interaction, we compute recommended items and compare them to the remaining interactions. We execute the training for each model in the Google cloud on a `n1-highmem-8` instance with a `nvidia-tesla-t4` GPU.

*Metrics*. We report four metrics computed from the top 20 recommended items: Mean Average Precision (MAP@20), Precision (P@20) and Recall (R@20) evaluate to what extent an algorithm is able to predict the next items in a session, while Mean Reciprocal Rank (MRR@20) evaluates to what extent an algorithm is able to predict the immediate next item in a session. We also do not report prediction times for ABO, which are hard to compare to the times of the other approaches, as ABO executes key-value lookups in BigTable in GCP, which includes network communication.

*Hyperparameter optimization*. Hyperparameter search for neural-based recommendation methods can be very time consuming, even on small data [7]. We therefore apply the following approach to tune the hyperparameters of all methods. We explore 100 combinations of hyperparameters with a random search on a data subset comprised of 100,000 interactions, and select the hyperparameters that result in the best MRR@20. Note that we had to schedule the hyperparameter search in parallel on different machines in the cloud to retrieve results in a reasonable time, even when allowing the methods to use GPUs. The hyperparameter search for NARM for example took more than six days of compute time, (in contrast to less than two hours for the VS-KNN approach)!

**Results**. Figure 2 shows the results of the predictive performance measured by our metrics, and the train/test time for neural-based and nearest-neighbor based approaches averaged over our five different data samples. We do not provide results for the NEXTITNET approach because it repeatedly crashed during training, which we attribute to a dependency issue with the provided implementation.

We find that the nearest neighbor approach VS-KNN consistently outperforms all neural-based approaches, and even provides a higher evaluation score for MRR@20, the metric for which the neural-based approaches have been designed. S-KNN outperforms all three neural networks in four out of five metrics as well. ABO outperforms the neural-based method STAMP in MAP and Precision, as well as the baseline methods AR and SR (even though it has not been designed and optimised for the academic task of session-based recommendation). The time required to train the neural approaches is at least an order of magnitude larger than the training time of the neighbor based approaches. Additionally, two of the three neural based methods require much more time for inference than the nearest neighbor methods. The 90th percentile of the time needed for a prediction with GRU4REC and NARM on a small dataset is already higher than 100 milliseconds.

**Discussion**. The experimental results on our proprietary data confirm the findings from the original study [7]: simple nearest neighbor methods outperform recent neural network based approaches for session-based recommendation on e-commerce data. In addition, we make several observations with respect to the deployability of the neural methods in a real world production scenario: $(i)$ These methods exhibit extremely long training times for hyperparameter search and model training even on very small datasets (100k observations for hyperparameter search and 1M observations for training). It is unclear whether they would even scale to our current production workload of several billion interactions at a reasonable training cost and time; $(ii)$ In addition, we saw the time to produce a recommendation with GRU4REC and NARM on our small evaluation dataset is already in a range that is far from usable in a real world serving system, which has to guarantee response times in the low millisecond range.

## IV. Impact of Serving Latency on Recommendation Performance

Most research focuses on improving the predictive performance of recommender systems via algorithmic changes. Motivated by our production setup, we are interested in the impact of orthogonal, systems-related improvements. These are in general hard to study for academic researchers without access to real world systems. Work from [16], [17] indicates that a reduction of the response latency has a positive impact on the acceptance rate of a search engine. Therefore we decide to investigate the impact of response latency on our recommender system as well.

**Data center migration**. As part of a bigger reorganization of infrastructure we migrated our serving component and its database from our proprietary data center (DC) to the Google Cloud Platform (GCP). This migration included several changes such as upgrading the version of Java and the libraries we use in our serving component, as well as rewriting code for retrieving records from the database and serving them. We use Bigtable (BT) in GCP as alternative to Apache Cassandra from the DC setup. In the DC, we ran the serving component in VMWare, while in GCP we run it via Docker images. Note that the serving component has to adhere to a strict service level agreement in each setup: If the webshop does not receive a response within 150ms it will discard the request and render the web page without the recommendation.

**Optimization for serving latency during bulk updates**. We notice that this SLA is likely to be violated during the daily bulk update of our pre-computed recommendations in the prediction server (Section II). In order to adhere to the latency SLA, the recommendations must be inserted without raising the CPU load on BT too much. This is challenging because the CPU load varies during the day and even while writing the data. We address this issue by sharing the responsibility of maintaining a low BT CPU load with the Avalanche job, which inserts updated recommendations into BT. We add a rate limite mechanism to control the insert rate of Avalanche as shown in Algorithm 1, aiming for an average CPU load of 35% in the BT nodes. We obtain the CPU load of the BT nodes every minute, and update the insertion rate of the Apache Beam job in Avalanche which conducts the bulk update.Note that the number 10,000 refers to the minimum amount of requests per second that a BigTable machine is guaranteed to answer.

**Algorithm 1** Control of the insertion rate of Avalanche.

**function** CONTROL_INSERTION_RATE($r, c, b, d$)
  *Input:* Insertion rate $r$ (insertions per sec) conducted by Avalanche, average CPU load in BigTable $c$ (percent), number of BigTable nodes $b$, number of Dataflow nodes $d$ in Avalanche.
  *Output:* Updated insertion rate limit for Avalanche.

  $r_{\max} \leftarrow (b \cdot 10000) \,/\, d$
  $\delta_{\text{cpu}} \leftarrow 40 - c$
  **if** $c \geq 40$ **return** $\max(1, r + 20 \cdot \delta_{\text{cpu}})$
  **elseif** $c \geq 35$ **return** $r$
  **elseif** $c \geq 30$ **return** $\min(r_{\max}, r + \delta_{\text{cpu}})$
  **else return** $\min(r_{\max}, r + 3 \cdot \delta_{\text{cpu}})$

### A. Experimental Evaluation

As part of our infrastructure migration we conduct a large scale online A/B test on our e-commerce platform to measure the impact of the reduction in serving latency on the acceptance of our recommendations.

**Experimental Setup**. Our website shows recommendations on every product page and makes them available as soon as the page loads. We run a large-scale online A/B test for a period of two weeks, where we divide visitors into two groups: The first group is served from the DC architecture, while the second group is served from the GCP architecture. We ensure that each visitor remains in the same A/B test group for the duration of the experiment via a persistent cookie that stores the group assignment. In total, we include more than 19 million user sessions in this experiment. The recommendations served from the DC and GCP architectures originate from the same algorithm and data, and are therefore identical. We measure the distribution of the latency between our prediction servers and webshops (deployed in DC and GCP), as well as two important business metrics based on orders and revenues. The circuit-breaker timeout in webshop for the recommender

service had been set to 150ms for this experiment, and our infrastructure is constantly monitored for requests discards by site reliability teams.

**Results & Discussion**. During the period of the experiment, we observe that the 90th percentile of the response latency distribution between the prediction service and our webshop[4] running in the DC is 32ms, while the GCP setup only requires 15ms. Furthermore, we observe a 2.19% increase in an important order-based metric, as well as a 2.31% increase in a corresponding revenue-based metric. We confirm that the resulting positive impact in metrics is statistically significant with a chi-squared test of independence. In summary, we find that important order and revenue based metrics correlate positively with a reduction in response latency by 17ms. Our results indicate that users are more likely to perform a purchase if recommendations are served with lower latency, given two content-wise identical recommender systems.

## V. Related Work

Recommender systems are an active research area [18], which is regularly fueled by industry challenges such as the "Netflix Prize" competition [19]. Unfortunately, it is often difficult to translate academic progress into deployable solutions which need to be able to handle billions of interactions. The winning solution of the Netflix challenge for example never went into production [20]. A classical approach to recommendation mining are nearest-neighbor methods [21]–[24], which are widely deployed in industry [10]–[12]. In the area of session-based recommendation, which is especially important for e-commerce scenarios, a lot of neural network-based approaches have recently been proposed [3]–[6], and there is an ongoing discussion of their relation to conceptually simpler nearest-neighbor methods, which outperform them on many datasets [7], [14].

## VI. Learnings & Future Work

In this, work we focused on how to improve the predictive performance of a real world recommendation system with both algorithmic and systems-related approaches. We confirm the finding from [7] that the simple nearest-neighbor-based VS-KNN approach outperforms modern neural network-based methods in session-based recommendation on e-commerce data. We additionally found that VS-KNN is orders of magnitude faster to train than the neural-based methods. This is an important property for a production systems that have to conduct regular retraining of their models, and adhere to strict time constraints for that.

Our second study indicates that reducing the response latency for serving has a significant impact on the acceptance of recommendations in e-commerce. We A/B tested this impact on over 19 million sessions where we were able to give visitors a better experience by improving the serving latency, which resulted in a significant increase in business-relevant metrics.

[4]Note that we measure the response latency at the webshop, our frontend server; The total latency for the customers includes additional network communication and the rendering of the web page on their devices.

We ran this study during an ongoing data center migration, which gave us the unique opportunity to investigate the effect of improving the latency, instead of making it artificially worse as done in previous studies for search engines [16], [17].

In the future, we will explore how to scale-up well-scoring algorithms for session-based recommendation (in particular VS-KNN) to a full production workload with several billion interactions. We think that our studies also outlined interesting research directions for improving the suitability of the neural-based recommendation algorithms for production settings.

*This research was supported by and carried out at bol.com. All content represents the opinion of the author(s), which is not necessarily shared or endorsed by their respective employers and/or sponsors.*

## References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *NeurIPS*, 2012.
[2] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *ICLR*, 2014.
[3] B. Hidasi, A. Karatzoglou, L. Baltrunas, et al., "Session-based recommendations with recurrent neural networks," *arXiv:1511.06939*, 2015.
[4] F. Yuan, A. Karatzoglou, I. Arapakis, J. M. Jose, and X. He, "A simple convolutional generative network for next item recommendation," *WSDM*, pp. 582–590, 2019.
[5] J. Li, P. Ren, Z. Chen, Z. Ren, T. Lian, and J. Ma, "Neural attentive session-based recommendation," *CIKM*, pp. 1419–1428, 2017.
[6] Q. Liu, Y. Zeng, R. Mokhosi, and H. Zhang, "Stamp: short-term attention/memory priority model for session-based recommendation," *KDD*, pp. 1831–1839, 2018.
[7] M. Ludewig, N. Mauro, S. Latifi, and D. Jannach, "Performance comparison of neural and non-neural approaches to session-based recommendation," *RecSys*, pp. 462–466, 2019.
[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, et al, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," *NSDI*, pp. 15–28, 2012.
[9] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," *WWW*, pp. 285–295, 2001.
[10] A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: scalable online collaborative filtering," *WWW*, 2007.
[11] J. Davidson, B. Liebald, J. Liu, P. Nandy, T. Van Vleet, U. Gargi, S. Gupta, Y. He, M. Lambert, B. Livingston, and D. Sampath, "The youtube video recommendation system," *RecSys*, pp. 293–296, 2010.
[12] Y. Huang, B. Cui, W. Zhang, J. Jiang, and Y. Xu, "Tencentrec: Real-time stream recommendation in practice," *SIGMOD*, pp. 227–238, 2015.
[13] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *JMLR*, vol. 17, no. 1, pp. 1235–1241, 2016.
[14] D. Jannach and M. Ludewig, "When recurrent neural networks meet the neighborhood for session-based recommendation," *RecSys*, 2017.
[15] B. Hidasi and A. Karatzoglou, "Recurrent neural networks with top-k gains for session-based recommendations," *CIKM*, pp. 843–852, 2018.
[16] R. Kohavi, A. Deng, B. Frasca, T. Walker, Y. Xu, and N. Pohlmann, "Online controlled experiments at large scale," *KDD*, 2013.
[17] I. Arapakis, X. Bai, and B. B. Cambazoglu, "Impact of response latency on user behavior in web search," *SIGIR*, pp. 103–112, 2014.
[18] F. Ricci, L. Rokach, and B. Shapira, "Introduction to recommender systems," in *Recommender systems handbook*, 2011, pp. 1–35.
[19] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, 2009.
[20] X. Amatriain, "Building industrial-scale real-world recommender systems," *RecSys*, pp. 7–8, 2012.
[21] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," *WWW*, pp. 285–295, 2001.
[22] J. J. Levandoski, M. Sarwat, M. F. Mokbel, and M. D. Ekstrand, "Recstore: an extensible and adaptive framework for online recommender queries inside the database engine," *EDBT*, pp. 86–96, 2012.
[23] B. Chandramouli, J. J. Levandoski, A. Eldawy, and M. F. Mokbel, "Streamrec: a real-time recommender system," *SIGMOD*, 2011.
[24] S. Schelter, C. Boden, and V. Markl, "Scalable similarity-based neighborhood methods with mapreduce," *RecSys*, pp. 163–170, 2012.