Recommender System - Project

Robert Fischer
01425684
e1425684@student.tuwien.ac.at

Elisabeth Harlander
01106899
e1106899@student.tuwien.ac.at

Stephan Lederer
0612193
e0612193@student.tuwien.ac.at

Stefan Schmid
00252354
e0252354@student.tuwien.ac.at

Lukas Steindl
11743494
e11743494@student.tuwien.ac.at

Roman Steiner
0326433
e0326433@student.tuwien.ac.at

ABSTRACT

To solve this years RecSys 2019-challenge we used Apache-Spark to preprocess and profile the trivago-dataset. We extended Mllib with specialized components to test simple baseline-techniques as well as more complex recommender-models. We realized that different models perform well for different session-types so we engineered session-features and learned to predict the most appropriate model. Using this ensemble-technique we achieved 0.607 on the trivago-submission-test-set.

ACM Reference Format:
Robert Fischer, Elisabeth Harlander, Stephan Lederer, Stefan Schmid, Lukas Steindl, and Roman Steiner. 2019. Recommender System - Project. In *Proceedings of RecSys '19: RecSys Challenge 2019 (RecSys '19).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In this document we describe our solution to the Trivago RecSys-Challenge '19 where we implemented different recommender models and combined them in an ensemble. We use "Apache Spark" to load, preprocess and profile the trivago-data. We also make extensive use of the MLLib-Machine-Learning-Library that builds on top of Spark.

In this assignment we mainly focused on the distribution of data intensive computation on large-scale clusters. We won't dive too deep into the individual models. This has been done for the recsys-team-project already and is available as an appendix to this report.

First we will detail about our custom environment (section 2), then our general strategy (section 3) on how to tackle the RecSys 2019 task. To get the data into a format suitable for our needs we describe the data processing methodology (section 4). As we employ an ensemble, which consists of several models, the following models will be detailed (section 5 to 8). The baseline approaches : display order, interaction order, interaction frequency, price and popularity. As well as not so successful ones: Content Based Recommender, Sequence Clustering and Model based Collaborative Filtering. Next one can read our approach to combine these models using manual weighting or several machine learned models (section 9).

Finally we will describe our results and a set of gotchas and other interesting findings during the course of the implementation.

An overview of the structure of the source code as well as a detailed description of the deep learning based model is given in the appendix.

RecSys '19, Summer 2019, Copenhagen

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 ENVIRONMENT

For our work we decided to use a cloud-based Spark-Cluster that can automatically scale up to 8 virtual nodes, each with 8 CPU-Cores and 28.0 GB Memory.

3 STRATEGY

Since MLLib does not provide many recommender-models out of the box, we decided to create custom transformers, estimators and evaluators by overwriting the base MLLib classes. This allowed us to reuse existing pipeline and cross-validation components.

Most of the baseline models are just transformers without fit methods. For collaborative filtering as well as for the popularity based model we had to overwrite also the fit method. Each custom-recommender outputs a weight column. For example: the popularitiy-recommender adds a new column with $\{9350916 = 0.1667, 9376634 = 0.3333, 2406373 = 0.5\}$ for the last clickout in a session with impressions list: 2406373|9376634|9350916. It may also happen that a recommender provides no output {}. However we ensured that the next pipeline-stages can deal with that.

To produce the final ranking we use a custom-transformer to combine all recommender-weights into a single weight-vector with one weight for each item in the impressions-list.

The combiner also accepts a weighting-vector as hyperparameter allowing us to give more credit to the output of stronger classifiers.

We also implemented a scoring and evaluation module to compute the reciprocal rank per session and finally the mean reciprocal rank over all sessions.

Every team-member worked on one or more models (depending on the complexity of the model) and evaluated them independently before integrating them into the final ensemble.

We realized that if we would always pick the best performing recommender for a session we would get a MRR-Score of 0.786 on our own test set. So we spent some time to engineer session features and trained one binary classifier per model to predict if it would rank the clickout-hotel to the first position. We then used the classifiers-confidence to decide which model to use.

In order to come up with good session features we did extensive data exploration.

4 DATA PREPROCESSING

All the baseline methods described in section 5 are performed on the same preprocessed data set. The data set is preprocessed in such a way that it also can be used with the Apache Spark CrossValidation. This leads to a data structure where each row represents one session. Each row is defined by the ids "`session_id`" and "`user_id`". For the submission some basic information about the predicted clickout item is need like "`timestamp`", "`step`" and "`impressions`". The whole content of the actions of the session is compressed into the column "`content`".

In the training data set we only use the last clickout item for prediction. If therefore the session has multiple clickcout items we only predict the last one.

There are a few data rows / sessions in the original file which have faulty values. Since they are very sparse we simply removed the following cases.

- Rows of the action type *'interaction item info'* with none numeric reference. [644 rows]
- Sessions which have duplicated steps (same step number). [19 sessions]
- Sessions which have not impressions (at the clickout event). [12 sessions]

At the end of the preprocessing we got 826.823 sessions with 14.263.960 actions in total. This data we save to disc (in parquet format) so we only have to compute them once. The preprocessing is done in the JupyterNotebook *prepare sessions*.

5 BASELINE METHODS

In this section we list a few very simple baseline models and analyze how well they would perform on our training-dataset.

All of those models are implemented as a *Apache Spark Transformer* or *Estimator*. Therefore they can be put into *Apache Spark pipelines* and evaluated by our custom Trivago Evaluator.

Display-Order

Our first baseline just provides the given impressions-list as recommendation-list. With that we already obtain a MRR-Score of 0.508.

Interaction-Order

With this model we assume that the next clickout will be the item the user has last interacted with. So we rerank the items in the impressions-list that the last interacted item is in the first position, the second last interacted item is in the second position and so on. With that we compute a MRR-Score of 0.289.

Interaction-Frequency

This model sorts the impressions by the number of times the user has interacted with the item in the session in descending order. Here we get a score of 0.266.

Price

If we recommend the cheapest hotels first we get a MRR-Score of 0.249. We realized that this model performs significantly better in countries with lower average income (see figure 1. The boxplot indicates that the MRR of the "cheapest-first"-model is significantly higher in Chile, Ecuador and Romania compared to countries like Germany, UK and Austria.

Popularity

This model reorders the hotels in the impressions-list by their overall-clickout-popularity. Here we get a MRR-Score of 0.314. This model is different to the other as it requires a fit-method to learn the popularity of the items. Therefore the MRR-Score reported on the same training-set could overestimate the performance of this baseline. And indeed the crossvalidated MRR-Score is 0.311.

Matrix Factorization

To represent a conventional algorithm as a baseline we did an experiment outside of spark based on matrix factorization. Compared to the more simple user-user and item-item collaborative filtering techniques matrix factorization is usually

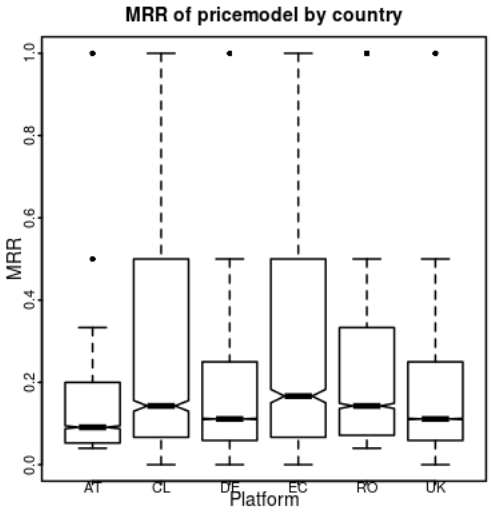

Figure 1: Pricemodel performance by platform-country

more effective because it allows discovering the latent features underlying the interactions between users and items. Moreover, we think that as the simplest model-based method it a great choice for a baseline.

To prepare the data for the model-based approach we performed a train-test split. But we did not shuffle the individual sets to keep the session together. Furthermore, we removed the last clickout reference of user/session combinations from the test set.

We treated sessions as users to leverage this technique for the sequence-based task. To do so, we created a sparse rating matrix R. Instead of the traditional user-item pairs and their respective rating, we used session-reference pairs and their respective weight by accumulating interactions within the session (simple interaction = 1, clickout-item = 2).

In order to be able to build the matrix factorization classes, we used one-hot-encoding for sessions (users) and references (items). Stochastic gradient descent was used to perform the training. We used an initial learning rate of 0.002.

Unfortunately, we experienced a sticky problem training our matrix factorization approach. After each iterated epoch the loss increased by approximately factor 10. Increasing the learning rate to 0.02 let the loss already explode at epoch 0. Setting the learning rate to 0.0002 kept the loss at the initial 189. Nevertheless, training our rather simple matrix factorization approach was lacking performance and took ages. Finally, we thought about altering the cost function used for SGD. Using a different cost function solved our problems. This simple change also increased overall performance to a reasonable level. Summed up, we concluded that matrix factorization is what it was meant to be - a baseline approach for session-based recommenders.

Content-Based Recommender

We did an experiment outside of spark where we loaded all data into Microsoft SQL-Servers column-storage-tables and created a semantic-search-fulltext-index on the metadata of each hotel. Our idea was to find the most similar hotel to the last hotel the user has interacted based on the hotels metadata. While we were successful in loading all the data into a database on our local notebook, we run into limitations of the semantic-full-text-feature. For performance reasons the software vendor restricted the output of the similarity function to 10 documents only. However these 10 hotels did not overlap with the hotels in the impressions list.We decided to abend on this approach. However we reused the created database in a subsequent experiment involving a proprietary data mining software (Microsoft Analysis Services) to create a sequence clustering model. More details can be found in the appendix.

6 NEIGHBORHOOD METHODS

Often one can provide good recommendations by constructing a neighborhood of similar users to the current user and then compute a score to weight the relevance of neighbour‘s items to the current user.

Model based Collaborative Filtering

Here we tried the Mllib standard-implementation of Matrix-Factorization. However the model did not work very well because we trained it with session-ids. This resulted in a severe cold-start-problem. The model did not find sessions similar to the new session resulting in the worst MRR-score of all models of 0.170.

7 SEQUENCE AWARE MODELS

We tried Sequence-Clustering (SC) as well as Recurrent Neural Networks (RNNs) to learn what will be the next best hotel to recommend. In the end we just used the RNN-model in the ensemble.

Sequence Clustering

This hybrid-model is a combination of clustering and markov-chain-analysis. The idea is to cluster all sessions based on some session-features such as device, platform, city of interest etc. In each cluster we just have a subset of all hotels. From the observed user interactions we can then learn the probability of a user who has interacted with hotel x to interact with hotel y next. Unfortunately even by partitioning the data into smaller clusters the number of possible state-transitions (between the hotels in each cluster) remains very large in comparison to the relatively small number of observed transitions in the training-set. Due to this sparsity issues we decided to discontinue this approach.

A detailed description of our experiments with this model and our ideas on how to make it work can be found in the appendix.

Deep Learning Model

Due to limitations of our Spark cluster we had to implement a clever work around: Training offline and import the data as CSV afterwards. More information can be read in the appendix B.

8 COMBINING MODELS

We integrated all the models described above into one framework. This has the advantage that different recommender models can be combined together to gain better performance.

Weighted Combination

All the recommender models emit a pairs of hotel id with some weight, where the weight are normalized (sum = 1). So a straight forward combination is to add up all the weights of the recommender models weighted by the importance of the recommender. This specially gives good result for all those recommenders which can not make a prediction in all situations (eg. interaction and deep learning model if there is not interaction with any hotel).

9 MACHINE LEARNING ON TOP

If there would be an oracle which could predict which recommender model should be used for which session, we would achieve a MRR score on the training set of 0.786. This insight lead to the idea of building a machine learning algorithm on top of all the recommender models.

First we extracted features from each session and made them available in a separate data set. This includes basic information like platform or device as well as counts of action items, last sorting order, time span and length (linear & log) of session.

We knew from exploratory data analysis that some features have significant discriminative power. For example the following chart shows that depending on the session-length we should pick the display-order model for very short sessions (figure 2).

Training ML models

Based on this features machine learning models (ML model) are trained. For each recommender model exactly one ML model is trained. The ML model is a binary classification model, which predicts 1 if and only if the target item id is ranked first by the recommender model, in all other cases the prediction is 0.

We use a Random Forest and hyper parameter tuning on the major parameters to obtain good models. Since we want to have models with high precision we implemented a custom

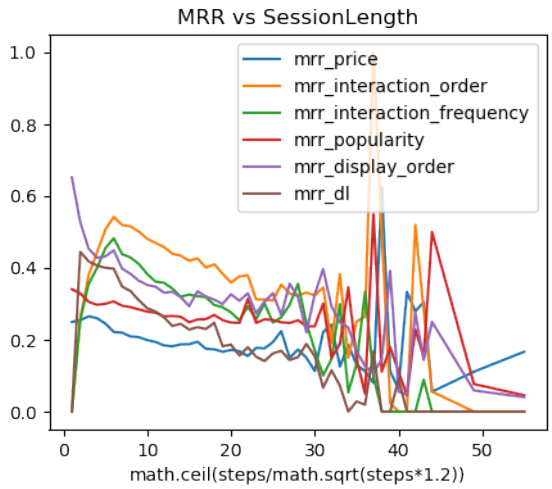

Figure 2: Model performance by session-length

Apache-Spark-Evaluator which calculates the precision for a binary classification model.

We also tried a linear SVM and gradient boosted trees. Neither of them result in better ML models. Moreover both of them have the disadvantage that Apache Spark does not reveal any probabilities of the predictions.

Applying ML models

The most intuitive and fastest way to select the final output, is to use the recommender model with the highest prediction probability. This approach gives a MRR score of 0.538 on the training data.

Another way is to define an order of the models and pick the first one where the ML model prediction is positive. The order of the models can either be hand crafted or is evaluated by brute force. Using 5 models gives 120 permutations, which can be evaluated within 12 seconds. The best scoring order is *'interaction_displayorder', 'dl', 'price', 'displayorder', 'popularity'* with a MRR score of 0.574 on the training set.

This approach can be improved if not the prediction, but the prediction probability is used. Running a brute force algorithm on the best 60 models determined above with different level for the threshold of prediction probabilities, the MRR score can be improved to 0.581 with a standard deviation of 0.001065 on the training set.

The best solution checks for models with at least 0.9 probability then of 0.8, 0.5 and 0.3. For each iteration the models are checked in the following order *'interaction_displayorder', 'displayorder', 'dl', 'popularity', 'price'*.

If this model is applied to the test data set, the MRR-score on the leader board is 0.607.

Model / Approach	Trivago Score (on Upload)
Ensemble with equal weighting of best 6 models	0.525
Deep Learning Model alone	0.536
Ensemble with weighting according to private Evaluation MRR	0.497
Models selected by machine learning model	0.607

Table 1: Scores on Test Set

Table Name	# Records	Data (KB)
Train	15,932,992	415,440
TrainImpressions	36,378,436	356,664
Metadata	927,142	275,480
TrainPrices	36,378,436	282,560
TestImpressions	12,145,848	112,384
Test	3,782,335	107,032
TestPrices	12,145,848	90,144

Table 2: We also loaded the trivago-dataset into SQL-Server Column-Store-Indexes to work with the entire dataset on a single developer-notebook.

10 RESULTS

Table 1 summarizes all our results for the individual models as well as for the combined ensemble model.

11 DISCLOSURE

Parts of this project are also handed in for the lecture *Recommender Systems* and *RecSys Team Project*. Moreover the team size for the other lectures have different sizes (for *Recommender System* 6 people, for *RecySys Team Project* just 2).

12 FINDINGS

We learned from our experiment with the matrix factorization outside of spark that it can be used as a baseline model for session-based recommends. In short, there are other options handling the task way better and improving overall accuracy and with it the general meaning of the recommendation.

Data Sets

We found that the datasets are rather small and the processing time was always very fast. Even running cross validation with different parameters did not take more than a couple of minutes because we did not tune many hyperparameters.

Due to the lack of recommender-algorithm-implementations in spark we also used external tools and frameworks such as pytorch, SQL-Server and Analysis Services to run some of our experiments. We managed to load the entire dataset into a relational SQL-Server database on our local notebooks within 15 Minutes using memory-based-hash-tables, where we also switched off the database log-file mechanism to avoid disk-bottlenecks. In a second step we compressed the data into a columnar storage format and dropped the hash-tables. Table 2 shows that we ended up with less than 2GB disk-space where we already normalized the impressions and price lists. However when we actually wanted to do something with the data we ran into severe scalability issues. For example it was difficult to denormalize the price and impressions columns into additional tables without running into database-file-disk-space problems. We also had issues running a sequence cluster model for the entire dataset. Instead we created views to limit the dataset to all sessions related to a single city before feeding the data into our local sequence clustering model in Analysis Services. Spark on the other hand scales well and deals with the entire dataset. However it just offers a tiny subset of distributed algorithms compared to the large number of advanced algorithms already implemented in non-distributed datamining software.

Moreover we hat the feeling that the data sets did not provide a lot of insight and already the sorting of the impression list was very good (MRR score approx 0.5).

Caching to HDD

Doing a lot of modification on a dataset can result in a huge execution plan for a data set. In this cases the model should be saved (cached) to hard drive after computations. This specially applies if multiple machine learning models are trained on a data set and the are also applied. In those cases it is useful to cache the results onto the hard disc between each model fit.

UDFs

We expected the UDFs to slow down the performance a lot. In our case this has not been the case. We reimplemented one two baseline recommender models in pure *Apache Spark*. Those implementations had approximately the same runtime. Implementing the transformer for combining different models turned out to be quite tricky. All the implementations we tried have so have been slower than the version using UDFs. Our conclusion is that using UDFs (specially during the development cycle) makes the code a lot more readable and maintable.

A SOURCE CODE

Since the Azure Databricks cluster was used for development the source code might need some minor adjustments for running on the TU cluster. This mainly involves data loading and displaying graphics. Unfortunately the Databricks cluster uses just 2 spaces for intend, where as Jupyter Notebooks generally use 4. This leads highlighting errors in the jupyter notebooks.

The data is once loaded by the the notebook */LoadData.py* as raw data to the cluster. All data analysis is collected together in the notebook *'/Data Analysis.ipynb'*, the data preprocessing as described in section 4 is performed in the notebook *'/prepare Sessions.ipynb'*.

All the implemented models in the Apache Spark framework can be found in the folder *'/models/'*. This folder also contains notebooks for tests as well as executions on the data sets. The training, evaluation and submission generation of the neural network can be found in the folder *'/nn/'*.

The machine learning mode (as described in section 9) is prepared, trained and evaluated in the 5 notebooks in the directory */model selection/*.

The performance analysis of Apache Spark code can be found in *'/Performance Analyze.ipynb'* and the evaluation of the performances and generation of graphics in this report is done in *'/Evaluation.ipynb'*.

B APPENDIX: DEEP LEARNING MODEL

This model tries to predict the clickout reference by applying a sequential deep neural network (DNN) architecture.

Goal of this approach is that the model extracts the features by itself, without sophisticated feature engineering. This would make it possible, that the same or a similar model may be applied for other use cases as well. Additionally a model like this might have applicability to other settings, which could solve the cold start problem of many recommender systems.

The main intuition is that a user will pick a similar item from what she/he has searched before.

For example: Whenever I search I have certain constraints in mind (e.g.: hotel must have a swimming pool and wifi). Therefore it will be more likely for me to pick a hotel which has these features. The model which is therefore learned tries to model this behaviour.

Architecture

First for each item a feature vector is constructed. This feature vector consists of the binary representation of the properties of the items (1 if it is present, 0 if it is not present). Each session is a sequence of these items represented by their feature vector.

The model consists of an RNN and a final regressor. The items (using their feature vectors) are fed into the RNN until the click out is reached, where finally the prediction is done. The prediction predicts for each element in the feature vector a weight between 0 and 1 which determines how much a user cares for a certain property.

After this âĂIJwants" vector of the user has been extracted the best match hotel is returned. This is done by picking the hotel with the highest cosine similarity among the impressions.

Many parts are inspired by âĂIJSession-based Recommendations with Recurrent Neural Networks" but using the feature vectors instead of a one-hot-encoding of the items.

Pre-Processing

The data for the deep learning model is filtered the following way:

- Each action_type which does not have a valid item ID is being filtered (e.g.: search for POI)
- Each session which does not end with a click out is filtered
- session_ids are encoded into integers, for performance reasons
- Action_typeś are one-hot-encoded
- Sessions with a length of 1 are filtered, due to the sequence based nature of the RNN based model, predictions for sessions like this will not work as intended
- Additionally each row with an invalid reference is removed

The metadata is then converted into a 202 dimensional vector where each item contains a binary vector where 1 denotes that the hotel has the property and 0 denotes the opposite. This results in a rather dense matrix describing each hotel.

Additionally the sessions are data augmented:

- Random slicing: The sessions are sliced into arbitrary slices of length 2 or more. A random start position and a random end position is picked and this is used as the new start/end positions of the temporary session
- Random reversing: The session is reversed with a certain chance.
- Random dropping: Random entries are dropped with a certain chance.

These measures combat overfitting very successfully and proved to be very useful tools.

Evaluation

Due to speed constraints evaluation is done by a simple holdout technique, where the train.csv is split again into a train set and a validation set, where the performance is measured. A cross validation would take too much resources.

Additionally since the train.csv is preprocessed in a way where certain invalid sessions are filtered the used validation set is biased anyway. This set should therefore only be used for comparative analysis to lead the implementation. A better result on the validation set still implies in general a better result on the final test set.

Experiments

Due to the high search space the goal was to probe as many solutions as possible in an efficient time frame. Then after a few attempts the most tempting approaches are explored more rigidly.

For this to work the experiments were done on much smaller subsets of the data. In the beginning only 10,000 lines of the CSV were used, then this was increased to 100,000 and 1,000,000 lines.

Baseline experiment. The baseline experiment was done just with the âĂIJwants" vector prediction based on the history. Important were the following aspects:

- BatchNormalization in the regression part of the final network, without it the performance dropped below 0.4 MRR
- Data Augmentation, without it the performance dropped below 0.4 MRR

The performance of this approach was 0.59 MRR on the validation set of the 10,000 lines CSV subset.

The following hyperparameter were tried:

- Batchsize (32, 64, 128, 256): Best 64
- Learning rate (0.1, 0.01, 0.001, 0.0001): Best 0.01
- Weight Decay: (0.001, 0.0001, 0.00001, 0.000001, 0.0): Best 0.0
- Wide (400 vs narrow (200) layer configuration: Narrow layer of 200 "decision" neurons
- Different layer counts (1, 2, 3, 4, 5): Best 1
- GRU vs LSTM: GRU (trained faster, but no significant impact on performance)
- Number of layers in GRU/LSTM: Both 1
- Dropout of 0.3, 0.2, 0.1, 0.0 after : Best 0.0 (Overfitting was not a problem)
- Batch Normalization vs No batch normalization: Batch Normalization proved to be very significant
- Different data augmentation techniques (random slicing, random reversing, random dropping): All. Random reversing improved result bei 0.05 MRR, without random slicing the performance was below 0.4 MRR.
- ReLU vs Leaky ReLU vs ReLU 6: Best ReLU
- Batch similarly length session together vs no batching: No performance impact, but batching by session length improved speed significantly
- MAE Loss vs Cosine Embedding Loss: Cosine Embedding performed by 0.02 MRR better compared to MAE Loss

The overall result on the test.csv was 0.51 MRR. Note that, because some sessions are dropped (too short, invalid reference, etc.) there are some sessions missing in the final submission. If this is the case, the entries from the baseline are used.

TFIDF Weighting Experiment. Another experiment tried was weighting the input feature vectors of the hotels by their Inverse Document Frequency. This made the results worse (below 0.4 MRR).

For another experiment the IDF weighting was applied on the predicted wants vector (and the item feature vectors). This seemed to cause minor improvements of around 0.01 MRR.

Add more features to Input Vector. Adding the following features to the input vector were tried:

- One-hot-encoding of the action type: No improvement
- Standard scaled (by sklearn) of the time between the current and the last action: No improvement
- Add a normalized âĂIJstep" field: Little improvement of 0.02 MRR on the validation set

Add Embedding. Another experiment was by adding embedding layers to the neural network. The thought was, that they might help the neural network to transform the input vectors into more meaningful data. The

This proved to be not the case. The following configurations were tried:

- Fully Connected Linear Layers (With/Without ReLU)
- Apply a convolution over the input data
  - Different sequences at once (2, 3, 5)
  - Different number of feature maps (100, 200, 300)
  - Different final pooling (Max Pooling, Average Pooling)

Ranking. This experiment tried to remove the cosine similarity nearest neighbor match, since it does not incorporate the fact that the user is much more likely to select the first impressions in comparison to the last impressions.

The first solution was a pairwise ranking approach. This is loosely based on RankNet, where two items are compared and the network should predict the rank. The general architecture was:

For a pair of items:

- Linear (202) => BatchNorm => ReLU => Linear(128) => BatchNorm => ReLU => Linear(32) => BatchNorm => ReLU => Linear(8)

They were compared using a fully connected linear classifier where the goal was to predict their respective similarity.

One experiment tried to achieve this by a binary assignment for the target label

Another experiment tried to achieve this by their respective ranking in the impressions list for the target label

Sadly incorporating this ranking based algorithm did not improve the MRR.

Different configurations were tried: More layers, less layers, different learning rate, different weight decay, different batch sizes, etc.

The next approach can be seen as a generalization two 25 ranks. It consisted of two steps:

For each (impression, reference) pairs:

- Linear (202*2 + 2) => BatchNorm => ReLU => Linear(202) => BatchNorm => ReLU => Linear(128) => BatchNorm => ReLU => Linear(16)
- 202*2 + 2 consists of
- ... 202 features of the wants vector,
- ... 202 features of the impression vector,
- ... 1 for the cosine similarity,
- ... 1 for the price of the impression.

This results in a 16*25 sized vector.

Which is then decided by a fully connected layer:

- Linear(16 * 25) => BatchNorm => ReLU => Linear(25)

So the input where the (impression, wants) vector pairs, the output is the predicted index. The index is encoded using a one-hot-encoding.

This approach actually improves the performance significantly. It boosts the MRR from 0.59 to 0.64 on the validation set.

Seq2Seq. The baseline approach was predicting the wants vector and the loss was computed only by the last hidden state. The loss was computed by the mean squared error between the last wants vector and the last reference.

This experiment tried to improve the training process by comparing the wants vector after each hidden state with the real reference. For each of these âĂIJtemporary" wants vectors the MSE loss was computed.

After trying different configurations:

- Weight decay: 0.001, 0.0001, 0.00001, 0.0
- Dropout: 0.5, 0.3, 0.0
- Number of layers in decision part: 1, 2, 5
- Number of hidden units per layer: 100, 200, 300

None of these improved the results, but made the results in general worse. Additionally due to the more complicated loss calculation training took significantly longer (10 minutes per epoch instead of 5, using a Nvidia GTX 1080).

Collapse rows. To make it easier for the RNN to memorize pattern multiple same references are collapsed into one with the normalized count. This âĂIJcompresses" the input sequence significantly. For example the following sequence is transformed:

- A A B C A A A A

Is transformed into

- A (2/8) B (1/8) C (1/8)A (4/8)

This step improved MRR by 0.02 on the validation set.

Design Decisions

Why no one hot encoding of items? There are two reasons: The first being, that there are many hotels, which would result in huge matrices. As the support of sparse matrices is not yet matured in PyTorch it was not possible to use a sparse matrix. This would mean some kind of dimensionality reduction must be applied, which could cause the performance to drop

Another reason is that it can be seen to be more âĂIJbeautiful" if the model learns the needs of the users, as that would make it theoretically possible to apply transfer learning. The needs of the user change less, than the product IDs

Why Deep Learning? Deep Learning has made a significant leap forward in the field of artificial intelligence. As this has not yet happened in the field of recommender systems it is interesting to try to challenge this.

Why Ranking Network? The cosine similarity is kind of arbitrary and the idea of deep learning in general is to do as much as possible without handcrafting. As it actually improves the performance, the idea seems to be correct.

Integration with Spark

The main issue we had to solve, was how to get the DL model running on Spark where we have no GPU available. We achieved this by offline training on one of our PCs which has a dedicated GPU.

Where it makes sense data preprocessing is done using Spark. Which is then exported to Parquet files, which can easily be loaded into Python. There some additional preprocessing (data augmentation and filtering) is done, if necessary.

After training the model, the predictions for the test/train data is generated and exported as a CSV file. These are then uploaded to Spark where they can be easily loaded. A dummy model which just looks for the session id inside the uploaded data returns the predictions. This allows for easy integration into Spark, without performance issues, while still integrating quite well into our pipeline. The only issue is, that we have no 100% clean train/test split during validation of the final models, which causes some skews towards the deep learning model in the reported performance metrics. With additional work this could be solved by ensuring, that both

the "online" Spark models, as well as the "offline" deep learning model are trained using the same split.

Improving the DL model

The current item feature vectors only include the itemś metadata, but this could be extended to a more elaborate feature vector, either through feature engineering (e.g.: average price of item) or collaborative filtering.

Additionally one could build a similar architecture for the user history instead, but this would probably not lead to much improvement, since only a small percentage of sessions actually has user id which is not unique.

Also the search history is ignored, which could also provide useful information to the model.

Using an encoder/decoder or transformer architecture could improve the wants vector prediction significantly, similar to how they improved various NLP tasks significantly.