Practical Lessons from Developing a Large-Scale Recommender System at Zalando

Antonino Freno
Zalando SE
Berlin, Germany
antonino.freno@zalando.de

ABSTRACT

Developing a *real-world* recommender system, i.e. for use in large-scale online retail, poses a number of different challenges. Interestingly, only a small part of these challenges are of algorithmic nature, such as how to select the most accurate model for a given use case. Instead, most technical problems usually arise from *operational constraints*, such as: adaptation to novel use cases; cost and complexity of system maintenance; capability of reusing pre-existing signal and integrating heterogeneous data sources.

In this paper, we describe the system we developed in order to address those constraints at Zalando, which is one of the most popular online fashion retailers in Europe. In particular, we explain how moving from a collaborative filtering approach to a learning-to-rank model helped us to effectively tackle the challenges mentioned above, while improving at the same time the quality of our recommendations. A fairly detailed description of our software architecture is provided, along with an overview of the algorithmic approach. On the other hand, we present some of the offline and online experiments that we ran in order to validate our models.

CCS CONCEPTS

• **Information systems** → **Recommender systems**; *Learning to rank*; • **Computing methodologies** → Batch learning; Regularization;

KEYWORDS

learning to rank; large-scale learning; recommender system architecture

1 INTRODUCTION

Zalando (http://www.zalando.com/) provides a unique hub for digital fashion content in Europe. In the online shop, machine-learned product recommendations are disseminated over many different contexts, ranging from general navigation aid on the homepage to personalized sorting in the catalog and item-to-item recommendations on product detail pages. In order to keep up with an ever-changing market and the constantly growing customer needs, algorithmic progress has always been a crucial requirement for our recommendation services. Yet, when viewed from the trenches of online retail industry, recommender systems development assumes a significantly different shape than we are used to think from an academic perspective.

From a scientific point of view, research on machine-learned recommender systems tries to identify *optimal* recommendation algorithms. Here, optimality is typically defined in terms of *maximum achievable accuracy* (possibly weighted by computational efficiency) with respect to specific recommendation goals, settings, and constraints. Accuracy is measured by a number of different metrics, which formally measure the quality of recommendations in terms of the ranking they induce on a set of candidate items. However, real-world recommender systems, such as those powering all major e-commerce platforms, have to face a sensibly different question, i.e. how to *maximize business value* in the long term. In this context, the notion of business value has at least two different sides. On the one hand, we want to generate value for our customers, e.g. in terms of user engagement, retention, conversion rate, revenue, and related performance indicators. On the other hand, we want to generate internal value, in terms of the long-term profitability of our technology assets. Here, profitability means *operational excellence*, on at least three different dimensions: (*i*) possibility of adapting the recommender system to novel use cases; (*ii*) cost and complexity of the involved maintenance; (*iii*) capability of capitalizing on pre-existing signal and effectively integrating newly available data sources. In this sense, an optimal recommender system is one that achieves operational excellence along at least these three dimensions, while also generating value for customers.

Back in the past, most of the recommendation contexts on Zalando were powered by machine learning models based on collaborative filtering (CF). As we tried to improve on these models and replace them by more sophisticated approaches, we carefully redesigned our system by keeping in mind the three guidelines of operational excellence mentioned above. To this purpose, we moved to a recommendation framework based on learning to rank (L2R). In this paper, we provide an in-depth overview of our L2R architecture. In particular, we discuss how the chosen framework enabled our recommendation stack to address operational concerns. Our goal is to stress the importance of treating the operational quality of recommender systems as a major criterion for the scientific analysis of those systems. We believe that by achieving a tighter integration of operational and algorithmic concerns, faster and more

RecSys '17, August 27-31, 2017, Como, Italy

ACM ISBN 978-1-4503-4652-8/17/08…$15.00
https://doi.org/10.1145/3109859.3109897

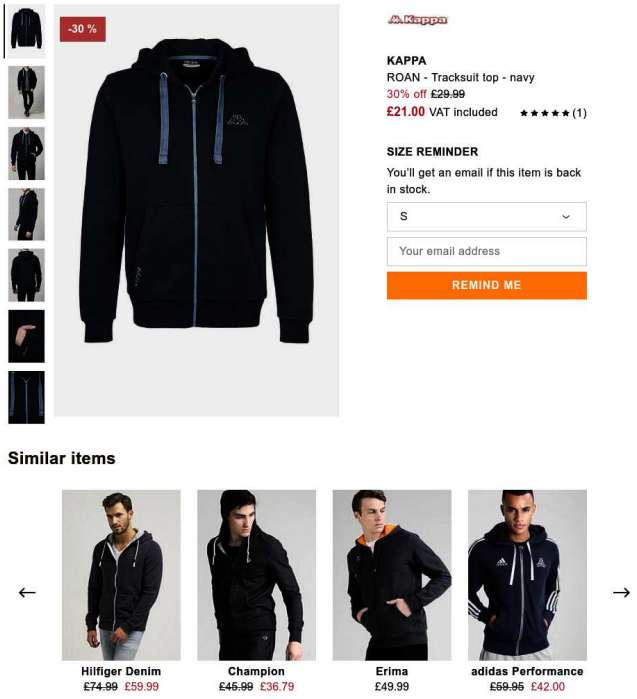

Figure 1: Item-based recommendations on a product detail page.

effective technology transfer can be achieved for the outcome of recommender systems research. In Sec. 2, we describe the main contexts where we serve recommendations in the fashion store. Sec. 3 reviews the theoretical framework underlying our machine learning system, whereas the system architecture is illustrated in Sec. 4. Sec. 5 presents some results from offline and online experiments. Finally, in Sec. 6 we draw some conclusions and sketch directions for future research.

2 USE CASES AND APPROACH

In Secs. 2.1–2.3 we review three different recommendation contexts from the Zalando web store. At the same time, we sketch the backbone of our modeling approach.

2.1 Item-based Recommendations

The first type of recommendations we serve at Zalando are the ones available on product detail pages, namely the ones usually presented in a "Similar items" carousel. These recommendations are not necessarily personalized, in that they simply model the relevance of additional candidate products with respect to the item that the user is currently checking out (i.e. the main subject of the currently open page). An example is provided in Fig. 1.

The way we model the relevance of candidate products for a given reference item is via the scoring function s:

$$s(\boldsymbol{i}_i, \boldsymbol{i}_j) = \varphi(\psi^I(\boldsymbol{i}_i, \boldsymbol{i}_j)) \tag{1}$$

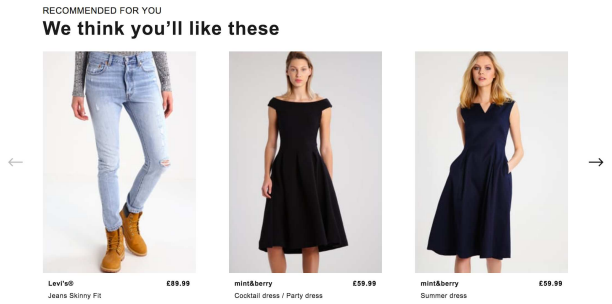

Figure 2: Personalized recommendations on the Zalando home.

where $\boldsymbol{i}_i$ is the reference item, and $\boldsymbol{i}_j$ is a candidate product for which we want to compute the relevance score. Once we score all relevant candidates $\boldsymbol{i}_j$, we rank them by score, and then recommend the top k.

The key ingredients to define are on the one hand the scoring function φ, and on the other hand ψ^I, which combines the base article and the candidate recommendation into the feature vector to be fed into ϕ. We will address the definition of ϕ in Sec. 3.2. While ϕ will be estimated by a general L2R routine, ψ^I encodes instead some of our domain knowledge by specifying how the attributes of $\boldsymbol{i}_i$ and $\boldsymbol{i}_j$ interact to one another. For example, if articles have a color attribute, their interaction $\psi^I(\boldsymbol{i}_i, \boldsymbol{i}_j)$ might contain an attribute specifying whether the two articles have the same color or not. Another interaction attribute might be given by the price difference of the two products, and so on. Therefore, we refer to ψ^I as an *interaction function*. Virtually, there is no limit to the type and quantity of interaction attributes we can encode into our model, as long as the information required to calculate them is available from the attributes of the input items. And of course, whenever relevant to the task at hand, non-relational attributes from the interacting items (e.g. the popularity of candidate articles) can also be incorporated into ψ^I.

2.2 Personalized Recommendations

General personalized recommendations are served to customers whenever no specific context or intention is assumed for the current context. For example, we serve personalized recommendations on the Zalando home page, usually presenting them in a "Recommended for you" box.

The scoring function for personalized recommendations can be modeled as follows:

$$s(\boldsymbol{u}_i, \boldsymbol{i}_j) = \varphi(\psi^U(\boldsymbol{u}_i, \boldsymbol{i}_j)) \tag{2}$$

where $\boldsymbol{u}_i$ is a user, and $\boldsymbol{i}_j$ is the candidate product we want to score with respect to $\boldsymbol{u}_i$. The key difference between Eq. 1 and Eq. 2 is that now, the function ψ^U has to model the interaction between user and article attributes (rather than an article-to-article interaction). For example, if we know the user's favorite brand, we can encode a boolean attribute specifying whether the brand of $\boldsymbol{i}_j$ is the same as $\boldsymbol{u}_i$'s favorite brand. In other words, ψ^U encapsulates

the feature-engineering side of our user-to-item recommendation model.

2.3 Personalized Item-based Recommendations

Another recommendation context arises when we attempt to personalize the recommendations we show in product detail pages, such as the ones that appear in the carousel already displayed in Fig. 1. In this case, there are three different entities to be accounted for in our scoring function, namely the user $\boldsymbol{u}_i$, the base article $\boldsymbol{i}_j$, and the candidate product $\boldsymbol{i}_k$:

$$s(\boldsymbol{u}_i, \boldsymbol{i}_j, \boldsymbol{i}_k) = \varphi(\psi^{U,I}(\boldsymbol{u}_i, \boldsymbol{i}_j, \boldsymbol{i}_k)) \tag{3}$$

To address this context, we have to leverage both item-to-item and user-to-item interactions, or possibly come up with ternary interaction attributes. For example, we might simply define $\psi^{U,I}$ as a concatenation of the values returned by the previously defined functions ψ^I and ψ^U, i.e. $\psi^{U,I}(\boldsymbol{u}_i, \boldsymbol{i}_j, \boldsymbol{i}_k) = (\psi^U(\boldsymbol{u}_i, \boldsymbol{i}_k), \psi^I(\boldsymbol{i}_j, \boldsymbol{i}_k))$. Also, we might want to include ternary interaction attributes. For example, we can have a boolean feature specifying whether the user's most frequently purchased brand, the base item's brand, and the candidate article's brand are all the same or not, or a real-valued feature calculating the fraction of pairs (x, y) from the set $\{(\boldsymbol{u}_i, \boldsymbol{i}_k), (\boldsymbol{i}_j, \boldsymbol{i}_k)\}$ such that x and y can be assigned to the same brand.

3 LEARNING TO RANK

We now review the L2R framework underlying our recommender system. In Sec. 3.1, we summarize some of the vast literature available on the topic, whereas Sec. 3.2 outlines the approach we use to estimate our models from data.

3.1 Related Work

L2R approaches are usually classified into three broad families: pointwise, pairwise, and listwise methods [16]. Pointwise models attempt to estimate the relevance score of each item as an independent data point. The supervision for learning the relevance score is typically extracted from previously (e.g. manually) ranked lists of candidates, by using the positions of candidates within the lists as the respective labels. At prediction time, the items returned for a query are sorted according to their scores. Linear or logistic regression are examples of scoring functions used in pointwise methods. Pairwise approaches score ordered pairs of items instead of individual items. Here, the goal is to learn the correct order of these pairs. In other words, the goal is to score the more relevant items higher than less relevant ones. The advantage of this approach over pointwise methods is that it does not require to learn absolute relevance scores. RankSVM [12, 13] is one of the most popular pairwise models. It formalizes ranking as a binary classification problem for item pairs and uses support vector machines as the underlying classifier. Another one is Rank Logistic Regression [21]. RankBoost [10] is also a pairwise model, where the ranking is learned through boosting. The idea is to construct a sequence of "weak" rankers in an iterative fashion, and then to predict ranks using a linear combination of the weak learners. Finally, listwise approaches rely on ranked lists as training examples. In particular, they try to minimize a loss function defined over full lists instead of ordered pairs sampled from those lists. ListNet [4] is a listwise ranking algorithm, which performs gradient descent over a loss function based on cross-entropy. AdaRank [24] is also a listwise approach, based instead on boosting. Gradient-boosted trees (GBTs) have recently become quite popular in learning to rank [3, 19]. Although GBTs have been shown to outperform simpler approaches (such as linear models), training them over web-scale datasets can be very expensive. Moreover, scoring latency is a serious issue for GBTs whenever we are not able to precompute and cache predictions. An alternative model, which is very suitable for the large-scale setting, is the WSABIE algorithm [22], which relies on low-dimensional embeddings and data sub-sampling. Yet another model is given by ElasticRank [9], inspired by a former approach known as LambdaRank [2]. ElasticRank inherits from LambdaRank the idea of weighting the loss function by a listwise penalization scheme. The loss is minimized by stochastic gradient descent, where the training algorithm is allowed to perform only one pass through the training data. Moreover, the special focus of this model is given by sparsity-inducing regularization schemes.

3.2 Ranking Loss Minimization

Let our training sample be a set $\mathcal{X}$ containing the pairs $(\boldsymbol{x}_1^+, \boldsymbol{x}_1^-), \ldots, (\boldsymbol{x}_n^+, \boldsymbol{x}_n^-)$, where each pair $(\boldsymbol{x}_i^+, \boldsymbol{x}_i^-)$ is such that $\boldsymbol{x}_i^+$ should be ranked higher than $\boldsymbol{x}_i^-$. If the setting is given by personalized recommendation, as described in Sec. 2.2, each training pair will be defined as follows:

$$(\boldsymbol{x}_i^+, \boldsymbol{x}_i^-) = (\psi^U(\boldsymbol{u}_j, \boldsymbol{i}^+), \psi^U(\boldsymbol{u}_j, \boldsymbol{i}^-)) \tag{4}$$

where item $\boldsymbol{i}^+$ is more relevant to user $\boldsymbol{u}_j$ than item $\boldsymbol{i}^-$. Relevance labeling might be obtained in several different ways, via implicit or explicit feedback. For example, $\boldsymbol{u}_j$ might have clicked (or purchased) $\boldsymbol{i}^+$ but not $\boldsymbol{i}^-$, assuming that both items where available in the same candidate set. Or, $\boldsymbol{u}_j$ might have explicitly labeled some articles, by rating them or placing them in a wishlist. The way we sample user feedback clearly depends on the available data (and especially on the quality and limitations of user feedback tracking), the specific use case, and possibly on computational requirements, and it is tuned for each application both by offline and online experimentation. The definition given in Eq. 4 can easily be cast to suitable forms for the item-based and personalized item-based recommendation setting, respectively.

For each training pair $(\boldsymbol{x}_i^+, \boldsymbol{x}_i^-)$, our ranking loss is defined as follows:

$$\ell(\boldsymbol{x}_i^+, \boldsymbol{x}_i^-; \varphi) = \max\left\{0, \varphi(\boldsymbol{x}_i^-) - \varphi(\boldsymbol{x}_i^+) + \epsilon\right\} \tag{5}$$

for some slack parameter $\epsilon \geqslant 0$. Different choices are possible other than the hinge loss specified in Eq. 5, e.g. the logistic loss $\log\{1 + \exp(\varphi(\boldsymbol{x}_i^+) - \varphi(\boldsymbol{x}_i^-))\}$. For our applications, we found the hinge loss completely satisfying.

Given a training pair $(\boldsymbol{x}_i^+, \boldsymbol{x}_i^-)$, we can optimize the parameters in φ by minimizing the ranking loss over $(\boldsymbol{x}_i^+, \boldsymbol{x}_i^-)$. This task can be accomplished via standard (sub-)gradient descent methods. Any choice for the parametric form of φ, ranging from simple linear regression to a multilayer perceptron, will be compatible with the ranking metric we are adopting as long as we are able to compute the corresponding (sub-)gradients. Throughout our applications, we assume a linear form for the scoring function ϕ, i.e. $\varphi(\boldsymbol{x}) = \boldsymbol{w}^\intercal \boldsymbol{x}$.

This is not only practical from the computational point of view, but it also preserves the convexity of the ranking loss defined in (5), which is a convenient property for the purpose of parameter optimization.

In order to induce sparsity in the learned scoring function, i.e. to completely drop some components in the weight vector $\boldsymbol{w}$, we can add ℓ_1-based regularization to the loss defined in Eq. 5. The resulting objective function will be convex, with both theoretical guarantees and efficient optimization schemes [20]. As previously advocated in the large-scale setting [17], we additionally consider a squared ℓ_2-term, hence leading to the following ranking loss:

$$\ell^*(\boldsymbol{x}_i^+, \boldsymbol{x}_i^-; \varphi_{\boldsymbol{w}}) = \ell(\boldsymbol{x}_i^+, \boldsymbol{x}_i^-; \varphi_{\boldsymbol{w}}) + \lambda_1 \|\boldsymbol{w}\|_1 + \frac{1}{2}\lambda_2 \|\boldsymbol{w}\|_2^2 \quad (6)$$

where λ_1 and λ_2 are non-negative hyperparameters determining, respectively, the weight of the ℓ_1 and ℓ_2 penalties within the overall loss. The resulting regularization model is usually referred to as elastic-net [25]. It is worth stressing that, for our web-scale application, sparsity is an extremely important requirement for the learned model. The reason is that, to extract useful signal from the available training data, we often need to exploit very high-dimensional feature representations, e.g. by considering all possible pairwise interactions of categorical attributes through a one-hot encoding scheme. This easily leads to feature vectors having billions of components. Therefore, dropping as many irrelevant features as possible is especially useful both to improve generalization and to reduce the memory footprint (and possibly the latency) of the learned model.

In order to learn our parameter vector from a training set $\mathcal{X} = \{(\boldsymbol{x}_1^+, \boldsymbol{x}_1^-), \ldots, (\boldsymbol{x}_n^+, \boldsymbol{x}_n^-)\}$, we use gradient descent to address the following, averaged regularized problem:

$$\min_{\boldsymbol{w}} \frac{1}{n} \sum_{i=1}^{n} \ell(\boldsymbol{x}_i^+, \boldsymbol{x}_i^-; \varphi_{\boldsymbol{w}}) + \lambda_1 \|\boldsymbol{w}\|_1 + \frac{1}{2}\lambda_2 \|\boldsymbol{w}\|_2^2 \quad (7)$$

In our applications, the number of training data points is typically in the order of several millions. To cope with the dataset size, we distribute the training load over several workers using the in-memory cluster computing capabilities offered by Apache Spark (http://spark.apache.org/). This way, we can easily calculate and sum up the gradient of ℓ^* over the full dataset in no more than a couple of seconds, and hence perform a few hundreds of gradient descent iterations in just a few minutes.

As proposed in [15], we can simply enforce sparsity by adding a pruning operation to gradient descent. The pruning step, scheduled every k gradient iterations, simply consists of setting to 0 all the weights w_i such that, for a chosen threshold θ, $|w_i| < \theta$. Clearly, the higher the value we choose for θ, the sparser the model will become. The experiments reported in [9] show that this simple technique performs surprisingly well as compared to way more sophisticated approaches available from the literature [7, 23]. Therefore, we adopt such regularization scheme as our default choice.

4 SYSTEM ARCHITECTURE

This section provides an overview of our system architecture. Our stack is fully deployed on Amazon AWS (https://aws.amazon.com/), and it is organized into a collection of offline jobs on the one hand, and a group of online services on the other hand. Sec. 4.1 describes the former, whereas the web services are presented in Sec. 4.2. A critical outlook is then proposed in Sec. 4.3.

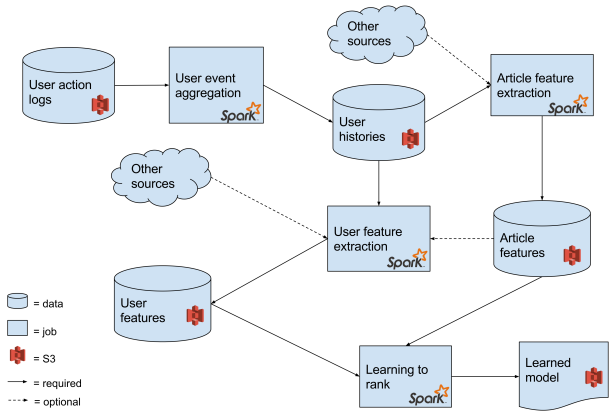

Figure 3: A diagram of our offline jobs: event aggregation, feature extraction, and learning to rank.

4.1 Offline Jobs

Let us look at how we organize all of the necessary batch calculations into a graph of interdependent offline jobs. Here, the goal is to produce and persist all data that will be necessary in order for the live recommendation engine to serve fresh results to our visitors. The jobs architecture is depicted in Fig. 3.

Everything starts from user action logs, which record every action performed by our customers on the Zalando web store. The first job aggregates all actions on a per-user basis. The difficulty here is that the volume of server logs is massive and distributed over multiple locations. Therefore, we cannot afford to crunch and re-aggregate all of this data every time we need to retrieve actions for a given customer. Pre-aggregation of customer actions allows subsequent jobs to selectively retrieve the relevant information in a much more convenient way.

Once the user histories have been aggregated, they are used for two purposes. On the one hand, we can extract a number of dynamic article features (i.e. quantities that change over time), such as number of clicks received by a product within a given time window, number of purchases, and so on. Here, we might be tempted to extract such features directly from the (non-aggregated) event logs. The drawback of such an idea is that it would prevent us from filtering out undesirable data points, such as events generated by robots or crawlers, which pollute our data by introducing noise and distorting their distribution. These problematic events can typically be identified only once we analyze the behavior of the respective cookies/user identifiers, and this behavior only materializes once the relevant event sets are aggregated and some basic statistics are derived from them.

On the other hand, we extract a wide range of user attributes, such as time elapsed since the user was last active on the shop, how the user choices are distributed over different product brands, or the price distribution of purchased articles. User feature extraction

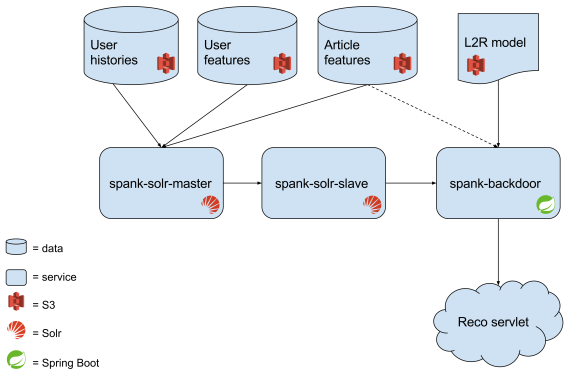

Figure 4: A diagram of our web services: data indexing, real-time ranking, and post-processing. 'Spank' is the name we use to refer to our content-based recommendation stack.

can also exploit already computed article features. For example, if we want to measure how user purchases are affected by product popularity, then for each purchase, we need to check how popular the corresponding product was at the time it was purchased (e.g. in terms of click-through rate).

Finally, the L2R job estimates the scoring function from the extracted historical data, relying on both user and article features. A dedicated job has to be run for each different recommendation setup (i.e. for each one of the different models specified in Eqs. 1–3), since they lean on different data representations and features. The learned scoring functions are then deployed to our live ranking services.

The full offline job pipeline is run at short, regular intervals (e.g. daily). This is important to ensure that new events ingested in our browsing logs can contribute to the calculation of user and article features, as well as to the adaptation of the machine-learned scoring function to drifting data distributions.

4.2 Web Services

The purpose of our recommendation services is to provide a RESTful API [8], returning sorted sets of results for queries that conform to one of the following use cases: (*i*) get top-k recommendations for item $\boldsymbol{i}_i$; (*ii*) get top-k recommendations for user $\boldsymbol{u}_i$; (*ii*) get top-k recommendations for user $\boldsymbol{u}_i$ and item $\boldsymbol{i}_j$. These patterns correspond to the three use cases described in Sec. 2. A diagram of our online architecture is given in Fig. 4.

The first step in setting up our live recommendation stack is to make the user and article data available for real-time processing, as they are originally stored in large, distributed batches. To this aim, we index our data into Apache Solr (http://lucene.apache.org/solr/), which allows us to retrieve single feature vectors by (user/article) id within an average latency of less than 5 milliseconds. Our Solr service is set up with a master/slave architecture. That is, the master server only takes care of indexing data coming from the offline jobs whenever new batches become available, which is a computationally expensive operation. On the other hand, the slaves mirror the content available from the master by mere data replication, and they expose it to the ranking engine for real-time querying. This way, we make sure that the computational load incurred while indexing new data batches will not affect the latency of Solr queries, which would have a tremendous impact on the responsiveness of the recommendation API.

A second component of the live system is given by the ranking servlet, exposing the recommendation API mentioned above. We refer to this component as the *backdoor engine*, for the following reason. As illustrated in Fig. 4, the backdoor service does not serve directly customer requests. Such requests go first to the stack labeled as 'Reco servlet', which then forwards the relevant requests to the backdoor engine. The goal behind this choice is to separate the problem of calculating and ranking recommendations from the problem of post-processing (e.g. filtering or rendering) them for the final customers. Such post-processing tasks are both complex to perform and extrinsic to the genuine recommendation problem, as they typically come from *ad hoc* business requirements or from interoperating with external services, such as sales, campaigns, hand-coded rules, advertising, de-duplication, front-end restrictions, and so on. Therefore, the backdoor-based architecture frees up the recommendation engine from the complexity of managing extrinsic logic, hence streamlining both software development and system operation work.

As shown in the diagram, the L2R model artifact can be loaded directly from the location where it is saved by the offline jobs, i.e. it does not need any sort of indexing, since this artifact is nothing but a self-contained weight-vector (possibly enriched with some additional configuration). This happens as soon as the backdoor servlet is started, while the model is also immediately refreshed whenever more recent artifacts become available. On the other hand, article features can be usefully cached in the same servlet, to a more or less significant extent. Caching will be crucial to minimize serving latency. Differently from user data, which have to be queried according to more unpredictable patterns, article queries tend to be generated according to a power-law distribution, because of the popularity patterns which are intrinsic to fashion shopping. Because of this, we can not only achieve relatively high cache-hit rates, but also initialize the cache with the most popular articles whenever new instances are added to the stack, which will be immensely beneficial to the initial responsiveness of these instances.

4.3 Discussion

We now proceed to discuss, in Secs. 4.3.1–4.3.3, how the system presented so far copes, respectively, with the three operational concerns motivating our general approach.

4.3.1 Adaptation to Diverse Use cases. As described in Sec. 2, product recommendation takes different forms as soon as we move through different contexts, e.g. depending on whether personalization is involved, whether a reference article is currently the focus, and so on. Therefore, being able to seamlessly adapt the available infrastructure to the different use cases is crucial to maximize the value of the adopted recommendation technology. This is the main consideration we had in mind when choosing a model such that only one specific component (i.e. the interaction function ψ) has to be modified whenever we tackle a new context. ψ^I, ψ^U, and $\psi^{U,I}$ are self-contained, plug and play modules, which are completely decoupled from anything but the offline L2R job and the online

backdoor servlet. Of course, these last two components need to be aware of which interaction function has to be loaded for each specific use case. But the logic of the encompassing system remains completely untouched whenever we move from one context to another.

4.3.2 System Operation. Maintaining a recommendation architecture can require significant efforts, for a variety of reasons. The amount of data to be processed, both for offline modeling and for online serving, is typically huge, which requires distributed processing infrastructure on both sides. Also, because of the system complexity and the need to continuously improve over time (in order to tackle new challenges and meet continuously evolving customer needs), the system components need to be easily replaceable and modifiable, while avoiding the risk of breaking the whole system whenever a local change is operated. While the system described in Secs. 4.1–4.2 takes a non-trivial *cognitive* effort to be operated and mastered, it does not require *expensive* operations whenever a component has to be modified in some way. For example, extracting some more user features from the historical data only requires a change in the corresponding feature extraction job, and the outcome will be automatically available for all subsequent stages, independent on when and how the other components will start using the new input. Similarly, adding one more L2R job, e.g. in order to move from non-personalized to personalized item-based recommendations, only requires to implement and add one job to the offline pipeline, without affecting any other component, and independent of whether and when the new model will go live. In other words, while designing the presented architecture, we strived to decouple as much as possible all different system components, so that the cost of operating the whole system could be minimized by isolating the scope of the most typically required interventions. Furthermore, the sharp separation between offline and online stack encapsulates the live system from unpredictable, possibly breaking changes happening in any of the ingested data sources, allowing us to detect the problem from its impact on the offline jobs, and to fix it before it starts affecting our customers through its downstream effects.

4.3.3 Exploitation of Pre-existing Signal. Whenever new data sources become available or whenever new signal can be exploited from a (possibly pre-existing) predictive model, the novel information can be homogeneously fed into the L2R engine in the form of additional features. Such an incremental modeling approach can lead to continuous improvement in recommendation quality, due to the seamless exploitation of the novel information. As we will illustrate in Sec. 5, we found this strategy to generate a lot of momentum in the process of replacing the engines currently used in production by increasingly accurate ones.

An extremely simple example of how we incorporated a pre-existing recommender into L2R is given by how we integrated the output of our previous CF engine for item-to-item scoring. Here, we just started to ingest the predicted item-to-item relevance into the article feature-extraction job, so that the similarities were made available to the interaction function ψ^I as pre-calculated features. A more sophisticated example of how we added a new type of signal is offered by the integration of a Word2Vec-based article embedding model [18]. For the purpose of article recommendation, we were interested in extracting a latent-vector representation of our products, by capturing the contextual relation of article views with the articles explored within the same browsing sessions. To this purpose, we simply model sessions as sequences of article views, and have a Word2Vec learning routine run on these sequences. The job flow is described in Fig. 5. As shown in the diagram, a simple entry point for the new embeddings into our offline job graph is provided by the feature extraction job for articles. The new data source is thereby made available to the L2R stage.

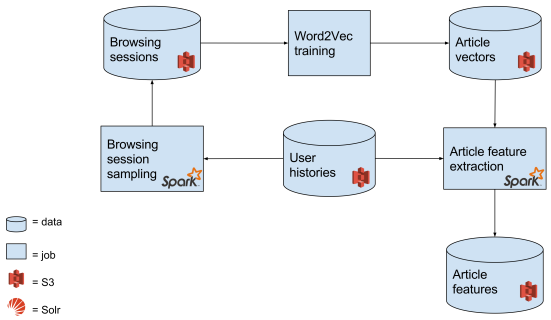

Figure 5: Integration of Word2Vec-based article embeddings into our offline pipeline: the entry point is provided by the article feature extraction job.

5 EXPERIMENTAL RESULTS

As Zalando is mainly focused on online shopping, the most reliable way for us to choose between competing recommendation algorithms and models is by comparing their live (i.e. online) performance in terms of a number of metrics, such as click-through rate, customer conversion rate, number of sales, generated revenue, effective catalog size [11], and additional indicators as well. However, when the candidates for online testing are too many, the only way to prioritize the different options is given by offline testing, i.e. by measuring the performance of the alternative models on historical data. The advantage of offline experiments is not only that the same data can be used to benchmark an arbitrary number of models, but also that poor models can be prevented from hurting our customers by exposing inaccurate recommendations. Moreover, whenever a new machine learning model is developed, extensive hyperparameter tuning and hypothesis testing needs to be done on validation data. Performing this task on online data is simply prohibitive both in terms of time and in terms of business risk. On the other hand, the drawback of offline testing is both that it suffers from bias (as the historical data are generated by the very same models we are trying to supersede), and that it does not capture all subtleties involved in a live system (such as the interaction with other components, the impact on user experience of higher/lower latencies, and so on). Therefore, we usually select between competing models by first deciding, through offline experimentation, which ones we want to benchmark in a live setting, and then by running dedicated A/B tests to determine whether the current production model should be replaced by a new one.

The purpose of this section is to illustrate how we learned one of the lessons discussed in the previous sections, namely the one considered in Sec. 4.3.3. One point that became clear to us both via offline and online experimentation is that focusing on the feature-based model described so far allows us to significantly speed-up algorithmic innovation, throughout our recommendation use cases, via inherently incremental improvements. The reason is that, without forcing us to go for the operational risk and expense of substituting new systems for the current production model, the adopted L2R framework makes room for a seamless integration of heterogeneous models and data sources, by merging all of them into the final scoring model in the form of additional features. Secs. 5.1–5.2 give an example of how this incremental benefit was detected and brought live in one of our personalized recommendation contexts, namely on the Zalando Home.

5.1 Offline Experiment

One way to test the ranking accuracy of a given recommendation engine from historical data is the following. We build our training set by sampling browsing sessions from user action logs, covering whatever time interval is suitable for training our algorithms. Typically, we use data from at least 7 consecutive days, going up at most to 4 consecutive weeks. Then, if our training sample extends to day d at latest, we use day $d+1$ for testing. For hyperparameter tuning, day d is actually held out of the training data, so that it can be used to build a validation set, and the algorithms are trained using data generated no later than on day $d-1$. Such a training sample is enough to provide several millions of user sessions, whereas for the test set we usually subsample about half a million sessions. To construct test queries from a user session, we proceed as follows. All articles that the user clicks on (or purchases) within the session are labeled as relevant items, i.e. as items that the ranking/recommendation algorithm should rank/recommend on top of anything else. Then, we check the position of the relevant items in the top-k recommendation lists produced by the competing algorithms, and we measure the corresponding ranking accuracy.

Our chosen metric for this measurement is normalized discounted cumulative gain (NDCG) at k [6]. A formal comparison of different evaluation metrics goes beyond the scope of this paper. However, it is worth mentioning that, on our datasets, alternative metrics (such as precision or recall at k) usually give consistent indications about which models are most accurate. NDCG is a normalized version of the following metric:

$$DCG@k(\boldsymbol{r},\boldsymbol{l}) = \sum_{i=1}^{\min\{k,n\}} \frac{2^{l_i}-1}{\log_2(r_i)+1} \tag{8}$$

where $\boldsymbol{r} = (r_1,\ldots,r_n)$ is the ranking induced by a given algorithm on n candidate products, and l_i is the relevance label of the i-th product in the list. We set the relevance label to 1 if the product was clicked/purchased, whereas $l_i = 0$ otherwise. DCG has non-negative values, and larger DCG values correspond to better rankings. NDCG normalizes the right-hand side of Eq. 8, dividing it by the DCG of a perfect ranking, i.e. the ranking induced by the relevance labels themselves.

As we started experimenting with L2R, the very first feature to use was simply the score calculated by the pre-existing CF engine.

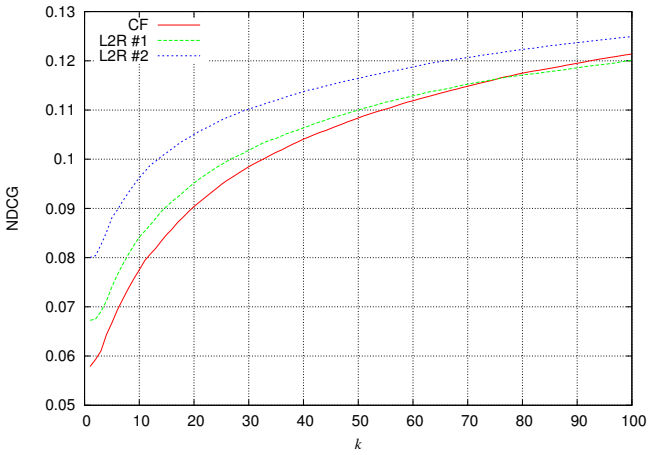

Figure 6: $NDCG@k$ measurements on a one-day traffic sample for: (*i*) CF; (*ii*) L2R using CF-based features plus static product attributes (L2R #1); (*iii*) L2R model using all the features from L2R #1 plus Word2Vec-based similarity (L2R #2).

Basically, the CF engine was an adaptation of the approach presented in [1]. As a number of static article features were available from our product catalog, such as brand, price, category, colors, and a variety of high-level tags, we simply added all such attributes to the L2R model. To this purpose, we engineered several types of attribute interactions and encoded them into our ψ^U function, using the strategy described in Sec. 2.2. We then compared this first version of our L2R model (L2R #1) to the CF baseline, leading to the NDCG measurements plotted in Fig. 6.

As the figure shows, L2R #1 achieves some noticeable improvement over CF, although the ranking deteriorates for large values of k. We notice that the measurement was seen to be statistically significant, as the experiment was repeated on several occasions, for different data samples. The evidence was enough for us to set up and run an A/B test, which we describe in Sec. 5.2. It is important to note that, in order to see the mentioned improvement in NDCG, it was not enough to add a bunch of features to the CF score, as described above, and then just hope for the learned model to regularize away all noise while learning the optimal scoring. Instead, extensive feature selection was necessary, which we performed by repeated cross-validation on held-out data. Unfortunately, when the features are poorly engineered, or just too noisy, the model accuracy can be hurt to a more or less significant extent, and regularization does not seem to be enough to isolate the good signal from noise.

Once the Word2Vec-based article embeddings were available from our modeling pipeline, we also started to experiment with features extracted from the embeddings. The simplest way to engineer such a feature into ψ^U is, for example, to calculate the cosine similarity between the latent vector representation of the candidate product and a suitable vector representation of the user. As a heuristic, the user vector might be calculated as the average of the article vectors from the relevant shopping history. We then add the new feature to the ones already used in L2R #1, and refer to the resulting model as L2R #2. As plotted in Fig. 6, L2R #2 significantly improves over L2R #1. Presumably, this happens because the newly

Country	ΔCTR	ΔCR> 0 & $p < 0.1$
#1	+15.22%	×
#2	+20.44%	✓
#3	+33.13%	×
#4	+23.93%	×
#5	+37.88%	✓

Table 1: Click-through rate and conversion rate results from A/B tests run over five different countries. Country names are masked in order to not disclose possibly sensitive information. L2R is used for treatment, while CF is used for the control group.

engineered feature conveys signal which is not captured by the pre-existing features. This makes the L2R #2 model a second, even more promising candidate for further A/B testing (which is currently work in progress). As the Word2Vec model requires a dedicated training routine to be run, relying on specific hyperparameter tuning, the chosen hyperparameters turn out to have a non-trivial effect on the accuracy of the final L2R model. However, a convenient property of the used L2R framework is that, while we observed a poor hyperparameter setup to make the Word2Vec-based similarity completely inaccurate if used as a stand-alone article-relevance predictor, using it instead as one of several L2R input features makes the final predictions way more robust to inaccuracies in the learned embeddings. In particular, if we refer to the plot in Fig. 6, what we observed is that, for less accurate settings of the Word2Vec hyperparameters, the NDCG curve of L2R #2 would get closer and closer to L2R #1, without however falling below that level. On the other hand, we did not see a stand-alone prediction model based only on the Word2Vec similarity coming anywhere close to the CF baseline.

5.2 Online Experiments

Offline NDCG measurements do not tell us with certainty how the L2R model will impact our business performance indicators in production. Therefore, we ran some A/B tests in order to compare L2R to the CF baseline. The tested L2R model is L2R #1 from the previous section (i.e. not including Word2Vec-based features, which were not yet available when the online tests were run). The tests concern the Zalando Home, where general personalized recommendations are shown to visitors (along with other content) as hints for navigation. The metrics we used for the evaluation were mainly click-through rate (CTR) and conversion rate (CR). The former is defined as the fraction of displayed recommendations that receive at least one-click, while the latter is the fraction of visitors who end up making at least one purchase during the relevant time window. The results are summarized in Table 1 for five different countries. To avoid disclosing any sensitive information with respect to our business performance, for CTR we report the relative difference between control (CF) and treatment group (L2R), whereas for CR, instead of mentioning explicit values, we report whether a statistically significant improvement ($p < 0.1$) was achieved or not. For an overview of the A/B testing methodology on the web and the involved significance calculations, we refer the reader to [14].

As shown in the table, the treatment group consistently outperforms control in CTR. All CTR measurements were statistically significant, where the p values very quickly converged to 0. An interesting point to remark is that, although the gap between L2R and CF is significantly positive everywhere, different countries display a noticeably different behavior in this respect, as the relative difference between the two algorithms exhibits a relatively large variance over different domains. This country-specific behavior is the reason why it is extremely important to run a dedicated test for each involved domain, in order to avoid making false (and financially risky) generalizations. On the other hand, our previous experience with the subtleties of fashion shopping shows that CR, as compared to CTR, is much more difficult to increase by mere algorithmic change. Nevertheless, CR increases significantly in two out of the five reported experiments, i.e. in countries #2 and #5, while it does not deviate significantly from the control in all other experiments. Besides the metrics discussed here, we also monitored additional revenue- and engagement-based measurements, which also showed, overall, either an improvement or no statistically significant difference. Therefore, the L2R engine was rolled out to production in all involved domains. In our interpretation, the key strategy by which L2R superseded CF in our production systems is by enriching the signal provided by CF through the integration of additional data sources into the final predictions, rather than discarding the old system and building on entirely different foundations.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we argued that the operational challenges involved in running recommender systems in the real world should be included as major dimensions in a formal analysis of such systems. In particular, we sketched three basic criteria by which such an analysis should be conducted, namely (*i*) flexibility of the system with respect to different use cases, (*ii*) cost of operations, and (*iii*) ease of integration of heterogeneous signal. To illustrate the importance of these criteria, we showed how they inspired us to redesign the recommender systems at Zalando from the ground up.

Although several directions are open for further development, we wish to mention one which is especially appealing as a next step, namely to complement our system with deep learning-based signal. Deep neural networks were recently shown to have significant potential also in the recommendation domain [5]. In particular, they promise to alleviate the burden of the feature engineering process, which unfortunately requires, in our current setup, a non-trivial amount of manual work and quite a few iterations in order to deliver satisfying results.

7 ACKNOWLEDGMENTS

I am extremely grateful to everyone in the Recommendation Team at Zalando for the outstanding help and support they provided throughout all phases of this project. A special thought goes out to my friend Ilaria Castelli.

REFERENCES

[1] Fabio Aiolli. 2013. Efficient top-n recommendation for very large scale binary rated datasets. In *Seventh ACM Conference on Recommender Systems, RecSys '13,*

Hong Kong, China, October 12-16, 2013, Qiang Yang, Irwin King, Qing Li, Pearl Pu, and George Karypis (Eds.). ACM, 273–280.

[2] Christopher J. C. Burges, Robert Ragno, and Quoc Viet Le. 2006. Learning to Rank with Nonsmooth Cost Functions. In *Advances in Neural Information Processing Systems (NIPS)*. 193–200.

[3] Christopher J. C. Burges, Krysta Marie Svore, Paul N. Bennett, Andrzej Pastusiak, and Qiang Wu. 2011. Learning to Rank Using an Ensemble of Lambda-Gradient Models. In *Proceedings of the Yahoo! Learning to Rank Challenge, held at ICML 2010, Haifa, Israel, June 25, 2010*. 25–35.

[4] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. 2007. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th International Conference on Machine learning (ICML 2007)*. ACM, New York, NY, USA, 129–136.

[5] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems (RecSys '16)*. ACM, New York, NY, USA, 191–198.

[6] Bruce Croft, Donald Metzler, and Trevor Strohman. 2009. *Search Engines: Information Retrieval in Practice*. Addison-Wesley, Boston (MA).

[7] John C. Duchi and Yoram Singer. 2009. Efficient Online and Batch Learning Using Forward Backward Splitting. *Journal of Machine Learning Research* 10 (2009), 2899–2934.

[8] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. Dissertation. University of California, Irvine.

[9] Antonino Freno, Martin Saveski, Rodolphe Jenatton, and Cédric Archambeau. 2015. One-Pass Ranking Models for Low-Latency Product Recommendations. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*. ACM, 1789–1798.

[10] Yoav Freund, Raj Iyer, Robert E. Schapire, and Yoram Singer. 2003. An Efficient Boosting Algorithm for Combining Preferences. *Journal of Maching Learning Research* 4 (2003), 933–969.

[11] Carlos A. Gomez-Uribe and Neil Hunt. 2016. The Netflix Recommender System: Algorithms, Business Value, and Innovation. *ACM Trans. Management Inf. Syst.* 6, 4 (2016), 13:1–13:19.

[12] Ralf Herbrich, Thore Graepel, and Klaus Obermayer. 2000. Large Margin Rank Boundaries for Ordinal Regression. In *Advances in Large Margin Classifiers*, Smola, Bartlett, Schölkopf, and Schuurmans (Eds.). MIT Press, Chapter 7, 115–132.

[13] Thorsten Joachims. 2002. Optimizing Search Engines Using Clickthrough Data. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, USA, 133–142.

[14] Ron Kohavi, Roger Longbotham, Dan Sommerfield, and Randal M. Henne. 2009. Controlled experiments on the web: survey and practical guide. *Data Mining and Knowledge Discovery* 18, 1 (2009), 140–181.

[15] John Langford, Lihong Li, and Tong Zhang. 2009. Sparse Online Learning via Truncated Gradient. *Journal of Machine Learning Research* 10 (2009), 777–801.

[16] Tie-Yan Liu. 2009. Learning to Rank for Information Retrieval. *Foundations and Trends in Information Retrieval* 3, 3 (2009), 225–331.

[17] H Brendan McMahan, Gary Holt, D Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, et al. 2013. Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD 2013)*. ACM, 1222–1230.

[18] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems (NIPS 2013)*, Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger (Eds.). 3111–3119.

[19] Ananth Mohan, Zheng Chen, and Kilian Q. Weinberger. 2011. Web-Search Ranking with Initialized Gradient Boosted Regression Trees. In *Yahoo! Learning to Rank Challenge*. 77–89.

[20] S. Negahban, P. Ravikumar, M. J. Wainwright, and B. Yu. 2009. A unified framework for high-dimensional analysis of M-estimators with decomposable regularizers. In *Advances in Neural Information Processing Systems 22 (NIPS 2009)*, Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta (Eds.). 1348–1356.

[21] D. Sculley. 2009. Large scale learning to rank. In *NIPS Workshop on Advances in Ranking*.

[22] Jason Weston, Samy Bengio, and Nicolas Usunier. 2011. WSABIE: Scaling Up to Large Vocabulary Image Annotation. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*. 2764–2770.

[23] Lin Xiao. 2010. Dual Averaging Methods for Regularized Stochastic Learning and Online Optimization. *Journal of Machine Learning Research* 11 (2010), 2543–2596.

[24] Jun Xu and Hang Li. 2007. AdaRank: A Boosting Algorithm for Information Retrieval. In *SIGIR '07: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, New York, NY, USA, 391–398.

[25] H. Zou and T. Hastie. 2005. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society. Series B* 67, 2 (2005), 301–320.