Pointwise, Pairwise and Listwise Learning to Rank

Information Retrieval 1: Homework 3

Christoph Hönes
University of Amsterdam
Stud. nr.: 12861944

Tamara Czinczoll
University of Amsterdam
Stud. nr.: 12858609

Yke Rusticus
University of Amsterdam
Stud. nr.: 11306386

1 INTRODUCTION

In the field of Information Retrieval, a wide range of methods have been examined to retrieve relevant documents for a given query. In this work, we focus on three approaches to learning to rank (LTR): pointwise, pairwise and listwise. In the pointwise approach, we approach ranking as a regression problem. However, this a naive approach, because ranking should in fact not be considered as either a regression or classification problem, as will be discussed further in this work. With the pairwise approach we addressed this problem, by considering pairs of documents at once, using two different versions of RankNet [1]. The limitation of both the pairwise and pointwise approach remains. That is, in both cases ranking quality is not optimized for directly. This problem is addressed by the listwise approach, using LambdaRank [3]. In the following section, we will answer some relevant theoretical questions. In Section 3 the models are explained and the experimental setup is covered in Section 4. The results are presented and discussed in Sections 5 and 6.

2 THEORETICAL QUESTIONS

TQ1.1

In this assignment you are already given the feature vector x for a given document and query pair. Assuming you have control over the design over this feature vector, how would you design it for the e-commerce domain?

Preferably, the features are independent of each other. Then a model does not have to learn these dependencies first before it can make useful predictions. This also implies that the feature vectors should not contain redundancies. Redundancies would waste computations during training. Specifically, for the e-commerce domain, we might want to keep underlying information secret (specifically, the information that the feature vectors encode) in ethical sensitive fields. How exactly this would be taken into account is a field on it's own, but it is important to keep it in mind.

TQ1.2

Briefly explain the main limitations of Offline LTR.

In offline LTR, we evaluate by essentially attempting to model a user's happiness with the produced results. The measures that are used for this (think about precision, recall, NDCG, etc.) can however, never truly capture a user's true experience - an inadequate measure implies focusing on the wrong problem. Furthermore, oftentimes annotated data sets are used, build by trained human annotators. Also in this case there is no guarantee that the annotators and the eventual users will agree on what is relevant and what is not.

TQ1.3

Contrast the problem of learning to rank with ordinal regression/ classification.

In regression and classification examples are considered independently. In LTR that would mean that each document-query pair is given a score, without considering the other document-query pairs. However, document scores should not be considered independently, because ranking is neither a regression nor classification problem. In contrast to the individual assessment of examples, ranking is about the ordering of each document, with respect to all other documents. Therefore, ordinal regression/classification methods would not suffice in the problem of learning to rank, because ranking quality is not optimized for directly.

TQ1.4

Is it possible to directly optimize a given metric? For instance, can we use ERR as a loss function? Explain your answer.

Ideally, in order to optimize for ranking quality directly, we would use known metrics such as precision, NDCG or ERR to optimize for. However, these metrics are non-continuous and non-differentiable, so these can not be used in gradient descent directly. We can however, optimize for the lower or upper bounds of a given metric, which is often done in LTR.

TQ2.1

Briefly explain the primary limitation of pointwise LTR with an example.

This answer has much overlap with the answer given in TQ1.3. The pointwise method uses regression or classification loss to optimize for, i.e. query-document pairs are scored independently of all other query-document pairs. Ranking is however, neither a regression nor classification problem. In the pointwise method, ranking quality is not directly optimized for, which is a fundamental flaw. (For more details, see TQ1.3)

TQ3.1

Comment on the complexity of training the RankNet (hint: In terms of n, the number of documents for a given query).

The RankNet is trained with a pairwise loss which means for each query the loss needs to be computed for every possible pair of documents. In terms of n (the number of retrieved documents for a query) the computational complexity is quadratic ($O(n^2)$).

TQ3.2

Comment on the complexity of training the sped-up version of RankNet and contrast it with the first version you implemented. (Hint: Consider the number of pairs of documents for a given query)

The most computational expensive part of training a neural network is usually the backpropagation. In the sped up version we save some time by directly computing one part of the gradient instead of computing a loss first (with approximately same computational cost) and then additionally letting auto-grad compute the gradients for the loss. But probably even more important is the fact that we can aggregate the contribution of each document by summing the lambdas for all pairings with the other documents. This reduces the computational complexity of the model with respect to the number of retrieved documents per query n from quadratic to almost linear ($O(n)$).

TQ3.3

Mention the key issues with pairwise approaches. Illustrate it with an example.

The main issue with pairwise approaches is that they minimize the number of incorrect inversions treating every pair of documents as equally important. This does not reflect very well a good ranking because it is way more important for the user to have highly relevant documents in the first couple of retrieved documents than getting the order of very little relevant documents at the bottom of the ranking correct. For example, consider a ranking with two relevant documents and 100 irrelevant documents. The pairwise approach would prefer a ranking that has the first only irrelevant documents in the first 48 ranks followed by the two relevant ones than having one of the relevant documents at rank one and the other one at the bottom of the ranking. As most users only look at the first couple of retrieved documents the latter ranking is preferable over the one preferred by the pairwise approach.

TQ4.1

What does the quantity λ_{ij} represent in LambdaRank?

The λ_{ij} is the gradient of the cost with respect to the model scores for a document pair (i, j). In LambdaRank this gradient is computed directly, leaving out actually calculating the cost function itself, since it is not necessary to optimize the model. The lambdas can also be seen as forces pushing on the ranking of the documents (i, j). If $i > j$, document i will get a push upward in the ranking of size $|\lambda_{ij}|$. Likewise, document j is pushed downwards by an equal strength. In LambdaRank, the lambdas are scaled by the change in the information retrieval metric we want to optimize that arises form swapping documents (i, j) in the ranking. It was empirically shown that this leads to good results [2, 3].

TQ4.2

You compute a similar λ_{ij} in RankNet, but the approach is still pairwise. In your opinion, does this mean that LambdaRank is still a pairwise approach? Justify your claim.

LambdaRank has properties of both approaches. Like a pairwise approach, it takes a document pair (i, j) as input and computes the probability that i is ranked higher than j. However, instead of minimizing the average number of inversions in the ranking, LambdaRank directly optimizes any information retrieval metric, like NDCG, through a lower-bound. This listwise learning-to-rank property makes it, in our opinion more of a listwise than a pairwise approach.

3 MODELS AND IMPLEMENTATION

3.1 Pointwise model

The pointwise model is taken to be a simple Multi-Layer-Perceptron (MLP) that uses the Mean-Squared-Error (MSE) regression loss to predict the given relevance grades for each query-document pair seperately. The model takes as input the query-document feature vector and maps it to a single number. Using this result, the loss and gradients are calculated and the model is updated accordingly, using the Adam optimizer [4]. Ranking is subsequently done by sorting all the predicted relevance scores, such that NDCG can be calculated.

3.2 Pairwise model (RankNet)

The architecture of the RankNet model is as well a simple MLP with ReLU activation functions after each layer. The loss is calculated according to equation 3 on the assignment. The combinations are generated by python itertools product which is a fast way to get all possible combinations of a sequence. Again we use the Adam optimizer for updating the weights.

The sped up version of RankNet inherits from the original RankNet class such that the basic functionality stays the same. The only changes made are in the training and the loss function. Instead of the pairwise loss the lambdas (gradients of the loss) are computed and aggregated for each document according to 6 and 7 on the assignment. In the training function instead of calling *.backward()* on the pairwise loss, *.backward(λ)* is called on the scores that were the output of the model (where λ is a vector of the λ_is see equation 7 of the assignment).

Each of the two versions has two training functions. One of them performs a forward and backward pass for every query and the other one uses batches of queries. Even though the per query update is already kind of a mini-batch version as it considers multiple documents at once I will from now on refer to it as stochastic gradient decent and call the other train function the batched version.

3.3 Listwise model (LambdaRank)

LambdaRank also uses a simple MLP to predict a score for each document, given a feature vector. The ranking is a list of documents sorted based on their scores. When training, the model is optimized to maximize a lower-bound for an arbitrary ranking metric, like Normalized Discounted Cumulative Gain (NDCG) or Expected reciprocal Rank (ERR), and thus directly optimizes the ranking metric, making it a listwise approach. The model calculates the gradients (λ_{ij}) in the same way as the sped-up pairwise approach. However, it also scales them with the difference of the ranking measure of the original ranking and when documents i and j are swapped. Our implementation uses the Adam optimizer and calculates the gradients based on equations 8 and 6 in the assignment. The model trains per query and does not use query batches. We vectorized as many computations as possible, but to get the ΔIRM values we used two abbreviated for-loops. We also added another NDCG function in evaluate.py which only calculates the ideal NDCG for the labels once per query.

4 EXPERIMENTAL SETUP

4.1 Pointwise

For the final pointwise model, we used two hidden layers, each having 400 nodes, and a learning rate of 5e−4, motivated by the results of the parameter comparison given in Figure 1. We trained the model for a maximum of 3000 steps, of which each step was an iteration with batch size 1000. We applied early stopping, when the model showed no change in NDCG over the validation set in the average of the last 5 evaluations, compared the the 5 evaluations before that. Then, we compared loss and NDCG over training steps and we examined the distribution of relevance scores within the annotations and within the predictions.

4.2 Pairwise model (RankNet)

For the hyperparameter search we used the batch version of the train function in both cases because especially for the original RankNet it reduces the computational time a lot. Note that due to the nature of the implementation the advantage of the sped-up RankNet in terms of mere computational time is not that clearly visible but this effect is unrelated to the approaches. For the single query training function the sped-up version is about three times faster. The query IDs are shuffled every epoch as there is some parts in the data where a sequence of queries with either very few or a lot of documents is present. For hyperparameter tuning and early stopping the NDCG metric is used on the validation set. NDCG is evaluated every 100 queries. If the model does not improve for eight consecutive evaluations then the training is stopped concluding the model has converged. We use a batch size of 500 queries per backward pass. We perform a grid-search over the following hyperparameters: learning rate: 2e-3, 1e-3, 5e-4 number of layers (and neurons per layer): [200], [200, 100], [200, 100, 50]

We evaluate the best model on the test set and report the performance. To compare the convergence of the original RankNet and the sped up version we contrast the rate of improvement in NDCG over increasing number of iterations for the respective best models. We also compare the performance of the best models with respect to two different metrics, NDCG and Average Relevant Rank (ARR).

4.3 Listwise model (LambdaRank)

We first performed hyperparameter tuning using grid search. We tested learning rates of 10^{-4}, 10^{-5}, 10^{-6}, and two hidden layers with $(200, 100)$ neurons as well as three hidden layers with $(200, 100, 50)$ neurons. We also tested 0.5 1 and 2 as sigma values, keeping the other parameters fixed at 10^{-4} and $(200, 100)$.

We implemented a convergence criterion for early stopping that uses the ndcg values evaluated every 100 iterations on the validation set. If the ndcg value has not surpassed the current best one within five evaluations, the criterion is triggered.

We report the performance on the test set for the model performing best on the validation set and plot the NDCG and ERR throughout training. Finally, we compare the performances of the pointwise, pairwise and listwise approaches.

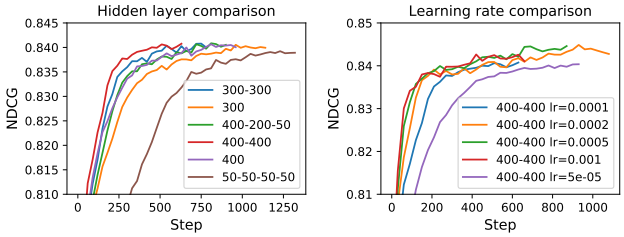

Figure 1: Parameter comparisons for the pointwise model. Left: NDCGs for varying layer number and dimensions, with learning rate 1e−4**. Right: NDCGs for varying learning rates, with two hidden layers with 400 nodes.**

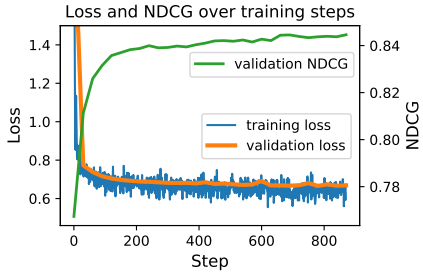

Figure 2: Loss and NDCG over training for two hidden layers with 400 nodes and learning rate 5e−4**.**

5 RESULTS

AQ2.1

Train a new model with the best performing model's hyperparameters. Every few batches (depending on your available compute power), compute NDCG over the validation set. Plot the loss and NDCG together. Comment on what you observe.

Figure 2 shows training and validation losses, and validation NDCG over training steps. Validation loss and NDCG were evaluated every 30 steps. The figure shows that the model quickly reaches a performance close to where it stops at the end. Moreover, we see that NDCG increases as the loss decreases. This means that ranking improves when a model simply learns to predict individual relevance scores. The final model was evaluated on the test set and had an NDCG of 0.85 ± 0.14.

AQ2.2

Compute a distribution of the scores (if you're using a classification loss, use the argmax) output by your model on the validation and test sets. Compare this with the distribution of the actual grades. If your distributions don't match, reflect on how you can fix this and if your solution is sufficient for LTR.

In the comparison of the score distributions, shown in Figure 3, predicted values are rounded to the closest integer in $\{0, 1, 2, 3, 4\}$. The figure shows, that in each case the model under-predicts the number of scores with relevance 0 or 3, and over-predicts the number of scores with relevance 1 or 2. This could be fixed, by

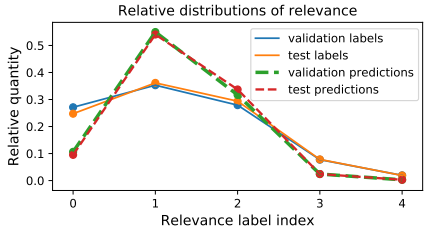

Figure 3: Score distribution comparison for the "true" labels of the test and validation set and the predicted labels from the best model.

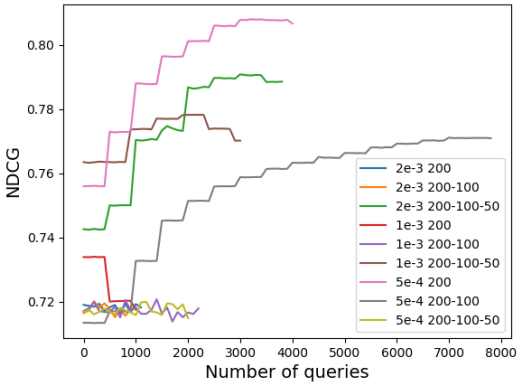

Figure 4: Grid Search over learning rate and number of layers for original RankNet.

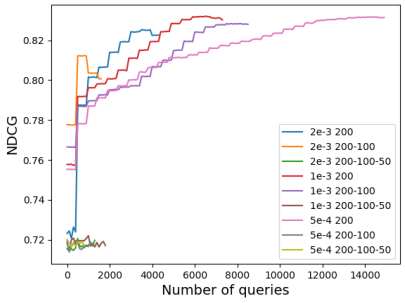

Figure 5: Grid Search over learning rate and number of layers for sped-up RankNet.

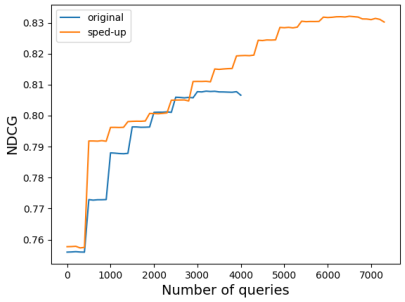

Figure 6: Comparison of original RankNet and sped-up RankNet convergence.

adding a KL-divergence term in the loss that penalized any deviation from the actual distribution. We can do this for LTR, since the KL-divergence term would be differentiable. However, it is still not optimal since the pointwise approach is fundamentally flawed, as discussed in TQ2.1.

AQ3.1

Contrast the convergence of the sped up version of RankNet with the previous version.

Figure 4 and 5 show the hyperparameter search for the original RankNet and the sped-up version respectively. In both cases the model is very sensitive to the learning rate and for some combinations the convergence criterion stops the training after a few iterations because the model is not learning. Even with only one hidden layer both versions can get very good results. We limited the maximal training time to three epochs but all the models already converged within the first epoch. The training is in general pretty unstable and especially for the original model the success also depends a bit on lucky batch shuffling. The best combination of hyperparameters is the single layer architecture with a learning rate of 5e-4 for the original version and the same architecture with a learning rate of 1e-3 for the sped up version. The original model could achieve an NDCG score of 80.48% (+/- 14.7%) and an ERR score of 61.59% (+/- 24.2%) on the final test set evaluation. The test results of the sped-up version are 83.14% (+/- 14%) for NDCG and 66.8% (+/- 24.9%) for ERR. Figure 6 compares the RankNet and its sped-up version's NDCG performance for the best hyperparameters over increasing training time. The sped-up version of the RankNet converges slightly faster in the beginning and is then roughly on par with the original version later on. The sped-up version can also reach a higher validation performance compared to the original version, however the convergence criterion might have kicked in a little early in this run. Note that the sped up version uses a slightly higher learning rate as well.

AQ3.2

Train a new model with the best performing model's hyperparameters. Every few batches (depending on your available compute power), compute NDCG and ARR (Average Relevant Rank) over the validation set. Plot these numbers. Explain what you observe. In particular, comment on which metric is better optimized and what you can conclude from it.

Figure 7 and 8 contrast the NDCG and ARR performances of the original and the sped-up RankNet on the validation set respectively. For both models the NDCG steadily increases while the ARR decreases. This makes sense because a lower ARR means a better ranking. Therefore the two metrics are consistent. Due to the simplicity of the dataset both metrics are optimized pretty well.

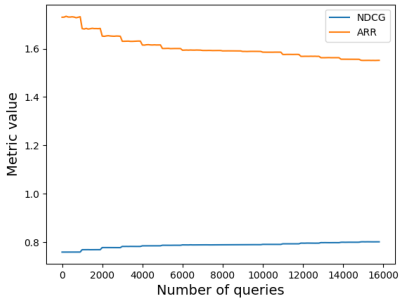

Figure 7: Comparison of NDCG and ARR over training time for the original RankNet.

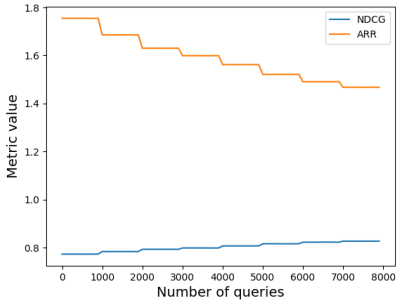

Figure 8: Comparison of NDCG and ARR over training time for sped-up RankNet.

However, the decrease in ARR seems to be bigger than the increase in NDCG and hence ARR is better optimized. This is due to the nature of pairwise approaches and is much related to the effect discussed in TQ3.3. For ARR it does not matter if 50% of the relevant documents are on top of the ranking while 50% are on the very bottom or if all of the relevant documents are in the middle of the ranking. In the same example NDCG would score much higher in the first scenario compared to the second. This makes NDCG more compatible with human judgement.

AQ4.1

Train a new model with the best performing model's hyperparameters. Every few batches (depending on your available compute power), compute NDCG over the validation set. Plot these results. How does this compare to the previous two plots for RankNet and the Pointwise model?

First we present the hyperparameter search and then answer the question.

The results of the grid search to find the best hyperparameters are plotted in Figure 9. The plots show clearly that the model with a learning rate of 10^{-4} and with two hidden layers of sizes (200, 100) not only achieves the best NDCG score, but also converges fastest out of the three models that successfully start learning. Three other models seem to not be able to learn anything and are stuck

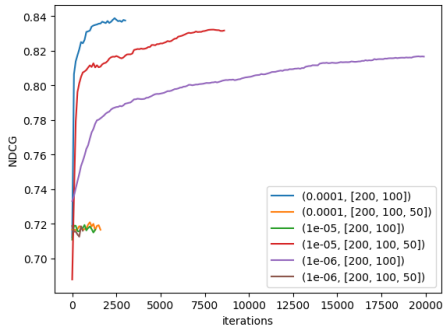

Figure 9: NDCG over training on the validation set during the hyperparameter grid search.

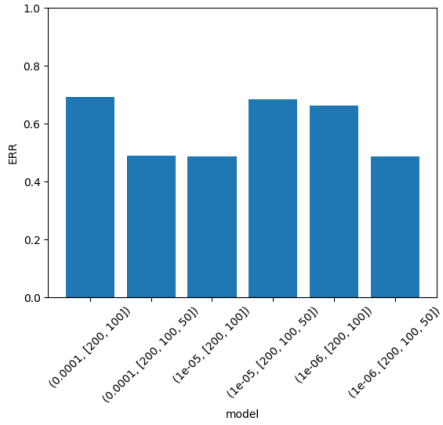

Figure 10: ERR on the validation set for the models of the hyperparameter grid search.

at around 0.72. The model versions with three hidden layers only seem to be able to learn with learning rates lesser or equal to 10^{-5}.

The ERR values of these models are shown in Figure 10. The best model with an ERR score of 0.6913 is the same one with the best NDCG score. The models show a similar performance when measured with ERR as with NDCG. The results for the different sigmas are shown in Figure 11. Whereas a sigma of 0.5 results in no training progress, the performances of the model with sigma set to 1 and 2 is very similar, with no clear winner. As a result of the hyperparameter tuning, we decide to use a learning rate of 10^{-4}, a sigma of 1 and two hidden layers of sizes 200 and 100 for further analysis.

Figure 12 shows the NDCG measured on the validation set for the fine-tuned model. When compared to the pointwise model, both approaches reach similar NDCG values, relatively fast. The pairwise approach takes more time to converge. LambdaRank's best model converges very fast in terms of iterations, however in computation time it is the slowest. The hyperparameter tuning for the pointwise approach seems the most stable, with all configurations achieving similar NDCG scores. RankNet and LambdaRank are more unstable in this regard, with some models never taking off at all.

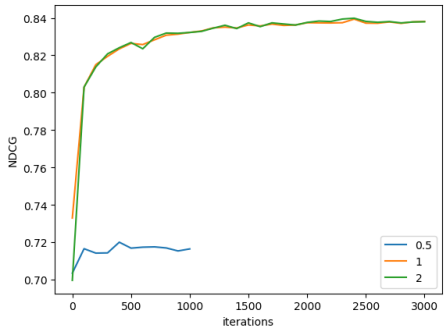

Figure 11: NDCG on the validation set for different sigma values.

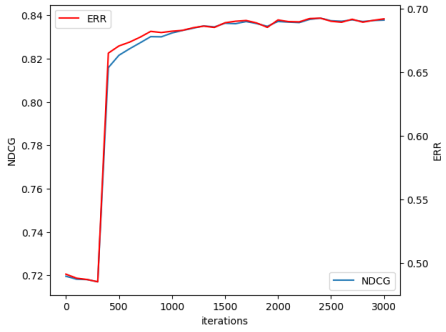

Figure 12: NDCG and ERR on the validation set for the fine-tuned model.

On the test set, the fine-tuned model achieves an NDCG score of 0.85±0.14 and an ERR score of 0.70±0.25.

AQ4.2

We have in total 3 models: Pointwise, RankNet, and LambdaRank, and 3 corresponding losses. Compute the losses we've discussed so far against each model (select the best performing model). Report these in a table. Reflect on what you observe.

Table 1 lists the test losses and NDCG values of the three approaches. Since LambdaRank does not explicitly calculate a loss, only its NDCG is given. The task of comparing the models' losses is somewhat confusing to us, since they do not all depend on the same cost function, and the whole point of the fast RankNet and LambdaNet is that the loss function is only computed indirectly. In terms of NDCG performance, we observe that no method clearly outperforms the other two. They all achieve similar NDCG values. We expected the pointwise approach to perform worst and LamdaRank to perform best, with RankNet closer to LambdaRank. In practice, LambdaRank and the pointwise approach have the same NDCG, but LambdaRank takes much (!) longer with regard to computation time. RankNet performed worst of all approaches. We did not give a loss value for the standard version of RankNet, because its loss curve is extremely noisy and we are not able to give a representative value. It fits our expectations that RankNet performs worse than LambdaNet. The pointwise model's strong performance could be due to its fast training times or due to the dataset and should be explored in future work.

Model	Loss	NDCG
Pointwise	0.7	0.85 ± 0.14
Pairwise original	?	0.81 ± 0.15
Pairwise sped-up	–	0.83 ± 0.14
Listwise	–	0.85 ± 0.14

Table 1: Loss and NDCG scores for the three models.

6 CONCLUSIONS

In summary, three approaches to learning-to-rank have been evaluated and compared: pointwise, pairwise and listwise. Regarding NDCG scores, all three methods show similar results. Surprisingly though, the pointwise approach scored slightly higher than the latter two methods, even though this method does not directly optimize for ranking quality. The pairwise approach showed the worst results out of the three methods. Training time took longer than the listwise approach, and the pairwise approach scored slightly lower on performance. In conclusion, the listwise approach is preferred over the two others: concerning information retrieval, it is more theoretically sound than the pointwise approach, and is showed better perfomance and faster training than the pairwise approach. However, in future work, statistical testing might be applied to reliably draw conclusions from performance comparisons.

REFERENCES

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

[2] Chris J.C. Burges. 2010. *From RankNet to LambdaRank to LambdaMART: An Overview*. Technical Report MSR-TR-2010-82. https://www.microsoft.com/en-us/research/publication/from-ranknet-to-lambdarank-to-lambdamart-an-overview/

[3] Christopher J Burges, Robert Ragno, and Quoc V Le. 2007. Learning to rank with nonsmooth cost functions. In *Advances in neural information processing systems*. 193–200.

[4] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).