Alpenglow Documentation

Release 0.1

MTA SZTAKI

Jun 24, 2021

GETTING STARTED:

CHAPTER

ONE

INTRODUCTION

Welcome to Alpenglow introduction!

Alpenglow is an open source recommender systems research framework, aimed at providing tools for rapid prototyping and evaluation of algorithms for streaming recommendation tasks.

The framework is composed of a large number of components written in C++ and a thin python API for combining them into reusable experiments, thus enabling ease of use and fast execution at the same time. The framework also provides a number of preconfigured experiments in the *alpenglow.experiments* package and various tools for evaluation, hyperparameter search, etc.

1.1 Requirements

Anaconda environment with Python >= 3.5

1.2 Installing

```
conda install -c conda-forge alpenglow
```

In case you also intend to run sample code and tutorials, you should install matplotlib as well:

```
conda install matplotlib
```

If you encounter any conflict or error, try installing Alpenglow in a clean conda environment.

1.3 Installing from source on Linux

```
cd Alpenglow
conda install libgcc sip
conda install -c conda-forge eigen
pip install .
```

1.4 Development

- For faster recompilation, use `export CC="ccache cc"`
- To enable compilation on 4 threads for example, use `echo 4 > .parallel`
- Reinstall modified version using `pip install --upgrade --force-reinstall --no-deps .`
- To build and use in the current folder, `use pip install --upgrade --force-reinstall --no-deps -e .` and `export PYTHONPATH="$(pwd)/python:$PYTHONPATH"`

CHAPTER

TWO

EXAMPLE USAGE

```
from alpenglow.experiments import FactorExperiment
from alpenglow.evaluation import DcgScore
import pandas as pd
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

data = pd.read_csv("http://info.ilab.sztaki.hu/~fbobee/alpenglow/alpenglow_sample_dataset
↪")

factor_model_experiment = FactorExperiment(
    top_k=100,
    seed=254938879,
    dimension=10,
    learning_rate=0.14,
    negative_rate=100
)
fac_rankings = factor_model_experiment.run(data, verbose=True)
fac_rankings['dcg'] = DcgScore(fac_rankings)
fac_rankings['dcg'].groupby((fac_rankings['time']-fac_rankings['time'].min())//86400).
↪mean().plot()
plt.savefig("factor.png")
```

CHAPTER

THREE

FIVE MINUTE TUTORIAL

In this tutorial we are going to learn the basic concepts of using Alpenglow by evaluating various baseline models on real world data.

3.1 The data

We will use the dataset at http://info.ilab.sztaki.hu/~fbobee/alpenglow/alpenglow_sample_dataset. This is a processed version of the 30M dataset, where we

- only keep users above a certain activity threshold
- only keep the first events of listening sessions
- recode the items so they represent artists instead of tracks

Let's start by importing standard packages and Alpenglow; and then reading the csv file using pandas. To avoid waiting too much for the experiments to complete, we limit the amount of records read to 200000.

```
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import alpenglow as ag

data = pd.read_csv('http://info.ilab.sztaki.hu/~fbobee/alpenglow/alpenglow_sample_dataset
↪', nrows=200000)
print(data.columns)
```

Output:

```
Index(['time', 'user', 'item', 'score', 'eval', 'category'], dtype='object')
```

To run online experiments, you will need time-series data of user-item interactions in similar format to the above. The only required columns are the `'user'` and `'item'` columns – the rest will be autofilled if missing. The most important columns are the following:

- **time**: integer, the timestamp of the record. Controls various things, like evaluation timeframes or batch learning epochs. Defaults to `range(0,len(data))` if missing.
- **user**: integer, the user the activity belongs to. This column is required.
- **item**: integer, the item the activity belongs to. This column is required.
- **score**: double, the score corresponding to the given record. This could be for example the rating of the item in the case of explicit recommendation. Defaults to constant 1.

- **eval**: boolean, whether to run ranking-evaluation on the record. Defaults to constant `True`.

3.2 Our first model

Let's start by evaluating a very basic model on the dataset, the popularity model. To do this, we need to import the preconfigured experiment from the package `alpenglow.experimens`.

```
from alpenglow.experiments import PopularityExperiment
```

When creating an instance of the experiment, we can provide various configuration options and parameters.

```
pop_experiment = PopularityExperiment(
    top_k=100, # we are going to evaluate on top 100 ranking lists
    seed=12345, # for reproducibility, we provide a random seed
)
```

You can see the list of the available options of online experiments in the documentation of *alpenglow.OnlineExperiment* and the parameters of this particular experiment in the documentation of the specific implementation (in this case *alpenglow.experiments.PopularityExperiment*) or, failing that, in the source code of the given class.

Running the experiment on the data is as simple as calling `run(data)`. Multiple options can be provided at this point, for a full list, refer to the documentation of *alpenglow.OnlineExperiment.OnlineExperiment.run()*.

```
result = pop_experiment.run(data, verbose=True) #this might take a while
```

The `run()` method first builds the experiment out of C++ components according to the given parameters, then processes the data, training on it and evaluating the model at the same time. The returned object is a `pandas.DataFrame` object, which contains various information regarding the results of the experiment:

```
print(result.columns)
```

Output:

```
Index(['time', 'score', 'user', 'item', 'prediction', 'rank'], dtype='object')
```

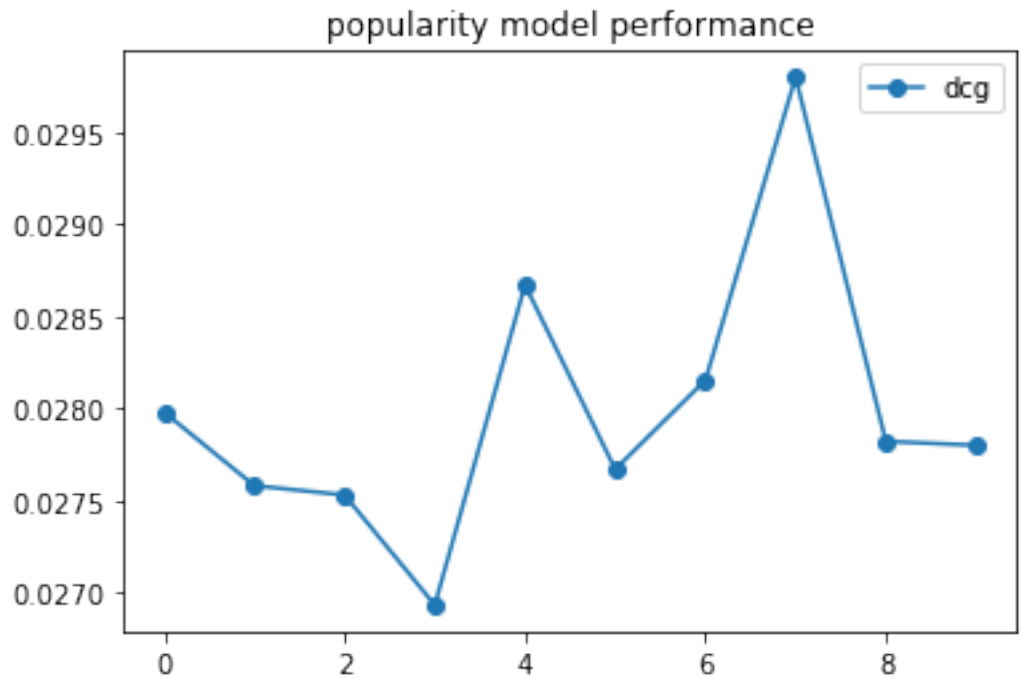
Prediction is the score estimate given by the model and rank is the rank of the item in the toplist generated by the model. If the item is not on the toplist, rank is `NaN`.

The easiest way interpret the results is by using a predefined evaluator, for example *alpenglow.evaluation.DcgScore*:

```
from alpenglow.evaluation import DcgScore
result['dcg'] = DcgScore(result)
```

The `DcgScore` class calculates the NDCG values for the given ranks and returns a `pandas.Series` object. This can be averaged and plotted easily to visualize the performance of the recommender model.

```
daily_avg_dcg = result['dcg'].groupby((result['time']-result['time'].min())//86400).
↪mean()
plt.plot(daily_avg_dcg,"o-", label="popularity")
plt.title('popularity model performance')
plt.legend()
```



Putting it all together:

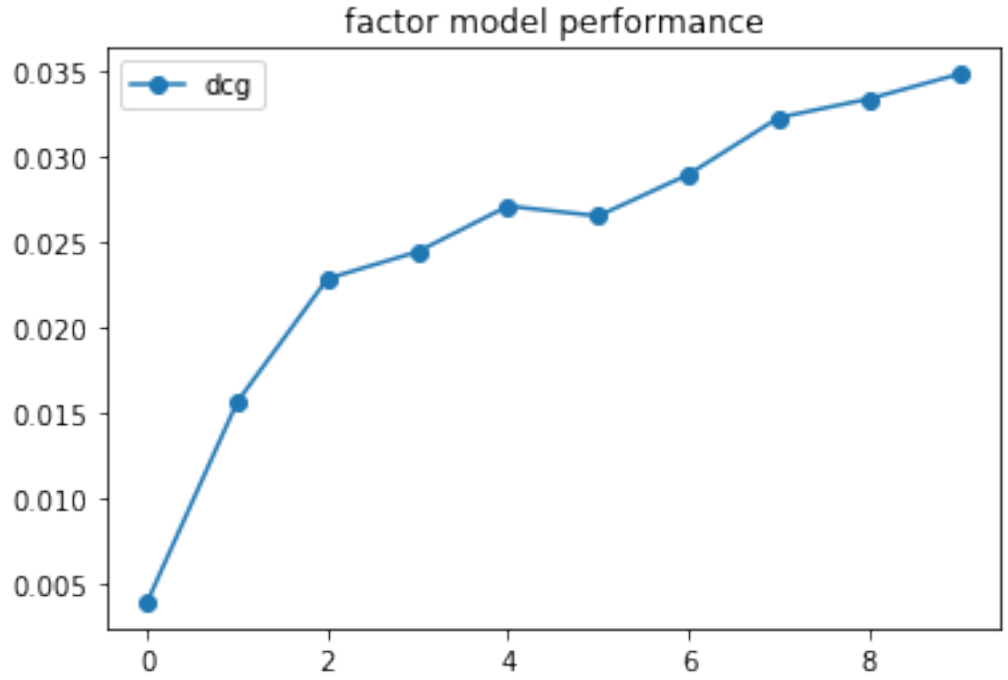
```
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
from alpenglow.evaluation import DcgScore
from alpenglow.experiments import PopularityExperiment

data = pd.read_csv('http://info.ilab.sztaki.hu/~fbobee/alpenglow/alpenglow_sample_dataset
↪', nrows=200000)

pop_experiment = PopularityExperiment(
    top_k=100,
    seed=12345,
)
results = pop_experiment.run(data, verbose=True)
results['dcg'] = DcgScore(results)
daily_avg_dcg = results['dcg'].groupby((results['time']-results['time'].min())//86400).
↪mean()

plt.plot(daily_avg_dcg,"o-", label="popularity")
plt.title('popularity model performance')
plt.legend()
```

3.3 Matrix factorization, hyperparameter search

The *alpenglow.experiments.FactorExperiment* class implements a factor model, which is updated in an online fashion. After checking the documentation / source, we can see that the most relevant hyperparameters for this model are `dimension` (the number of latent factors), `learning_rate`, `negative_rate` and `regularization_rate`. For this experiment, we are leaving the factor dimension at the default value of 10, and we don't need regularization, so we'll leave it at its default (0) as well. We will find the best negative rate and learning rate using grid search.

We can run the `FactorModelExperiment` similarly to the popularity model:

```
from alpenglow.experiments import FactorExperiment

mf_experiment = FactorExperiment(
    top_k=100,
)
mf_results = mf_experiment.run(data, verbose=True)
mf_results['dcg'] = DcgScore(mf_results)
mf_daily_avg = mf_results['dcg'].groupby((mf_results['time']-mf_results['time'].min())//
↪86400).mean()

plt.plot(mf_daily_avg,"o-", label="factorization")
plt.title('factor model performance')
plt.legend()
```



The default parameters are chosen to perform generally well. However, the best choice always depends on the task at hand. To find the best values for this particular dataset, we can use Alpenglow's built in multithreaded hyperparameter search tool: `alpenglow.ThreadedParameterSearch`.

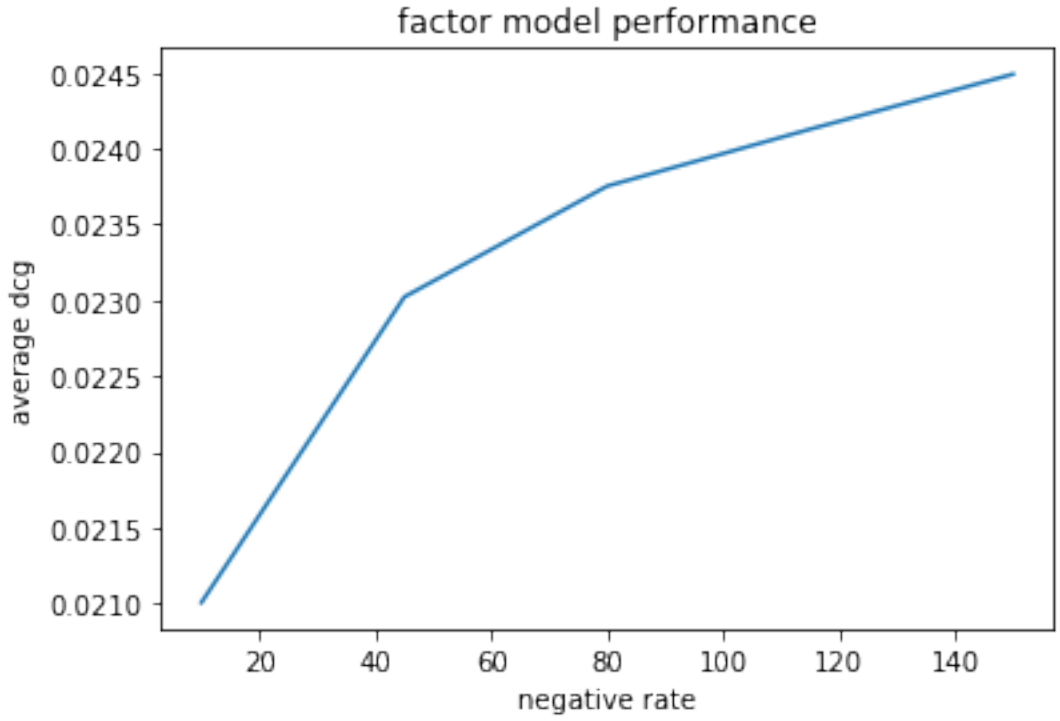
```
mf_parameter_search = ag.utils.ThreadedParameterSearch(mf_experiment, DcgScore,␣
↪threads=4)
mf_parameter_search.set_parameter_values('negative_rate', np.linspace(10, 100, 4))
```

The `ThreadedParameterSearch` instance wraps around an `OnlineExperiment` instance. With each call to the function `set_parameter_values`, we can set a new dimension for the grid search, which runs the experiments in parallel accoring to the given `threads` parameter. We can start the hyperparameter search similar to the experiment itself: by calling `run()`.

```
neg_rate_scores = mf_parameter_search.run(data, verbose=False)
```

The result of the search is a pandas DataFrame, with columns representing the given parameters and the score itself.

```
plt.plot(neg_rate_scores['negative_rate'], neg_rate_scores['DcgScore'])
plt.ylabel('average dcg')
plt.xlabel('negative rate')
plt.title('factor model performance')
```



CHAPTER

FOUR

FURTHER READING

If you want to get familiar with Alpenglow quickly, we collected a list of resources for you to read.

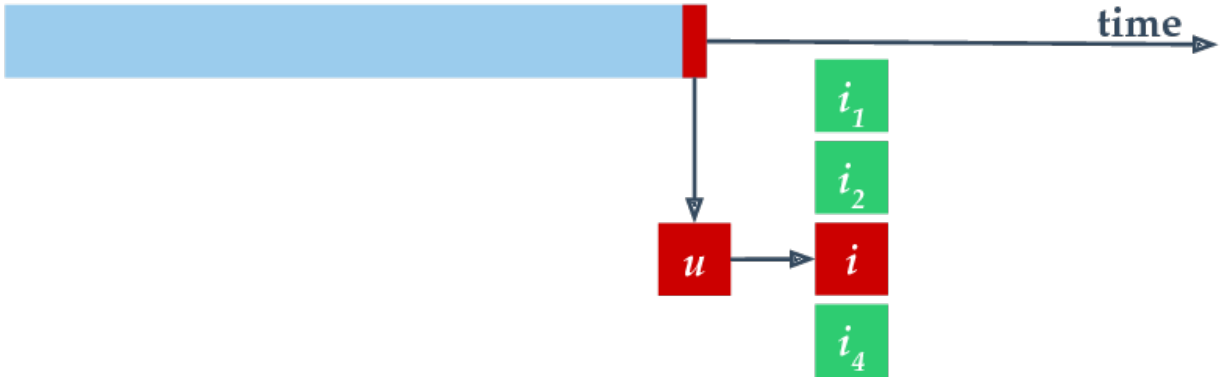
1. The documentation of *alpenglow.OnlineExperiment*. This describes basic information about running online experiments with alpenglow, and the parameters that are shared between all implementations.
2. The documentation of implemented experiments in the `alpenglow.experimens` package, which briefly describe the algorithms themselves and their parameters.
3. The documentation of *alpenglow.offline.OfflineModel*, which describes how to use Alpenglow for traditional, scikit-learn style machine learning.
4. The documentation of implemented offline models in the *alpenglow.offline.models* package.
5. Any pages from the the General section of this documentation

CHAPTER

FIVE

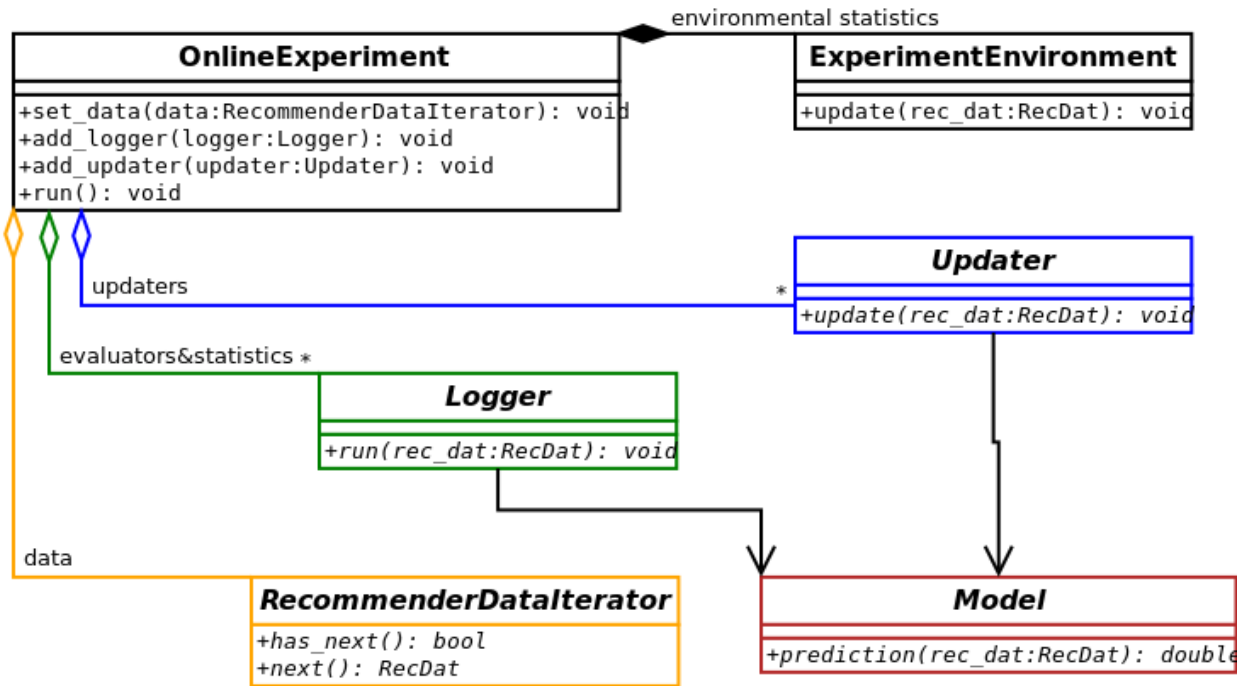
THE ANATOMY OF AN ONLINE EXPERIMENT

5.1 General structure of the online experiment



The online experiment runs on a time series of samples, each containing a user-item pair. We treat the time series as a stream, performing two steps for each sample. First, we evaluate the recommender, using the sample as an evaluation sample. One possible evaluation method is to query a toplist for the user (without revealing the item), check if the correct item is included and to compute the rank of the correct item. Second, after the evaluation, we append the sample to the end of the available training data and allow the recommender model to update itself. Normally, we perform an incremental update step using only the newest item.

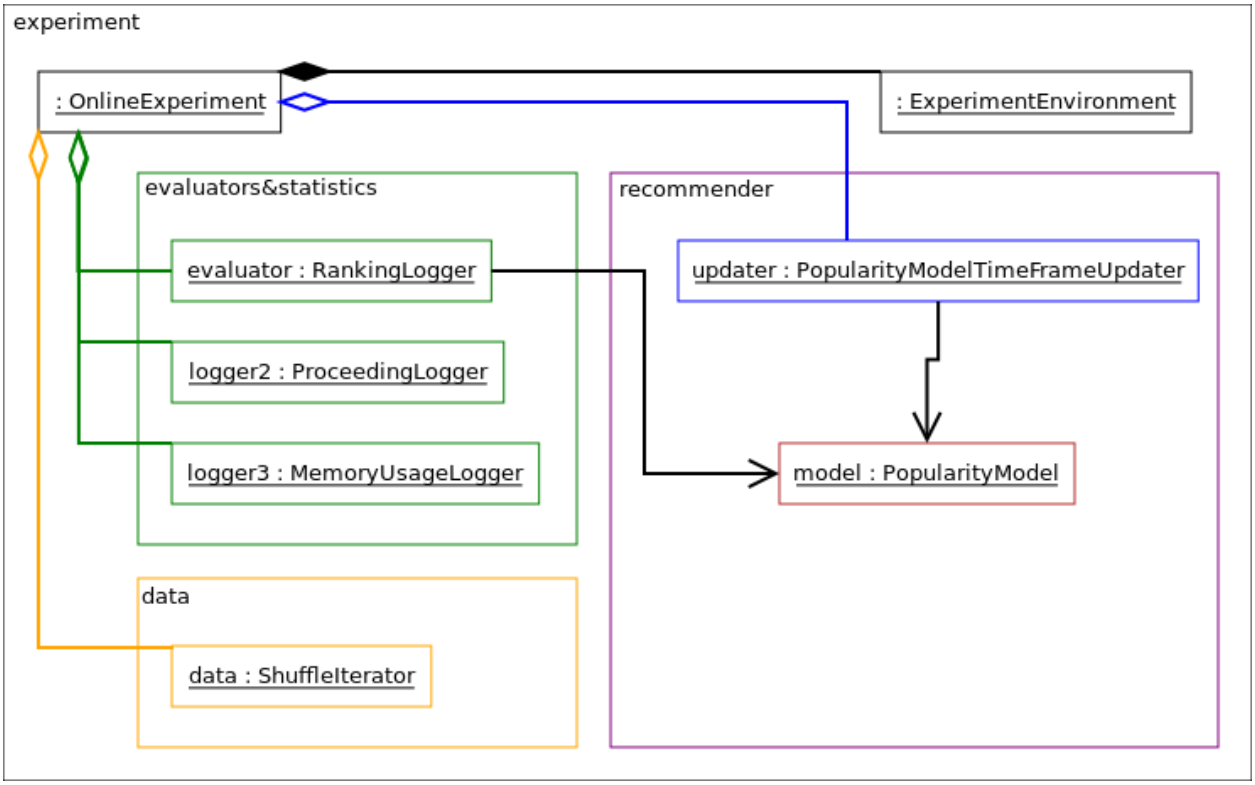
In our implementation, the central class that manages the process described above is *alpenglow.cpp.OnlineExperiment*. The data, the evaluators and the training algorithms are set into this class, using the appropriate function. They have to implement the appropriate interfaces, as depicted on the UML class diagram.



Read more about the interfaces in *C++ API*.

5.2 An example: time frame based popularity model experiment

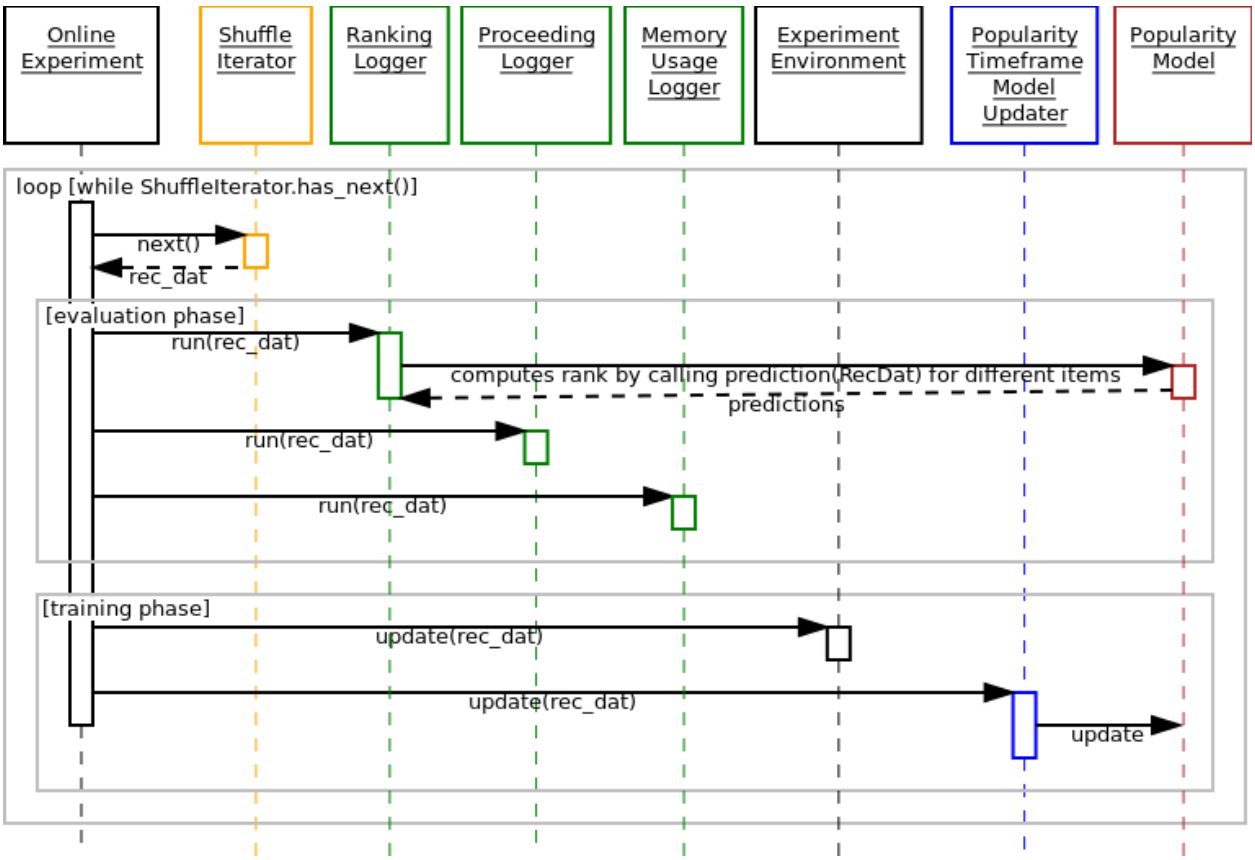
Consider a time frame based popularity model experiment for example. Below you can see the object diagram of this experiment.



This experiment contains

- a *alpenglow.cpp.OnlineExperiment* that is the central class of the experiment,
- a *alpenglow.cpp.ShuffleIterator* that contains the time series of the data,
- a *alpenglow.cpp.ExperimentEnvironment* that contains common statistics etc.,
- a *alpenglow.cpp.PopularityModel* in the role of the recommender model,
- a *alpenglow.cpp.MemoryRankingLogger* in the role of the evaluator,
- a *alpenglow.cpp.ProceedingLogger* and a *alpenglow.cpp.MemoryUsageLogger* that log some info about the state of the experiment,
- a *alpenglow.cpp.PopularityTimeFrameModelUpdater* in the role of an updater.

The building and wiring of such experiment will be explained later. Now consider the function call sequence of *alpenglow.cpp.OnlineExperiment.run()* that runs the experiment. Below you can see the sequence diagram of this function.



The sequence diagram contains a huge loop on the timeline of the samples. Each sample is considered only once.

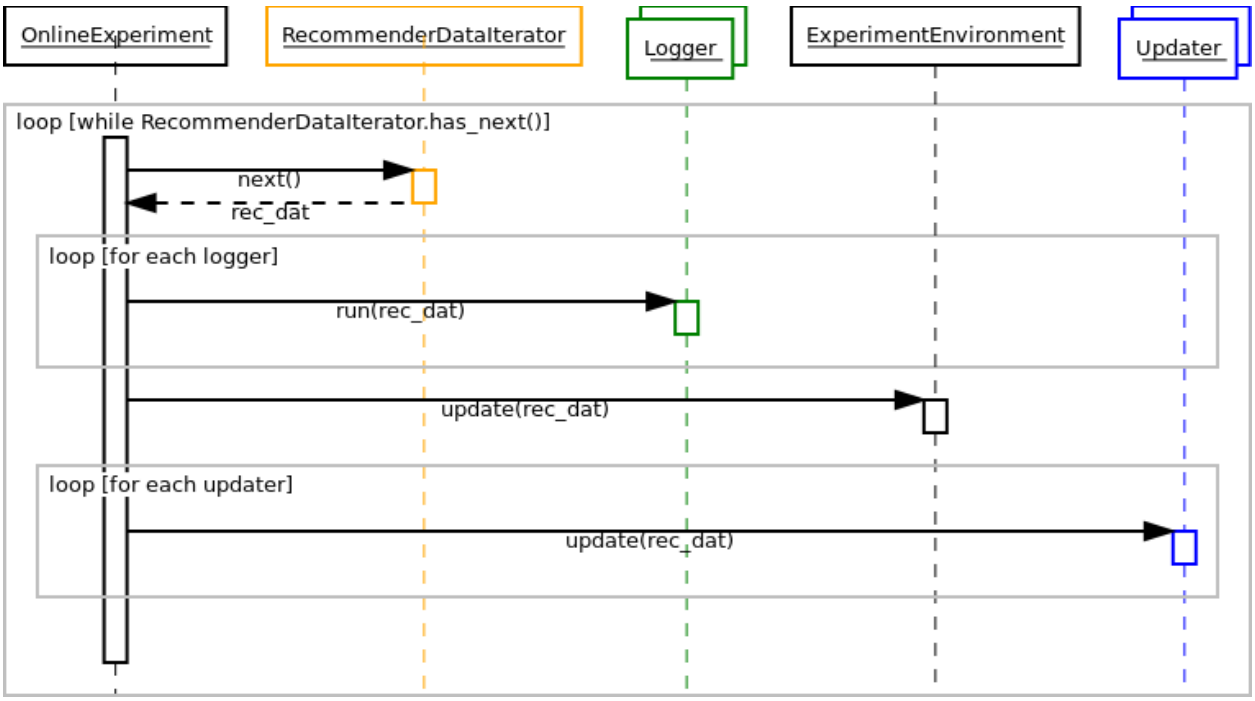
There are two phases for each sample, the evaluation and the training phase. In the evaluation phase, we call the *alpenglow.cpp.Logger.run()* function of the loggers. The function of the three loggers in this experiment:

- *alpenglow.cpp.MemoryRankingLogger* computes the rank of the correct item by querying the score of the known items (calling *alpenglow.cpp.PopularityModel.prediction()*) and writes it into a file and/or into a container. Note that while `prediction` is not a `const` function, it doesn't change the state of the model. Doing so would ruin the correctness of the experiment. Read more about rank computation in *Rank computation optimization*.
- *alpenglow.cpp.ProceedingLogger* logs the state of progress of the experiment to the screen, i.e. how many percents of the data is already processed.
- *alpenglow.cpp.MemoryUsageLogger* logs the current memory usage into a file.

In the training phase, first the central class updates the common statistic container,:py:class:*alpenglow.cpp.ExperimentEnvironment*. After that, the updater of the model is called. The updater contains model-specific code and updates the model directly through friendship.

In the next cycle, all of these is called with the next saple, and so on, until the last sample is processed.

5.3 General call sequence



The general function call sequence of `OnlineExperiment.run()` that runs the online experiment is depicted on the sequence diagram. The recommender model is not depicted here, although loggers and updaters may access it as necessary, see the popularity model above for an example.

During the evaluation phase, `online_exeriment` passes the sample to each *alpenglow.cpp.Logger* object that are added into it. Loggers can evaluate the model or log out some statistics as well. This is the evaluation phase for the sample, consequently, to keep the validity of the experiment, the loggers are not allowed to update the model or change its state.

During the second phase, when the sample becomes a training sample, `online_experiment` calls `update()` to each updater notify them about the new sample. First update is called to *alpenglow.cpp.ExperimentEnvironment* that updates some common containers and statistics of the training data, e.g. the number of the users, the list of most popular items.

Then the updaters of the recommender models are called also. In the general case, model updating algorithms are organised into a chain, or more precisely into a DAG. You can add any number of *alpenglow.cpp.Updater* objects into the experiment, and the system will pass the positive sample to each of them. Some *alpenglow.cpp.Updater* implementations can accept further *alpenglow.cpp.Updater* objects and passes them further the samples, possibly completed with extra information (e.g. gradient value) or mixed with generated samples (e.g. generated negative samples). Note that while the updating algorithms are allowed to retrain the model using the complete training data from the past, most of them uses only the newest sample or only a few more chosen from the past.

The experiment finishes when there are no more samples in the time series.

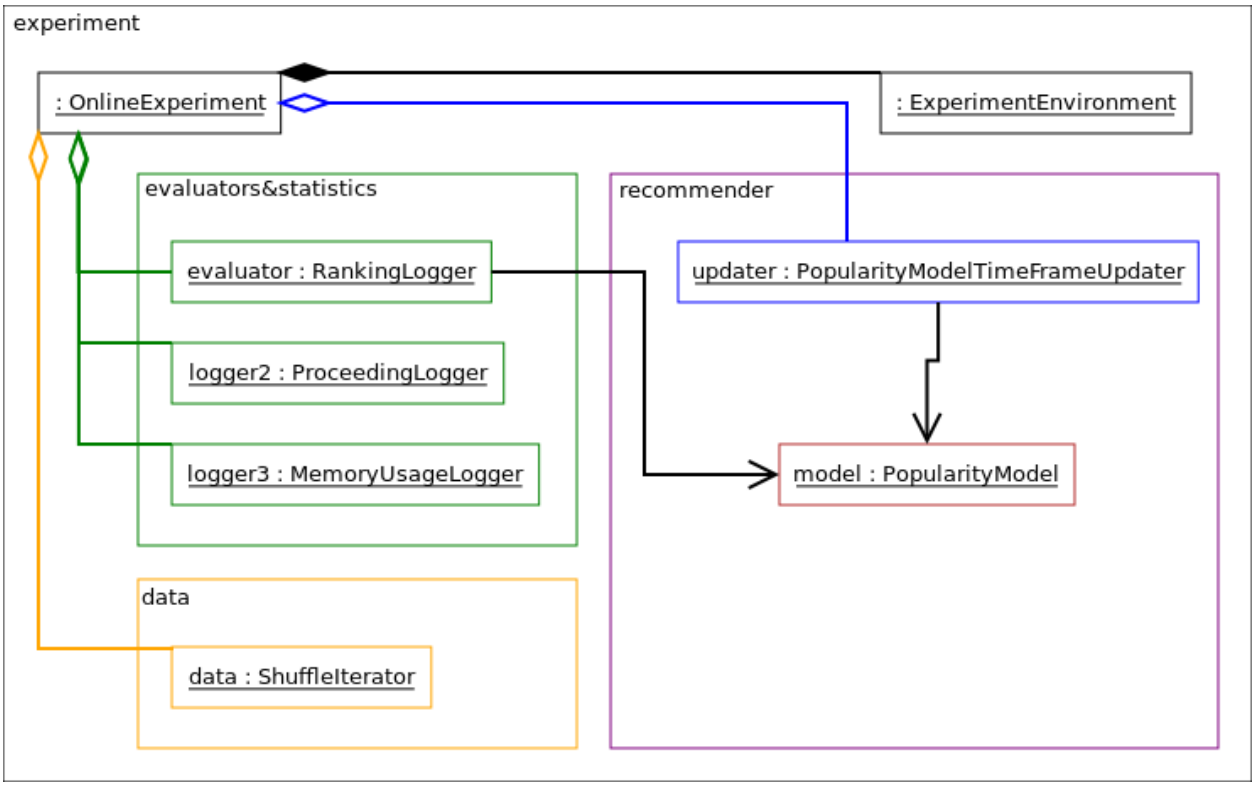
5.4 Examples

In what follows, we give object diagrams for a few experiments.

The depenedency injection mechanism in our python framework sets automatically *alpenglow.cpp.ExperimentEnvironment* to objects that require it (see *alpenglow.Getter* and *alpenglow.cpp.NeedsExperimentEnvironment* for details). Through this class, the experiment data (*alpenglow.cpp.RecommenderDataIterator*) is also accessible. As these two are available for any objects in the experiment, we omit the connections between these two and other objects.

5.4.1 Time-frame based popularity experiment

Recall the object diagram.



The python code that builds this experiment is the following. Note that most of the connections on the UML diagram correspond to a `set_xxxx()` or an `add_yyyy()` call.

This code is mostly for illustration. In most of the cases, one can use the pre-built experiments in *alpenglow.experiments*, see *alpenglow.experiments.PopularityTimeframeExperiment*.

```
from alpenglow.Getter import Getter as cpp
import alpenglow
import pandas as pd


cpp.collect() #see general/memory usage

#data
```

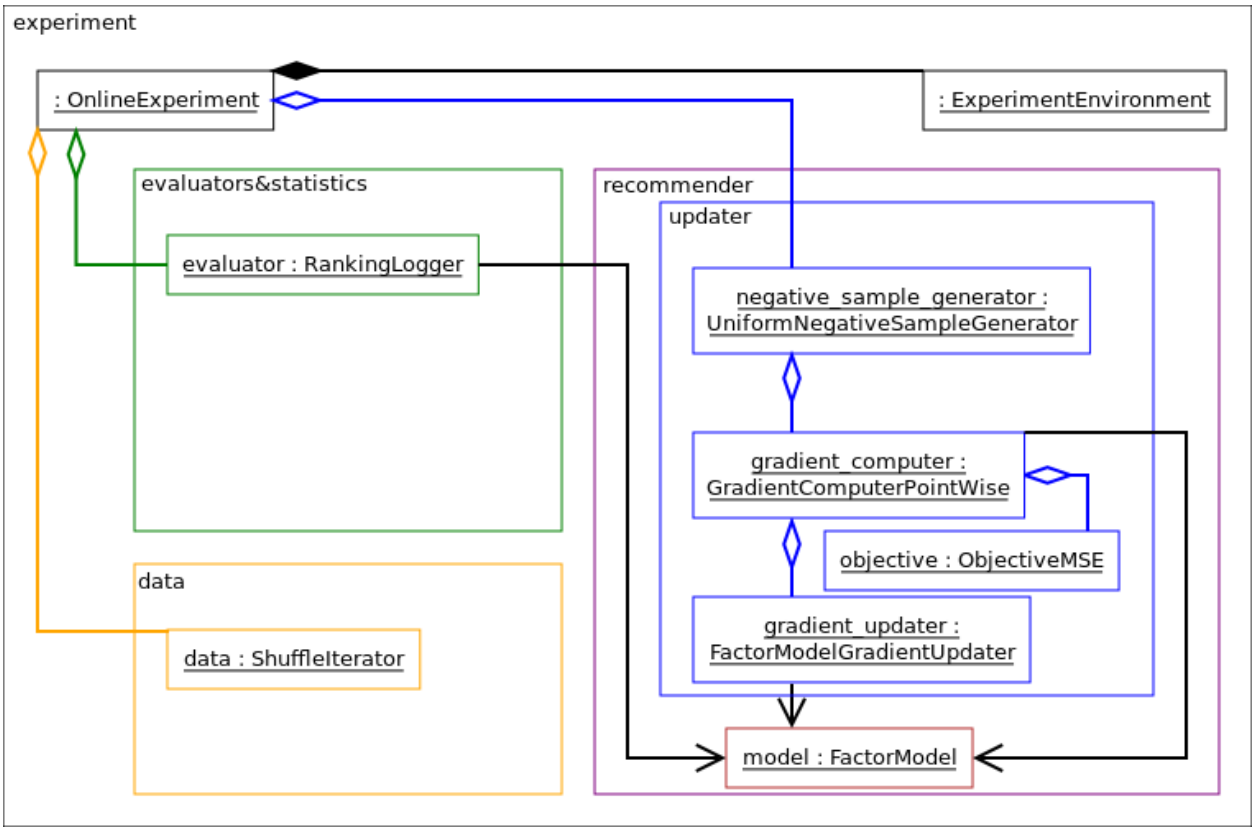
(continues on next page)

(continued from previous page)

```
data_python = pd.read_csv("http://info.ilab.sztaki.hu/~fbobee/alpenglow/alpenglow_sample_
↪dataset")
data_cpp_bridge = alpenglow.DataframeData(data_python)
data = cpp.ShuffleIterator(seed=12345)
data.set_recommender_data(data_cpp_bridge)

#recommender: model+updater
model = cpp.PopularityModel()
updater = cpp.PopularityTimeFrameModelUpdater(
    tau = 86400
)
updater.set_model(model)

#loggers: evaluation&statistics
logger1 = cpp.MemoryRankingLogger(
    memory_log = True
)
logger1.set_model(model)
ranking_logs = cpp.RankingLogs() #TODO get rid of these 3 lines
ranking_logs.top_k = 100
logger1.set_ranking_logs(ranking_logs)
logger2 = cpp.ProceedingLogger()
logger3 = cpp.MemoryUsageLogger()

#online_experiment
#Class experiment_environment is created inside.
online_experiment = cpp.OnlineExperiment(
    random_seed=12345,
    top_k=100,
    exclude_known=True,
    initialize_all=False
)
online_experiment.add_logger(logger1)
online_experiment.add_logger(logger2)
online_experiment.add_logger(logger3)
online_experiment.add_updater(updater)
online_experiment.set_recommender_data_iterator(data)

#clean, initialize, test (see general/cpp api)
objects = cpp.get_and_clean()
cpp.set_experiment_environment(online_experiment, objects)
cpp.initialize_all(objects)
for i in objects:
    cpp.run_self_test(i)

#run the experiment
online_experiment.run()

result = logger1.get_ranking_logs()
```

5.4.2 Matrix factorization experiment

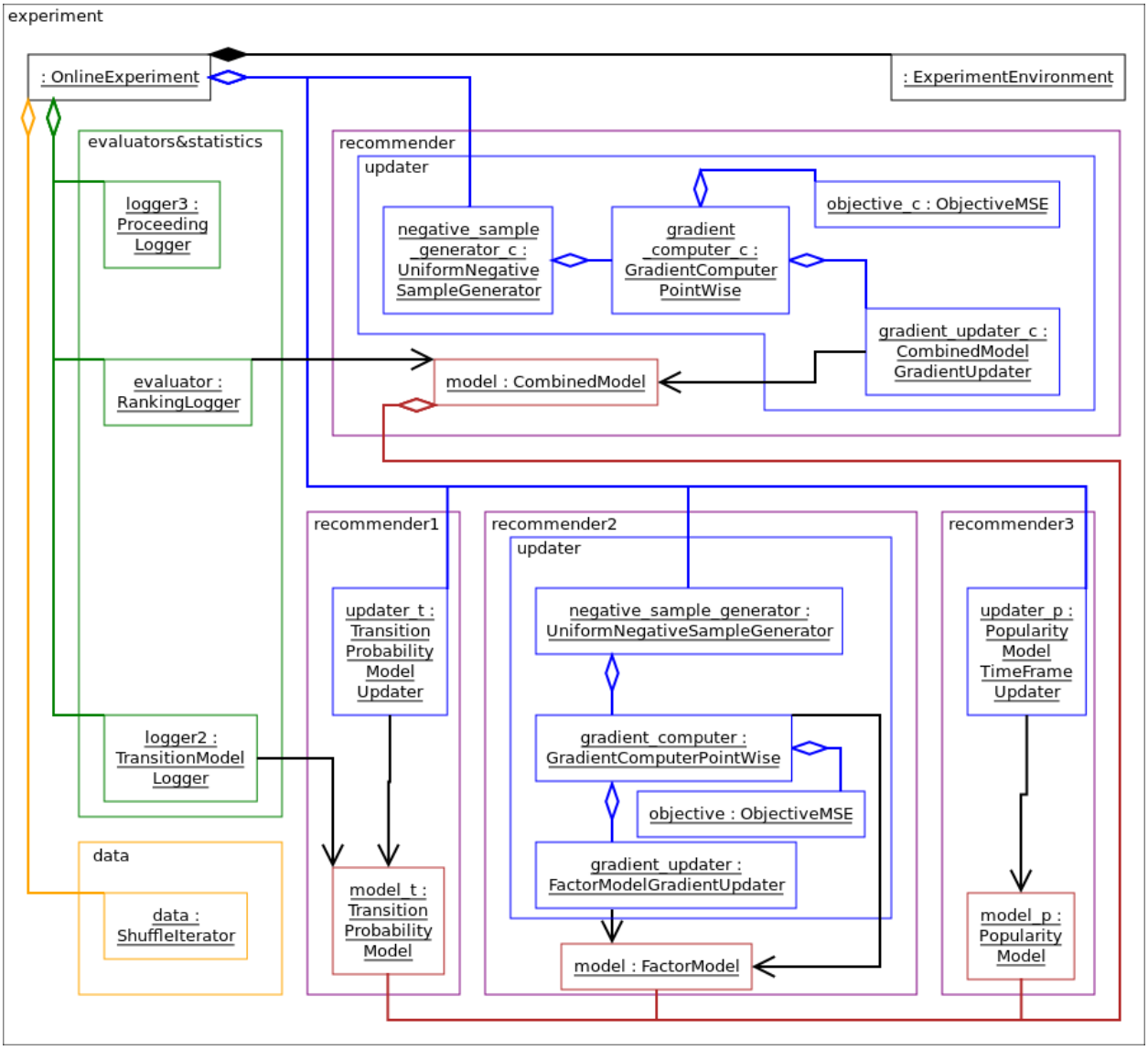
In this experiment, we have multiple updaters, chained into eachother.



See `alpenglow.experiments.MatrixFactorizationExperiment`.

5.4.3 Combined model experiment

In this experiment, the DAG of updaters is more complex.



See *Model combination*.

CHAPTER

SIX

ADJUSTABLE PROPERTIES OF EVALUATION

When running an online experiment, the user can control some modeling decisions and the flow of the experiment through global parameters. These modeling decisions, control options and the parameters are described below.

6.1 Global properties of the online experiment

The components in the online experiment can obtain the value of these parameters through query functions of *alpenglow.cpp.ExperimentEnvironment*. Note that some models ignore the value set in the common parameter container and always use the default value or a locally set value.

These are parameters of :*alpenglow.OnlineExperiment* that affect the results of experiments.

parameter name	description
`exclude_known`	Excludes items from evaluation that the actual user already interacted with. Besides evaluation, influences negative sample generation in gradient training. sample generation in gradient training. The `eval` columns of the input data should be set accordingly.
`initialize_all`	Set true to treat all users and items as existing from the beginning of the experiment. Technically, the largest user and item is searched in the time series and all ids starting from 0 will be treated as existing. By default this parameters is set to false, meaning that users and items come into existence by their first occurrence in a training sample.
`top_k`	Sets the toplist length. Models may treat scores liberally that belong to items that are under the limit to optimize running time.
`evaluation_start_time`	Do not evaluate samples having smaller timestamp.
`experiment_end_time`	Terminate experiment after the first sample having equal or larger time stamp.

6.2 Other possibilities to control evaluation

6.2.1 The eval column

This field controls which datapoints are to be evaluated before training. Defaults to True for all samples. It should be set in accordance with the `exclude_known` parameter of the experiment.

6.2.2 Calculating toplists

By default, Alpenglow doesn't actually compute toplists, see *Rank computation optimization*. However it is still possible to actually calculate them using the `calculate_toplists` parameter of the online experiment. If simply `True` then all of the toplists are calculated. The other possibility is to provide a list of boolean values, specifying which training instances are the toplists to be calculated for.

The toplists themselves can be retrieved after the end of the run using *alpenglow.OnlineExperiment.OnlineExperiment.get_predictions()*.

6.2.3 Filtering available items

It is possible to filter the evaluation to only consider a certain whitelist-set of available items at any time point. This can be useful for usecases such as TV recommendation, when not all items are available all the time.

For this, you'll need to configure your experiment and include an `AvailabilityFilter`. For this, please refer to *alpenglow.utils.AvailabilityFilter* and *alpenglow.OnlineExperiment*.

6.2.4 Delaying training

In a real system the engine might not be capable of processing every item immediately after it arrives. To better simulate this, it is possible to delay the training, i.e. evaluate each item with a 1 hour old model. This is done by desynchronizing the training and evaluation timelines by wrapping the updater of the experiment in a delay wrapper.

For this, you'll need to configure your experiment and wrap your updater in a *alpenglow.cpp.LearnerPeriodicDelayedWrapper*. This class is capable of simple delayed online updates and also delayed periodic updates, based on the values of the `delay` and `period` parameters.

CHAPTER

SEVEN

RANK COMPUTATION OPTIMIZATION

7.1 How do we optimize rank and top list computation?

The recommender models implement different interfaces based on what type of output can they easily generate. Examples: a popularity based model can easily generate a top list. A matrix factorization based model provides a score for each user-item pair. Using a `alpenglow.cpp.LempContainer`, the same model can efficiently serve the items roughly sorted descending by score, that lets us optimize rank computation.

7.2 Available interfaces and tools

1. `alpenglow.cpp.ToplistRecommender` interface -> returns filtered toplist of items for user.
2. *alpenglow.cpp.RankingScoreIteratorProvider* interface -> implementing class provides a `alpenglow.cpp.RankingScoreIterator` for itself. The iterator lets the rank computer or toplist creator iterate on the items in a roughly score-descending order.
3. `alpenglow.cpp.ModelFilter` interface -> DEPRECATED for this type of usage in favor of `alpenglow.cpp.RankingScoreIterator`. Has similar functionality + can filter out some items.
4. *alpenglow.cpp.Model* -> original, default interface. Returns score.

7.3 Rank computation methods

Using the interfaces listed above:

1. get the toplist, find the active item.
2. get the RankingScoreIterator, iterate on items score descending. Count items that have higher score than current one. Break computation if the score of remaining items is lower than the score of the current item.
3. like the previous one
4. iterate on the items (sorted by popularity, that correlates with the score to some extent). Count items that have higher score than the current item. Break cycle if found top_k items having higher score than the current item, because in that case the current item is not included in the top list.

7.4 Top list computation methods

Using the interfaces listed above:

1. get the toplist.
2. get the RankingScoreIterator, iterate on items score descending. Use a fixed size heap to keep an up-to-date list of items having the highest score. No more items having higher score than heap.min -> the toplist is the heap in a reverse order. This is impelemented in `alpenglow.cpp.ToplistFromRankingScoreRecommender`.
3. like the previous one, but no reusable implementation
4. trivial brute-force algorithm

CHAPTER

EIGHT

C++ API

The core of Alpenglow is written in C++. The C++ code is wrapped in python (see https://riverbankcomputing.com/software/sip/intro). We instantiate the C++ objects from python and wire them together there.

Below we describe the most important C++ interfaces.

8.1 Online experiment

Read *The anatomy of an online experiment* to get a general picture of the structure of the online experiments. Here we describe the most important interfaces and they purpose in the online experiment.

8.1.1 Model

Interface: *alpenglow.cpp.Model*

The recommender model of the experiment. Even though the implementation of *alpenglow.cpp.OnlineExperiment* does not specify the interface of the recommender model, all currently implemented models implement the *alpenglow.cpp.Model* interface. This interface provides function *alpenglow.cpp.Model.prediction()* to query a score for a user-item pair. The most commonly used evaluator, `alpenglow.cpp.RankingLogger` that computes the rank of the correct item on the top list generated for the current user, expects interface `Model`. However, some models implement further interfaces like `alpenglow.cpp.ToplistRecommender`, that let `RankingLogger` optimize rank computation. See *Rank computation optimization* for details.

8.1.2 Updater

Interface: *alpenglow.cpp.Updater*

In the online experiment, we iterate on a time series of samples. The framework processes samples one by one. For each sample, after the evaluation, in the training phase, the current sample becomes a training sample.

The central class of the experiment, *alpenglow.cpp.OnlineExperiment* that manages the iteration on the samples, notifies the components in the experiment about the new sample through the interface *alpenglow.cpp.Updater*. It calls the function *alpenglow.cpp.Updater.update()* for each sample. Even though the updaters have access to all the samples from the past, most updaters use only the newest sample that they get through the update function.

The central class of the experiment accepts multiple `Updater` instances, and calls each of them during the training phase. See *The anatomy of an online experiment* for details.

8.1.3 Logger

Interface: *alpenglow.cpp.Logger*

In the online experiment, we iterate on a time series of samples. The framework processes samples one by one. For each sample, during the evaluation phase, the central class of the experiment, *alpenglow.cpp.OnlineExperiment* that manages the iteration on the samples, calls *alpenglow.cpp.Logger.run()* function of loggers that are set into it. See *The anatomy of an online experiment* for details.

Loggers can serve different purposes. Their purpose can be to evaluate the experiment (see `alpenglow.cpp.RankingLogger` as an example), log some info about the state of the experiment (e.g. *alpenglow.cpp.MemoryUsageLogger*, *alpenglow.cpp.ProceedingLogger*) or some statistics about the state of the recommender model (e.g. *alpenglow.cpp.TransitionModelLogger*).

To log some data just before the termination of the experiment, set end loggers to the online experiment. These loggers have to implement the same interface and be added to the central class of the online experiment using function *alpenglow.cpp.OnlineExperiment.add_end_logger()*. The central class calls *alpenglow.cpp.Logger.run()* function of the end loggers after the training phase of the last sample is finished. The parameter of the call is a `NULL` pointer.

8.1.4 RecommenderDataIterator

Interface: *alpenglow.cpp.RecommenderDataIterator*.

The data must implement the interface *alpenglow.cpp.RecommenderDataIterator*. This class behaves like an iterator, but provides random access availability to the time series also. The two most commonly used implementations, that are available in preconfigured experiments also are *alpenglow.cpp.ShuffleIterator* and *alpenglow.cpp.SimpleIterator*. a While the latter keeps the original order of the samples, the former shuffles the samples that have identical timestamp in order to get rid of any artificial order. Use the parameter `shuffle_same_time` in the preconfigured experiments to choose the appropriate implementation.

8.1.5 Components for gradient based learning algorithms

Updating gradient based recommenders require some common tasks independetly from the acutal algorithm. These are described below together with the interfaces that are used to carry them out.

Negative sample generators

Interface: *alpenglow.cpp.NegativeSampleGenerator*

In implicit datasets, normally all samples are positive samples. Training gradient based recommenders using only positive samples would result in doubtful outcome. To avoid this problem, we generate negative samples. We treat all user-item pairs that are not present in the dataset as a negative sample. The negative sample generators select from the set of these "missing" pairs using different strategies.

The simplest strategy is choosing uniformly randomly a fixed size set of items for the current user from the set of items that this user have not yet iteracted with. This strategy is implemented in *alpenglow.cpp.UniformNegativeSampleGenerator*.

In the implementation, the negative sample generators are present in the chain of the updaters. They get the positive sample, generate negative ones and call to the next updater(s) for the original positive sample and for each negative one. See *The anatomy of an online experiment* to learn more about the chain of the updaters.

Gradient computers and objectives

Interface: *alpenglow.cpp.GradientComputer*, *alpenglow.cpp.ObjectivePointWise*

In the alpenglow framework, the objective-dependent and model-dependent part of the gradient computation is separated, as much as this is (mathematically) possible. The objective-depentent part is implemented in the gradient computer class, that passes the update call providing the gradient value to gradient updaters (see next section).

Gradient updaters

Interface: *alpenglow.cpp.ModelGradientUpdater*

The gradient updater computes the model-dependent part of the gradient and updates the model.

8.2 General interfaces

These are administrative things, nothing to do with the recommender algorithm. These make some administrative things, solved in a centralized way:

- injecting the common `ExperimentEnvironment` object into classes that require it (only in the online experiments),
- notify the classes about the end of the wiring phase,
- run self-checks to find wiring errors and faulty parameters.

In the preconfigured experiments (*alpenglow.experiments*, *alpenglow.offline*) these administration tasks are automatically performed.

8.2.1 NeedsExperimentEnvironment

Interface: *alpenglow.cpp.NeedsExperimentEnvironment*.

In the online experiment, the common data, centrally updated statistics and common simulation features are available to all objects through *alpenglow.cpp.ExperimentEnvironment*. The system can automatically inject this dependency to the objects using *alpenglow.Getter.MetaGetter.set_experiment_environment()*.

In the offline experiments, `ExperimentEnvironment` is not available. The common objects and parameters that would be available through it need to be set locally.

8.2.2 Initializable

Interface: *alpenglow.cpp.Initializable*

The C++ objects are instantiated in python and then wired together using `set_xxx()` and `add_xxx()` functions. When the wiring is finished, some object require a notification to make some initial tasks that depend on the final configuration (e.g. depend on the number of subobjects that were added).

Use *alpenglow.Getter.MetaGetter.initialize_all()* to notify objects by calling *alpenglow.cpp.Initializable.initialize()* when wiring is finished.

8.2.3 self_test() function

Example: *alpenglow.cpp.FactorModel.self_test()*

The wiring of the experiment is very error-prone. Wiring errors may lead to segmentation faults and undefined behaviour. To mitigate this problem, most of the classes can test themselves for missing subcomponents and contradictory parameters. Use *alpenglow.Getter.MetaGetter.run_self_test()* to call `self_test` for each object that implements this function.

8.3 Offline experiments

The batch style experiments that have a fixed train-test split need some separate classes. See *alpenglow.cpp.OfflineLearner* and *alpenglow.cpp.OfflineEvaluator*.

The models that are trained in batch style can be embedded in the online framework. See *alpenglow.experiments.ALSOnlineFactorExperiment* and *alpenglow.experiments.BatchFactorExperiment*. The embedding works in the other direction, see *alpenglow.offline.models.PopularityModel*.

CHAPTER

NINE

PYTHON API

9.1 General information

Alpenglow grew out of an internal framework that was originally written and used in C++. The basic premise of this framework was that a large number of componenst were implemented that one could piece together to create experiments. When we decided to provide a Python framework, this architecture stayed in place, only the framework now also provides precomposed, parametrizable experiments.

This document serves to describe the overall structure and workings of the Python package.

9.1.1 Python bindings in SIP

We use SIP to create Python bindings for C++ classes and interfaces. This allows us to access almost the entirety of the C++ codebase from Python. Python types like lists dictinaries and tuples are automatically converted to C++ stdlib types, and we can instantiate or even subclass C++ classes from Python. There are some caveats though, which we list under the appropriate sections here.

9.1.2 Type conversion

Generally type conversion work automatically (if the type conversion is implemented in SIP files, see the *sip/std* folder). However generally conversions must do a copy of the data, thus incurring additional memory requirements. This is not a problem when model hyperparameters are copied, but with training datasets it can become problematic. If you are short on memory, what you can do is to run the experiments on data that is read from file, so it is directly read into the appropriate C++ structures and skips Python.

9.1.3 Instantiating C++ classes in Python

C++ classes can be instantiated in Python without problem. However, since C++ doesn't support named arguments, these have a special way of being implemented. When a class instance is being created and named parameters are provided for a class named *Klass*, the system looks for a C++ struct named *KlassParameters* and sets the same named attribute on this, then provides it to the constructor of the C++ class *Klass* as its single parameter. This convention is followed throughout the codebase.

This process is done by the class `alpenglow.Getter`: instead of importing the package *alpenglow.cpp*, you should include this class and use it described in the examples.

One more thing this class does is help with memory management. Since Python is garbage collected and C++ is not, memory management takes a little extra effort. Specifically, when a C++ class is instantiated in Python, is given to another C++ class instance, but then Python loses the reference, this causes a problem: the Python garbage collector frees the memory and the C++ code segfaults. This is avoided by keeping a reference of the objects in Python, tied to

the experiment instance that is being run. This is also handled by the class *alpenglow.Getter*: it can be asked to keep a reference of all objects created through it. For an example of this, see the implementation of the *run* method of *alpenglow.OnlineExperiment*.

9.1.4 Implementing C++ interfaces in Python

Implementing C++ interfaces in Python generally works well. The main caveat is that multiple inheritance is not supported by SIP, thus it may be advisable to create a C++ interaface first specifically for being subclassed in Python. Another way to "cheat" this is to use composition instead of inheritance. You can see an example of this in the class *alpenglow.PythonModel.SelfUpdatingModel*.

9.1.5 GIL and multithreading

While Python doesn't play nice with multithreading because of the global interpreter lock, this restriction does not apply to C++ code. Alpenglow explicitly releases the GIL at the start of the run of experiments, thus in theory multiple experiments can be run simultaneously in the same process. However, this is not very well tested, so use it carefully.

9.2 Package structure

The package is divided into three main parts. The first one is the most often used one, the online recommendation experiment API. This spans over the root package and the *experiments*, *eval*, and *utils* subpackages. The other two are *alpenglow.cpp* which provides raw access to the C++ classes and *alpenglow.offline* which stores the scikit-learn like api.

CHAPTER

TEN

IMPLEMENTING A NEW MODEL IN C++

10.1 Creating C++ model class and google tests

The models must inherit from *alpenglow.cpp.Model*. The bare minimal model implementation consists of the following two files.

File *cpp/src/main/models/MyNewModel.h*:

```
#ifndef MY_NEW_MODEL_H
#define MY_NEW_MODEL_H

#include "Model.h"
#include <gtest/gtest_prod.h>

class MyNewModel
  : public Model
{
  public:
    double prediction(RecDat* rec_dat) override;
};

#endif /* MY_NEW_MODEL_H */
```

File *cpp/src/main/models/MyNewModel.cpp*:

```
#include "MyNewModel.h"

double MyNewModel::prediction(RecDat* rec_dat){
  return 0; //TODO method stub
}
```

Unit test in file *cpp/src/test/models/TestMyNewModel.cpp*:

```
#include <gtest/gtest.h>
#include "../../main/models/MyNewModel.h"

namespace {

class TestMyNewModel : public ::testing::Test {
  public:
    TestMyNewModel(){}
```

(continues on next page)

(continued from previous page)

```
    virtual ~TestMyNewModel(){}
    void SetUp() override {
    }
    void TearDown() override {
    }
};

} //namespace

TEST_F(TestMyNewModel, test){
  MyNewModel model;
}


int main (int argc, char **argv) {
  testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

You may need to install gtest in directory *cpp/dep* before compiling your code:

```
cd cpp/dep
mv gtest gtest_old
wget https://github.com/google/googletest/archive/release-1.7.0.zip
unzip -q release-1.7.0.zip
mv googletest-release-1.7.0 gtest
cd gtest
GTEST_DIR=`pwd`
mkdir build
cd build
g++ -isystem ${GTEST_DIR}/include -I${GTEST_DIR} -pthread \
    -c ${GTEST_DIR}/src/gtest-all.cc
ar -rv libgtest.a gtest-all.o
```

To compile your model and link the test binary, step into directory *cpp/src* and run *scons*. To make compilation faster, you can run it on multiple threads, e.g. *scons -j4* uses 4 threads. Note that for the sake of simplicity, all *.o* files are linked to all test binaries, so all of them are regenerated if any *.h* or *.cpp* file changes, making the linking process a bit slow.

The test binaries are generated to *cpp/bin/test*.

10.2 Making the new model available in python

To make the model available in python, you will need the appropriate *sip/src/models/MyNewModel.sip* file. For simple C++ headers, the sip file can be easily generated using a script:

```
sip/scripts/header2sip cpp/src/main/models/MyNewModel.h overwrite
```

Note that the conversion script may fail for too complicated C++ files and also for ones that do not follow the formatting conventions of the project. To mark your header as automatically convertible, add the comment line

```
//SIP_AUTOCONVERT
```

to the header file. However, the conversion does not run automatically before compiling, you need to run it yourself, if you update the header file.

Add your sip file to *sip/recsys.sip* to include it in python compilation:

```
%Include src/models/MyNewModel.sip
```

Then reinstall alpenglow:

```
pip install --upgrade --force-reinstall --no-deps .
```

Now the new model is available in python:

```
import alpenglow.Getter as rs
my_new_model = rs.MyNewModel()
rd = rs.RecDat()
rd.time = 0
rd.user = 10
rd.item = 3
my_new_model.prediction(rd)
```

10.3 Constructor parameters

The constructor parameters are organized into a struct for each class, that has the same name as the class, appended *Parameters*. To add a parameter named *fading_factor*, extend the header file like that:

```
struct MyNewModelParameters {
  double fading_factor = 0.9;
};

class MyNewModel
  : public Model
{
  public:
    MyNewModel(MyNewModelParameters* params){
      fading_factor_ = params->fading_factor;
    }
  double prediction(RecDat* rec_dat) override;
  private:
    double fading_factor_ = 0;
};
```

Update the unit test:

```
TEST_F(TestMyNewModel, test){
  MyNewModelParameters model_params;
  model_params.fading_factor = 0.5;
  MyNewModel model(&model_params);
}
```

Recompile using *scons* before running the unit test. If all is fine on the cpp level, update the sip file and reinstall the pyton package:

```
sip/scripts/header2sip cpp/src/main/models/MyNewModel.h overwrite
pip install --upgrade --force-reinstall --no-deps .
```

Now the parameters is available in python:

```
import alpenglow.Getter as rs
my_new_model = rs.MyNewModel(fading_factor=0.8)
rd = rs.RecDat()
rd.time = 0
rd.user = 10
rd.item = 3
my_new_model.prediction(rd)
```

10.4 Updater for the model

The updater class, that performs the incremental update, must implement interface *alpenglow.cpp.Updater*. The following is the minimal implementation.

File *cpp/src/main/models/MyNewModelUpdater.h*:

```
#ifndef MY_NEW_MODEL_UPDATER_H
#define MY_NEW_MODEL_UPDATER_H

//SIP_AUTOCONVERT

#include "../general_interfaces/Updater.h"
#include "MyNewModel.h"

class MyNewModelUpdater: public Updater{
  public:
    void update(RecDat* rec_dat) override;
    void set_model(MyNewModel* model){
      model_ = model;
    }
  private:
    MyNewModel* model_ = NULL;
};

#endif /* MY_NEW_MODEL_UPDATER_H */
```

File *cpp/src/main/models/MyNewModelUpdater.cpp*:

```
#include "MyNewModelUpdater.h"

void MyNewModelUpdater::update(RecDat* rec_dat){
  return; //TODO perform incremental update here
}
```

Declare the updater as a friend of the model class, so the updater can update the private state fields of the model:

```
class MyNewModel
  : public Model
{
  // ...
  friend class MyNewModelUpdater;
};
```

Normally the unit test for the model and the updater is implemented as a common test. Extend the unit test of the model:

```
#include "../../main/models/MyNewModelUpdater.h"
TEST_F(TestMyNewModel, test){
  // ...
  MyNewModelUpdater updater;
  updater.set_model(&model);
}
```

Compile with *scons*, run the test, then generate sip file:

```
sip/scripts/header2sip cpp/src/main/models/MyNewModelUpdater.h overwrite
```

Add the sip file to *sip/recsys.sip*

```
%Include src/models/MyNewModelUpdater.sip
```

Reinstall the python module:

```
pip install --upgrade --force-reinstall --no-deps .
```

Now the updater is available in python:

```
import alpenglow.Getter as rs
my_new_model = rs.MyNewModel(fading_factor=0.8)
my_new_updater = rs.MyNewModelUpdater()
my_new_updater.set_model(my_new_model)
rd = rs.RecDat()
my_new_updater.update(rd) #does nothing (yet)
```

Similarly, to create a logger and log some statistics about your model, create a class that implements interface *alpenglow.cpp.Logger*.

10.5 Add logic to the model and the updater

We implement a fading popularity model, that computes item popularity discounting exponentially in time.

```
TEST_F(TestMyNewModel, test){
  MyNewModelParameters model_params;
  model_params.fading_factor = 0.5;
  MyNewModel model(&model_params);
  MyNewModelUpdater updater;
  updater.set_model(&model);
```

(continues on next page)

(continued from previous page)

```
  RecDat rec_dat;
  rec_dat.time=0;
  rec_dat.user=1;
  rec_dat.item=2;

  EXPECT_EQ(0,model.prediction(&rec_dat));

  rec_dat.time=1;
  rec_dat.item=2;
  updater.update(&rec_dat);
  EXPECT_EQ(1,model.prediction(&rec_dat));
  rec_dat.item=3;
  EXPECT_EQ(0,model.prediction(&rec_dat));

  rec_dat.time=2;
  rec_dat.item=2;
  EXPECT_DOUBLE_EQ(0.5,model.prediction(&rec_dat));
  rec_dat.item=3;
  EXPECT_EQ(0,model.prediction(&rec_dat));
  rec_dat.item=2;
  updater.update(&rec_dat);
  EXPECT_DOUBLE_EQ(1+0.5,model.prediction(&rec_dat));
}
```

Now this test naturally fails. We implement the model:

```
class MyNewModel
  : public Model
{
    // ...
    std::vector<double> scores_;
    std::vector<double> times_;
    // ...
};
```

```
double MyNewModel::prediction(RecDat* rec_dat){
  int item = rec_dat->item;
  if (scores_.size() <= item) return 0;
  double time_diff = rec_dat->time-times_[item];
  return scores_[item]*std::pow(fading_factor_,time_diff);
}
```

And the updater:

```
void MyNewModelUpdater::update(RecDat* rec_dat){
  int item = rec_dat->item;
  int time = rec_dat->time;
  if (item >= model_->scores_.size()) {
    model_->scores_.resize(item+1,0);
    model_->times_.resize(item+1,0);
  }
  double time_diff = time-model_->times_[item];
```

(continues on next page)

(continued from previous page)

```
  model_->scores_[item]*=std::pow(model_->fading_factor_,time_diff);
  model_->scores_[item]+=1;
  model_->times_[item]=time;
}
```

After recompiling with *scons*, the test passes. These modifications are irrelevant for the sip files, but the python package needs to be reinstalled.

10.6 Creating an experiment using the new model

To create a preconfigured experiment using the new class, inherit from *alpenglow.OnlineExperiment* and implement *_config()*. Calling *run()* for the new class will run the experiment, i.e. for each sample, compute rank and then call the updater to update the model. Create file *python/alpenglow/experiments/MyNewExperiment.py*:

```
import alpenglow.Getter as rs
import alpenglow as prs

class MyNewExperiment(prs.OnlineExperiment):
    """Recommends the most popular item from the set of items seen so far,
       discounting exponentially by time.
    """

    def _config(self, top_k, seed):
        model = rs.MyNewModel(**self.parameter_defaults(
          fading_factor=0.8
        ))
        updater = rs.MyNewModelUpdater()
        updater.set_model(model)

        return (model, updater, [], [])
```

Append the new class to *python/alpenglow/experiments/__init__.py*:

```
from .MyNewExperiment import *
```

Create the corresponding integration test in *python/test_alpenglow/experiments/test_MyNewExperiment.py*:

```
import alpenglow as prs
import alpenglow.Getter as rs
import alpenglow.experiments
import pandas as pd
import math


class TestMyNewExperiment:
    def test_MyNewExperiment(self):
        data = pd.read_csv(
            "python/test_alpenglow/test_data_4",
            sep=' ',
            header=None,
            names=['time', 'user', 'item', 'id', 'score', 'eval']
```

(continues on next page)

(continued from previous page)

```
        )
        experiment = alpenglow.experiments.MyNewExperiment(
            top_k=100,
            seed=254938879,
            fading_factor=0.9
        )
        popRankings = experiment.run(data, verbose=True, exclude_known=True)
        print(list(popRankings["rank"].fillna(101)))
        assert popRankings.top_k == 100
        desired_ranks = [] #TODO
        assert list(popRankings["rank"].fillna(101)) == desired_ranks
```

Reinstall the python package and all the tests using *pytest* or only the new test using the following command:

```
pytest python/test_alpenglow/experiments/test_MyNewExperiment.py
```

The test will fail, but it will print the ranks produced by the model. It would be very time consuming to check whether all values are correct, but simple errors (e.g. all values are 101 because the *set_model()* call is missing) might be obvious. If all seems to be fine, then copy the actual output to the expected output field. This way the test will catch unintentional modifications of the logic of the model.

Now the new experiment is available in python, using similar code to the test.

10.7 Document your model

To document the C++ clasees, use java-style documentation comments in the header files. Note that the comment describing the class is after the opening bracket of the class declaration, and the comment that belongs to the function is after the function declaration.

```
class MyNewModel
  : public Model
{
/**
  Item popularity based model.  The popularity of the items fades
  in time exponentially.
*/
  public:
    MyNewModel(MyNewModelParameters* params){
      fading_factor_ = params->fading_factor;
    }
    double prediction(RecDat* rec_dat) override;
    /**
      prediction(RecDat)

      Computes prediction score for the sample.  Uses only the time
      and item fields, the user is ignored.

      Parameters
      ----------
      rec_dat : RecDat*
          The sample.
```

(continues on next page)

(continued from previous page)

```
        Returns
        -------
        double
            The prediction score.
    */
    // ...
}
```

Then transform the comments by the header->sip converter, reinstall the python package and regenerate the documentation. The reinstallation step is necessary, as the documentation generator acquires the documentation from the installed alpenglow package.

```
sip/scripts/header2sip cpp/src/main/models/MyNewModel.h overwrite
pip install --upgrade --force-reinstall --no-deps .
cd docs
make dirhtml
```

The process is similar for the updater. To document an experiment, add a docstring (already shown in the example above).

10.8 Implement further functions of the Model interface

The *Model* interface provides 4 more functions to override, and the framework provides one:

```
//void add(RecDat* rec_dat) override; //not applicable in our case
void write(ostream& file) override;
void read(istream& file) override;
void clear() override;
bool self_test();
```

Function *add()* is called before gradient updates. It notifies the model about the existance of a user and an item, and its responsibility is the (random) initialization of the model w.r.t. the item and user in the parameter. As an example, consider the random initialization of factors in case of a factor model.

Functions *write()* and *read()* implement serialization. While serialization possibilities are not complete in the framework, it is possible to write out and read back models in *alpenglow.experiment.BatchFactorExperiment* and *alpenglow.experiment.BatchAndOnlineExperiment*. For details, see *Serialization*.

Function *clear()* must clear and reinitialize the model.

Function *self_test()* must check whether all components are properly set, the parameters are sane etc. The main goal is to prevent hard-to-debug segmentation faults caused by missing *set_xxx()* calls. Note that *self_test()* is not virtual, it is called by the framework for the appropriate type and it is the functions responsibility to call *self_test()* of its ancestors.

Here are the expanded testcases:

```
TEST_F(TestMyNewModel, test){
  // ...

  //read, write
  std::stringstream ss;
  model.write(ss);
```

(continues on next page)

(continued from previous page)

```
  model.write(ss);

  MyNewModel model2(&model_params);
  model2.read(ss);
  EXPECT_DOUBLE_EQ(model.prediction(&rec_dat), model2.prediction(&rec_dat));
  MyNewModel model3(&model_params);
  model3.read(ss);
  EXPECT_DOUBLE_EQ(model.prediction(&rec_dat), model3.prediction(&rec_dat));

  //clear
  model.clear();

  for(int item : {0,1,2,3,4,5}){
    rec_dat.item=item;
    EXPECT_EQ(0,model.prediction(&rec_dat));
  }
}
```

```
TEST_F(TestMyNewModel, self_test){
  MyNewModelParameters model_params;
  model_params.fading_factor = 0.5;
  MyNewModel model(&model_params);
  EXPECT_TRUE(model.self_test());

  model_params.fading_factor = 0;
  MyNewModel model2(&model_params);
  EXPECT_TRUE(model2.self_test());

  model_params.fading_factor = -0.2;
  MyNewModel model3(&model_params);
  EXPECT_FALSE(model3.self_test());
}
```

And the implementations:

```
void MyNewModel::write(ostream& file){
  file << scores_.size() << " ";
  for (double score : scores_){
    file << score << " ";
  }
  file << times_.size() << " ";
  for (double time : times_){
    file << time << " ";
  }
}
void MyNewModel::read(istream& file){
  int scores_size;
  file >> scores_size;
  scores_.resize(scores_size);
  for (uint i=0;i<scores_.size();i++){
    file >> scores_[i];
  }
```

(continues on next page)

(continued from previous page)

```
  int times_size;
  file >> times_size;
  times_.resize(times_size);
  for (uint i=0;i<times_.size();i++){
    file >> times_[i];
  }
}
void MyNewModel::clear(){
  scores_.clear();
  times_.clear();
}
```

Normally *self_test()* is implemented in the header:

```
bool self_test() {
  bool ok = Model::self_test();
  if (fading_factor_<0) ok = false;
  return ok;
}
```

Regenerate the sip file and reinstall the python package to make the new functions available in python and visible for the online experiment framework.

10.9 Access common data

In the online experiment, the framework provides some common parameters and statistics through class *alpenglow.cpp.ExperimentEnvironment* (see details there). To access them, the class needs to implement interface *alpenglow.cpp.NeedsExperimentEnvironment*, so the online experiment framework will set the *ExperimentEnvironment* object.

Typically such classes also implement `Initializable`, asking the framework to call their *autocalled_initialize()* function when the experiment is already built (after the *set_xxx()* calls), and in that function, they copy the pointers to the common objects. See the example below.

```
#include "../general_interfaces/NeedsExperimentEnvironment.h"
#include "../general_interfaces/Initializable.h"
// ...
class MyNewModel
  : public Model
  , public NeedsExperimentEnvironment
  , public Initializable
{
  public:
    // ...
    void set_items(const vector<int>* items){ items_ = items; }
    bool self_test() {
      bool ok = Model::self_test();
      if (fading_factor_<0) ok = false;
      if (items_==NULL) ok=false;
      return ok;
    }
```

(continues on next page)

(continued from previous page)

```
  private:
    bool autocalled_initialize(){
      if (items_ == NULL) { //items_ is not set
        if (experiment_environment_!=NULL){ //exp_env is available
          items_ = experiment_environment_->get_items();
        } else {
          return false; //can't set items
        }
      }
      return true;
    }
    const std::vector<int>* items_ = NULL;
    // ...
};
```

We also need to update the unit test:

```
class TestMyNewModel : public ::testing::Test {
  public:
    vector<int> items;
    // ...
};

TEST_F(TestMyNewModel, test){
  // ...
  MyNewModel model(&model_params);
  model.set_items(&items);
  // ...
  items.push_back(2);
  updater.update(&rec_dat);
  // ...
}
```

```
TEST_F(TestMyNewModel, self_test){
  MyNewModelParameters model_params;
  model_params.fading_factor = 0.5;
  MyNewModel model(&model_params);
  model.set_items(&items);
  EXPECT_TRUE(model.self_test());

  model_params.fading_factor = 0;
  MyNewModel model2(&model_params);
  model2.set_items(&items);
  EXPECT_TRUE(model2.self_test());

  model_params.fading_factor = -0.2;
  MyNewModel model3(&model_params);
  model3.set_items(&items);
  EXPECT_FALSE(model3.self_test());

  model_params.fading_factor = 0.5;
  MyNewModel model4(&model_params);
```

(continues on next page)

(continued from previous page)

```
  EXPECT_FALSE(model4.self_test());

  model_params.fading_factor = 0.5;
  MyNewModel model5(&model_params);
  ExperimentEnvironment expenv;
  model5.set_experiment_environment(&expenv);
  EXPECT_TRUE(model5.initialize());
  EXPECT_TRUE(model5.self_test());
}
```

However, changing the test of *MyNewExperiment* is not necessary as the framework automatically sets *experiment_environment_* and calls *autocalled_initialize()*. The alternative setting method, *set_items()* is necessary for offline experiments where *exp_env* is not available and might be useful in unit tests.

10.10 Make evaluation faster

In the online experiment, the rank of the relevant item is computed by default by comparing its score to the score of the other items one-by-one. The computation can halt when we already found more better items, than the rank threshold, or when all items were compared to the relevant. By processing the items first that has higher score, we make the process faster. This is the goal of the interface `alpenglow.cpp.RankingScoreIterator`. In addition, in case of some model implementation, keeping an up-to-date toplist is computationally easy. For these models, we can query the rank directly. Such models should implement the interface `alpenglow.cpp.ToplistRecommender`.

CHAPTER

ELEVEN

IMPLEMENTING A NEW MODEL IN PYTHON

While the core of the framework runs in C++, the fact that Alpenglow uses SIP for its Python bindings allows us to implement models in Python, inheriting from the necessary C++ classes. Please note though that this feature is still experimental and may be a little rough arond the edges.

Let's use a very simple example for demonstration: the empirical transition probability model. This model records how often items follow each other, and always recommenders the empirically most likely next item based on this log. Note that even though the name implies that our model should output probabilities, in fact outputting raw counts is the same from an evaluation perspective, since the empirical probability is monotonic as a function of counts.

The following code demonstrates a very simple implementation of this:

```
import pandas as pd
from collections import defaultdict

from alpenglow.evaluation import DcgScore
from alpenglow import SelfUpdatingModel, OnlineExperiment


data = pd.read_csv("http://info.ilab.sztaki.hu/~fbobee/alpenglow/alpenglow_sample_dataset
↪")


class TransitionProbabilityModel(SelfUpdatingModel):
    def __init__(self):
        super(TransitionProbabilityModel, self).__init__()
        self.last_item = defaultdict(lambda: -1)
        self.num_transitions = defaultdict(lambda: 0)

    def update(self, rec_dat):
        self.num_transitions[(self.last_item[rec_dat.user], rec_dat.item)] += 1
        self.last_item[rec_dat.user] = rec_dat.item

    def prediction(self, rec_dat):
        return self.num_transitions[(self.last_item[rec_dat.user], rec_dat.item)]


class TransitionProbabilityExperiment(OnlineExperiment):
    def _config(self, top_k, seed):
        model = TransitionProbabilityModel()
        return (model._model, model._updater, [])
```

(continues on next page)

(continued from previous page)

```
experiment = TransitionProbabilityExperiment(top_k=5)
rankings = experiment.run(data.head(100000))
rankings['dcg'] = DcgScore(rankings)
averages = rankings['dcg'].groupby((rankings['time']-rankings['time'].min())//86400).
↪mean()
print(averages)
```

We import the neccessary packages, load the data, define the model and define the experiment. The model definition is done by subclassing `alpenglow.SelfUpdatingModel`. Note, that this itself is not a C++ class, but a Python class that handles a few neccessary steps for us. It is possible to go deeper and implement the model and its updater separately for example. For this and other, more fine-grained possibilities, please refer to the source of `alpenglow.SelfUpdatingModel` and the page *C++ API*.

We define three functions for the model: initialization, update and prediction. Initialization is self-explanatory. Update is called after each evaluation step, and receives the training sample as parameter. For the definition of the type of rec_dat, please refer to *alpenglow.cpp.RecDat*. Prediction is called for scoring positive samples, as well as to determine the ranking of items during evaluation.

The implemented logic in the above example is quite simple: we store two dictionaries - one contains the last visited items of each user, the other counts the number of occurrences of items after each other. The prediction is simply the latter number.

Let's run the experiment:

```
:$ time python first_example.py
running experiment...
0%-1%-2%-3%-4%-5%-6%-7%-8%-9%-10%-11%-12%-13%-14%-15%-16%-17%-18%-19%-20%-21%-22%-23%-24
↪%-25%-26%-27%-28%-29%-30%-31%-32%-33%-34%-35%-36%-37%-38%-39%-40%-41%-42%-43%-44%-45%-
↪46%-47%-48%-49%-50%-51%-52%-53%-54%-55%-56%-57%-58%-59%-60%-61%-62%-63%-64%-65%-66%-67
↪%-68%-69%-70%-71%-72%-73%-74%-75%-76%-77%-78%-79%-80%-81%-82%-83%-84%-85%-86%-87%-88%-
↪89%-90%-91%-92%-93%-94%-95%-96%-97%-98%-99%-OK
time
0.0    0.003957
1.0    0.004454
2.0    0.006304
3.0    0.006426
4.0    0.009281
Name: dcg, dtype: float64

real    2m50.635s
user    2m56.236s
sys     0m10.479s
```

We can see that the score is nicely improving from week to week: the model is able to learn incrementally. We can compare it to the builtin transition model:

```
:$ time python transition_builtin.py
running experiment...
0%-1%-2%-3%-4%-5%-6%-7%-8%-9%-10%-11%-12%-13%-14%-15%-16%-17%-18%-19%-20%-21%-22%-23%-24
↪%-25%-26%-27%-28%-29%-30%-31%-32%-33%-34%-35%-36%-37%-38%-39%-40%-41%-42%-43%-44%-45%-
↪46%-47%-48%-49%-50%-51%-52%-53%-54%-55%-56%-57%-58%-59%-60%-61%-62%-63%-64%-65%-66%-67
↪%-68%-69%-70%-71%-72%-73%-74%-75%-76%-77%-78%-79%-80%-81%-82%-83%-84%-85%-86%-87%-88%-
↪89%-90%-91%-92%-93%-94%-95%-96%-97%-98%-99%-OK
time
```

(continues on next page)

(continued from previous page)

```
0.0    0.002760
1.0    0.003982
2.0    0.005773
3.0    0.006265
4.0    0.009061
Name: dcg, dtype: float64

real    0m5.217s
user    0m20.329s
sys     0m3.401s
```

There are two things to note here. First, the scores are slightly worse. The reason for this is that our implementation implicitly handles cold-start user cases to some degree: we predict the score for the nonexistent previous item with id -1, which basically learns to predict based on item popularity. The builtin model doesn't do this - but this effect is only significant in the very beginning of usual data timelines (and is achievable via model combination using builtin models).

The second thing to note is speed: the builtin experiment runs about 35x faster. This is in part due to the fact that it's implemented in C++ rather than Python - but also due to the fact that it implements something called a *ranking score iterator*. We'll learn more about this in the next section.

Note: The first time an item is seen in the timeline, it is always because a user just interacted with it for the first time, thus we know that it is in fact a positive sample. If the model for some reason gives higher scores for new items, this could lead to misleading results. In our experience, unfortunately, this happens sometimes unintentionally. To avoid it, the first time an item is seen, the system always returns zero for the ranking. It is thus not possible right now to evaluate completely cold-start item situations. An optional flag is planned for future versions of Alpenglow to selectively re-allow evaluating these records.

11.1 Speeding up the evaluation: ranking iterators

One way to learn about ranking iterators is to read *Rank computation optimization*. However, let's do a quick recap here as well.

When Alpenglow evaluates a record in the timeline, first it asks the model for a prediction for the given (user, item) pair. Then, to determine the rank of the positive item, it starts asking the model for predictions for other items and counts larger, smaller and equal scores. When the number of larger scores is more than the given top K value we are evaluation for, this process stops: the positive item is not on the toplist. This method has the advantage that it is usually much faster than evaluating on all items.

However, it can be made even faster: the model may be able to give hints about items with larger scores, so that the evaluation might stop faster. This can be done in Python models as well, by defining a *prediction_iterator* method. Let's see an example of this:

```
class TransitionProbabilityModel(SelfUpdatingModel):
    def __init__(self):
        super(TransitionProbabilityModel, self).__init__()
        self.last_item = defaultdict(lambda: -1)
        self.transitions = defaultdict(lambda: 0)
        self.nonzero_transitions = defaultdict(lambda: set())
        self.itemset = set()
```

(continues on next page)

(continued from previous page)

```
    def update(self, rec_dat):
        self.transitions[(self.last_item[rec_dat.user], rec_dat.item)] += 1
        self.nonzero_transitions[self.last_item[rec_dat.user]].add(rec_dat.item)
        self.last_item[rec_dat.user] = rec_dat.item
        self.itemset.add(rec_dat.item)

    def prediction(self, rec_dat):
        return self.transitions[(self.last_item[rec_dat.user], rec_dat.item)]

    def prediction_iterator(self, user, bound):
        nonzero_pred_items = self.nonzero_transitions[self.last_item[user]]
        for i in self.nonzero_transitions[self.last_item[user]]:
            yield (i, self.transitions[(self.last_item[user], i)])

        remaining_items = self.itemset - nonzero_pred_items
        for i in remaining_items:
            if bound() > 0:
                break
            yield (i, 0)
```

The main difference from the previous one is the fact that our model now has an additional method, which is actually a generator. This iterates over all of the items that the model is aware of and produces item-score tuples. However, the items with nonzero scores are listed first.

There's one more very important part: the bound parameter of the method. This receives a function that always returns the score under which we are no longer interested in listing the items. I.e. if the bound is 1.0 and somehow we can guarantee that all the remaining items have a score below 1.0, we can stop iterating. When simply running an experiment this stays constant - the score of the positive item. However, in other cases, such as when the toplists are actually calculated, it may change based on the progress of the calculation.

We could further optimize this function by first sorting the nonzero transitions, but the above implementation already achieves a significant speedup:

```
:$ time python first_example.py
running experiment...
0%-1%-2%-3%-4%-5%-6%-7%-8%-9%-10%-11%-12%-13%-14%-15%-16%-17%-18%-19%-20%-21%-22%-23%-24
↪%-25%-26%-27%-28%-29%-30%-31%-32%-33%-34%-35%-36%-37%-38%-39%-40%-41%-42%-43%-44%-45%-
↪46%-47%-48%-49%-50%-51%-52%-53%-54%-55%-56%-57%-58%-59%-60%-61%-62%-63%-64%-65%-66%-67
↪%-68%-69%-70%-71%-72%-73%-74%-75%-76%-77%-78%-79%-80%-81%-82%-83%-84%-85%-86%-87%-88%-
↪89%-90%-91%-92%-93%-94%-95%-96%-97%-98%-99%-OK
time
0.0    0.003903
1.0    0.004307
2.0    0.006239
3.0    0.006659
4.0    0.009002
Name: dcg, dtype: float64

real    0m12.604s
user    0m53.413s
sys     0m8.367s
```

That's a nice improvement! Of course, being able to impement an iterator can be useful in other ways as well - for

example if the model can more efficiently calculate scores for batches of items, we could first calculate a batch and then yield the scores one at a time.

Note: Sometimes the results of an experiment can slightly differ after implementing a ranking iterator. This happens because after the number of larger, smaller and equal items is calculated, the evaluator randomly chooses each equally scored item to be either under or above the positive item in the toplist. The randomness for this is consistent across runs based on the seed, but it's unfortunately not consistent between evaluation methods yet.

Warning: Not listing all the items in the iterator (or erronously stopping too soon based on the bound) could incorrectly produce higher results than it should. Please take extra care when implementing ranking iterators and try to cross-check against the unoptimized version of the same model.

11.2 Speeding up the evaluation: toplists

There's one more optional method for Python models: *get_top_list*. This is also used automatically for speeding up evaluation, and it takes preference over *prediction_iterator*. Below is an example of this.

```
class TransitionProbabilityModel(SelfUpdatingModel):
    def __init__(self):
        super(TransitionProbabilityModel, self).__init__()
        self.last_item = defaultdict(lambda: -1)
        self.transitions = defaultdict(lambda: 0)
        self.nonzero_transitions = defaultdict(lambda: set())
        self.itemset = set()

    def update(self, rec_dat):
        self.transitions[(self.last_item[rec_dat.user], rec_dat.item)] += 1
        self.nonzero_transitions[self.last_item[rec_dat.user]].add(rec_dat.item)
        self.last_item[rec_dat.user] = rec_dat.item
        self.itemset.add(rec_dat.item)

    def prediction(self, rec_dat):
        return self.transitions[(self.last_item[rec_dat.user], rec_dat.item)]

    def get_top_list(self, user, k, exclude):
        last_item = self.last_item[user]
        nonzero = self.nonzero_transitions[last_item]
        nonzero_tuples = [(i, self.transitions[(last_item, i)]) for i in nonzero if not␣
↪i in exclude]
        sorted_nonzero = sorted(nonzero_tuples, key=lambda x: x[1], reverse=True)
        return sorted_nonzero[:k]
```

The idea is pretty straightforward: we implement a get_top_list method that return a list of (item, score) pairs of length k, in descending order of rank. The parameter *exclude* is used to provide the model with information about items that should be excluded from the toplist. This is used for example when `exclude_known=True`.

```
:$ time python toplist_example.py
running experiment...
0%-1%-2%-3%-4%-5%-6%-7%-8%-9%-10%-11%-12%-13%-14%-15%-16%-17%-18%-19%-20%-21%-22%-23%-24
↪%-25%-26%-27%-28%-29%-30%-31%-32%-33%-34%-35%-36%-37%-38%-39%-40%-41%-42%-43%-44%-45%-
↪46%-47%-48%-49%-50%-51%-52%-53%-54%-55%-56%-57%-58%-59%-60%-61%-62%-63%-64%-65%-66%-67
↪%-68%-69%-70%-71%-72%-73%-74%-75%-76%-77%-78%-79%-80%-81%-82%-83%-84%-85%-86%-87%-88%-
↪89%-90%-91%-92%-93%-94%-95%-96%-97%-98%-99%-OK
```

(continues on next page)

(continued from previous page)

```
time
0.0    0.003675
1.0    0.004191
2.0    0.006286
3.0    0.006221
4.0    0.009494
Name: dcg, dtype: float64

real    0m49.867s
user    1m1.892s
sys     0m4.906s
```

Faster than the first version, slower than ranking iterators. This makes sense: while ranking iterators may stop early, creating the toplist is slower as it always considers all nonzero items. Moreover, the above implementation is not optimal: we could either keep the items in a priority list for each user, or simply do an O(n) top k selection instead of sorting. Another improvement we could make is to complete the toplist when it's too short or break ties using e.g. popularity.

Note: Once again the result is different. This is, again, due to equally scored items. In toplist models, it's the responsibility of the model to handle this question correctly. Note though that the effect of equally scored items is unusually strong in case of the transition probability model, and is much less pronounced in others, such as matrix factorization.

CHAPTER

TWELVE

EVALUATING EXTERNAL MODELS

The goal of Alpenglow is to evaluate models in an on-line recommendation setting. This is primarily done through implementing the models in either C++ or Python and running them within Alpenglow experiments. However, there's also a builtin way to evaluate external models as periodically trained models in the same setting. This works by preparing an experiment that writes training files to disk, running the external model to train on these and predict toplists for given users, then run another experiment that reads these toplists back and provides them as predictions.

This logic is implemented thorugh *alpenglow.experiments.ExternalModelExperiment*. Below is an example of an experiment that prepares the training data:

```
exp = ExternalModelExperiment(
    period_length=60 * 60 * 24 * 7 * 4,
    out_name_base="batches/batch",
    mode="write"
)
```

When run, this experiment creates files such as *batches/batch_1_train.dat* and *batches/batch_1_test.dat*. The first is a CSV containing the training data, the second is a list of users that the model should generate toplists for. The predictions themselves should be saved in the file *batches/batch_1_predictions.dat* as CSV, containing 'user', 'item' and 'pos' columns. Then, the following code can be used to evaluate the results:

```
exp = ExternalModelExperiment(
    period_length=60 * 60 * 24 * 7 * 4,
    in_name_base="batches/batch",
    mode="read",
)
```

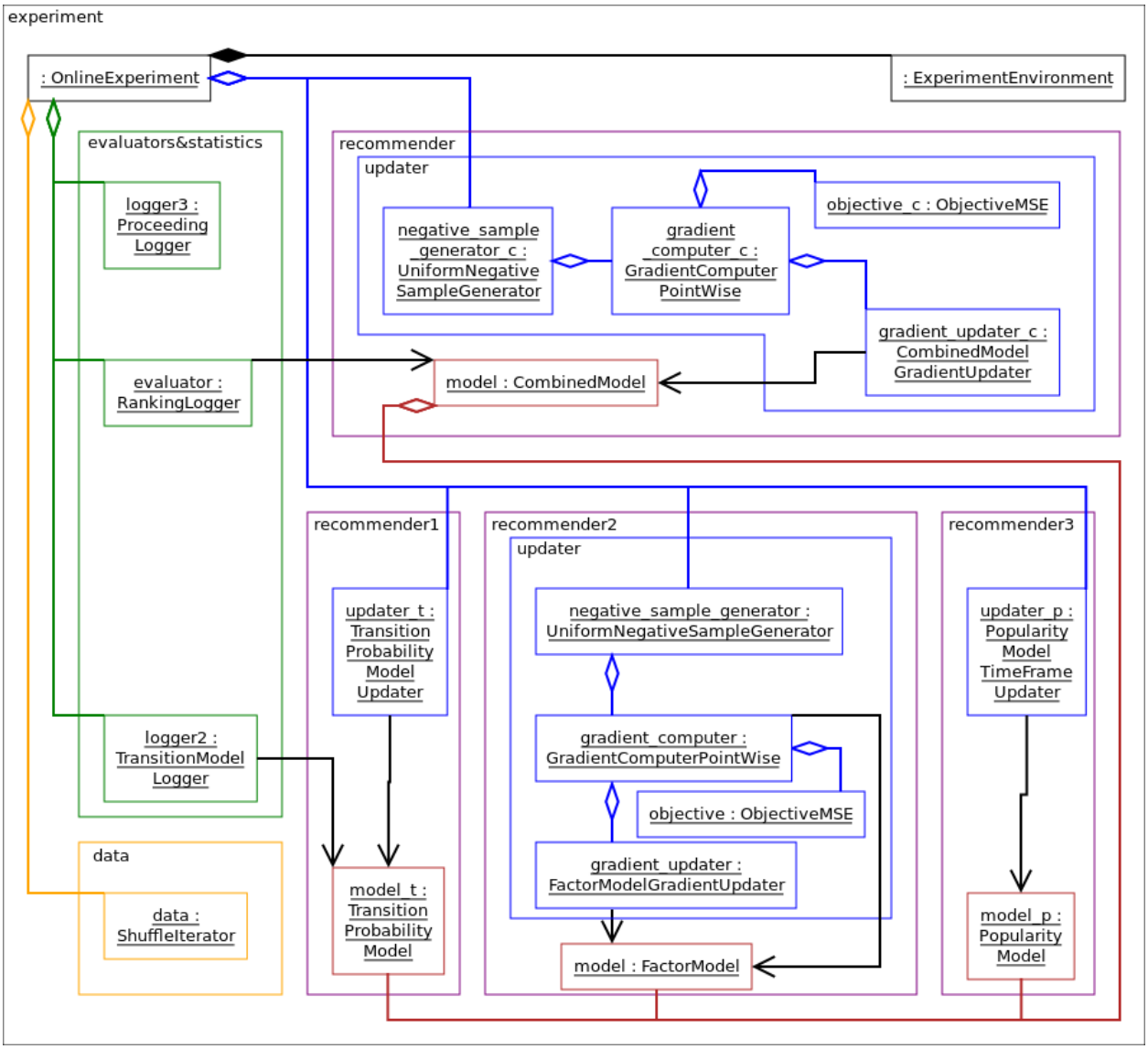
For working examples, please check out the examples/external_models directory of the repository, where this process is demonstrated through multiple examples, such as LibFM, LightFM and Turicreate.

CHAPTER

THIRTEEN

MODEL COMBINATION

While some model combination methods are implemented in Alpenglow, there are no preconfigured combined experiments. Here is an example that contains the linear combination of three models. The combination weights are trained with SGD.



The code below is quite long and building experiments this way is error-prone, but currently no graphical building tool is

implemented. The typical fault is to miss some `add_xxxx()` or `set_xxxx()`. Sometimes the result is blatantly invalid and catched by the `self_test()` call (see the last few lines). However, sometimes you can end up with hard-to-debug segfaults or invalid results.

The order of `online_experiment.add_updater()` calls is important. In the updating phase, the order of `update()` calls is indentical to the order here. This way the combination weights are updated first, then the individual models.

```
from alpenglow.Getter import Getter as cpp
import alpenglow
import pandas as pd


cpp.collect() #see general/memory usage

#data
data_python = pd.read_csv("http://info.ilab.sztaki.hu/~fbobee/alpenglow/alpenglow_sample_
↪dataset", nrows=2000)
data_cpp_bridge = alpenglow.DataframeData(data_python)
data = cpp.ShuffleIterator(seed=12345)
data.set_recommender_data(data_cpp_bridge)

#recommender1: model+updater
model1 = cpp.TransitionProbabilityModel()
updater1 = cpp.TransitionProbabilityModelUpdater(
  mode="normal"
)
updater1.set_model(model1)

#recommender3: model+updater
model3 = cpp.PopularityModel()
updater3 = cpp.PopularityTimeFrameModelUpdater(
    tau = 86400
)
updater3.set_model(model3)

#recommender2:
model2 = cpp.FactorModel(
  dimension = 10,
  begin_min = -0.1,
  begin_max = 0.1
)
negative_sample_generator_f = cpp.UniformNegativeSampleGenerator(
  negative_rate = 10
)
gradient_computer_f = cpp.GradientComputerPointWise()
gradient_computer_f.set_model(model2)
gradient_updater_f = cpp.FactorModelGradientUpdater(
  learning_rate = 0.08,
  regularization_rate = 0.0
)
gradient_updater_f.set_model(model2)
gradient_computer_f.add_gradient_updater(gradient_updater_f)
objective_f = cpp.ObjectiveMSE()
gradient_computer_f.set_objective(objective_f)
```

(continues on next page)

(continued from previous page)

```
#recommender: combined model
model = cpp.CombinedModel(
  log_file_name="xxx",
  log_frequency=1000000,
  use_user_weights=False
)
model.add_model(model1)
model.add_model(model2)
model.add_model(model3)
objective_c = cpp.ObjectiveMSE()
negative_sample_generator_c = cpp.UniformNegativeSampleGenerator(
  negative_rate = 10
)
gradient_computer_c = cpp.GradientComputerPointWise()
gradient_computer_c.set_model(model)
negative_sample_generator_c.add_updater(gradient_computer_c)
gradient_updater_c = cpp.CombinedDoubleLayerModelGradientUpdater(
  learning_rate = 0.05
)
gradient_computer_c.add_gradient_updater(gradient_updater_c)
gradient_computer_c.set_objective(objective_c)
gradient_updater_c.set_model(model)

#loggers: evaluation&statistics
logger1 = cpp.MemoryRankingLogger(
    memory_log = True
)
logger1.set_model(model)
ranking_logs = cpp.RankingLogs()
ranking_logs.top_k = 100
logger1.set_ranking_logs(ranking_logs)
logger2 = cpp.TransitionModelLogger(
    toplist_length_logfile_basename = "test",
    timeline_logfile_name = "log",
    period_length = 100000
)
logger2.set_model(model1)
logger3 = cpp.ProceedingLogger()

#online_experiment
#Class experiment_environment is created inside.
online_experiment = cpp.OnlineExperiment(
    random_seed=12345,
    top_k=100,
    exclude_known=True,
    initialize_all=False
)
online_experiment.add_logger(logger1)
online_experiment.add_logger(logger2)
online_experiment.add_logger(logger3)
online_experiment.add_updater(negative_sample_generator_c) #this will be called first
```

(continues on next page)

(continued from previous page)

```
online_experiment.add_updater(updater1)
online_experiment.add_updater(negative_sample_generator_f)
online_experiment.add_updater(updater3)
online_experiment.set_recommender_data_iterator(data)

#clean, initialize, test (see general/cpp api)
objects = cpp.get_and_clean()
cpp.set_experiment_environment(online_experiment, objects)
cpp.initialize_all(objects)
for i in objects:
    cpp.run_self_test(i)

#run the experiment
online_experiment.run()

result = logger1.get_ranking_logs()
```

CHAPTER

FOURTEEN

SERIALIZATION

Serialization is parially implemented in the Alpenglow framework. See the code samples below to discover the current serialization possibilities.

14.1 Interfaces for serialization

Many C++ classes have *write(ostream& file)* and *read(istream& file)* functions for serialization. However, these functions are not available directly through the python interface, and also might left unimplemented by some classes (throwing exceptions).

In case of `alpenglow.cpp.Model`, one can use *write(std::string file_name)* and *read(std::string file_name)*.

14.2 Serialization of periodically retrained models in the online framework

Use the parameters *write_model=True* and *base_out_file_name* to write trained models in `alpenglow.experiments.BatchFactorExperiment` to disk. See the example below. Note that the model output directory (*/path/to/your/output/dir/models/* in the example) must exist or no models will be written out. The model files will be numbered (e.g. *model_1*, *model_2* etc. in the example).

```
from alpenglow.experiments import BatchFactorExperiment

data = "/path/to/your/data"
out_dir = "/path/to/your/output/dir/"

factor_model_experiment = BatchFactorExperiment(
    out_file=out_dir+"/output_legacy_format",
    top_k=100,
    seed=254938879,
    dimension=10,
    write_model=True,
    base_out_file_name=out_dir+"/models/model",
    learning_rate=0.03,
    number_of_iterations=10,
    period_length=100000,
    period_mode="samplenum",
    negative_rate=30
)
```

(continues on next page)

(continued from previous page)

```
rankings = factor_model_experiment.run(
    data, exclude_known=True, experimentType="online_id")
```

You can read back your models using the same class, changing the parameters. Note that the model size parameters (*dimension*, *period_length*, *period_mode*) must agree. However, the training parameters (*learning_rate*, *negative_rate*, *number_of_iterations*) may be omitted if *learn* is set to *False*.

```
from alpenglow.experiments import BatchFactorExperiment

data = "/path/to/your/data"
out_dir = "/path/to/your/output/dir/"

factor_model_experiment = BatchFactorExperiment(
    out_file=out_dir+"/output_legacy_format",
    top_k=100,
    seed=254938879,
    dimension=10,
    learn=False,
    read_model=True,
    base_in_file_name=out_dir+"/models/model",
    period_length=100000,
    period_mode="samplenum"
)

rankings = factor_model_experiment.run(
    data, exclude_known=True, experimentType="online_id")
```

Alternatively, one could read back the models using *alpenglow.experiments.BatchAndOnlineFactorExperiment* and apply online updates on top of the pretrained batch models.

14.3 Serialization in offline experiments

See the example below:

```
import pandas as pd
from alpenglow.offline.models import FactorModel
import alpenglow.Getter as rs

data = pd.read_csv(
    "/path/to/your/data",
    sep=' ',
    header=None,
    names=['time', 'user', 'item', 'id', 'score', 'eval']
)
model = FactorModel(
    factor_seed=254938879,
    dimension=10,
    negative_rate=9,
    number_of_iterations=20,
)
```

(continues on next page)

(continued from previous page)

```
model.fit(data)

model.model.write("output_file") #writes model to output_file
rd = rs.RecDat()
rd.user = 3
rd.item = 5
print("prediction for user=3, item=5:", model.model.prediction(rd))

#model2 must have the same dimension
model2 = FactorModel(
    factor_seed=1234,
    dimension=10,
    negative_rate=0,
    number_of_iterations=0,
)
#to create the inner model but avoid training, we need to run fit()
#on an empty dataset
data2=pd.DataFrame(columns=['time', 'user', 'item'])
model2.fit(data2)
model2.model.read("output_file") #reads back the same model
print("prediction for user=3, item=5 using the read-back model:",
    model2.model.prediction(rd))
```

CHAPTER

FIFTEEN

COMPILING ALPENGLOW USING CLANG AND LIBC++ ON LINUX

If you wish to compile Alpenglow using clang and libc++ instead of g++ and libstdc++ (for example to have the same behavior on linux and MacOS), you can do that with a few simple changes.

First, you need to set the CC environment variable to "clang++" by using the command `export CC="clang++"`. Next, you need to make the following changes to the "setup.py" file in the root directory of the package:

1. Under the platform specific flags for linux, add

```
'-stdlib=libc++',
'-mfpmath=sse',
```

and remove

```
'-mfpmath=sse,387',
```

2. Add the following parameter to the 'alpenglow.cpp' extension in the call to the 'setup' function:

```
extra_link_args=[
    '-nodefaultlibs',
    '-nostdinc++',
    '-lc++',
    '-lm',
    '-lc',
    '-lgcc_s',
    '-lpthread',
    '-lgcc',
],
```

Now reinstall using `pip install --force-reinstall --no-deps --upgrade .`. You can check if you were successful by running the following code:

```
import alpenglow
print(alpenglow.cpp.__compiler, alpenglow.cpp.__stdlib)
# expected output: clang libc++
```

CHAPTER

SIXTEEN

ALPENGLOW PACKAGE

16.1 Subpackages

16.1.1 alpenglow.evaluation package

Submodules

alpenglow.evaluation.DcgScore module

alpenglow.evaluation.DcgScore.**Dcg**(*rank*)

alpenglow.evaluation.DcgScore.**DcgScore**(*rankings*)

alpenglow.evaluation.MseScore module

alpenglow.evaluation.MseScore.**MseScore**(*rankings*)

alpenglow.evaluation.PrecisionScore module

alpenglow.evaluation.PrecisionScore.**Precision**(*rank*)

alpenglow.evaluation.PrecisionScore.**PrecisionScore**(*rankings*)

alpenglow.evaluation.RecallScore module

alpenglow.evaluation.RecallScore.**Recall**(*rank*, *top_k*)

alpenglow.evaluation.RecallScore.**RecallScore**(*rankings*, *top_k=None*)

alpenglow.evaluation.RrScore module

alpenglow.evaluation.RrScore.**Rr**(*rank*)

alpenglow.evaluation.RrScore.**RrScore**(*rankings*)

Reciprocial rank, see https://en.wikipedia.org/wiki/Mean_reciprocal_rank .

Module contents

16.1.2 alpenglow.experiments package

Submodules

alpenglow.experiments.ALSFactorExperiment module

class alpenglow.experiments.ALSFactorExperiment.**ALSFactorExperiment**(*dimension=10*, *begin_min=- 0.01*, *begin_max=0.01*, *number_of_iterations=15*, *regularization_lambda=1e-3*, *alpha=40*, *implicit=1*, *clear_before_fit=1*, *period_length=86400*)

Bases: *alpenglow.OnlineExperiment.OnlineExperiment*

This class implements an online version of the well-known matrix factorization recommendation model [Koren2009] and trains it via Alternating Least Squares in a periodic fashion. The model is able to train on explicit data using traditional ALS, and on implicit data using the iALS algorithm [Hu2008].

Parameters

- **dimension** (*int*) – The latent factor dimension of the factormodel.
- **begin_min** (*double*) – The factors are initialized randomly, sampling each element uniformly from the interval (begin_min, begin_max).
- **begin_max** (*double*) – See begin_min.
- **number_of_iterations** (*int*) – The number of ALS iterations to perform in each period.
- **regularization_lambda** (*double*) – The coefficient for the L2 regularization term. See [Hu2008]. This number is multiplied by the number of non-zero elements of the user-item rating matrix before being used, to achieve similar magnitude to the one used in traditional SGD.
- **alpha** (*int*) – The weight coefficient for positive samples in the error formula. See [Hu2008].
- **implicit** (*int*) – Valued 1 or 0, indicating whether to run iALS or ALS.
- **clear_before_fit** (*int*) – Whether to reset the model after each period.
- **period_length** (*int*) – The period length in seconds.
- **timeframe_length** (*int*) – The size of historic time interval to iterate over at every batch model retrain. Leave at the default 0 to retrain on everything.

alpenglow.experiments.ALSOnlineFactorExperiment module

class alpenglow.experiments.ALSOnlineFactorExperiment.**ALSOnlineFactorExperiment**(*dimension=10*, *begin_min=-0.01*, *begin_max=0.01*, *number_of_iterations=15*, *regularization_lambda=1e-3*, *alpha=40*, *implicit=1*, *clear_before_fit=1*, *period_length=86400*)

Bases: *alpenglow.OnlineExperiment.OnlineExperiment*

Combines ALSFactorExperiment and FactorExperiment by updating the model periodically with ALS and continously with SGD.

Parameters

- **dimension** (*int*) – The latent factor dimension of the factormodel.
- **begin_min** (*double*) – The factors are initialized randomly, sampling each element uniformly from the interval (begin_min, begin_max).
- **begin_max** (*double*) – See begin_min.
- **number_of_iterations** (*double*) – Number of times to optimize the user and the item factors for least squares.
- **regularization_lambda** (*double*) – The coefficient for the L2 regularization term. See [Hu2008]. This number is multiplied by the number of non-zero elements of the user-item rating matrix before being used, to achieve similar magnitude to the one used in traditional SGD.
- **alpha** (*int*) – The weight coefficient for positive samples in the error formula. See [Hu2008].
- **implicit** (*int*) – Valued 1 or 0, indicating whether to run iALS or ALS.
- **clear_before_fit** (*int*) – Whether to reset the model after each period.
- **period_length** (*int*) – The period length in seconds.
- **timeframe_length** (*int*) – The size of historic time interval to iterate over at every batch model retrain. Leave at the default 0 to retrain on everything.
- **online_learning_rate** (*double*) – The learning rate used in the online stochastic gradient descent updates.
- **online_regularization_rate** (*double*) – The coefficient for the L2 regularization term for online update.
- **online_negative_rate** (*int*) – The number of negative samples generated after online each update. Useful for implicit recommendation.

alpenglow.experiments.AsymmetricFactorExperiment module

class alpenglow.experiments.AsymmetricFactorExperiment.**AsymmetricFactorExperiment**(*dimension=10*, *begin_min=-0.01*, *begin_max=0.01*, *learning_rate=0.05*, *regularization_rate=0.0*, *negative_rate=20*, *cumulative_item_updates=True*, *norm_type='exponential'*, *gamma=0.8*)

Bases: *alpenglow.OnlineExperiment.OnlineExperiment*

Implements the recommendation model introduced in [Koren2008].

Parameters

- **dimension** (*int*) – The latent factor dimension of the factormodel.
- **begin_min** (*double*) – The factors are initialized randomly, sampling each element uniformly from the interval (begin_min, begin_max).
- **begin_max** (*double*) – See begin_min.
- **learning_rate** (*double*) – The learning rate used in the stochastic gradient descent updates.
- **regularization_rate** (*double*) – The coefficient for the L2 regularization term.
- **negative_rate** (*int*) – The number of negative samples generated after each update. Useful for implicit recommendation.
- **norm_type** (*str*) – Type of time decay; either “constant”, “exponential” or “disabled”.
- **gamma** (*double*) – Coefficient of time decay in the case of **norm_type** == “exponential”.

alpenglow.experiments.BatchAndOnlineFactorExperiment module

class alpenglow.experiments.BatchAndOnlineFactorExperiment.**BatchAndOnlineFactorExperiment**(*dimension=10*, *begin_min=-0.01*, *begin_max=0.01*, *batch_learning_r batch_regulariza batch_negative_r online_learning_ra online_regularizatio online_negative_rat period_length=864*

Bases: *alpenglow.OnlineExperiment.OnlineExperiment*

Combines BatchFactorExperiment and FactorExperiment by updating the model both in batch and continously.

Parameters

- **dimension** (*int*) – The latent factor dimension of the factormodel.
- **begin_min** (*double*) – The factors are initialized randomly, sampling each element uniformly from the interval (begin_min, begin_max).
- **begin_max** (*double*) – See begin_min.
- **batch_learning_rate** (*double*) – The learning rate used in the batch stochastic gradient descent updates.
- **batch_regularization_rate** (*double*) – The coefficient for the L2 regularization term for batch updates.
- **batch_negative_rate** (*int*) – The number of negative samples generated after each batch update. Useful for implicit recommendation.
- **timeframe_length** (*int*) – The size of historic time interval to iterate over at every batch model retrain. Leave at the default 0 to retrain on everything.
- **online_learning_rate** (*double*) – The learning rate used in the online stochastic gradient descent updates.
- **online_regularization_rate** (*double*) – The coefficient for the L2 regularization term for online update.
- **online_negative_rate** (*int*) – The number of negative samples generated after online each update. Useful for implicit recommendation.

alpenglow.experiments.BatchFactorExperiment module

class alpenglow.experiments.BatchFactorExperiment.**BatchFactorExperiment**(*dimension=10*, *begin_min=- 0.01*, *begin_max=0.01*, *learning_rate=0.05*, *regularization_rate=0.0*, *negative_rate=0.0*, *number_of_iterations=3*, *period_length=86400*, *timeframe_length=0*, *clear_model=False*)

Bases: *alpenglow.OnlineExperiment.OnlineExperiment*

Batch version of *alpenglow.experiments.FactorExperiment.FactorExperiment*, meaning it retrains its model periodically nd evaluates the latest model between two training points in an online fashion.

Parameters

- **dimension** (*int*) – The latent factor dimension of the factormodel.
- **begin_min** (*double*) – The factors are initialized randomly, sampling each element uniformly from the interval (begin_min, begin_max).
- **begin_max** (*double*) – See begin_min.
- **learning_rate** (*double*) – The learning rate used in the stochastic gradient descent updates.
- **regularization_rate** (*double*) – The coefficient for the L2 regularization term.
- **negative_rate** (*int*) – The number of negative samples generated after each update. Useful for implicit recommendation.
- **number_of_iterations** (*int*) – The number of iterations over the data in model retrain.
- **period_length** (*int*) – The amount of time between model retrains (seconds).
- **timeframe_length** (*int*) – The size of historic time interval to iterate over at every model retrain. Leave at the default 0 to retrain on everything.
- **clear_model** (*bool*) – Whether to clear the model between retrains.

alpenglow.experiments.ExternalModelExperiment module

class alpenglow.experiments.ExternalModelExperiment.**ExternalModelExperiment**(*period_length=86400*, *timeframe_length=0*, *period_mode='time'*)

Bases: *alpenglow.OnlineExperiment.OnlineExperiment*

Parameters

- **period_length** (*int*) – The period length in seconds (or samples, see period_mode).
- **timeframe_length** (*int*) – The size of historic time interval to iterate over at every batch model retrain. Leave at the default 0 to retrain on everything.

- **period_mode** (*string*) – Either "time" or "samplenum", the unit of period_length and timeframe_length.

alpenglow.experiments.FactorExperiment module

class alpenglow.experiments.FactorExperiment.**FactorExperiment**(*dimension=10*, *begin_min=- 0.01*, *begin_max=0.01*, *learning_rate=0.05*, *regularization_rate=0.0*, *negative_rate=100*)

Bases: *alpenglow.OnlineExperiment.OnlineExperiment*

This class implements an online version of the well-known matrix factorization recommendation model [Koren2009] and trains it via stochastic gradient descent. The model is able to train on implicit data using negative sample generation, see [X.He2016] and the **negative_rate** parameter.

Parameters

- **dimension** (*int*) – The latent factor dimension of the factormodel.
- **begin_min** (*double*) – The factors are initialized randomly, sampling each element uniformly from the interval (begin_min, begin_max).
- **begin_max** (*double*) – See begin_min.
- **learning_rate** (*double*) – The learning rate used in the stochastic gradient descent updates.
- **regularization_rate** (*double*) – The coefficient for the L2 regularization term.
- **negative_rate** (*int*) – The number of negative samples generated after each update. Useful for implicit recommendation.

alpenglow.experiments.FmExperiment module

class alpenglow.experiments.FmExperiment.**FmExperiment**(*dimension=10*, *begin_min=- 0.01*, *begin_max=0.01*, *learning_rate=0.05*, *negative_rate=0.0*, *user_attributes=None*, *item_attributes=None*)

Bases: *alpenglow.OnlineExperiment.OnlineExperiment*

This class implements an online version of the factorization machine algorithm [Rendle2012] and trains it via stochastic gradient descent. The model is able to train on implicit data using negative sample generation, see [X.He2016] and the **negative_rate** parameter. Note that interactions between separate attributes of a user and between separate attributes of an item are not modeled.

The item and user attributes can be provided through the **user_attributes** and **item_attributes** parameters. These each expect a file path pointing to the attribute files. The required format is similar to the one used by libfm: the i. line describes the attributes of user **i** in a space sepaterated list of **index:value** pairs. For example the line "3:1 10:0.5" as the first line of the file indicates that user 0 has 1 as the value of attribute 3, and 0.5 as the value of attribute 10. If the files are omitted, an identity matrix is assumed.

Notice: once an attribute file is provided, the identity matrix is no longer assumed. If you wish to have a separate latent vector for each id, you must explicitly provide the identity matrix in the attribute file itself.

Parameters

- **dimension** (*int*) – The latent factor dimension of the factormodel.

- **begin_min** (*double*) – The factors are initialized randomly, sampling each element uniformly from the interval (begin_min, begin_max).
- **begin_max** (*double*) – See begin_min.
- **learning_rate** (*double*) – The learning rate used in the stochastic gradient descent updates.
- **negative_rate** (*int*) – The number of negative samples generated after each update. Useful for implicit recommendation.
- **user_attributes** (*string*) – The file containing the user attributes, in the format described in the model description. Set None for no attributes (identity matrix).
- **item_attributes** (*string*) – The file containing the item attributes, in the format described in the model description. Set None for no attributes (identity matrix).

alpenglow.experiments.NearestNeighborExperiment module

class alpenglow.experiments.NearestNeighborExperiment.**NearestNeighborExperiment**(*gamma=0.8*, *direction='forward'*, *gamma_threshold=0*, *num_of_neighbors=10*)

Bases: *alpenglow.OnlineExperiment.OnlineExperiment*

This class implements an online version of a similarity based recommendation model. One of the earliest and most popular collaborative filtering algorithms in practice is the item-based nearest neighbor [Sarwar2001] For these algorithms similarity scores are computed between item pairs based on the co-occurrence of the pairs in the preference of users. Non-stationarity of the data can be accounted for e.g. with the introduction of a time-decay [Ding2005] .

Describing the algorithm more formally, let us denote by U_i the set of users that visited item i, by I_u the set of items visited by user u, and by s_{ui} the index of item i in the sequence of interactions of user u. The frequency based time-weighted similarity function is defined by $sim(j,i) = \frac{\sum_{u \in U_j \cap U_i} f(s_{ui} - s_{uj})}{|U_j|}$, where $f(\tau) = \gamma^\tau$ is the time decaying function. For non-stationary data we sum only over users that visit item j before item i, setting $f(\tau) = 0$ if $\tau < 0$. For stationary data the absolute value of τ is used. The score assigned to item i for user u is $score(u,i) = \sum_{j \in I_u} f\left(|I_u| - s_{uj}\right) sim(j,i)$. The model is represented by the similarity scores. Since computing the model is time consuming, it is done periodically. Moreover, only the most similar items are stored for each item. When the prediction scores are computed for a particular user, all items visited by the user can be considered, including the most recent ones. Hence, the algorithm can be considered semi-online in that it uses the most recent interactions of the current user, but not of the other users. We note that the time decay function is used here to quantify the strength of connection between pairs of items depending on how closely are located in the sequence of a user, and not as a way to forget old data as in [Ding2005].

Parameters

- **gamma** (*double*) – The constant used in the decay function. It shoud be set to 1 in offline and stationary experiments.
- **direction** (*string*) – Set to "forward" to consider the order of item pairs. Set to "both" when the order is not relevant.
- **gamma_thresold** (*double*) – Threshold to omit very small members when summing similarity. If the value of the decay function is smaller than the threshold, we omit the following members. Defaults to 0 (do not omit small members).
- **num_of_neighbors** (*int*) – The number of most similar items that will be stored in the model.

alpenglow.experiments.OldFactorExperiment module

class alpenglow.experiments.OldFactorExperiment.**OldFactorExperiment**(*dimension=10*, *begin_min=- 0.01*, *begin_max=0.01*, *learning_rate=0.05*, *regularization_rate=0.0*, *negative_rate=0.0*)

Bases: *alpenglow.OnlineExperiment.OnlineExperiment*

Deprecated, use FactorExperiment. This class implements an online version of the well-known matrix factorization recommendation model [Koren2009] and trains it via stochastic gradient descent. The model is able to train on implicit data using negative sample generation, see [X.He2016] and the **negative_rate** parameter.

Parameters

- **dimension** (*int*) – The latent factor dimension of the factormodel.
- **begin_min** (*double*) – The factors are initialized randomly, sampling each element uniformly from the interval (begin_min, begin_max).
- **begin_max** (*double*) – See begin_min.
- **learning_rate** (*double*) – The learning rate used in the stochastic gradient descent updates.
- **regularization_rate** (*double*) – The coefficient for the L2 regularization term.
- **negative_rate** (*int*) – The number of negative samples generated after each update. Useful for implicit recommendation.

alpenglow.experiments.PersonalPopularityExperiment module

class alpenglow.experiments.PersonalPopularityExperiment.**PersonalPopularityExperiment**(***parameters*)

Bases: *alpenglow.OnlineExperiment.OnlineExperiment*

Recommends the item that the user has watched the most so far; in case of a tie, it falls back to global popularity. Running this model in conjunction with **exclude_known** == True is not recommended.

alpenglow.experiments.PopularityExperiment module

class alpenglow.experiments.PopularityExperiment.**PopularityExperiment**(***parameters*)

Bases: *alpenglow.OnlineExperiment.OnlineExperiment*

Recommends the most popular item from the set of items seen so far.

alpenglow.experiments.PopularityTimeframeExperiment module

class alpenglow.experiments.PopularityTimeframeExperiment.**PopularityTimeframeExperiment**(*tau=86400*)

Bases: *alpenglow.OnlineExperiment.OnlineExperiment*

Time-aware version of PopularityModel, which only considers the last **tau** time interval when calculating popularities. Note that the time window ends at the timestamp of the last updating sample. The model does not take into consideration the timestamp of the sample for that the prediction is computed.

Parameters **tau** (*int*) – The time amount to consider.

alpenglow.experiments.PosSamplingFactorExperiment module

class alpenglow.experiments.PosSamplingFactorExperiment.**PosSamplingFactorExperiment**(*dimension=10*, *begin_min=-0.01*, *begin_max=0.01*, *base_learning_rate=0.2*, *base_regularization_rate*, *positive_learning_rate=0.05*, *positive_regularization_rate=*, *negative_rate=40*, *positive_rate=3*, *pool_size=3000*)

Bases: *alpenglow.OnlineExperiment.OnlineExperiment*

This model implements an online, efficient technique that approximates the learning method of *alpenglow.experiments.BatchAndOnlineFactorExperiment*, using fewer update steps. Similarly to the online MF model (*alpenglow.experiments.FactorExperiment*), we only use a single iteration for the model in a temporal order. However, for each interaction, we generate not only negative but also positive samples. The positive samples are selected randomly from past interactions, i.e. we allow the model to re-learn the past. We generate *positive_rate* positive samples along with *negative_rate* negative samples, hence for t events, we only take $(1+negative\_rate+positive\_rate)\cdot t$ gradient steps.

The samples are not drawn uniformly from the past, but selected randomly from pool S with maximum size *pool_size*. This avoids oversampling interactions that happened at the beginning of the data set. More specifically, for each observed new training instance, we

- update the model by positive_rate samples from pool S,
- delete the selected samples from pool S if it already reached *pool_size*,
- and add the new instance *positive_rate* times to the pool.

For more details, see [frigo2017online] .

Parameters

- **dimension** (*int*) – The latent factor dimension of the factormodel.
- **begin_min** (*double*) – The factors are initialized randomly, sampling each element uniformly from the interval (begin_min, begin_max).
- **begin_max** (*double*) – See begin_min.
- **negative_rate** (*int*) – The number of negative samples generated after each update. Useful for implicit recommendation.
- **positive_rate** (*int*) – The number of positive samples generated for each update.
- **pool_size** (*int*) – The size of pool for generating positive samples. See the article for details.
- **base_learning_rate** (*double*) – The learning rate used in the stochastic gradient descent updates for the original positive sample and the generated negative samples.
- **base_regularization_rate** (*double*) – The coefficient for the L2 regularization term.

- **positive_learning_rate** (*double*) – The learning rate used in the stochastic gradient descent updates for the generated positive samples.
- **positive_regularization_rate** (*double*) – The coefficient for the L2 regularization term.

alpenglow.experiments.SvdppExperiment module

class alpenglow.experiments.SvdppExperiment.**SvdppExperiment**(*begin_min=- 0.01*, *begin_max=0.01*, *dimension=10*, *learning_rate=0.05*, *negative_rate=20*, *use_sigmoid=False*, *norm_type='exponential'*, *gamma=0.8*, *user_vector_weight=0.5*, *history_weight=0.5*)

Bases: *alpenglow.OnlineExperiment.OnlineExperiment*

This class implements an online version of the SVD++ model [Koren2008] The model is able to train on implicit data using negative sample generation, see [X.He2016] and the **negative_rate** parameter. We apply a decay on the user history, the weight of the older items is smaller.

Parameters

- **begin_min** (*double*) – The factors are initialized randomly, sampling each element uniformly from the interval (begin_min, begin_max).
- **begin_max** (*double*) – See begin_min.
- **dimension** (*int*) – The latent factor dimension of the factormodel.
- **learning_rate** (*double*) – The learning rate used in the stochastic gradient descent updates.
- **negative_rate** (*int*) – The number of negative samples generated after each update. Useful for implicit recommendation.
- **norm_type** (*string*) – Normalization variants.
- **gamma** (*double*) – The constant in the decay function.
- **user_vector_weight** (*double*) – The user is modeled with a sum of a user vector and a combination of item vectors. The weight of the two part can be set using these parameters.
- **history_weight** (*double*) – See user_vector_weight.

alpenglow.experiments.TransitionProbabilityExperiment module

class alpenglow.experiments.TransitionProbabilityExperiment.**TransitionProbabilityExperiment**(*mode='normal*

Bases: *alpenglow.OnlineExperiment.OnlineExperiment*

A simple algorithm that focuses on the sequence of items a user has visited is one that records how often users visited item i after visiting another item j. This can be viewed as particular form of the item-to-item nearest neighbor with a time decay function that is non-zero only for the immediately preceding item. While the algorithm is more simplistic, it is fast to update the transition fre- quencies after each interaction, thus all recent information is taken into account.

Parameters **mode** (*string*) – The direction of transitions to be considered, possible values: normal, inverted, symmetric.

Module contents

16.1.3 alpenglow.offline package

Subpackages

alpenglow.offline.evaluation package

Submodules

alpenglow.offline.evaluation.NdcgScore module

alpenglow.offline.evaluation.NdcgScore.**NdcgScore**(*test*, *recommendations*, *top_k=100*)

alpenglow.offline.evaluation.PrecisionScore module

alpenglow.offline.evaluation.PrecisionScore.**PrecisionScore**(*test*, *recommendations*, *top_k*)

alpenglow.offline.evaluation.RecallScore module

alpenglow.offline.evaluation.RecallScore.**RecallScore**(*test*, *recommendations*, *top_k*)

Module contents

alpenglow.offline.models package

Submodules

alpenglow.offline.models.ALSFactorModel module

class alpenglow.offline.models.ALSFactorModel.**ALSFactorModel**(*dimension=10*, *begin_min=- 0.01*, *begin_max=0.01*, *number_of_iterations=3*, *regularization_lambda=0.0001*, *alpha=40*, *implicit=1*)

Bases: *alpenglow.offline.OfflineModel.OfflineModel*

This class implements the well-known matrix factorization recommendation model [Koren2009] and trains it using ALS and iALS [Hu2008].

Parameters

- **dimension** (*int*) – The latent factor dimension of the factormodel.
- **begin_min** (*double*) – The factors are initialized randomly, sampling each element uniformly from the interval (begin_min, begin_max).
- **begin_max** (*double*) – See begin_min.
- **number_of_iterations** (*double*) – Number of times to optimize the user and the item factors for least squares.

- **regularization_lambda** (*double*) – The coefficient for the L2 regularization term. See [Hu2008]. This number is multiplied by the number of non-zero elements of the user-item rating matrix before being used, to achieve similar magnitude to the one used in traditional SGD.
- **alpha** (*int*) – The weight coefficient for positive samples in the error formula in the case of implicit factorization. See [Hu2008].
- **implicit** (*int*) – Whether to treat the data as implicit (and optimize using iALS) or explicit (and optimize using ALS).

alpenglow.offline.models.AsymmetricFactorModel module

class alpenglow.offline.models.AsymmetricFactorModel.**AsymmetricFactorModel**(*dimension=10*, *begin_min=- 0.01*, *begin_max=0.01*, *learning_rate=0.05*, *regularization_rate=0.0*, *negative_rate=0*, *number_of_iterations=9*)

Bases: *alpenglow.offline.OfflineModel.OfflineModel*

Implements the recommendation model introduced in [Paterek2007].

Parameters

- **dimension** (*int*) – The latent factor dimension of the factormodel.
- **begin_min** (*double*) – The factors are initialized randomly, sampling each element uniformly from the interval (begin_min, begin_max).
- **begin_max** (*double*) – See begin_min.
- **learning_rate** (*double*) – The learning rate used in the stochastic gradient descent updates.
- **regularization_rate** (*double*) – The coefficient for the L2 regularization term.
- **negative_rate** (*int*) – The number of negative samples generated after each update. Useful for implicit recommendation.
- **number_of_iterations** (*int*) – Number of times to iterate over the training data.

alpenglow.offline.models.FactorModel module

class alpenglow.offline.models.FactorModel.**FactorModel**(*dimension=10*, *begin_min=- 0.01*, *begin_max=0.01*, *learning_rate=0.05*, *regularization_rate=0.0*, *negative_rate=0.0*, *number_of_iterations=9*)

Bases: *alpenglow.offline.OfflineModel.OfflineModel*

This class implements the well-known matrix factorization recommendation model [Koren2009] and trains it via stochastic gradient descent. The model is able to train on implicit data using negative sample generation, see [X.He2016] and the **negative_rate** parameter.

Parameters

- **dimension** (*int*) – The latent factor dimension of the factormodel.
- **begin_min** (*double*) – The factors are initialized randomly, sampling each element uniformly from the interval (begin_min, begin_max).
- **begin_max** (*double*) – See begin_min.
- **learning_rate** (*double*) – The learning rate used in the stochastic gradient descent updates.
- **regularization_rate** (*double*) – The coefficient for the L2 regularization term.
- **negative_rate** (*int*) – The number of negative samples generated after each update. Useful for implicit recommendation.
- **number_of_iterations** (*int*) – Number of times to iterate over the training data.

alpenglow.offline.models.NearestNeighborModel module

class alpenglow.offline.models.NearestNeighborModel.**NearestNeighborModel**(*num_of_neighbors=10*)

Bases: *alpenglow.offline.OfflineModel.OfflineModel*

One of the earliest and most popular collaborative filtering algorithms in practice is the item-based nearest neighbor [Sarwar2001] For these algorithms similarity scores are computed between item pairs based on the co-occurrence of the pairs in the preference of users. Non-stationarity of the data can be accounted for e.g. with the introduction of a time-decay [Ding2005] .

Describing the algorithm more formally, let us denote by U_i the set of users that visited item i, by I_u the set of items visited by user u, and by s_{ui} the index of item i in the sequence of interactions of user u. The frequency based similarity function is defined by $sim(j,i) = \frac{\sum_{u \in U_j \cap U_i} 1}{|U_j|}$. The score assigned to item i for user u is $score(u,i) = \sum_{j \in I_u} sim(j,i)$. The model is represented by the similarity scores. Only the most similar items are stored for each item. When the prediction scores are computed for a particular user, all items visited by the user are considered.

Parameters **num_of_neighbors** (*int*) – Number of most similar items that will be stored in the model.

alpenglow.offline.models.PopularityModel module

class alpenglow.offline.models.PopularityModel.**PopularityModel**

Bases: *alpenglow.offline.OfflineModel.OfflineModel*

Recommends the most popular item from the set of items.

alpenglow.offline.models.SvdppModel module

class alpenglow.offline.models.SvdppModel.**SvdppModel**(*dimension=10*, *begin_min=- 0.01*, *begin_max=0.01*, *learning_rate=0.05*, *negative_rate=0.0*, *number_of_iterations=20*, *cumulative_item_updates=false*)

Bases: *alpenglow.offline.OfflineModel.OfflineModel*

This class implements the SVD++ model [Koren2008] The model is able to train on implicit data using negative sample generation, see [X.He2016] and the **negative_rate** parameter.

Parameters

- **dimension** (*int*) – The latent factor dimension of the factormodel.
- **begin_min** (*double*) – The factors are initialized randomly, sampling each element uniformly from the interval (begin_min, begin_max).
- **begin_max** (*double*) – See begin_min.
- **learning_rate** (*double*) – The learning rate used in the stochastic gradient descent updates.
- **negative_rate** (*int*) – The number of negative samples generated after each update. Useful for implicit recommendation.
- **number_of_iterations** (*int*) – Number of times to iterate over the training data.
- **cumulative_item_updates** (*boolean*) – Cumulative item updates make the model faster but less accurate.

Module contents

Submodules

alpenglow.offline.OfflineModel module

class alpenglow.offline.OfflineModel.**OfflineModel**(***parameters*)

Bases: *alpenglow.ParameterDefaults.ParameterDefaults*

OfflineModel is the base class for all traditional, scikit-learn style models in Alpenglow. Example usage:

```
data = pd.read_csv('data')
train_data = data[data.time < (data.time.min()+250*86400)]
test_data = data[ (data.time >= (data.time.min()+250*86400)) & (data.time < (data.
↪time.min()+300*86400))]

exp = ag.offline.models.FactorModel(
    learning_rate=0.07,
    negative_rate=70,
    number_of_iterations=9,
)
exp.fit(data)
test_users = list(set(test_data.user)&set(train_data.user))
recommendations = exp.recommend(users=test_users)
```

fit(*X*, *y=None*, *columns={}*)

Fit the model to a dataset.

Parameters

- **X** (*pandas.DataFrame*) – The input data, must contain the columns **user** and **item**. May contain the **score** column as well.
- **y** (*pandas.Series or list*) – The target values. If not set (and X doesn't contain the score column), it is assumed to be constant 1 (implicit recommendation).
- **columns** (*dict*) – Optionally the mapping of the input DataFrame's columns' names to the expected ones.

predict(*X*)
Predict the target values on X.

Parameters **X** (*pandas.DataFrame*) – The input data, must contain the columns **user** and **item**.

Returns List of predictions

Return type list

recommend(*users=None*, *k=100*, *exclude_known=True*)
Give toplist recommendations for users.

Parameters

- **users** (*list*) – List of users to give recommendation for.
- **k** (*int*) – Size of toplists
- **exclude_known** (*bool*) – Whether to exclude (user,item) pairs in the train dataset from the toplists.

Returns DataFrame of recommendations, with columns **user**, **item** and **rank**.

Return type pandas.DataFrame

Module contents

16.1.4 alpenglow.utils package

Submodules

alpenglow.utils.AvailabilityFilter module

class alpenglow.utils.AvailabilityFilter.**AvailabilityFilter**(*availability_data*)
Bases: *alpenglow.cpp.AvailabilityFilter*

Python wrapper around *alpenglow.cpp.AvailabilityFilter*.

alpenglow.utils.DataShuffler module

class alpenglow.utils.DataShuffler.**DataShuffler**(*seed=254938879*, *shuffle_mode=complete*, *input_file*, *output_file*)
Bases: *alpenglow.ParameterDefaults.ParameterDefaults*

This class is for shuffling datasets.

Parameters

- **seed** (*int*) – The seed to initialize RNG-s. Should not be 0.
- **shuffle_mode** (*string*) – Possible values: complete, same_timestamp.
- **input_file** (*string*) – Input file name.
- **output_file** (*string*) – Output file name.
- **data_format** (*string*) – Input file format. Available values: online, online_id, online_id_noeval, online_attribute, offline, offlineTimestamp, category. See RecommenderData.cpp for details. Default: online_id.

run()

alpenglow.utils.DataframeData module

class alpenglow.utils.DataframeData.**DataframeData**(*df*, *columns={}*)
Bases: *alpenglow.cpp.DataframeData*

Python wrapper around *alpenglow.cpp.DataframeData*.

alpenglow.utils.FactorModelReader module

alpenglow.utils.FactorModelReader.**readEigenFactorModel**(*file*)

alpenglow.utils.FactorModelReader.**readFactorModel**(*file*, *dimensions*)

alpenglow.utils.ParameterSearch module

class alpenglow.utils.ParameterSearch.**DependentParameter**(*format_string*, *parameter_names=None*)
Bases: object

eval(*parameters*)

class alpenglow.utils.ParameterSearch.**ParameterSearch**(*model*, *Score*)
Bases: object

Utility for evaluating online experiments with different hyperparameters. For a brief tutorial on using this class, see *Five minute tutorial*.

run(**run_paremeters*, ***run_kw_parameters*)

set_parameter_values(*parameter_name*, *parameter_values*)

alpenglow.utils.ThreadedParameterSearch module

class alpenglow.utils.ThreadedParameterSearch.**ThreadedParameterSearch**(*model*, *Score*, *threads=4*, *use_process_pool=True*)
Bases: *alpenglow.utils.ParameterSearch.ParameterSearch*

Threaded version of *alpenglow.utils.ParameterSearch*.

run(**run_parameters*, ***run_kw_parameters*)

Module contents

16.2 Submodules

16.3 alpenglow.Getter module

class alpenglow.Getter.**Getter**
Bases: object

Responsible for creating and managing cpp objects in the alpenglow.cpp package.

collect_ = {}

items = {}

class alpenglow.Getter.**MetaGetter**(*a*, *b*, *c*)

Bases: type

Metaclass of *alpenglow.Getter.Getter*. Provides utilities for creating and managing cpp objects in the alpenglow.cpp package. For more information, see *Python API*.

collect()

get_and_clean()

initialize_all(*objects*)

run_self_test(*i*)

set_experiment_environment(*online_experiment*, *objects*)

16.4 alpenglow.OnlineExperiment module

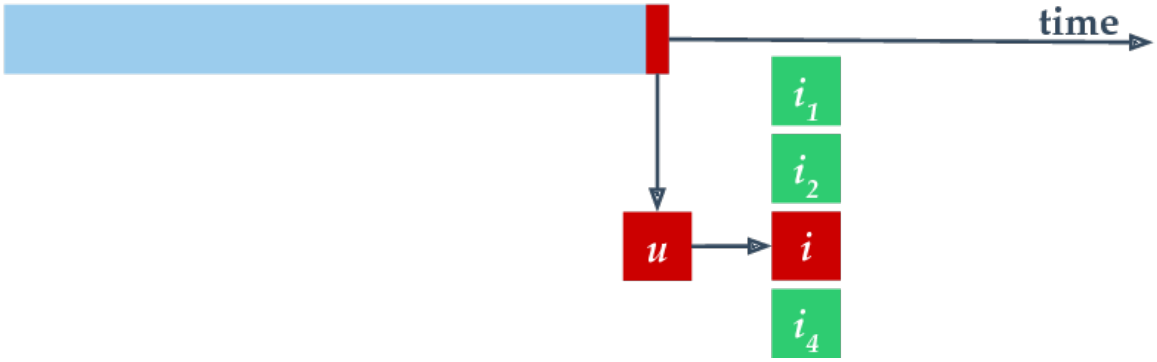
class alpenglow.OnlineExperiment.**OnlineExperiment**(*seed=254938879*, *top_k=100*)

Bases: *alpenglow.ParameterDefaults.ParameterDefaults*

This is the base class of every online experiment in Alpenglow. It builds the general experimental setup needed to run the online training and evaluation of a model. It also handles default parameters and the ability to override them when instantiating an experiment.

Subclasses should implement the config() method; for more information, check the documentation of this method as well.

Online evaluation in Alpenglow is done by processing the data row-by-row and evaluating the model on each new record before providing the model with the new information.



Evaluation is done by ranking the next item on the user's toplist and saving the rank. If the item is not found in the top top_k items, the evaluation step returns NaN.

For a brief tutorial on using this class, see *Five minute tutorial*.

Parameters

- **seed** (*int*) – The seed to initialize RNG-s. Should not be 0.
- **top_k** (*int*) – The length of the toplists.
- **network_mode** (*bool*) – Instructs the experiment to treat data as a directed graph, with source and target columns instead of user and item.

get_predictions()

If the calculate_toplists parameter is set when calling run, this method can used to acquire the generated toplists.

Returns

DataFrame containing the columns record_id, time, user, item, rank and prediction.

- **record_id** is the index of the record begin evaluated in the input DataFrame. Generally, there are `top_k` rows with the same record_id.
- **time** is the time of the evaluation
- **user** is the user the toplist is generated for
- **item** is the item of the toplist at the **rank** place
- **prediction** is the prediction given by the model for the (user, item) pair at the time of evaluation.

Return type pandas.DataFrame

run(*data*, *experimentType=None*, *columns={}*, *verbose=True*, *out_file=None*, *exclude_known=False*, *initialize_all=False*, *calculate_toplists=False*, *experiment_termination_time=0*, *memory_log=True*, *shuffle_same_time=True*, *recode=True*)

Parameters

- **data** (*pandas.DataFrame or str*) – The input data, see *Five minute tutorial*. If this parameter is a string, it has to be in the format specified by `experimentType`.
- **experimentType** (*str*) – The format of the input file if `data` is a string
- **columns** (*dict*) – Optionally the mapping of the input DataFrame's columns' names to the expected ones.
- **verbose** (*bool*) – Whether to write information about the experiment while running
- **out_file** (*str*) – If set, the results of the experiment are also written to the file located at `out_file`.
- **exclude_known** (*bool*) – If set to True, a user's previosly seen items are excluded from the toplist evaluation. The `eval` columns of the input data should be set accordingly.
- **calculate_toplists** (*bool or list*) – Whether to actually compute the toplists or just the ranks (the latter is faster). It can be specified on a record-by-record basis, by giving a list of booleans as parameter. The calculated toplists can be acquired after the experiment's end by using `get_predictions`. Setting this to non-False implies shuffle_same_time=False
- **experiment_termination_time** (*int*) – Stop the experiment at this timestamp.
- **memory_log** (*bool*) – Whether to log the results to memory (to be used optionally with out_file)
- **shuffle_same_time** (*bool*) – Whether to shuffle records with the same timestamp randomly.
- **recode** (*bool*) – Whether to automatically recode the entity columns so that they are indexed from 1 to n. If `False`, the recoding needs to be handled before passing the DataFrame to the `run` method.

Returns Results DataFrame if memory_log=True, empty DataFrame otherwise

Return type DataFrame

16.5 alpenglow.ParameterDefaults module

class alpenglow.ParameterDefaults.**ParameterDefaults**(***parameters*)
Bases: object

Base class of OnlineExperiment and OfflineModel, providing utilities for parameter defaults and overriding.

check_unused_parameters()

parameter_default(*name*, *value*)

parameter_defaults(***defaults*)

set_parameter(*name*, *value*)

16.6 alpenglow.PythonModel module

class alpenglow.PythonModel.**SelfUpdatingModel**
Bases: object

class alpenglow.PythonModel.**SubModel**(*parent*)
Bases: *alpenglow.cpp.PythonModel*

prediction(*rec_dat*)

class alpenglow.PythonModel.**SubRankingIteratorModel**(*parent*)
Bases: *alpenglow.cpp.PythonRankingIteratorModel*

iterator_get_next_(*id*, *user*)

iterator_has_next_(*id_*, *user*, *upper_bound*)

prediction(*rec_dat*)

class alpenglow.PythonModel.**SubToplistModel**(*parent*)
Bases: *alpenglow.cpp.PythonToplistModel*

get_top_list(*u*, *k*, *exclude*)

prediction(*rec_dat*)

class alpenglow.PythonModel.**SubUpdater**(*parent*)
Bases: *alpenglow.cpp.Updater*

update(*RecDat* rec_dat*)
Updates the associated model or other object of the simulation.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

16.7 Module contents

CHAPTER

SEVENTEEN

ALPENGLOW.CPP PACKAGE

The classes in this module are usually not used directly, but instead through the *alpenglow.Getter* class. For more info, read

- *Implementing a new model in C++*
- *C++ API*
- *Adjustable properties of evaluation*
- *Python API*

Note that there are some C++ classes that have no python interface. These are not documented here.

17.1 Filters

The function of the filter interface is limiting the available set of items. Current filters are whitelist-type filters, implementing *alpenglow.cpp.WhitelistFilter*.

To use a filter in an experiment, wrap the model into the filter using *alpenglow.cpp.WhitelistFilter2ModelAdapter*.

Example:

```
class LabelExperiment(prs.OnlineExperiment):
    '''Sample experiment illustrating the usage of LabelFilter. The
    experiment contains a PopularityModel and a LabelFilter.'''
    def _config(self, top_k, seed):
        model = ag.PopularityModel()
        updater = ag.PopularityModelUpdater()
        updater.set_model(model)
        label_filter = ag.LabelFilter(**self.parameter_defaults(
            label_file_name = ""
        ))
        adapter = ag.WhitelistFilter2ModelAdapter()
        adapter.set_model(model)
        adapter.set_whitelist_filter(label_filter)
```

class alpenglow.cpp.**WhitelistFilter**

Bases: sip.wrapper

Filter interface for classes that implement white list type filtering.

active(*RecDat* rec_dat*)

Returns whether the item is active for the user.

Parameters **rec_dat** (*RecDat**) – The sample containing the user and the item.

Returns Whether the item is available for the user.

Return type bool

get_whitelist(*int user*)
Returns the set of active items for the user.

Parameters **user** (*int*) – The whitelist will be computed for the given user.

Returns The list of allowed items for the given user. The second element of the pair is an upper bound for the score of the item for the given user (or the score itself).

Return type std::vector<int,double>

class alpenglow.cpp.**WhitelistFilter2ModelAdapter**
Bases: *alpenglow.cpp.Model*

Adapter class to filter the output of a model.

By chaining the adapter in front of a model, we can filter the output of the model, allowing only items on the whitelist filter to the toplist.

Note that as the currently implemented whitelists contain only a few elements, the adapter interface algorithm is optimized for short whitelists. We ignore the RSI of the model.

prediction(*RecDat**)
Returns filtered prediction value. The score of the items that are not on the whitelist is set to 0. Overrides method inherited from *alpenglow.cpp.Model.prediction()*, see also documentation there.

Parameters **rec_dat** (*RecDat**) – The sample we query the prediction for.

Returns The prediction score, modified based on the filter. If the item is not on the whitelist, the returned score is 0, otherwise the score returned by the model.

Return type double

self_test()
Tests whether the model and the whitelist filter is set.

Returns Whether all necessary objects are set.

Return type bool

set_model(*Model* model*)
Sets model whose output is filtered.

Parameters **model** (*Model**) – The model whose output is filtered.

set_whitelist_filter(*WhitelistFilter* whitelist_filter*)
Sets whitelist filter.

Parameters **whitelist_filter** (*WhitelistFilter**) – The whitelist filter we use for filtering the output of the model.

class alpenglow.cpp.**LabelFilterParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.LabelFilter*. See documentation there.

label_file_name

class alpenglow.cpp.**LabelFilter**
Bases: *alpenglow.cpp.WhitelistFilter*, *alpenglow.cpp.Updater*

White list filter class that allows items having the same label (e.g. artist) as the item that was previously interacted by the user. Requires updates (i.e., add the object to the online experiment as an updater).

Sample usage:

```
import alpenglow as ag

model = ag.PopularityModel()
updater = ag.PopularityModelUpdater()
updater.set_model(model)
label_filter = ag.LabelFilter(
    label_file_name = "/path/to/file/"
)
adapter = ag.WhitelistFilter2ModelAdapter()
adapter.set_model(model)
adapter.set_whitelist_filter(label_filter)
```

active(*RecDat**)
Implements *alpenglow.cpp.WhitelistFilter.active()*.

get_whitelist(*int user*)
Implements *alpenglow.cpp.WhitelistFilter.get_whitelist()*.

self_test()
Returns true.

update(*RecDat* rec_dat*)
Implements *alpenglow.cpp.Updater.update()*.

class alpenglow.cpp.**AvailabilityFilter**
Bases: *alpenglow.cpp.WhitelistFilter*, *alpenglow.cpp.NeedsExperimentEnvironment*

This filter filters the set of available items based on (time,itemId,duration) triplets. These have to be preloaded before using this filter.

Sample code

```
1 f = rs.AvailabilityFilter()
2 f.add_availability(20,1,10) #item 1 is available in the time interval (20,30)
```

active(*RecDat* rec_dat*)
Returns whether the item is active for the user.

Parameters **rec_dat** (*RecDat**) – The sample containing the user and the item.

Returns Whether the item is available for the user.

Return type bool

add_availability()

get_whitelist(*int user*)
Returns the set of active items for the user.

Parameters **user** (*int*) – The whitelist will be computed for the given user.

Returns The list of allowed items for the given user. The second element of the pair is an upper bound for the score of the item for the given user (or the score itself).

Return type std::vector<int,double>

self_test()

17.2 Offline evaluators

Use offline evaluators in traditional, fixed train/test split style learning. Check the code of *alpenglow.offline.OfflineModel.OfflineModel* descendants for usage examples.

class alpenglow.cpp.**PrecisionRecallEvaluatorParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.PrecisionRecallEvaluator*. See documentation there.

cutoff

test_file_name

test_file_type

time

class alpenglow.cpp.**PrecisionRecallEvaluator**
Bases: *alpenglow.cpp.OfflineEvaluator*

evaluate()

self_test()

set_model()

set_train_data()

class alpenglow.cpp.**OfflineRankingComputerParameters**
Bases: sip.wrapper

top_k

class alpenglow.cpp.**OfflinePredictions**
Bases: sip.wrapper

items

ranks

scores

users

class alpenglow.cpp.**OfflineRankingComputer**
Bases: sip.wrapper

compute()

set_items()

set_toplist_creator()

set_users()

class alpenglow.cpp.**OfflineEvaluator**
Bases: sip.wrapper

evaluate()

self_test()

17.3 Recommender data

This module contains the classes that are responsible for reading in the dataset and serving it to other classes of the experiment.

Interface *alpenglow.cpp.RecommenderData* is the anchestor for classes that read in the dataset. The two most frequently used implementations are *alpenglow.cpp.DataframeData* and *alpenglow.cpp.LegacyRecommenderData*.

Interface *alpenglow.cpp.RecommenderDataIterator* is the anchestor for classes that serve the data to the classes in the online experiment. See *The anatomy of an online experiment* for general information. The most frequently used implementations are *alpenglow.cpp.ShuffleIterator* and *alpenglow.cpp.SimpleIterator*.

class alpenglow.cpp.**RandomOnlineIteratorParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.RandomOnlineIterator*. See documentation there.

seed

class alpenglow.cpp.**RandomOnlineIterator**
Bases: *alpenglow.cpp.RecommenderDataIterator*

This RecommenderDataIterator shuffles the samples keeping the timestamps ordered and serves them in a fixed random order. Note that the samples are modified, the nth sample of the random order gets the timestamp of the nth sample of the original data.

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

get(*int index*)
See *alpenglow.cpp.RecommenderDataIterator.get()*

get_actual()
See *alpenglow.cpp.RecommenderDataIterator.get_actual()*

get_following_timestamp()
See *alpenglow.cpp.RecommenderDataIterator.get_following_timestamp()*

get_future(*int index*)
See *alpenglow.cpp.RecommenderDataIterator.get_future()*

next()
See *alpenglow.cpp.RecommenderDataIterator.next()*

self_test()
Tests if the class is set up correctly.

class alpenglow.cpp.**ShuffleIteratorParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.ShuffleIterator*. See documentation there.

seed

class alpenglow.cpp.**ShuffleIterator**
Bases: *alpenglow.cpp.RecommenderDataIterator*

autocalled_initialize()
See *alpenglow.cpp.Initializable.autocalled_initialize()*

get(*int index*)
See *alpenglow.cpp.RecommenderDataIterator.get()*

get_actual()
See *alpenglow.cpp.RecommenderDataIterator.get_actual()*

get_following_timestamp()
See *alpenglow.cpp.RecommenderDataIterator.get_following_timestamp()*

get_future(*int index*)
See *alpenglow.cpp.RecommenderDataIterator.get_future()*

next()
See *alpenglow.cpp.RecommenderDataIterator.next()*

self_test()
Tests if the class is set up correctly.

class alpenglow.cpp.**DataframeData**
Bases: *alpenglow.cpp.RecommenderData*

add_recdats()

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

get()

size()

class alpenglow.cpp.**SimpleIterator**
Bases: *alpenglow.cpp.RecommenderDataIterator*

This RecommenderDataIterator serves the samples in the original order.

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

get(*int index*)
See *alpenglow.cpp.RecommenderDataIterator.get()*

get_actual()
See *alpenglow.cpp.RecommenderDataIterator.get_actual()*

get_following_timestamp()
See *alpenglow.cpp.RecommenderDataIterator.get_following_timestamp()*

get_future(*int index*)
See *alpenglow.cpp.RecommenderDataIterator.get_future()*

next()
See *alpenglow.cpp.RecommenderDataIterator.next()*

class alpenglow.cpp.**RecommenderDataIterator**
Bases: *alpenglow.cpp.Initializable*

Iterator-like interface that serves the dataset as a time series in the online experiment. The class also provides random access to the time series. Implementations assume that the data is already sorted by time.

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

get(*int index*)
This method provides random access to the past samples of the time series. It reaches an error if index is larger than the index of the current sample, i.e. if one tries to access data from the future through this function.

Use *get_counter()* to get the index of the newest available sample. Use *get_future()* to get data from the future.

Parameters **index** (*int*) – The index of sample to return.

Returns A pointer to the sample.

Return type RecDat*

get_actual()

Returns A pointer to the actual sample.

Return type RecDat*

get_counter()

Returns Index of the actual sample.

Return type int

get_following_timestamp()

Returns The timestamp of the next sample in the future, i.e., when will the next event happen.

Return type double

get_future(*int index*)
This method provides random access to the complete time series, including future.

Use *get()* to safely get data from the past.

Parameters **index** (*int*) – The index of sample to return.

Returns A pointer to the sample.

Return type RecDat*

has_next()

Returns Whether the iterator has't reached the end of the time series.

Return type bool

next()
Advances the iterator and returns a pointer to the following sample.

Returns A pointer to the following sample.

Return type RecDat*

restart()
Restarts the iterator.

self_test()
Tests if the class is set up correctly.

set_recommender_data(*RecommenderData* data*)
Sets the dataset that we iterate on.

Parameters **data** (*RecommenderData**) – The dataset.

size()

Returns The number of the samples.

Return type int

class alpenglow.cpp.**FactorRepr**
Bases: sip.wrapper

entity

factors

class alpenglow.cpp.**UserItemFactors**
Bases: sip.wrapper

item_factors

user_factors

class alpenglow.cpp.**FactorModelReader**
Bases: sip.wrapper

read()

class alpenglow.cpp.**EigenFactorModelReader**
Bases: sip.wrapper

read()

class alpenglow.cpp.**RandomIteratorParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.RandomIterator*. See documentation there.

seed

shuffle_mode

class alpenglow.cpp.**RandomIterator**
Bases: *alpenglow.cpp.RecommenderDataIterator*

RecommenderDataIterator class that completely shuffles data. Note that the timestamps won't stay in increasing order, that may cause faults in some time-dependent models.

Sample offline usage: *alpenglow.cpp.OfflineIteratingOnlineLearnerWrapper*

autocalled_initialize()
See *alpenglow.cpp.Initializable.autocalled_initialize()*.

get(*int index*)
See *alpenglow.cpp.RecommenderDataIterator.get()*

get_actual()
See *alpenglow.cpp.RecommenderDataIterator.get_actual()*

get_following_timestamp()
See *alpenglow.cpp.RecommenderDataIterator.get_following_timestamp()*

get_future(*int index*)
See *alpenglow.cpp.RecommenderDataIterator.get_future()*

next()
See *alpenglow.cpp.RecommenderDataIterator.next()*

restart()
Restarts the iterator. In auto_shuffle mode it also reshuffles the dataset.

self_test()
Tests if the class is set up correctly.

shuffle()
Reshuffles the dataset.

class alpenglow.cpp.**InlineAttributeReader**
Bases: sip.wrapper

read_attribute()

self_test()

class alpenglow.cpp.**RecDat**
Bases: sip.wrapper

Struct representing a training instance.

category
model specific, mostly deprecated

eval
whether this record is to be used for evaluation

id
index of row in DataFrame; 0 based index if missing

item
item id

score
value of score column; 1 if column missing

time
value of time column; 0 based index if missing

user
user id

class alpenglow.cpp.**RecPred**
Bases: sip.wrapper

prediction

score

class alpenglow.cpp.**RecommenderData**
Bases: *alpenglow.cpp.Initializable*

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

clear()

get()

get_all_items()

get_all_users()

get_full_matrix()

get_items_into()

get_max_item_id()

get_max_user_id()

get_rec_data()

get_users_into()

set_rec_data()

size()

class alpenglow.cpp.**LegacyRecommenderDataParameters**

Bases: sip.wrapper

experiment_termination_time

file_name

type

class alpenglow.cpp.**LegacyRecommenderData**

Bases: *alpenglow.cpp.RecommenderData*

autocalled_initialize()

Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

read_from_file()

set_attribute_container()

17.4 Utils

This module contains miscellaneous helper classes.

class alpenglow.cpp.**PeriodComputerParameters**

Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.PeriodComputer*. See documentation there.

period_length

period_mode

start_time

class alpenglow.cpp.**PeriodComputer**

Bases: *alpenglow.cpp.Updater*, *alpenglow.cpp.NeedsExperimentEnvironment*, *alpenglow.cpp.Initializable*

Helper class to compute periods in the time series, e.g., update a model weekly or log statistics after every 10000th sample.

The class has two modes:

- period_mode==”time”: the time is based on the timestamp queried from recommender_data_iterator.
- period_mode==”samplenum”: the time is based on the number of calls to *update()*.

The class is notified about the progress of time by calls to the *update()* function.

autocalled_initialize()

Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

end_of_period()

True if the current sample is the last in the current period.

If period_mode==time, the function returns true if the timestamp of the next sample falls into the next timeframe. If period_mode==samplenum, the function returns true if the next call to *update()* will increase the number of the timeframe.

Returns True if the current sample is the last in the current period.

Return type bool

get_period_num()

Returns the number of the current period, i.e. timestamp/period_length+1.

If period_mode==time, timestamp is the time field of the current recdat. If period_mode==samplenum, timestamp is the number of calls to the *update()* function.

Returns The index of the current period.

Return type int

self_test()

Returns true.

set_parameters()

set_recommender_data_iterator()

update(*RecDat**)

Notifies the class that time has changed.

If period_mode==samplenum, the time is increased by 1. If period_mode==time, the timestamp of the next sample is queried from recommender_data_iterator. The parameter value is not used at all.

Parameters **rec_dat** (*RecDat**) – The current sample. Not used by the implementation.

class alpenglow.cpp.**SpMatrix**

Bases: sip.wrapper

Implements a sparse matrix. Obtaining a row is O(1), obtaining a value in a row is logarithmic in the length of the row.

clear()

Deletes all elements and rows from the matrix.

erase(*int row_id*, *int col_id*)
Removes the value from (row_id,col_id). If no value exists at that position, nothing happens.

Parameters

- **row_id** (*int*) – The index of the row.
- **col_id** (*int*) – The index of the column.

get(*int row_id*, *int col_id*)
Returns the value of (row_id,col_id). If no value exists at that position, returns 0.

Parameters

- **row_id** (*int*) – The index of the row.
- **col_id** (*int*) – The index of the column.

Returns The value of (row_id,col_id) or 0 if no value.

Return type double

has_value(*int row_id*, *int col_id*)

Parameters

- **row_id** (*int*) – The index of the row.
- **col_id** (*int*) – The index of the column.

Returns Whether the matrix has value at (row_id,col_id).

Return type bool

increase(*int row_id*, *int col_id*, *double value*)
Increases (row_id,col_id) with value or inserts value if (row_id,col_id) doesn't exist (i.e. not existing value is equivalent to 0 value). The matrix is expanded and the row is created as necessary.

Parameters

- **row_id** (*int*) – The index of the row.
- **col_id** (*int*) – The index of the column.
- **value** (*double*) – The new value.

insert(*int row_id*, *int col_id*, *double value*)
Inserts value to (row_id,col_id). If the value already exists, the insertion fails silently (the container doesn't change. The matrix is expanded and the row is created as necessary.

Parameters

- **row_id** (*int*) – The index of the row.
- **col_id** (*int*) – The index of the column.
- **value** (*double*) – The value to be inserted.

read_from_file(*std::string file_name*)
Reads matrix from file file_name. Format of the lines of the file is "row_id col_id value" . In case of repeating row_id col_id pairs, the first value will be used. Writes the size of the matrix to stderr. If the file can't be opened, fails silently not changing the container.

Parameters **file_name** (*std::string*) – The name of the file to read from.

resize(*int row_id*)
Expands the matrix to contain at least row_id rows. The matrix won't be shrunken. Creates an empty row object at index row_id.

Parameters **row_id** (*int*) – The index of the row.

row_size(*int row_id*)
Returns the number of the elements of the sparse row row_id. Returns 0 if the row doesn't exist.

Parameters **row_id** (*int*) – The index of the row.

Returns The size of the row or 0 if the row doesn't exist.

Return type int

size()

Returns

Return type The largest row index.

update(*int row_id*, *int col_id*, *double value*)
Updates (row_id,col_id) to value or inserts value if (row_id,col_id) doesn't exist. The matrix is expanded and the row is created as necessary.

Parameters

- **row_id** (*int*) – The index of the row.
- **col_id** (*int*) – The index of the column.
- **value** (*double*) – The new value.

write_into_file(*std::string file_name*)
Writes matrix into file file_name. Format of the lines of the file is "row_id col_id value" , space separated. If the file can't be opened, fails silently.

Parameters **file_name** (*std::string*) – The name of the file to write into.

class alpenglow.cpp.**Random**
Bases: sip.wrapper

This class implements a pseudorandom generator.

The next state is computed as *state = state*multiplier % mod* where *multiplier = 48271* and *mod = 2147483647*.

The initial state value can be set through the parameter of the constructor, or using the *set()* function.

Most of the functions are available without the max parameter providing a double value between [0,1) with a similar distribution as the discrete functions.

get(*int max*)
Get a uniform pseudorandom value between 0 and max-1.

Largest possible value is max-1.

Parameters **max** (*int*) – The upper bound of the random value.

Returns The pseudorandom value.

Return type int

get_arctg(*double y*, *int max*)
Get a pseudorandom value between 0 and max-1 with decaying distribution.

Probability of smaller values is larger. The largest possible value is max-1.

Parameters

- **y** (*double*) – The parameter of the distribution.
- **max** (*int*) – The upper bound of the random value.

Returns The pseudorandom value.

Return type double

get_boolean(*double prob*)

Get a pseudorandom true-or-false value.

Parameters **prob** (*double*) – The probability of the true value.

Returns The pseudorandom value.

Return type bool

get_discrete(*std::vector<double>& distribution*)

Get a pseudorandom value following a given discrete distribution.

The sum of the values in the given vector should be 1. If the sum is more or less, the probability of the largest value(s) will differ from the specified probability. The values should be non-negative.

Parameters **distribution** (*std::vector<double>&*) – The probability of output value i is distribution[i].

Returns The pseudorandom value.

Return type int

get_geometric(*double prob*, *int max*)

Get a pseudorandom value between 0 and max-1 with geometric distribution.

Probability of smaller values is larger. The largest possible value is max-1. The probability of value i is proportional to (1-prob)*prob^i`.

Parameters

- **prob** (*double*) – The parameter of the distribution.
- **max** (*int*) – The upper bound of the random value.

Returns The pseudorandom value.

Return type int

get_linear(*int max*)

Get a pseudorandom value between 0 and max-1 with linear distribution.

Probability of smaller values is smaller. The largest possible value is max-1.

Parameters **max** (*int*) – The upper bound of the random value.

Returns The pseudorandom value.

Return type int

self_test()

set(*int seed*)

Set the state of the random generator.

class alpenglow.cpp.**RankComputerParameters**

Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.RankComputer*. See documentation there.

exclude_known

random_seed

top_k

class alpenglow.cpp.**RankComputer**
Bases: *alpenglow.cpp.NeedsExperimentEnvironment*, *alpenglow.cpp.Initializable*

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

get_rank()

self_test()

set_items()

set_model()

set_parameters()

set_top_pop_container()

set_train_matrix()

class alpenglow.cpp.**Bias**
Bases: sip.wrapper

clear()

get()

init()

update()

class alpenglow.cpp.**SparseAttributeContainerParameters**
Bases: sip.wrapper

class alpenglow.cpp.**SparseAttributeContainer**
Bases: sip.wrapper

get_max_attribute_index()

class alpenglow.cpp.**FileSparseAttributeContainer**
Bases: *alpenglow.cpp.SparseAttributeContainer*

load_from_file()

class alpenglow.cpp.**Recency**
Bases: sip.wrapper

get()

update()

class alpenglow.cpp.**PowerLawRecencyParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.PowerLawRecency*. See documentation there.

delta_t

exponent

class alpenglow.cpp.**PowerLawRecency**
Bases: *alpenglow.cpp.Recency*

get()

update()

class alpenglow.cpp.**PopContainer**
Bases: sip.wrapper

Container for storing the popularity of items.

clear()
Clears the container. After this operations, the popularity of all the items is 0.

get(*int item*)
Returns the popularity value of the item.

Parameters **item** (*int*) – The item.

Returns The popularity value of the item.

Return type int

increase(*int item*)
Increases the popularity of the item.

Parameters **item** (*int*) – The item.

reduce(*int item*)
Reduces the popularity of the item.

Parameters **item** (*int*) – The item.

class alpenglow.cpp.**TopPopContainer**
Bases: sip.wrapper

Helper class for storing the items sorted by popularity.

Sample code

```
x = rs.TopPopContainer()
x.increase(1)
x.increase(1)
x.increase(3)
x.increase(4)
x.increase(1)

print("The most popular item is")
print(x.get_item(0)) #returns 1
print("The second most popular item is")
print(x.get_item(1)) #returns 3 or 4
```

create(*int item*)
Adds an item to the container. The item will have 0 popularity, but will be counted in *size()*.

get_item(*int index*)
Returns the index'th item from the popularity toplist.

Parameters **index** (*int*) – Index of the item in the popularity toplist. The index of the most popular item is 0.

Returns The appropriate item from the toplist.

Return type int

increase(*int item*)
Increases the popularity of the item.

size()
Returns the length of the complete toplist, i.e. the number of items having at least a popularity of 1 or that were added through *create()*.

Returns The length of the complete toplist.

Return type int

class alpenglow.cpp.**ToplistCreatorParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.ToplistCreator*. See documentation there.

exclude_known

top_k

class alpenglow.cpp.**ToplistCreator**
Bases: *alpenglow.cpp.NeedsExperimentEnvironment*, *alpenglow.cpp.Initializable*

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

run()

self_test()

set_items()

set_model()

set_train_matrix()

class alpenglow.cpp.**ToplistCreatorGlobalParameters**
Bases: *alpenglow.cpp.ToplistCreatorParameters*

Constructor parameter struct for alpenglow.cpp.ToplistCreatorGlobalParameters : public ToplistCreator. See documentation there.

initial_threshold

class alpenglow.cpp.**ToplistCreatorGlobal**
Bases: *alpenglow.cpp.ToplistCreator*

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

run()

self_test()

set_filter()

class alpenglow.cpp.**ToplistCreatorPersonalizedParameters**
Bases: *alpenglow.cpp.ToplistCreatorParameters*

Constructor parameter struct for alpenglow.cpp.ToplistCreatorPersonalizedParameters : public ToplistCreator. See documentation there.

class alpenglow.cpp.**ToplistCreatorPersonalized**
Bases: *alpenglow.cpp.ToplistCreator*

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

run()

self_test()

17.5 Gradient computers

This module contains the gradient computer classes that implement gradient computation necessary in gradient methods. See *alpenglow.experiments.FactorExperiment* for an example.

class alpenglow.cpp.**GradientComputer**
Bases: *alpenglow.cpp.Updater*

add_gradient_updater()

self_test()
Returns true.

set_model()

class alpenglow.cpp.**GradientComputerPointWise**
Bases: *alpenglow.cpp.GradientComputer*

self_test()
Returns true.

set_objective()

update(*RecDat* rec_dat*)
Updates the associated model or other object of the simulation.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

17.6 Objectives

This module contains the implementation of objective functions that are necessary for gradient computation in gradient learning methods. See *alpenglow.experiments.FactorExperiment* for a usage example.

class alpenglow.cpp.**ObjectiveMSE**
Bases: *alpenglow.cpp.ObjectivePointWise*

get_gradient()

class alpenglow.cpp.**ObjectivePointWise**
Bases: sip.wrapper

get_gradient()

self_test()

class alpenglow.cpp.**ObjectivePairWise**
Bases: sip.wrapper

self_test()

class alpenglow.cpp.**ObjectiveListWise**
Bases: sip.wrapper

get_gradient()

self_test()

17.7 General interfaces

This module contains the general interfaces that are implemented by classes belonging to different modules.

class alpenglow.cpp.**Initializable**
Bases: sip.wrapper

This interface signals that the implementing class has to be initialized by the experiment runner. The experiment runner calls the *initialize()* method, which in return calls the class-specific implementation of *autocalled_initialize()* and sets the *is_initialized()* flag if the initialization was successful. The *autocalled_initialize()* method can check whether the neccessary dependencies have been initialized or not before initializing the instance; and should return the success value accordingly.

If the initialization was not successful, the experiment runner keeps trying to initialize the not-yet initialized objects, thus resolving dependency chains.

Initializing and inheritance. Assume that class Parent implements Initializable, and the descendant Child needs further initialization. In that case Child has to override *autocalled_initialize()*, and call *Parent::autocalled_initialize()* in the overriding function first, continuing only if the parent returned true. If the init of the parent was succesful, but the children failed, then the children has to store the success of the parent and omit calling the initialization of the parent later.

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

initialize()

Returns Whether the initialization was successful.

Return type bool

is_initialized()

Returns Whether the component has already been initialized.

Return type bool

class alpenglow.cpp.**Updater**
Bases: sip.wrapper

Interface for updating *alpenglow.cpp.Model* instances or other objects of the simulation. Objects may implement this interface themselves or have one or more associated Updater types.

Examples:

- *alpenglow.cpp.TransitionProbabilityModel* and *alpenglow.cpp.TransitionProbabilityModelUpdater*
- *alpenglow.cpp.PopularityModel* has two updating algorithms:
- *alpenglow.cpp.PopularityModelUpdater*
- alpenglow.cpp.PopularityTimeframeModelUpdater
- *alpenglow.cpp.PeriodComputer* implements the Updater interface

In the online experiment, updaters are organized into a chain. See *The anatomy of an online experiment* for details.

self_test()
Returns true.

update(*RecDat* rec_dat*)
Updates the associated model or other object of the simulation.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

class alpenglow.cpp.**NeedsExperimentEnvironment**
Bases: sip.wrapper

set_experiment_environment()

17.8 Negative sample generators

All the samples in an implicit dataset are positive samples. To make gradient methods work, we need to provide negative samples too. This module contains classes that implement different negative sample generation algorithms. These classes implement *alpenglow.cpp.NegativeSampleGenerator*. The most frequently used implementation is *alpenglow.cpp.UniformNegativeSampleGenerator*.

class alpenglow.cpp.**UniformNegativeSampleGeneratorParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.UniformNegativeSampleGenerator*. See documentation there.

filter_repeats

negative_rate

seed

class alpenglow.cpp.**UniformNegativeSampleGenerator**
Bases: *alpenglow.cpp.NegativeSampleGenerator*, *alpenglow.cpp.Initializable*, *alpenglow.cpp.NeedsExperimentEnvironment*

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

generate()

self_test()
Returns true.

set_items()

set_train_matrix()

class alpenglow.cpp.**PooledPositiveSampleGeneratorParameters**
Bases: sip.wrapper

pool_size

positive_rate

seed

class alpenglow.cpp.**PooledPositiveSampleGenerator**
Bases: *alpenglow.cpp.NegativeSampleGenerator*

Generates positive samples from a pool.

For details, see:

Frigó, E., Pálovics, R., Kelen, D., Kocsis, L., & Benczúr, A. (2017). Online ranking prediction in non-stationary environments. Section 3.5.

generate()

get_implicit_train_data()

self_test()
Returns true.

class alpenglow.cpp.**NegativeSampleGenerator**
Bases: *alpenglow.cpp.Updater*

add_updater()

self_test()
Returns true.

update(*RecDat* rec_dat*)
Updates the associated model or other object of the simulation.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

17.9 Offline learners

Use offline learners in traditional, fixed train/test split style learning. Check the code of *alpenglow.offline.OfflineModel.OfflineModel* descendants for usage examples.

class alpenglow.cpp.**OfflineIteratingOnlineLearnerWrapperParameters**
Bases: sip.wrapper

number_of_iterations

seed

shuffle

class alpenglow.cpp.**OfflineIteratingOnlineLearnerWrapper**
Bases: *alpenglow.cpp.OfflineLearner*

add_early_updater()

add_iterate_updater()

add_updater()

fit()

self_test()

class alpenglow.cpp.OfflineEigenFactorModelALSLearnerParameters
Bases: sip.wrapper

alpha

clear_before_fit

implicit

number_of_iterations

regularization_lambda

class alpenglow.cpp.OfflineEigenFactorModelALSLearner
Bases: *alpenglow.cpp.OfflineLearner*

fit()

iterate()

self_test()

set_copy_from_model()

set_copy_to_model()

set_model()

class alpenglow.cpp.OfflineLearner
Bases: sip.wrapper

fit()

self_test()

class alpenglow.cpp.OfflineExternalModelLearnerParameters
Bases: sip.wrapper

in_name_base

mode

out_name_base

class alpenglow.cpp.OfflineExternalModelLearner
Bases: *alpenglow.cpp.OfflineLearner*

fit()

set_model()

17.10 Loggers

Loggers implement evaluators, statistics etc. in the online experiment. These classes implement interface *alpenglow.cpp.Logger*. See *The anatomy of an online experiment* for a general view.

class alpenglow.cpp.**OnlinePredictions**
Bases: sip.wrapper

ids

items

ranks

scores

times

users

class alpenglow.cpp.**PredictionLogger**
Bases: *alpenglow.cpp.Logger*

get_predictions()

run(*RecDat* rec_dat*)
Evaluates the model and logs results, statistics, simulation data, debug info etc.

In the online experiment, *alpenglow.cpp.OnlineExperiment* calls this method. It is not allowed to modify the model or other simulation objects through this function.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

self_test()
Returns true.

set_prediction_creator()

class alpenglow.cpp.**RankingLog**
Bases: sip.wrapper

id

item

prediction

rank

score

time

user

class alpenglow.cpp.**RankingLogs**
Bases: sip.wrapper

logs

top_k

class alpenglow.cpp.**MemoryRankingLoggerParameters**
Bases: sip.wrapper

evaluation_start_time

memory_log

out_file

random_seed

top_k

class alpenglow.cpp.**MemoryRankingLogger**

Bases: *alpenglow.cpp.Logger*, *alpenglow.cpp.NeedsExperimentEnvironment*, *alpenglow.cpp.Initializable*

autocalled_initialize()

Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

get_ranking_logs()

run(*RecDat* rec_dat*)

Evaluates the model and logs results, statistics, simulation data, debug info etc.

In the online experiment, *alpenglow.cpp.OnlineExperiment* calls this method. It is not allowed to modify the model or other simulation objects through this function.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

self_test()

Returns true.

set_items()

set_model()

set_ranking_logs()

set_top_pop_container()

set_train_matrix()

class alpenglow.cpp.**ProceedingLogger**

Bases: *alpenglow.cpp.Logger*, *alpenglow.cpp.Initializable*, *alpenglow.cpp.NeedsExperimentEnvironment*

autocalled_initialize()

Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

run(*RecDat* rec_dat*)

Evaluates the model and logs results, statistics, simulation data, debug info etc.

In the online experiment, *alpenglow.cpp.OnlineExperiment* calls this method. It is not allowed to modify the model or other simulation objects through this function.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

self_test()

Returns true.

set_data_iterator()

class alpenglow.cpp.**TransitionModelLoggerParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.TransitionModelLogger*. See documentation there.

period_length

timeline_logfile_name

top_k

toplist_length_logfile_basename

class alpenglow.cpp.**TransitionModelLogger**
Bases: *alpenglow.cpp.Logger*, *alpenglow.cpp.NeedsExperimentEnvironment*, *alpenglow.cpp.Initializable*

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

run(*RecDat* rec_dat*)
Evaluates the model and logs results, statistics, simulation data, debug info etc.

In the online experiment, *alpenglow.cpp.OnlineExperiment* calls this method. It is not allowed to modify the model or other simulation objects through this function.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

self_test()
Returns true.

set_model()

set_pop_container()

set_train_matrix()

class alpenglow.cpp.**Logger**
Bases: sip.wrapper

Interface for evaluating the model and logging results, statistics, simulation data, debug info etc.

In the online experiment, alpenglow.cpp.OnineExperiment calls loggers for each sample and at the end of the experiment. See alpenglow.cpp.OnineExperiment for details.

run(*RecDat* rec_dat*)
Evaluates the model and logs results, statistics, simulation data, debug info etc.

In the online experiment, *alpenglow.cpp.OnlineExperiment* calls this method. It is not allowed to modify the model or other simulation objects through this function.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

self_test()
Returns true.

class alpenglow.cpp.**OnlinePredictorParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.OnlinePredictor*. See documentation there.

evaluation_start_time

file_name

time_frame

class alpenglow.cpp.**OnlinePredictor**
Bases: *alpenglow.cpp.Logger*, *alpenglow.cpp.NeedsExperimentEnvironment*, *alpenglow.cpp.Initializable*

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

run(*RecDat* rec_dat*)
Evaluates the model and logs results, statistics, simulation data, debug info etc.

In the online experiment, *alpenglow.cpp.OnlineExperiment* calls this method. It is not allowed to modify the model or other simulation objects through this function.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

self_test()
Returns true.

set_prediction_creator()

class alpenglow.cpp.**MemoryUsageLogger**
Bases: *alpenglow.cpp.Logger*, *alpenglow.cpp.Initializable*, *alpenglow.cpp.NeedsExperimentEnvironment*

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

run(*RecDat* rec_dat*)
Evaluates the model and logs results, statistics, simulation data, debug info etc.

In the online experiment, *alpenglow.cpp.OnlineExperiment* calls this method. It is not allowed to modify the model or other simulation objects through this function.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

self_test()
Returns true.

set_data_iterator()

class alpenglow.cpp.**InterruptLogger**
Bases: *alpenglow.cpp.Logger*

run(*RecDat* rec_dat*)
Evaluates the model and logs results, statistics, simulation data, debug info etc.

In the online experiment, *alpenglow.cpp.OnlineExperiment* calls this method. It is not allowed to modify the model or other simulation objects through this function.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

class alpenglow.cpp.**ConditionalMetaLogger**
Bases: *alpenglow.cpp.Logger*

run(*RecDat* rec_dat*)
Evaluates the model and logs results, statistics, simulation data, debug info etc.

In the online experiment, *alpenglow.cpp.OnlineExperiment* calls this method. It is not allowed to modify the model or other simulation objects through this function.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

self_test()
Returns true.

set_logger()

should_run()

class alpenglow.cpp.**ListConditionalMetaLoggerParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.ListConditionalMetaLogger*. See documentation there.

should_run_vector

class alpenglow.cpp.**ListConditionalMetaLogger**
Bases: *alpenglow.cpp.ConditionalMetaLogger*

should_run()

class alpenglow.cpp.**InputLoggerParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.InputLogger*. See documentation there.

output_file

class alpenglow.cpp.**InputLogger**
Bases: *alpenglow.cpp.Logger*, *alpenglow.cpp.Initializable*

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

run(*RecDat* rec_dat*)
Evaluates the model and logs results, statistics, simulation data, debug info etc.

In the online experiment, *alpenglow.cpp.OnlineExperiment* calls this method. It is not allowed to modify the model or other simulation objects through this function.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

self_test()
Returns true.

17.11 Online experiment

The central classes of the online experiments.

class alpenglow.cpp.**OnlineExperimentParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.OnlineExperiment*. See documentation there.

evaluation_start_time

exclude_known

experiment_termination_time

initialize_all

max_item

max_user

random_seed

top_k

class alpenglow.cpp.**OnlineExperiment**
Bases: sip.wrapper

The central class of the online experiment.

It queries samples from the dataset, then one-by-one for each sample

- calls loggers that are set using *add_logger()*,
- updates the environment and common statistics, see *alpenglow.cpp.ExperimentEnvironment*,
- calls the updaters that are set using *add_updater()*.

At the end of the experiment, it calls end loggers that are set using *add_end_logger()*.

See *alpenglow.OnlineExperiment.OnlineExperiment* for details.

add_end_logger(*Logger* logger*)
Adds a logger instance, that will be called once at the end of the experiment.

Parameters **logger** (*Logger**) – Pointer to the logger to be added.

add_logger(*Logger* logger*)
Adds a logger instance.

Parameters **logger** (*Logger**) – Pointer to the logger to be added.

add_updater(*Updater* updater*)
Adds an updater.

Parameters **updater** (*Updater**) – Pointer to the updater to be added.

inject_experiment_environment_into(*NeedsExperimentEnvironment* object*)
Sets the experiment environment into another object that requires it.

In the online experiment, this method is automatically called with all the objects that implement *alpenglow.cpp.NeedsExperimentEnvironment*, injecting the dependency where it is necessary. See the code of *alpenglow.OnlineExperiment.OnlineExperiment* for details.

run()
Runs the experiment.

self_test()
Tests if the dataset is set.

Furthermore, the test produces a warning message if no loggers are set because in that case the the experiment will produce no output.

Returns Whether the tests were successful.

Return type bool

set_recommender_data_iterator(*RecommenderDataIterator* recommender_data_iterator*)
Sets the dataset of the experiment.

Parameters **recommender_data_iterator** (*RecommenderDataIterator**) – Pointer to the dataset.

class alpenglow.cpp.ExperimentEnvironment
Bases: sip.wrapper

Class that stores, updates and serves common simulation data and parameters, e.g. length of the top list, dataset and popularity of the items.

In the online experiment, the central class *alpenglow.cpp.OnlineExperiment* updates this class and the common statistic containers. This class is updated after calling loggers and before calling updaters (for each sample). See details there. Other objects are not allowed to modify this class or the statistic containers, even if they have non-const access (exception: the common Random).

get_evaluation_start_time()

Returns The beginning timestamp of evaluation. Elements in the time series before that timestamp will not be evaluated. Note that not all evaluator classes consider this value.

Return type int

get_exclude_known()

Returns Whether each user-item pair should be evaluated only at the first occurrence, i.e., known user-item pairs should not be evaluated at repeated occurrences.

Return type bool

get_experiment_termination_time()

Returns The last timestamp of evaluation. Elements in the time series after that timestamp will not be evaluated. Note that not all evaluator classes consider this value.

Return type int

get_initialize_all()

Returns Whether all the users and items exist from the beginning of the experiment, or they appear only when they are mentioned first in a sample. If set, recode the dataset so that the users and items are numbered starting 0 or 1 continuously. Skipped ids are treated as existing too.

Return type bool

get_items()

Returns A pointer to the list of known items. In the online experiment, the list is updated by this class for each sample after the call of the loggers and before the call to the updaters. If `initialize_all==True`, the list is filled with items at the beginning of the experiment.

Return type std::vector<int>*

`get_max_item_id()`

Returns The maximal item id int the whole experiment.

Return type int

`get_max_user_id()`

Returns The maximal user id int the whole experiment.

Return type int

`get_popularity_container()`

Returns A pointer to a container containing the popularity statistics of known items.

Return type PopContainer*

`get_popularity_sorted_container()`

Returns A pointer to a container containing the popularity statistics of known items. The items can be acquired in popularity order. The container contains all known items.

Return type TopPopContainer*

`get_recommender_data_iterator()`

Returns A pointer to the data iterator containing the time series of the experiment.

Return type RecommenderDataIterator*

`get_time()`

`get_top_k()`

Returns The top list length in the current experiment. Note that not all classes consider this value.

Return type int

`get_train_matrix()`

Returns A pointer to the current training data in a sparse matrix form.

Return type SpMatrix*

`get_users()`

Returns A pointer to the list of known users. In the online experiment, the list is updated by this class for each sample after the call of the loggers and before the call to the updaters. If `initialize_all==True`, the list is filled with users at the beginning of the experiment.

Return type std::vector<int>*

`is_first_occurrence_of_item()`

Returns Whether the current item is mentioned for the first time in the current sample. If `initialize_all==False`, equals to *is_new_item()*.

Return type bool

`is_first_occurrence_of_user()`

Returns Whether the current user is mentioned for the first time in the current sample. If `initialize_all==False`, equals to *is_new_user()*.

Return type bool

`is_item_existent()`

Returns Whether the item exists. If `initialize_all==True`, returns constant true value for items<=max_item_id, because all items exsist from the begininning of the experiment. Note that new items come into existence after te call to loggers, before the call to updaters.

Return type bool

`is_item_new_for_user()`

Returns Whether the current item is new for the current user, i.e., this is the first occurrence of this user-item pair in the time series. Note that the value is updated only when the loggers had been called already.

Return type bool

`is_new_item()`

Returns Whether the current item is new, i.e. come to existence with the current sample. If `initialize_all==True`, returns constant false value, because all items exsist from the begininning of the experiment. Note that new items come into existence after te call to loggers, before the call to updaters.

Return type bool

`is_new_user()`

Returns Whether the current user is new, i.e. come to existence with the current sample. If `initialize_all==True`, returns constant false value, because all users exsist from the begininning of the experiment. Note that new users come into existence after te call to loggers, before the call to updaters.

Return type bool

`is_user_existing()`

Returns Whether the user exists. If `initialize_all==True`, returns constant true value for users<=max_user_id, because all users exsist from the begininning of the experiment. Note that new users come into existence after te call to loggers, before the call to updaters.

Return type bool

`self_test()`

set_parameters()
Sets the parameters of the experiment. Called by *alpenglow.cpp.OnlineExperiment*.

set_recommender_data_iterator()
Sets the dataset (the time series) of the experiment. Called by *alpenglow.cpp.OnlineExperiment*.

update(*RecDat* rec_dat*)
Updates the container.

In the online experiment, *alpenglow.cpp.OnlineExperiment* calls this function after the call to the loggers and before the call to the updaters. Other classes should not call this function. :param rec_dat: The newest available sample of the experiment. :type rec_dat: RecDat*

17.12 Models

The prediction models in the experiments. The model interface is *alpenglow.cpp.Model*. See *Rank computation optimization* about different evaluation methods.

17.13 Factor models

This module contains the matrix factorization based models.

class alpenglow.cpp.**EigenFactorModelParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.EigenFactorModel*. See documentation there.

begin_max

begin_min

dimension

lemp_bucket_size

seed

class alpenglow.cpp.**EigenFactorModel**
Bases: *alpenglow.cpp.Model*, *alpenglow.cpp.Initializable*

add()

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

clear()

prediction()

resize()

self_test()

class alpenglow.cpp.**FactorModelUpdater**
Bases: *alpenglow.cpp.Updater*

self_test()
Returns true.

set_model()

update(*RecDat* rec_dat*)
Updates the associated model or other object of the simulation.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

class alpenglow.cpp.**FmModelUpdaterParameters**
Bases: sip.wrapper

learning_rate

class alpenglow.cpp.**FmModelUpdater**
Bases: *alpenglow.cpp.Updater*

self_test()
Returns true.

set_model()

update(*RecDat* rec_dat*)
Updates the associated model or other object of the simulation.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

class alpenglow.cpp.**FmModelParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.FmModel*. See documentation there.

begin_max

begin_min

dimension

item_attributes

seed

user_attributes

class alpenglow.cpp.**FmModel**
Bases: *alpenglow.cpp.Model*, *alpenglow.cpp.Initializable*

add()

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

clear()

prediction()

self_test()

class alpenglow.cpp.**AsymmetricFactorModelGradientUpdaterParameters**
Bases: sip.wrapper

cumulative_item_updates

learning_rate

class alpenglow.cpp.**AsymmetricFactorModelGradientUpdater**
Bases: *alpenglow.cpp.ModelGradientUpdater*

self_test()

set_model()

update()

class alpenglow.cpp.**SvdppModelUpdater**
Bases: *alpenglow.cpp.Updater*

self_test()
Returns true.

set_model()

update(*RecDat* rec_dat*)
Updates the associated model or other object of the simulation.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

class alpenglow.cpp.**FactorModelParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.FactorModel*. See documentation there.

begin_max

begin_min

dimension

initialize_all

lemp_bucket_size

max_item

max_user

seed

use_item_bias

use_sigmoid

use_user_bias

class alpenglow.cpp.**FactorModel**
Bases: *alpenglow.cpp.Model*, *alpenglow.cpp.SimilarityModel*, *alpenglow.cpp.NeedsExperimentEnvironment*, *alpenglow.cpp.Initializable*

add()

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

clear()

prediction()

self_test()

set_item_recency()

set_user_recency()

similarity()

class alpenglow.cpp.**AsymmetricFactorModelUpdater**
Bases: *alpenglow.cpp.Updater*

self_test()
Returns true.

set_model()

update(*RecDat* rec_dat*)
Updates the associated model or other object of the simulation.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

class alpenglow.cpp.**SvdppModelGradientUpdaterParameters**
Bases: sip.wrapper

cumulative_item_updates

learning_rate

class alpenglow.cpp.**SvdppModelGradientUpdater**
Bases: *alpenglow.cpp.ModelGradientUpdater*

self_test()

set_model()

update()

class alpenglow.cpp.**FactorModelGlobalRankingScoreIterator**
Bases: *alpenglow.cpp.GlobalRankingScoreIterator*, *alpenglow.cpp.NeedsExperimentEnvironment*

autocalled_initialize()

get_global_items()

get_global_users()

run()

self_test()

set_experiment_environment()

set_items()

set_model()

set_users()

class alpenglow.cpp.**SvdppModelParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.SvdppModel*. See documentation there.

begin_max

begin_min

dimension

gamma

history_weight

initialize_all

max_item

max_user

norm_type

seed

use_sigmoid

user_vector_weight

class alpenglow.cpp.SvdppModel
Bases: *alpenglow.cpp.Model*, *alpenglow.cpp.NeedsExperimentEnvironment*, *alpenglow.cpp.Initializable*

add()

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

clear()

prediction()

self_test()

class alpenglow.cpp.FactorModelGradientUpdaterParameters
Bases: sip.wrapper

learning_rate

learning_rate_bias

regularization_rate

regularization_rate_bias

turn_off_item_bias_updates

turn_off_item_factor_updates

turn_off_user_bias_updates

turn_off_user_factor_updates

class alpenglow.cpp.FactorModelGradientUpdater
Bases: *alpenglow.cpp.ModelGradientUpdater*

self_test()

set_model()

update()

class alpenglow.cpp.AsymmetricFactorModelParameters
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.AsymmetricFactorModel*. See documentation there.

begin_max

begin_min

dimension

gamma

initialize_all

max_item

norm_type

seed

use_sigmoid

class alpenglow.cpp.**AsymmetricFactorModel**

Bases: *alpenglow.cpp.Model*, *alpenglow.cpp.NeedsExperimentEnvironment*, *alpenglow.cpp.Initializable*

add()

autocalled_initialize()

Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

clear()

prediction()

self_test()

17.14 Baseline models

This submodule contains the simple baseline models like nearest neighbor or most popular.

class alpenglow.cpp.**TransitionProbabilityModel**

Bases: *alpenglow.cpp.Model*

clear()

prediction()

self_test()

class alpenglow.cpp.**NearestNeighborModelParameters**

Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.NearestNeighborModel*. See documentation there.

direction

gamma

gamma_threshold

norm

num_of_neighbors

class alpenglow.cpp.**NearestNeighborModel**
Bases: *alpenglow.cpp.Model*

Item similarity based model.

See source of *alpenglow.experiments.NearestNeighborExperiment* for a usage example.

prediction(*RecDat* rec_dat*)
Implements *alpenglow.cpp.Model.prediction()*.

self_test()
Tests whether the model is assembled appropriately.

Returns Whether the model is assembled appropriately.

Return type bool

class alpenglow.cpp.**PopularityTimeFrameModelUpdaterParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.PopularityTimeFrameModelUpdater*. See documentation there.

tau

class alpenglow.cpp.**PopularityTimeFrameModelUpdater**
Bases: *alpenglow.cpp.Updater*

Time-aware updater for PopularityModel, which only considers the last tau time interval when calculating popularities. Note that the time window ends at the timestamp of the last updating sample. the timestamp of the sample in the prediction call is not considered.

self_test()
Returns true.

set_model()

update(*RecDat* rec_dat*)
Updates the associated model or other object of the simulation.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

class alpenglow.cpp.**PersonalPopularityModel**
Bases: *alpenglow.cpp.Model*

prediction()

class alpenglow.cpp.**PersonalPopularityModelUpdater**
Bases: *alpenglow.cpp.Updater*

self_test()
Returns true.

set_model()

update(*RecDat* rec_dat*)
Updates the associated model or other object of the simulation.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

class alpenglow.cpp.**PopularityModel**
Bases: *alpenglow.cpp.Model*

prediction()

class alpenglow.cpp.**NearestNeighborModelUpdaterParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.NearestNeighborModelUpdater*. See documentation there.

compute_similarity_period

period_mode

class alpenglow.cpp.**NearestNeighborModelUpdater**
Bases: *alpenglow.cpp.Updater*

self_test()
Returns true.

set_model()

update(*RecDat* rec_dat*)
Updates the associated model or other object of the simulation.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

class alpenglow.cpp.**TransitionProbabilityModelUpdaterParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.TransitionProbabilityModelUpdater*. See documentation there.

filter_freq_updates

label_file_name

label_transition_mode

mode

class alpenglow.cpp.**TransitionProbabilityModelUpdater**
Bases: *alpenglow.cpp.Updater*

self_test()
Returns true.

set_model()

update(*RecDat* rec_dat*)
Updates the associated model or other object of the simulation.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

class alpenglow.cpp.**PopularityModelUpdater**
Bases: *alpenglow.cpp.Updater*

self_test()
Returns true.

set_model()

update(*RecDat* rec_dat*)
Updates the associated model or other object of the simulation.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

17.15 Model combination

This module contains the models that combine other models. The most frequently used class is *alpenglow.cpp.CombinedModel*. See *Model combination* for a usage example.

class alpenglow.cpp.**ToplistCombinationModelParameters**
Bases: sip.wrapper

seed

top_k

class alpenglow.cpp.**ToplistCombinationModel**
Bases: *alpenglow.cpp.Model*, *alpenglow.cpp.Initializable*, *alpenglow.cpp.NeedsExperimentEnvironment*

add()

add_model()

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

inject_wms_into()

prediction()

self_test()

class alpenglow.cpp.**WeightedModelStructure**
Bases: sip.wrapper

distribution_

is_initialized()

models_

class alpenglow.cpp.**WMSUpdater**
Bases: sip.wrapper

set_wms()

class alpenglow.cpp.**RandomChoosingCombinedModelParameters**
Bases: sip.wrapper

seed

class alpenglow.cpp.**RandomChoosingCombinedModel**
Bases: *alpenglow.cpp.Model*, *alpenglow.cpp.Initializable*

add()

add_model()

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

inject_wms_into()

prediction()

self_test()

class alpenglow.cpp.**CombinedModelParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.CombinedModel*. See documentation there.

log_file_name

log_frequency

use_user_weights

class alpenglow.cpp.**CombinedModel**
Bases: *alpenglow.cpp.Model*

add()

add_model()

prediction()

class alpenglow.cpp.**RandomChoosingCombinedModelExpertUpdaterParameters**
Bases: sip.wrapper

eta

loss_type

top_k

class alpenglow.cpp.**RandomChoosingCombinedModelExpertUpdater**
Bases: *alpenglow.cpp.Updater*, *alpenglow.cpp.WMSUpdater*, *alpenglow.cpp.Initializable*, *alpenglow.cpp.NeedsExperimentEnvironment*

autocalled_initialize()
Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

self_test()
Returns true.

set_experiment_environment()

set_wms()

update(*RecDat* rec_dat*)
Updates the associated model or other object of the simulation.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

class alpenglow.cpp.**Evaluator**
Bases: sip.wrapper

get_loss()

get_score()

self_test()

class alpenglow.cpp.**CombinedDoubleLayerModelGradientUpdaterParameters**
Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.CombinedDoubleLayerModelGradientUpdater*. See documentation there.

always_learn

global_learning_rate

global_regularization_rate

learning_rate

regularization_rate

start_combination_learning_time

class alpenglow.cpp.**CombinedDoubleLayerModelGradientUpdater**
Bases: *alpenglow.cpp.ModelGradientUpdater*

self_test()

set_model()

update()

class alpenglow.cpp.**TopListRecommender**
Bases: sip.wrapper

get_top_list()

class alpenglow.cpp.**MassPredictor**
Bases: sip.wrapper

predict()

set_model()

class alpenglow.cpp.**Model**
Bases: sip.wrapper

add()

clear()

prediction()

read()

self_test()

write()

class alpenglow.cpp.**ExternalModelParameters**
Bases: sip.wrapper

mode

class alpenglow.cpp.**ExternalModel**
Bases: *alpenglow.cpp.Model*

add()

clear()

prediction()

read_predictions()

self_test()

class alpenglow.cpp.PythonModel
Bases: alpenglow.cpp.Model

class alpenglow.cpp.PythonToplistModel
Bases: alpenglow.cpp.PythonModel, alpenglow.cpp.TopListRecommender

class alpenglow.cpp.PythonRankingIteratorModel
Bases: alpenglow.cpp.PythonModel, alpenglow.cpp.RankingScoreIteratorProvider

iterator_get_next_()

iterator_has_next_()

class alpenglow.cpp.SimilarityModel
Bases: sip.wrapper

self_test()

similarity()

class alpenglow.cpp.ModelGradientUpdater
Bases: sip.wrapper

self_test()

update()

class alpenglow.cpp.ModelMultiUpdater
Bases: sip.wrapper

self_test()

update()

class alpenglow.cpp.RankingScoreIteratorProvider
Bases: sip.wrapper

class alpenglow.cpp.GlobalRankingScoreIterator
Bases: sip.wrapper

run()

self_test()

17.16 Data generators

The classes in this module are responsible for generating data subsets from the past. This is necessary for embedding offline models into the online framework, that needs to be updated in a batch. See alpenglow.experiments.BatchFactorExperiment for a usage example.

class alpenglow.cpp.DataGenerator
Bases: sip.wrapper

generate_recommender_data()

self_test()

class alpenglow.cpp.SamplingDataGeneratorParameters
Bases: sip.wrapper

Constructor parameter struct for alpenglow.cpp.SamplingDataGenerator. See documentation there.

distribution

geometric_param

number_of_samples

seed

y

class alpenglow.cpp.**SamplingDataGenerator**

Bases: *alpenglow.cpp.DataGenerator*, *alpenglow.cpp.Initializable*, *alpenglow.cpp.NeedsExperimentEnvironment*

autocalled_initialize()

Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

generate_recommender_data()

self_test()

set_recommender_data_iterator()

class alpenglow.cpp.**CompletePastDataGenerator**

Bases: *alpenglow.cpp.DataGenerator*, *alpenglow.cpp.NeedsExperimentEnvironment*, *alpenglow.cpp.Initializable*

autocalled_initialize()

Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

generate_recommender_data()

self_test()

set_recommender_data_iterator()

class alpenglow.cpp.**TimeframeDataGeneratorParameters**

Bases: sip.wrapper

Constructor parameter struct for *alpenglow.cpp.TimeframeDataGenerator*. See documentation there.

timeframe_length

class alpenglow.cpp.**TimeframeDataGenerator**

Bases: *alpenglow.cpp.DataGenerator*, *alpenglow.cpp.NeedsExperimentEnvironment*, *alpenglow.cpp.Initializable*

autocalled_initialize()

Has to be implemented by the component.

Returns Whether the initialization was successful.

Return type bool

generate_recommender_data()

self_test()

set_recommender_data_iterator()

17.17 Online learners

This module contains classes that modifiy the learning process, e.g. delay the samples or feed them in a batch into offline learning methods.

class alpenglow.cpp.**LearnerPeriodicDelayedWrapperParameters**
Bases: sip.wrapper

delay

period

class alpenglow.cpp.**LearnerPeriodicDelayedWrapper**
Bases: *alpenglow.cpp.Updater*

self_test()
Returns true.

set_wrapped_learner()

update(*RecDat* rec_dat*)
Updates the associated model or other object of the simulation.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

class alpenglow.cpp.**PeriodicOfflineLearnerWrapperParameters**
Bases: sip.wrapper

base_in_file_name

base_out_file_name

clear_model

learn

read_model

write_model

class alpenglow.cpp.**PeriodicOfflineLearnerWrapper**
Bases: *alpenglow.cpp.Updater*

add_offline_learner()

self_test()
Returns true.

set_data_generator()

set_model()

set_period_computer()

update(*RecDat* rec_dat*)
Updates the associated model or other object of the simulation.

Parameters **rec_dat** (*RecDat**) – The newest available sample of the experiment.

- genindex
- Module index
- search

BIBLIOGRAPHY

[Hu2008] Hu, Yifan, Yehuda Koren, and Chris Volinsky. "Collaborative filtering for implicit feedback datasets." Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on. Ieee, 2008.

[Paterek2007] Arkadiusz Paterek. „Improving regularized singular value decomposition for collaborative filtering". In: Proc. KDD Cup Workshop at SIGKDD'07, 13th ACM Int. Conf. on Knowledge Discovery and Data Mining. San Jose, CA, USA, 2007, pp. 39–42.

[Koren2008] Koren, Yehuda. "Factorization meets the neighborhood: a multifaceted collaborative filtering model." Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2008.

[Koren2009] Koren, Yehuda, Robert Bell, and Chris Volinsky. "Matrix factorization techniques for recommender systems." Computer 42.8 (2009).

[X.He2016] X. He, H. Zhang, M.-Y. Kan, and T.-S. Chua. Fast matrix factorization for online recommendation with implicit feedback. In SIGIR, pages 549–558, 2016.

[Rendle2012] Rendle, Steffen. "Factorization machines with libfm." ACM Transactions on Intelligent Systems and Technology (TIST) 3.3 (2012): 57.

[Sarwar2001] B. Sarwar, G. Karypis, J. Konstan, and J. Reidl. Item-based collaborative filtering recommendation algorithms. In Proc. WWW, pages 285–295, 2001.

[Ding2005] Y. Ding and X. Li. Time weight collaborative filtering. In Proc. CIKM, pages 485–492. ACM, 2005.

[frigo2017online] Frigó, E., Pálovics, R., Kelen, D., Kocsis, L., & Benczúr, A. (2017). "Online ranking prediction in non-stationary environments." Section 3.5.

[Koren2008] Y. Koren, "Factorization Meets the Neighborhood: A Multifaceted Collaborative Filtering Model," Proc. 14th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining, ACM Press, 2008, pp. 426-434.

PYTHON MODULE INDEX

a

INDEX

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W

Y