Recommender Systems Course, Algorithm Porting Project A.Y. 2024-25*

Team Member 1[1] , **Team Member 2**[1]
[1]Politecnico di Milano, Student
todo@mail.polimi.it

1 Project purpose

The purpose of this project is the porting of recently published Recommender Systems algorithms in a standard framework we use. Reasonable programming skills in Python are required. The project will require to obtain the original implementation provided by the authors of the selected articles, then to write a *Wrapper* class around it in in order for it to be compatible with the BaseRecommender interface and reproduce the original experimental results. Each team of two will be assigned two or three algorithms, if an algorithm cannot be successfully ported another one may be assigned to replace it depending on the amount of work previously done. This document contains the relevant documentation and examples.

1.1 Delivery requirements

The delivery of this project will consist of a zip file whose name must be *"TeamMember1 TeamMember2.zip"* containing:

- A *team members.txt* file with the name, surname, matricola, PoliMi email of the two team members
- A *[algorithm_name]_github* folder containing the clone of the original repository provided by the authors
- A *[algorithm_name]_our_interface* folder containing the recommender wrapper plus all other scripts necessary and a sub-folder for each dataset with the corresponding data reader
- A *[algorithm_name]_Report.pdf* file for each algorithm using the template provided in Section **??**.

1.2 Source code requirements

The project involves writing both a Wrapper around the original model as well as a *DataReader* object for each of the datasets used in the experiments. The two should allow to run the experimental pipeline an example of which is in the provided source code (*run_CIKM_18_ExampleAlgorithm.py*). As a general criteria, whenever the original source contains a function that already does something you need (model instantiation, data parsing...), do not re-write it or copy it, but rather call that function. If necessary applying some pre and post processing to adapt the data input or output. If the function requires small changes, for example due to the different format of the input data, copy it and alter it as needed. You should *copy, move or modify the least amount possible of the original code*, ideally almost no changes should be made and the Wrapper should operate by calling the original functions.[1] [2] The code you write or modify will be thoroughly checked and compared to the original implementation, due to the **very delicate** nature of this type of work. In the DataReader the original reading functions should be used and then the data transformed into a sparse URM and/or ICM matrices. In a typical scenario you have that the DataReader uses the original reading function that creates the data structure the model needs, but you will also need to transform the data into a sparse matrix to ensure compatibility with the framework. If the model requires as input a different data structure you have two scenarios: (1) if the model uses simple data structures your wrapper must take as input only the sparse URM/ICM and any data conversion must be hidden within the wrapper, (2) if the model uses very complex data structures, for example graphs that use content-based entities which cannot be reconstructed easily from the sparse URM/ICM, you can write the wrapper so that it requires both the sparse (for framework compatibility) and model-specific (for simplicity of porting) formats but the two formats must be consistent.

*This is an internal document of the assignments for the Recommender Systems Course at Politecnico di Milano, and should not be distributed or shared for purposes other than this course activities.

[1]If the model is implemented in tensorflow, pytorch or keras you likely will be able to import the original model into a rather simple Recommender class (see an example in Recommenders/MatrixFactorization/PyTorchMF.py).

[2]Sometimes the authors provide multiple "official" implementations, prefer those written in pytorch to those in tensorflow because they are simpler and more flexible. Check with the teaching assistant.

Validation and test data The ported source code must not be provided with the validation or test data in any way (for example, in the constructor). If the original source code takes as input the test data this may mean it is used during the training process, which could invalidate the results reported in the paper. Instead of providing the real test/validation matrices, set them to None. If you see errors are raised check where the data is being used. If it is used actively during the train process, contact the teaching assistant via email. Otherwise, just comment-out the code in question, but do not delete it.

Evaluation Often the original source code evaluates the model during the training phase, the ported version will decouple the training and evaluation phases so that the evaluation will be done by the Evaluator object in the framework that will rely on the _compute_item_score function of the ported recommender. You can comment-out the parts of the original implementation that evaluate during training.

Overall, the source code will need to meet the following requirements:

- You should provide an implementation of the methods described in Appendix **??**, **??** and **??** but not the BaseTempFolder class described in **??**.
- The code should meet the interface specifications as per Appendix **??**, **??**, **??** and **??** and should pass the tests done in script *Test_BaseRecommender.py*.
- The code should be consistent with the example provided and meet the previous requirements.

1.3 Report requirements

Section **??** contains the template for the report, you should write one per algorithm. There are to types of notes:

- *[Fill this: something]*: Replace them providing the required information and description
- **[Check this: Yes/No]**: Answer the question choosing one of the options. Sometimes it may be required to provide additional information according to the answer. Leave the answer in the blue/bold format.

1.4 Project Evaluation criteria

The project will be considered successful if the following conditions are met:

- The DataReader and Wrapper implementations can be understood with reasonable effort and are consistent with the interface and the requirements. The original implementation has not been altered in substantial ways.
- The Wrapper model can be saved and loaded, what should be saved is the model itself, not the predictions.
- The Wrapper implementation passes the tests provided in script *Test/run_unittest_recommenders.py*.
- The DataReader and Wrapper implementation allows to achieve results consistent with the ones reported in the original article, provided that the original implementation allows to do the same.
- The report is complete, truthful and consistent with the other material provided
- The deadline has been met

1.5 Some annoying problems you may encounter

- On Windows when you pass the parameters to the Keras/Tensorflow model you may get the exception *Unsupported feed type*. Check if in the input data you have a numpy array with integer values, if so add ".astype(np.int32)".
- Sometimes you need to save sessions or other complex objects native of Keras or Tensorflow. You can save them in a subfolder and then compress it, in order to put all the components of a model in a single compressed file (an example is in the Neural/MultVAERecommender file). Alternatively, if you can extract arrays of weights or other parameters, you can put them in a dictionary and save them with the provided DataIO class, then you can re-create the original data structure when you do the loading. Sometimes saving/loading the state of a model is tricky and may produce erroneous results, make sure to check the loaded version is consistent with the saved one.
- When you use a Saver to save the state of a tensorflow model during training, be careful because the Saver also deletes old files. For this reason we recommend that a new Saver object is created when you do a load/save operation in the load_model and save_model functions to avoid accidentally removing files.

2 Publication title: *[Fill this: Insert article title]*

2.1 Introduction and brief description

In [?] *[Fill this: insert citation with reference in BibTex format in the references.bib file]*, published at *[Fill this: insert conference name and year]*, the authors propose *[Fill this: insert brief description of the algorithm, paraphrasing what stated in the abstract and introduction of the original paper (10-20 rows). Focus on the idea, avoid comments on how successful the model is supposed to be or how great the original authors think it is. If reporting a comment made in the paper, use* "in the original paper it is claimed that ...", *for example do not say "the algorithm allows to model non linear feature correlations outperforming the state of the art", instead say "the paper claims this algorithm is able to model non linear feature correlations". Do not report the model formula or figures of its architecture.]*

Introduction checklist:

- The model performs top-N item recommendations to a user: **[Check this: Yes/No, if No or you are not sure contact teaching assistant]**
- The model relies on sequential patterns or recurrent neural networks: **[Check this: Yes, if yes or you are not sure contact teaching assistant/No]**
- The model relies on non-structured input data such as images or text: **[Check this: Yes, if yes or you are not sure contact teaching assistant/No]**

2.2 Datasets:

The evaluation is performed on the following datasets, their statistics are reported in Table **??**:

- *[Fill this: Insert dataset name]*: The dataset contains *[Fill this: insert in 2-3 lines a description of its content and value range. For example: the dataset contains the number of user-shop checkpoints][Fill this: mention any preprocessing applied. For example, only users with more than 10 checkpoints are kept.]*[3]
- ...

Dataset	Interactions	Items	Users	Density
[Fill this: name]	-	-	-	$- \cdot 10^1$
[Fill this: name]	-	-	-	$- \cdot 10^1$
[Fill this: name]	-	-	-	$- \cdot 10^1$

Table 1: Dataset Statistics

Datasets checklist:

- At least one of the datasets is publicly available for download: **[Check this: Yes/No, if No contact the teaching assistant]**
- The characteristics (size and density) of the datasets you are using for the experiments (i.e., not those in the dataset description online, but those loaded by the DataReader considering any preprocessing that may be required) are consistent with the ones reported in the original article: **[Check this: Yes/No, if No state which are the differences. If None of the datasets has the same statistics contact the teaching assistant]**

2.3 Evaluation protocol:

The data splitting is performed in the following way:

[Fill this: Describe splitting method: holdout picking a certain quota of the interactions, holdout performed for each user, leave-one-out, leave-last-out etc...]

[Fill this: Mention how the validation data is split. If it is not described, state that it is not described.]

[Fill this: Mention if the evaluation is performed ranking a positive (test) item and a fixed number of negative items.]

The evaluation is performed reporting:

[Fill this: list the metric@cutoff used in the evaluation, for example Recall and MAP at cutoff 20, 30, 50.]

[3] *[Fill this: Insert link to the dataset if available]* todo

Evaluation protocol checklist:

- The original train-test split of the data is available on the Github repository? **[Check this: Yes/No, state for which datasets]**
- [If the evaluation requires it] The original negative item split used for the evaluation phase (not for training) is available on the Github repository? **[Check this: Yes/No/Not using this evaluation method]**
- Does the paper mention a validation set? **[Check this: Yes/No]**
- Does the paper state that the evaluation is performed in the same way (data, metrics) as another article? **[Check this: Yes, if so insert citation of that article/No]**

2.4 Baseline algorithms checklist:

- Is TopPopular among the baselines? **[Check this: Yes/No]**
- Are ItemKNN or UserKNN with cosine similarity among the baselines? **[Check this: Yes/No]**.
- [If KNNs are among the baselines] Does the paper state if they include a shrink term? **[Check this: Yes/No/No KNNs among baselines]**.

2.5 Hyperparameter tuning:

The hyperparameters are tuned using *[Fill this: insert tuning strategy (grid search, random search, Bayesian search... as well as the optimized metric, or "not described")]*. The values of the hyperparameters for the proposed method are described in Table **??**.

Hyperparameter	Described in	Value All datasets
[Fill this: name]	*[Fill this: Paper / source code]* [a]	- [b]
[Fill this: name]	*[Fill this: Paper / source code]*	-
[Fill this: name]	*[Fill this: Paper / source code]*	-

[a]If it is mentioned in the paper, state paper, otherwise source code
[b]If different hyperparameters are reported for different datasets, create a column for each dataset

Table 2: Hyperparameter Values

Hyperparameter tuning checklist:

- Does the paper mention the metric-cutoff optimised during hyperparameter tuning? **[Check this: Yes, if so state which/No]**
- Does the paper report the hyperparameter value range used during the tuning for the proposed method? **[Check this: Yes/No]**
- Does the paper report the hyperparameter value range used during the tuning for the baseline methods? **[Check this: Yes/No]**
- Does the paper state that the hyperparameters of a baseline are the *default* ones or are taken from another article? **[Check this: Yes, if so state which/No]**
- Does the paper mention how the optimal number of epochs is selected? **[Check this: Yes, if so describe it/No]**
- Does the paper state that the value of one of the hyperparameters is fixed for all or some of the baselines?[4] **[Check this: Yes, if so state which/No]**

2.6 Source code:

The source code is publicly available[5]. Table **??** reports the results contained in the original paper, the results obtained running the source code as provided by the authors (with the hyperparameters reported in Table **??**) and lastly the results obtained with the ported version of the algorithm when evaluated with the Evaluator object of the framework.[6]

[4]Examples of hyperparameters that may be fixed across different models are the number of latent factors for matrix factorization models, the architecture or size of the encoding layer in autoeconders etc... Sometimes it is stated this is done to ensure a *"fair"* comparison, while in reality it is the opposite. When present this almost certainly means the evaluation reported in the paper contains a fundamental error.

[5]*[Fill this: add link to the public repository]* todo

[6]If the training time of the algorithm is very long and you are not able to use your own hardware or free resources e.g., Colab, contact the teaching assistant.

Dataset	Result from	Metric@20	...
	Paper	- [a]	-
[Fill this: name]	Original source	-	-
	Ported source	-	-
	ItemKNN cosine [b]	-	-
	Paper	-	-
[Fill this: name]	Original source	-	-
	Ported source	-	-
	ItemKNN cosine	-	-

[a]For the *Paper* results just copy the results reported in the original article
[b]Use the baseline optimization code provided in run_CIKM_18_ExampleAlgorithm.py

Table 3: Results obtained by the algorithm on the various datasets and metrics

Source code checklist:

- Is the original source code executable with at most minor changes (e.g., fixing imports or downloading data)? **[Check this: Yes/No, if so mention what the problem is]**
- Is the original source code, with the correct hyperparameters, able to reproduce the results reported in the paper? **[Check this: Yes/No. Write you result for one metric in terms of its percentage of the original result (for example Recall@20 98%, or 127%...)]**
- Does the ported version of the original algorithm pass the test suite provided in Test/run_unittest_recommenders.py? **[Check this: Yes/No]**
- Is the ported version of the original algorithm, with the correct hyperparameters, able to reproduce the results obtained by the original source code? **[Check this: Yes/No. Write you result for one metric in terms of its percentage of the original result (for example Recall@20 98%, or 127%...)]**
- Is the source code delivered compliant with the requirements in Section **??**? **[Check this: Yes/No, if so describe how]**
- In the original source code is the test data used during the training in any way (for example to select the number of epochs or to select the negative samples...)? **[Check this: Yes, if so mention where and provide a reference to the code/No]**

A Source code documentation: BaseRecommender

Each RecommenderWrapper should inherit from the BaseRecommender class, which provides the following abstract methods:

- `fit`
- `_compute_item_score`
- `save_model`
- `load_model`

A.1 `fit`

This method has the purpose of fitting a recommender model, for example if the model is a KNN based on a certain similarity, then the fit function will compute all the item-item similarity matrix. If the recommender is a machine learning model, then the fit will contain the training loop.

Signature: `fit(custom_args, temp_file_folder=None, **earlystopping_kwargs)`
Parameters:

- **`custom_args`** any hyperparameter the algorithm needs, prefer keyword arguments
- **`temp_file_folder`** To be added only if the Wrapper needs to save files in a temporary folder (i.e., saving the best model during early-stopping)
- **`**earlystopping_kwargs`** the arguments to be provided to the early-stopping training function.

Returns:

- **None** No return value is required

A.2 `_compute_item_score`

This function has the purpose of computing the user-item scores, either in terms of predicted ratings or in terms of another scoring function. Items with the highest scores will be recommended. If you have an ItemKNN model this function simply computes the dot product between the user profile and the item similarity matrix, if you have a matrix factorization algorithm this function computes the product between item factors and user factors. If the algorithm you are working on belongs to either of these categories then you can use the implementation already available in the respective classes of methods (your class should inherit from BaseMatrixFactorizationRecommender or from one of the BaseSimilarityRecommender).

Signature: `_compute_item_score(user_id_array, items_to_compute = None)`
Parameters:

- **`user_id_array`** an array of any length that will contain the IDs of the users for which the scores should be computed
- **`items_to_compute`** an array of any length that will contain the IDs of the items we want to compute the scores of. If None, the scores will be computed for all items.

Returns:

- **`item_scores`** an array of shape (len(user_id_array), total_n_items) containing the user-item scores. The only admitted values are: any real number, -numpy.inf (to be used if the score of that user-item cannot or should not be computed). If items_to_compute is None, the scores of all items must be filled. If items_to_compute is not None only the scores of the selected items must be filled. The scores of items that are not selected will be associated with a -np.inf value. If the model does not allow to specify which items to score, a simple workaround is to instantiate an array of -np.inf values, call the original recommendation model prediction function and only copy the predictions for the items in the items_to_compute argument.

A.3 `save_model`

This function has the purpose of saving the current model state to allow its re-use. The following two sequences must work and produce the same identical results: create model instance, fit model, save model; create model instance, load model. This function must save all hyperparameters relevant for the state of the model in a dictionary with the DataIO class, as per example provided.

Signature: `save_model(folder_path, file_name = None)`
Parameters:

- **`folder_path`** the folder in which you want to save the model
- **`file_name`** the root name all save files will have. If the function requires to save multiple files, all will have file_name as prefix. If file_name is None the content of the attribute RECOMMENDER_NAME is used as prefix.

Returns:

- **None** No return value is required

Important Notice: If you save a model using Keras Saver, or other savers, consider that they may be developed to perform periodic saves during the training of the model in a transparent way, therefore they may update your saves or remove old ones without you noticing. Always save the model with a Saver instance used only to that purpose, within the save function. If you find that during the training of models requiring many epochs sometimes the "best_model" saved by the early-stopping disappears, then you are not using an independent saver in the save_model function and the best_model was automatically removed at some point during the training.

A.4 `load_model`

This function has the purpose of loading a previously saved model. The following two sequences must work and produce the same identical results: create model instance, fit model, save model; create model instance, load model. This function must load all hyperparameters relevant for the state of the model from a dictionary using the DataIO class, as per example provided.

Signature: `load_model(folder_path, file_name = None)`

Parameters:

- **`folder_path`** the folder from which you want to load the model
- **`file_name`** the root name all save files will have. If the function requires to load multiple files, all will have file_name as prefix. If file_name is None the content of the attribute RECOMMENDER_NAME is used as prefix.

Returns:

- **None** No return value is required

B Source code documentation: Incremental_Training_Early_Stopping

If a model is trained for a certain number of epochs it needs a way to select the optimal number of epochs and you should apply early-stopping. In such case the Wrapper should inherit from the Incremental_Training_Early_Stopping class.

The class Incremental_Training_Early_Stopping class provides the following abstract methods:

- `_run_epoch`
- `_prepare_model_for_validation`
- `_update_best_model`

The behaviour of this class and its usage are exemplified with the code below. A new instance of the recommender model is created and a clone is made and referred to as the initial "best model". Then the early-stopping can begin. Function `_train_with_early_stopping` takes care of handling the process and calls three functions which you will need to implement for the specific algorithm you are porting. The model is trained for a certain number of epochs with the `_run_epoch` method. After that the model needs to be evaluated, the function `_prepare_model_for_validation` is called and then the Recommender object is passed to an Evaluator object. After the evaluation is complete, if the current model status is better than the previously found "best model", then the "best model" is updated calling the `_update_best_model` function. After the training is complete, the "best model" must be loaded again. At a higher level, if the model uses Tensorflow, the fit function should look as in:

```
self._model_instance = get_model_instance(...)

# Set the initial "Best model"
self._update_best_model()

self._train_with_early_stopping(epochs_max=self.epochs,
                                algorithm_name=self.RECOMMENDER_NAME,
                                **earlystopping_kwargs)

# close session tensorflow and open another
self.sess.close()
self.sess = tf.Session()

# Load the final "Best model"
self.load_model(self.temp_file_folder, file_name="_best_model")

# Remove the temp directory where the "best model" had been saved
self._clean_temp_folder(temp_file_folder=self.temp_file_folder)

self._print("Training complete")
```

B.1 `_run_epoch`

This function trains the model for **one epoch**. It commonly contains a loop over a certain batch of data or the users. It should contain a copy of the train loop written by the original authors.

Signature: `_run_epoch(num_epoch)`

Parameters:

- **`num_epoch`** The number of the current epoch

Returns:

- **None** No return value is required

B.2 `_prepare_model_for_validation`

This function is called right before the model is evaluated. Commonly this function does nothing at all and only contains the `pass` command. It can be useful if the model is not available in a format that allows to compute the recommendations and a parsing is required, for example if the model is developed in a non-Python language and it is saved as a file. Since parsing the file into a python data structure may be expensive, instead of doing it for every epoch it can be done in this function only when required.

Signature: `_prepare_model_for_validation()`

Parameters:

- No parameter is present

Returns:

- **None** No return value is required

B.3 _update_best_model

This function is called when the current status of the model provides better recommendation quality than the previously best one and therefore the "best model" must be updated. This function only needs to create a persistent copy of the best model. The persistent copy should be done via a deep copy, to ensure that the model that will be trained and the "best model" will be disconnected. If the model is more complex, a basic but sufficient implementation simply saves the current model in a temporary folder in the following way

```
def _update_best_model(self):
    self.save_model(self.temp_file_folder, file_name="_best_model")
```

Signature: `_update_best_model()`
Parameters:

- No parameter is present

Returns:

- **None** No return value is required

C Source code documentation: BaseTempFolder

If a Wrapper needs to save a temporary file it should inherit from the BaseTempFolder class, which provides the following methods:

C.1 `_get_unique_temp_folder`

This function returns a unique folder path, to be used every time something needs to be saved. This function should be called at the beginning of the fit function.

Signature: `self._get_unique_temp_folder(input_temp_file_folder = None)`

Parameters:

- **`input_temp_file_folder`** The file path requested or None. If None is provided, a path based on the recommender name is used.

Returns:

- **`unique_temp_folder`** A folder guaranteed to be unique and not already existent.

C.2 `_clean_temp_folder`

This function deletes the temp folder previously created, it should be called at the very end of the fit function.

Signature: `self._clean_temp_folder(temp_file_folder = self.temp_file_folder)`

Parameters:

- **`temp_file_folder`** The temp folder to remove

Returns:

- **None** No return value is required

D Source code documentation: DataReader

The DataReader task is to read the data of a specific dataset and load it into a dictionary of sparse matrices. You should implement the constructor in order to fill the class attributes.

D.1 `__init__(pre_splitted_path, **kwargs)`

The constructor performs all the operation required, it contains two parts: first it tries to load a previously saved version of the data, if that is not available it parses the original data applying any preprocessing required.

Signature: `__init__(pre_splitted_path, **kwargs)`

Parameters:

- **`pre_splitted_path`** The folder where the experiments are being made, the data files will be saved in a further subfolder called "data_split/".

Returns:

- **None** No return value is required

D.2 Class attributes

- **`DATASET_NAME`** A string with the dataset name (e.g., "Movielens1M")
- **`URM_DICT`** A dictionary containing the name of the URM and the corresponding object. For example:

```
self.URM_DICT = {
    "URM_train": URM_train,
    "URM_test": URM_test,
    "URM_validation": URM_validation,
    "URM_test_negative": URM_test_negative,
    "URM_timestamp": URM_timestamp,
}
```

- **`ICM_DICT`** A dictionary containing the name of the ICM and the corresponding object, similarly to URM_DICT.