Chapter 8

Polara: a new open-source framework for recommender systems research

One of the critical aspects of creating new recommender models is their evaluation and fair comparison. Many software libraries have been developed to date; however, none of them are concerned with the idea of feedback polarity as described in Chap. 5, and the related implementation aspects of proper model evaluation are not taken into consideration in their system design. This has led the author of this work to the development of a new framework called *Polara* – the first recommendation framework that allows a more in-depth analysis of recommender systems' performance, based on the idea of feedback polarity.

Polara, however, is not just an evaluation library and it is not limited to polarity-driven evaluation paradigm. From the very beginning, it was envisioned as a general purpose framework for quick model prototyping and comprehensive comparative analysis, featuring various evaluation regimes and testing scenarios. The framework's design and its internal structure aim to minimize the risk of unintended mistakes in routine tasks, reduce the number of potential bugs in the code and ensure consistent experimental settings. All of this allows pursuing another higher-level goal – *research reproducibility*.

The framework has also become a convenient playground for students and helped facilitate teaching in classes. For those who have just started learning about recommender systems, it makes the learning experience generally more smooth. For more advanced students it allows to focus on creative tasks. Curious minds, however, can always make a deep dive into the code and see how everything works internally.

Polara provides exceptional flexibility in both model creation and experiment setup. At the same time, it is designed to follow *recommender system for humans* paradigm, providing a high-level abstraction of general workflows with a significant focus on the usability and the ease of use.

To achieve these goals, Polara is written in Python programming language[1] – de facto, the leading platform for data science and machine learning[2]. The framework supports both Python 2 and Python 3 versions. In addition to that Polara is boosted by the Python's scientific computing ecosystem, which helps to ensure efficient operations not only in model computations but also during the evaluation phase. The latter at first glance might seem like a minor point, however in many cases evaluation takes a much longer time than actual model training, which affects the way experiments are conducted. Polara avoids running experiments user by user and, where it is possible, takes advantage of highly optimized vector operations and parallel execution to reduce an overall experiment time.

Another essential feature of Polara is the possibility to easily extend its default set of models with the help of external libraries and frameworks. This allows conducting more rigorous research that requires comparison with various existing techniques. Implementing all of them in Polara from scratch would be a tedious task, and it would be hard to keep up with the most recent advances. Instead, Polara defines a clear protocol for such interoperability and implements many convenience methods that make this process transparent and straightforward.

Table 8.1 provides a brief comparison of Polara with some popular frameworks. Besides some basic characteristics, we assess additional aspects related to functionality and usability. For example, *Customizable evaluation* column indicates whether the framework supports and allows to chose from several evaluation scenarios, which includes various data splitting protocols, data sampling strategies and flexible configuration of experimental settings. The *Warm start regime* denotes the support of a new user/new item scenario, which includes appropriate data preprocessing and/or explicit implementation of folding-in for the provided models. The names of other columns are self-descriptive.

[1] https://www.python.org/

[2] https://www.kdnuggets.com/2017/09/python-vs-r-data-science-machine-learning.html

Table 8.1: Comparison with popular recommendation frameworks.

Framework	Languages	Polarity-based evaluation	Tensor models	API for custom models	Customizable evaluation	Warm start regime	Cold start regime	Side information support	Rich set of algorithms	Regularly updated**	License
Polara [125]	Python	✓	✓	✓	✓	✓	✓	✓		✓	MIT
Mrec [106]	Python							✓			BSD
Surpise [164]	Python			✓	✓			✓		✓	BSD-3 Clause
MyMediaLite* [107]	C# / Java			✓							GNU GPL
Turi / GraphLab* [175]	C++ / Python							✓		✓	Apache 2.0
Implicit* [83]	Python					✓				✓	MIT
RankSys [129]	Java			✓				✓			MPL 2.0
LensKit [97]	Java			✓	✓	✓	✓	✓			LGPL v2.1
LibRec [100]	Java		✓	✓	✓	✓	✓	✓	✓	✓	GNU GPL
RecommenderLab [131]	R			✓		✓				✓	GNU GPL v.2

* Supported as external models in Polara.

** Last checked: September 2018.

8.1 Core components

The framework has a modular structure and is built on top of three key components, *Recommender Data*, *Recommender Model* and *Evaluation*, which consist of basic classes and standalone methods. The components are designed to support a general workflow and take care of many technical aspects, related to the stable and reliable functioning of the framework as a whole. There is also much flexibility included in these components, allowing for a high degree of customization.

The general workflow is based on the following paradigm. An instance of *Recommender Data*, holding actual user-item interactions, provides a single entry point for all *Recommender Model* instances. Recommender Data instance has a mutable state, i.e., a specific configuration, corresponding to the desired experimental setup. In turn, Recommender Model instances, i.e., actual algorithmic implementations, take the data model instance as an input argument and depend on its state. In that sense, all depen-

dent recommender instances are *subscribers* that are immediately notified on the data state changes and take appropriate actions on their side.

For example, changing train-test splitting configuration in a data model instance will lead to recomputation of a dependent recommender model instance at the very next attempt of using it (e.g., when calling for recommendations or trying to evaluate the model's performance). Alternatively, changing the number of holdout items in the test data will leave the recommender model intact; however, will flush previously calculated recommendations and will ensure that evaluation scores are refreshed at next calculation. Worth noting, the subscriber interface is exposed to a user, and it is possible to define custom actions that are executed in response to certain state changes. This mechanism can be especially useful in non-standard user-defined experiments with specific evaluation pipelines.

Overall, an interplay of the described components allows to freely experiment with various evaluation settings and be sure that all changes in experimentation setup are taken into account by recommender models without any additional actions needed from the user side. This also minimizes the amount of code, required to conduct experiments. Below are the key implementation details of each component, also demonstrating the ease of use of the framework.

8.2 Recommender Data

Recommender Data component is the central part of the framework, implemented as a standalone class with pre-defined properties and methods. It provides a rich interface with a number of tuning parameters that opens up a great level of flexibility in experiment design and ensures consistent data state across all compared models.

The component is designed in a *data-agnostic* way. As an input it takes a history of transactions in the form of a Pandas[3] dataframe, which is internally transformed into a standardized representation, allowing for efficient data manipulation and quick conversion between internal and external representations. It only requires to define, which columns of the dataframe correspond to users, items, and feedback data. As

[3]https://pandas.pydata.org/

an example, the code presented on the listing below allows to start working with the Movielens-1M dataset.

Listing 8.1: Declaring data model.

```
from polara import RecommenderData
from polara import get_movielens_data
data = get_movielens_data() # load data from Grouplens website
data_model = RecommenderData(data, "userid", "movieid", "rating")
```

The component also implements various methods for data preprocessing, data splitting and data indexing. All configurable parameters for the data manipulation and their current values can be listed with the help of `data_model.get_configuration()` call. These parameters include `test_fold` to control the fold selection in the CV experiment, `test_ratio` to define the fraction of users for test, `holdout_size` to control the number of held out items, `warm_start` to exclude test users from the training, and some other parameters that control data randomization and sampling mechanisms.

One can easily achieve almost any data configuration by assigning the appropriate values to the aforementioned parameters. For example, standard 5-fold CV experiment with 20% of users marked for test and a single top-rated holdout item per each test user can be implemented with the following configuration setup: `data_model.test_ratio = 0.2`, `data_model.holdout_size = 1`. Alternatively, in order to hold out 5% of all consumed items from *every* available user, one needs to assign `data_model.test_ratio = 0`, `data_model.holdout_size = 0.05` and `data_model.warm_start = False`[4].

Configuration parameters are wrapped with "lazy update" routines in order to prevent early triggering of subscriber notification calls and avoid multiple execution of the same commands. Configuration is applied only after calling the `data_model.prepare()` method. An attempt to read an altered configuration before calling this method will result in a warning message, informing the user that some of the changes are not yet effective.

The system of state change notifications is based on a variant of the *Observer design pattern* and uses callback functionality for communication. Several default events trigger notifications. These events correspond to modification of parameters related to either training or test data. Notification processing for the default events is imple-

[4]More examples can be found at https://github.com/Evfro/polara/tree/master/examples

mented on the recommender model's side and does not require any user involvement. It is also possible to register custom events with notifications, handled by custom user routines if the standard functionality is not sufficient for the user.

8.3 Recommender Model

Recommender Model component provides a generic interface for creating new models ready for recommendations' generation and evaluation. As with the previous component, it holds some common properties and methods that are designed to support a unified workflow independently of specific implementation details. There are several standard recommender models already implemented for user convenience, including the models based on matrix and tensor factorization. The default models are subclassed directly from the abstract base class called `RecommenderModel`. New models can be subclassed either from the base class or from already defined models to inherit some of their unique properties and extend upon them.

When creating custom models, there are two primary methods that should be implemented prior to the usage: `build` and `get_recommendations`. The former computes a recommendation model, and the latter takes its result to generate recommendations for the test users. All generated recommendations are stored as an array within the model and can be used to evaluate the model's performance or asses its behaviour for further fine-tuning.

There are several control parameters shared by all models. Among them: `filter_seen` attribute defines whether the previously consumed items are allowed to be recommended again; `topk` attribute determines the number of recommendations generated by the model; `feedack_threshold` attribute defines whether only the feedback above a certain threshold value (e.g., only ratings above 4) should be used for computing the model. These parameters are all automatically initialized with some default values when a model is created (so that users do not have to set them every time manually) and can be redefined later. The default values can be found in the `polara.recommender.defaults` module. Listing 8.2 below demonstrates an example of creating a simple SVD-based model with Polara.

Listing 8.2: Define a simple SVD-based model.

```
1 from polara import RecommenderModel
```

```
from scipy.sparse.linalg import svds # for sparse matrices

class SimpleSVD(RecommenderModel):
    def __init__(self, data_model):
        super(SimpleSVD, self).__init__(data_model)
        self.rank = 40 # set the rank of SVD
        self.method = "SVD" # label that will be used in logging

    def build(self):
        # get sparse matrix of ratings
        train_matrix = self.get_training_matrix(dtype="f8")
        # find leading left singular vectors with truncated SVD
        _, _, items_factors = svds(train_matrix, k=self.rank,
                                   return_singular_vectors="vh")
        # remember the result
        self.items_factors = items_factors

    def get_recommendations(self):
        # gather test data
        test_data, test_shape, _ = self._get_test_data()
        # construct sparse rating matrix for all test users
        test_matrix, test_idx = self.get_test_matrix(test_data,
                                                     test_shape)
        # compute predicted scores for all test users at once
        v = self.items_factors
        svd_scores = (test_matrix.dot(v.T)).dot(v)
        if self.filter_seen:
            # prevent seen items from appearing in recommendations
            self.downvote_seen_items(svd_scores, test_idx)
        # compute recommendations
        top_recs = self.get_topk_elements(svd_scores)
        return top_recs
```

The newly created model can now be used in a general workflow. It only takes a few lines of code and the commands for that are self-explanatory, as illustrated below.

Listing 8.3: Create and evaluate the model.

```
svd = SimpleSVD(data_model)
svd.build()
svd.evaluate("relevance")
```

```
4 # output will be something like
5 # Relevance(precision=0.0994, recall=0.3314...
```

We note that defining the `build` method is entirely on the user's responsibility and depends only on the choice of a particular algorithmic implementation. In turn, the `get_recommendations` method provides two options. The first option is to manually implement all of its internal logic similarly to the example in Listing 8.2. In that case, the user is responsible for making the code efficient in terms of both computational resources and available memory. This can be a reasonable option when data is small, and the framework is used for learning purposes.

However, in order to deal with real data, special care must be taken on the process of recommendations generation. If the number of items as well as the number of test users is huge, intermediate calculation results may consume all available memory (e.g., line 27 of Listing 8.2, where a complete dense matrix is created). Moreover, as the memory I/O is generally slower than CPU operations, even if there is enough memory it is more efficient to limit its consumption by the model and expose memory resources in pieces of a fixed size.

In order to achieve that the default implementation of the `get_recommendations` method in the base class splits the test data into chunks. Every chunk will include a number of unique test users, typically more than one. This relies on the assumption that computing recommendations for a group of users at once is much more efficient than looping over every user individually. This is the case in a number of scenarios (e.g., standard scenario of recommending to the known users) and for a wide range of algorithms, including SVD, MF, TF, etc., as it may take advantage of BLAS operations.

In order to operate over the chunks of data, one only needs to declare a `slice_recommendations` method, which is by default the key part of the `get_recommendations` method. The code in Listing 8.4 indicates the necessary changes in the new model creation to activate this functionality.

Listing 8.4: More efficient variant of defining a model.

```
1 class SimpleSVD(RecommenderModel):
2     def __init__(self, data_model):
3         # same as before
4 
5     def build(self):
6         # same as before
```

```
7
8       def slice_recommendations(self, test_data, test_shape,
9                                 start, stop, test_users=None):
10          test_slice = (start, stop) # selecting users from a range
11          test_matrix, slice_data = self.get_test_matrix(test_data,
12                                                         test_shape,
13                                                         test_slice)
14          v = self.items_factors
15          scores = (test_matrix.dot(v)).dot(v.T)
16          return scores, slice_data
```

The `slice_recommendations` method here operates on a group of users, selected by an index range. The predicted scores are computed only for these users and then are returned to the `get_recommendations` method to generate final recommendations. The method also returns index data to allow filtering out previously seen items from recommendations. Note, that there's no need to define `get_recommendations` anymore and the code becomes slightly simpler.

The size of chunks (i.e., the number of test users in it) is controlled by a predefined memory limit, which can be set via the `MEMORY_HARD_LIMIT` attribute from the `polara.recommender.utils` module. Its optimal value depends on the hardware capabilities and should be determined empirically. It may range from one gigabyte to several dozens of gigabytes. Setting it to lower values will lead to a computational overhead with many small iterations, while too high values (if there's enough available memory) are unlikely to improve performance due to I/O bounds.

The I/O bound, however, can be alleviated with parallel execution. When data is large, I/O operations like reading the data of a group of test users may take more time than actual computations. Such operations typically lock Python's Global Interpreter Lock. In order to mitigate that limitation, the `slice_recommendations` method can be executed in parallel threads. This behavior is controlled by the `max_test_workers` parameter of a recommender model. Setting it to a non-zero value defines the number of parallel threads. The maximum amount of memory consumed by a model during the recommendations generation can be estimated as `MEMORY_HARD_LIMIT * max_test_workers` gigabytes.

Note that in the examples above only one model is created. However, as has been previously mentioned, several recommender models can share the same `data_model` in order to conduct bulk experiments with fair model comparison.

Polara can be easily extended with the help of external libraries and frameworks. It uses the concept of *wrapper* – an interface between internal methods and external sources. The general process of creating new wrappers is no different from creating new models within the framework and requires minimum efforts. By default, Polara already implements several wrappers for the well-known software tools, which extends the list of supported algorithms. This includes *MyMediaLite* [107], *GraphLab Create* [175] and *implicit* [83].

8.4 Evaluation

Unlike the previous components, the evaluation component is not a particular class but rather is a set of convenience methods, designed to support various evaluation scenarios in a unified way. The major focus of evaluation is shifted towards the relevance of recommendations and the quality of recommendation ranking. There are several standard evaluation metrics supported by this component, namely Precision, Recall, HR, MRR, nDCG, nDCL and a number of others.

Two key features distinguish this component from evaluation components in other recommendation libraries and frameworks. The first one is a native control over the false positive rate estimation. As described in Sec. 5.3.1, recommender models may recommend items that have no user feedback (this happens very often, in fact). Treating them as false positives in some cases leads to an undesired fp rate overestimation and spoils the precision-recall curve.

Polara allows users to assign more appropriate weighting in this case via the `not_rated_penalty` argument of the `model.evaluate()` method. Setting its value to 1 will force the evaluation process to count recommendations with unknown user feedback as false positives while setting it to 0 will filter out such recommendations from the final score calculation. Values between 0 and 1 will lead to a "smooth" fp rate estimation.

The second feature is a native support for the positivity threshold, also described in Sec. 5.3.1. As has been demonstrated in the results of Chap. 5, taking into account

the performance of models with respect to both positive and negative aspects of recommendations plays a crucial role in an understanding of the overall quality of recommendations. Hence, every recommender model is provided with the `switch_positive` trigger, which allows defining, what values of feedback should be treated as positive or negative examples when evaluating recommendation quality. This trigger not only affects how metrics are computed but also allows to calculate the nDCL score, introduced in Sec. 5.3.2. It also defines which recommendations are counted as false positive.

The technical implementation of the component relies on a sparse data representation and bulk computations without loops. This not only improves computational efficiency but also allows to conduct a more in-depth analysis of model performance, going beyond aggregated evaluation and in some cases helping to create a better picture of model behavior. One particular method worth mentioning in this regard is `assemble_scoring_matrices` from the `polara.recommender.evaluation` module. It takes generated recommendations and holdout data as an input and returns various indicators of correct and incorrect recommendations in the form of sparse matrices.

8.5 Supported scenarios and setups

As a multi-purpose evaluation framework, Polara provides the necessary instruments and controls for various setups that cover all major evaluation scenarios. There are three main experiment setups, supported natively by Polara. The *standard evaluation* scenario allows test users to be a part of the training data and only the items from holdout set remain unknown until the evaluation phase. In the *warm start* scenario the test users are also hidden from the training phase. During the evaluation phase, their known preferences are used to generate recommendations, which are then evaluated against the holdout items. Finally, the *cold start* scenario is represented by a separate `polara.recommender.coldstart` module, which currently provides item cold start functionality. The module additionally provides a few methods to manipulate content information to support cold start regime.

Polara supports both implicit and explicit feedback, independently of whether it is represented by rating values, binary data, frequency counts or other data formats.

Note that categorical feedback naturally fits the tensor-based representation and can also be handled within Polara.

In addition to standard data splitting methods, Polara also supports custom fields that can be used to order elements and split data. The simplest example is the timestamp data. Assuming there is an additional column named "timestamp" in the original pandas dataframe, the following modification of the Recommender Data constructor allows to take this information into account:

```
RecommenderData(data, "userid", "itemid", custom_order="timestamp"),
```

where for illustration purposes we also omit the `feedback` field to demonstrate how to handle purely implicit positive-only data.

An important part of the general evaluation framework is the ability to set custom test data, provided externally (e.g. in some online recommender system challenge). For example, if one is provided with some external holdout data, which is not a part of the training data, however contains only known users, the following setup allows to seamlessly work the data:

Listing 8.5: Preparing data model for experiments with custom holdout.

```
data_model = RecommenderData(data, "userid", "itemid", "feedback")
data_model.prepare_training_only() # do not attempt to split data
data_model.set_test_data(holdout=external_holdout,
                         warm_start=False)
```

It should be noted that by default Polara will reindex `external_holdout` data to conform with the internal data representation. This behavior can be disabled by providing `reindex = False` argument into the `set_test_data` method.

Fine-tuning of many recommendation models is not as simple as the tuning of SVD and often requires an extensive hyper-parameter search. Current implementation of Polara provides basic functionality for the random grid search, which can be accessed via the `random_grid` method from the `polara.evaluation.pipelines` module. This functionality will be extended in future versions and include customizable all-in-one pipelines.

As a final example of the framework functionality, the listing below demonstrates how to conduct a top-n recommendation experiment for several models in bulk with a few lines of code in the current version of the framework:

Listing 8.6: Example of cross-validation experiment for evaluating several models

```
from polara import PopularityModel
from polara import RandomModel
from polara.evaluation import evaluation_engine as ee

svd = SVDModel(data_model)
popular = PopularityModel(data_model)
random = RandomModel(data_model)

models = [svd, popular, random]
metrics = ["ranking", "relevance"]
topk_values = [1, 5, 10, 20, 50] # number of recommendations

topk_result = {}
for fold in [1, 2, 3, 4, 5] :
    data_model.test_fold = fold
    topk_result[fold] = ee.topk_test(models, topk_values, metrics)
```

This will store the result of all models' evaluation for all 5 folds in the `topk_result` variable, which can be further used to perform comparative analysis and report on findings.

8.6 Summary

In this chapter, we have described the key design aspects and demonstrated the main functionality of the Polara framework. It takes care of the most of the data processing and data handling hassles, providing a thin, abstract layer for the user with a rich set of controls. The framework also provides a number of convenient and flexible software tools for quick prototyping of recommender models and performing a comprehensive evaluation. Apart from the boilerplate functionality, Polara also supports several external frameworks and libraries, allowing to incorporate their models into the general workflow. Internally, the framework uses various tweaks and controls in order to perform operations efficiently and wisely consume system resources. The framework is suitable for both beginners and advanced users; it can be used in classes for teaching or as a part of a daily research.