Sim4Rec: Flexible and Extensible Simulator for Recommender Systems for Large-Scale Data

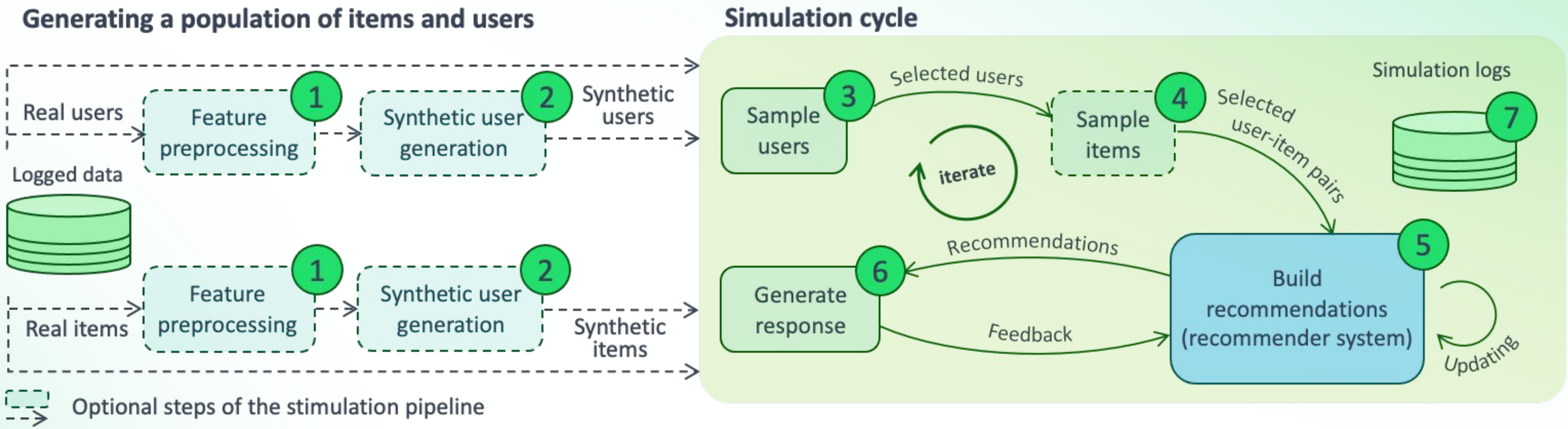
Contents

Motivation: Sim4Rec goals and features

- Evaluation and comparison of recommendation systems:
  simulator as a compromise between counterfactual policy evaluation and A/B testing
- Long-term performance evaluation of the RS for evaluation results that are close to reality
- What-if analysis
- An opportunity to conduct experiments on synthetic data to preserve privacy restrictions

- Ability to work with the large data volumes on industrial tech stack (PySpark)
- Pipeline flexibility and extensibility to be adopted for evaluation of various RS on various recommendations surfaces

Sim4Rec simulation pipeline



- The simulation cycle consists of several iterations to evaluate long-term performance of RS
- The simulation pipeline models work in a batch mode to generate responses for a batch of users and their recommendations
- The simulation cycle can be launched with both real and synthetic data

Sim4Rec architecture

- Open-source framework for Python
- Distributed computing with PySpark
- The main components of Sim4Rec simulation cycle are inherited from PySpark Mllib pipeline elements to be easily extended with the PySpark MLlib functional and embedded into PySpark MLlib Pipelines
- The Synthetic Data Vault (SDV) framework functionality is imported for synthetic data generation and quality assessment

Sim4Rec architecture

Embeddings 1

- Reducing the dimensionality of user/item vector representations

Pyspark MLLib estimator/transformer: PCA, Autoencoder

Generator 2 3

- Synthetic data generation (users/items)
- Sampling of synthetic and real data for simulation iteration
- Sampling from multiple sources in a given ratio (what if analysis)

RealDataGenerator, SDVDataGenerator, CompositeGenerator

Selector 4

- Selection of items available for recommendation to chosen users

Pyspark MLLib estimator/transformer. Base model, all items available

Response 6

- The generation of responses to recommendations
- Can be applied sequentially to obtain several types of response, generating a final response based on responses from several models

Pyspark MLLib estimator/transformer: Constant, Random, Cosine similarity, Bernoulli, Parametric (multicomponent) response function

Evaluation

Evaluation of the quality of :
- synthetic data generation
- the response function with logged responses as a ground truth
- recommendations with generated responses as a ground truth

Synthetic quality metrics (SDV)
Response and RS quality metrics (PySpark MLLib)

Simulator 7

- Store and check consistency of simulation log
- Provide an API to call the core simulation pipeline models

Each module implements base classes to be extended with the custom models

Core modules functional and APIs: Generator

The `generator` module implements functionality to store and sample user data and generate synthetic user and item data

The main methods:

- `fit` to store the real data
- `generate methods` to either subsample real data or generate synthetic data
- `sample` to return a sample of visiting users

`RealDataGenerator` works with the real data

```
real_data_generator = RealDataGenerator(
    label="users_real",
    seed=SEED
)
real_data_generator.fit(users_gen)
real_data_generator.generate(10000)
real_users = real_data_generator.sample(0.1)
print(f"n_samples = {real_users.count()}")
real_users.limit(5).toPandas()
```

n_samples = 1006

	user_idx
0	22079
1	49584
2	51157
3	11094
4	46009

`SDVDataGenerator` generates synthetic data

```
svd_data_generator = SDVDataGenerator(
    label="synth",
    id_column_name="user_id",
    model_name="copulagan",
    parallelization_level=4,
    device_name="cpu",
    seed=SEED,
)
svd_data_generator.fit(user_features.drop("id").sample(0.1))
synthetic_users = svd_data_generator.generate(user_features.sample(0.1).count())
synthetic_users.limit(5).toPandas()
```

	user_id	user_feature[0]	user_feature[1]	user_feature[2]	user_feature[3]
0	synth_0	0.325855	0.187837	0.320260	0.162454
1	synth_1	0.162295	0.158982	0.178573	0.115930
2	synth_2	0.453112	0.386497	0.436773	0.369445
3	synth_3	0.418272	0.172196	0.208842	0.172384
4	synth_4	0.514485	0.273235	0.245768	0.130478

Core modules functional and APIs: Response

Response pipeline

```
als = ALS(
    rank=10,
    maxIter=5,
    userCol="user_idx",
    itemCol="item_idx",
    ratingCol="relevance",
    seed=SEED,
)
als_model = als.fit(train_sim_positive)

va = VectorAssembler(inputCols=["prediction"], outputCol="features")

calibration = LogisticRegression(
    featuresCol="features",
    labelCol="relevance",
    predictionCol="lr_pred",
    probabilityCol="lr_prob",
    maxIter=500,
    tol=1e-2
)
calibration_model = calibration.fit(va.transform(als_model.transform(test_sim)))

vee = VectorElementExtractor(inputCol="lr_prob", outputCol="response_proba", index=1)
br = BernoulliResponse(inputCol="response_proba", outputCol="response", seed=SEED)

response_pipeline = PipelineModel(stages=[als_model, va, calibration_model, vee, br])
predictions = response_pipeline.transform(test_sim).select(
    "user_idx",
    "item_idx",
    "relevance",
    "response_proba",
    "response"
)
predictions.limit(5).toPandas()
```

	user_idx	item_idx	relevance	response_proba	response
0	33689.0	946.0	0	0.637381	0
1	60187.0	946.0	0	0.637381	0
2	93871.0	946.0	1	0.637381	0
3	98698.0	1021.0	1	0.637381	1
4	71965.0	1021.0	0	0.637381	1

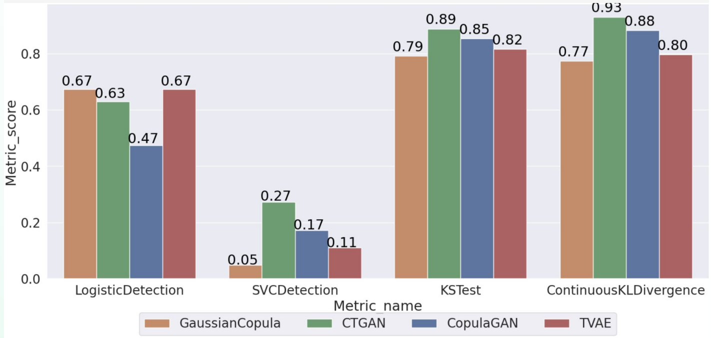
The `Response` module is dedicated to user response modeling. Sim4Rec allows building response pipelines to apply response model components one-by-one.

- Sim4Rec offers `NoiseResponse`, `ConstantResponse`, `CosineSimilatiry` response baselines out of the box.
- PySpark MLlib models, such as `LogisticRegression` or `ALS` could be embedded into the response pipeline.
- The `BernoulliResponse` applied over response probabilities models a non-deterministic nature of user response.

Core modules functional and APIs: Evaluation

Evaluation of synthetic data

```
real_users = user_features.sample(0.1)
synthetic_users = generator.generate(real_users.count())
gen_score = evaluate_synthetic(
    synthetic_users.drop("user_id"),
    real_users.drop("id")
)
```



The `evaluation` module provides functionality to evaluate the quality of the simulation models.

- The quality of synthetic data generation is evaluated with `evaluate_synthetic` function
- The quality of the response model and RS is evaluated with `EvaluateMetrics` class which utilizes PySpark MLlib metrics

The custom metrics are supported.

Evaluation of response model

```
pipeline_eval = EvaluateMetrics(
    userKeyCol="user_idx",
    itemKeyCol="item_idx",
    predictionCol="response_proba",
    labelCol="relevance",
    mllib_metrics=["areaUnderROC"],
)

predictions = response_pipeline.transform(test_sim)
predictions = predictions.withColumn(
    "response_proba", predictions["response_proba"].astype("double")
)
pipeline_eval(predictions)
print(f"ROC-AUC = {pipeline_eval(predictions)['areaUnderROC']}")
```

```
[Stage 3105:>
ROC-AUC = 0.6132784535037881
```

Evaluation of recommender system

```
metrics = []

recs = rs_model.predict(
    log=log, k=50, users=current_users, items=train_items_gen, filter_seen_items=False
).cache()

true_resp = (
    sim.sample_responses(
        recs_df=recs,
        user_features=current_users,
        item_features=train_items_gen,
        action_models=response_pipeline,
    )
    .select("user_idx", "item_idx", "relevance", "response")
    .cache()
)

sim.update_log(true_resp, iteration=0)

metrics.append(n_clicks_per_user(true_resp))
```

```
print(f"n_clicks = {round(metrics[0])}")
```

```
n_clicks = 32
```

Core modules functional and APIs: Simulator

Simulator initialization

```
sim = Simulator(
    user_gen=users_generator,
    item_gen=items_generator,
    data_dir=f"{CHECKPOINT_DIR}/pipeline",
    spark_session=spark
)
```

Users and responses sampling

```
current_users = sim.sample_users(0.1).cache()
log = sim.get_log(train_users_gen)
recs = rs_model.predict(
    log=log,
    k=50,
    users=current_users,
    items=train_items_gen,
    filter_seen_items=False
)
true_resp = sim.sample_responses(
        recs_df=recs,
        user_features=current_users,
        item_features=train_items_gen,
        action_models=response_pipeline,
)
sim.update_log(true_resp, iteration=0)
```

The `Simulator` class stores and checks the consistency of the simulation log with `update_log` function and provides an API to call the core simulation pipeline models.

The simulation data is stored on disk and partitioned by simulation iteration to allow convenient access to the most recent data without loading the entire simulation history into memory.

/ ··· / pipeline / log.parquet /

Name
__iter=0
__iter=1
__iter=2
__iter=3
__iter=4
__iter=5

/ ··· / log.parquet / __iter=0 /

Name
part-00003-d24a06ba-a62c-4198-83f8-b3c1cfe6f26d-c000.snappy.parquet
part-00000-d24a06ba-a62c-4198-83f8-b3c1cfe6f26d-c000.snappy.parquet
part-00002-d24a06ba-a62c-4198-83f8-b3c1cfe6f26d-c000.snappy.parquet
part-00001-d24a06ba-a62c-4198-83f8-b3c1cfe6f26d-c000.snappy.parquet
part-00004-d24a06ba-a62c-4198-83f8-b3c1cfe6f26d-c000.snappy.parquet
part-00007-d24a06ba-a62c-4198-83f8-b3c1cfe6f26d-c000.snappy.parquet

Case studies: Synthetic data generation

1 Fit non-negative ALS for users embeddings

```
# initialization of non-negative ALS
als = ALS(
    rank=64,
    maxIter=5,
    userCol="user_idx",
    itemCol="item_idx",
    ratingCol="relevance",
    seed=SEED,
    nonnegative=True,
)
# fit ALS
als_model = als.fit(train)
```

2 Get users features

```
user_features = als_model.userFactors.orderBy("id")
user_features = (user_features.withColumn("user_feature", col("features"))).select(
    ["id"] + [col("user_feature")[i] for i in range(64)]
)
user_features.limit(5).toPandas()
```

	id	user_feature[0]	user_feature[1]	user_feature[2]	user_feature[3]	user_feature[4]	user_feature[5]
0	0	0.098465	0.052241	0.125119	0.202401	0.178767	0.148965
1	1	0.083297	0.047674	0.103928	0.181666	0.161923	0.128280
2	2	0.089750	0.044578	0.118326	0.188298	0.157684	0.135538
3	3	0.065662	0.048028	0.068513	0.146848	0.143943	0.101966
4	4	0.076065	0.054559	0.085825	0.180649	0.170346	0.122832

5 rows × 65 columns

3 Generate users features with CopulaGAN

```
# initialization of data generator
sdv_data_generator = SDVDataGenerator(
    label="synth",
    id_column_name="user_id",
    model_name="copulagan",
    parallelization_level=4,
    device_name="cpu",
    seed=SEED,
)
# fit data generator
sdv_data_generator.fit(user_features.drop("id").sample(0.1))
# generate user embeddings
synthetic_users = sdv_data_generator.generate(user_features.sample(0.1).count())
synthetic_users.limit(5).toPandas()
```

	user_id	user_feature[0]	user_feature[1]	user_feature[2]	user_feature[3]	user_feature[4]	user_feature[5]
0	synth_0	0.091851	0.032960	0.095349	0.133471	0.118124	0.113068
1	synth_1	0.092978	0.012748	0.124107	0.186155	0.069900	0.149438
2	synth_2	0.110147	0.052691	0.128284	0.213336	0.113021	0.164293
3	synth_3	0.070364	0.015948	0.122228	0.146514	0.019321	0.088826
4	synth_4	0.025496	0.007917	0.074783	0.069922	0.041347	0.024782

5 rows × 65 columns

4 Evaluate generator

```
# sample users
real_users = user_features.sample(0.1)
# generate synthetic users
synthetic_users = sdv_data_generator.generate(real_users.count())
# evaluate the quality of synthetic data
gen_score = evaluate_synthetic(
    synthetic_users.drop("user_id"),
    real_users.drop("id")
)
gen_score
```

```
{'LogisticDetection': 0.017155137360620132,
 'SVCDetection': 0.000693956093678505,
 'KSTest': 0.9043275057028799,
 'ContinuousKLDivergence': 0.5256557092611626}
```

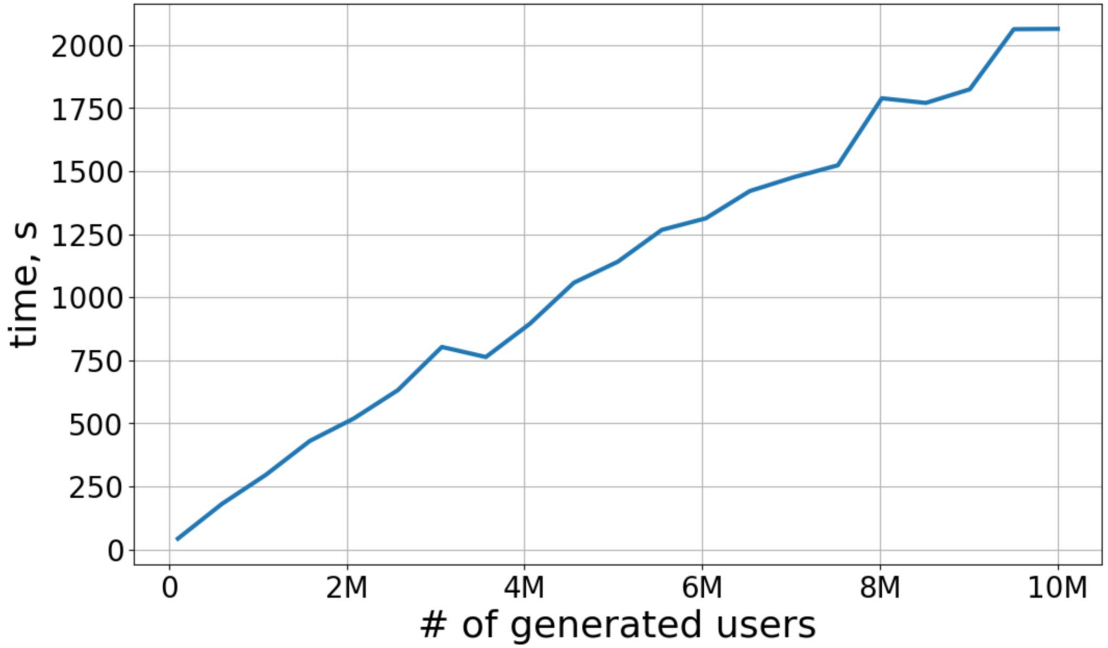
Case studies: Synthetic data generation

The speed of synthetic users' profiles generation with Copula GAN with respect to the number of users

Steps to produce synthetic data:

- train an ALS model on the history of users' interactions to generate the real user vectors representing their profiles
- train `SDVDataGenerator` with the CopulaGAN synthetic data model on the real user vectors
- generate users' profiles

The length of a dense user feature vector is 64.



The figure demonstrates the time to generate user profiles with the trained `SDVDataGenerator` for different user sample sizes.

Case studies: Long-term RS performance evaluation

RS evaluation with a custom response model

- **Users:** 5000 anonymous users
- **Items:** 100 items
- **Response function:** Users are more likely to choose the most popular items.

One iteration of simulation cycle step by step

1 Choice of users

```
user_generator.sample(0.1).count()
```

```
491
```

2 Choice of items

During the simulation cycle, all 100 items will be available at each iteration.

3 Initialization of recommender model

```
# initialization of recommender model
rs_model = UCB(sample=True, seed=SEED)
# fit recommender model
rs_model.fit(
    log=users.limit(1).crossJoin(items.limit(1)).withColumn("relevance", sf.lit(1))
)
# get top k items for each user
pred = rs_model.predict(log=None, users=users.limit(2), items=items, k=2)
pred.limit(5).toPandas()
```

	user_idx	item_idx	relevance
0	0	195	0.005
1	0	76	0.005
2	1	35	0.005
3	1	36	0.005

4 Response Function

```
# initialization of response function
popularity_model = PopBasedTransformer(inputCol="item_idx", outputCol="response_proba")
# initialization of Bernoulli response sampling
br = BernoulliResponse(inputCol="response_proba", outputCol="response", seed=SEED)
# get response pipeline
response_pipeline = Pipeline(stages=[popularity_model, br])
# fit response pipeline
response_model = response_pipeline.fit(items)
# get users' responses on recommended items
test_response = response_model.transform(pred)
test_response.limit(5).toPandas()
```

	user_idx	item_idx	relevance	response_proba	response
0	0	195	0.005	0.5	0
1	0	76	0.005	0.6	0
2	1	35	0.005	0.5	0
3	1	36	0.005	0.6	1

5 Fitting of new recommender model

```
new_log = test_response.drop("relevance", "response_proba").withColumnRenamed(
    "response", "relevance"
)
new_log.limit(5).toPandas()
```

	user_idx	item_idx	relevance
0	0	195	0
1	0	76	0
2	1	35	0
3	1	36	1

```
rs_model.fit(log=new_log)
rs_model.item_popularity.limit(5).toPandas()
```

	item_idx	relevance
0	195	1.442027
1	76	1.442027
2	35	1.442027
3	36	2.442027

6 Quality of recommendations

```
calc_metric(test_response)
```

```
0.5
```

Case studies: Long-term RS performance evaluation

Training the model in the simulator

- Simulator initialization

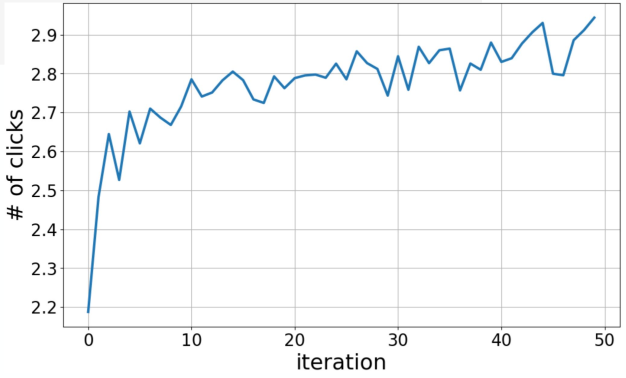
```
user_generator.initSeedSequence(SEED)
item_generator.initSeedSequence(SEED)

sim = Simulator(
    user_gen=user_generator,
    item_gen=item_generator,
    data_dir=f'{CHECKPOINT_DIR}/pipeline',
    user_key_col='user_idx',
    item_key_col='item_idx',
    spark_session=spark
)
```

- Response function initialization

```
response_model = response_pipeline.fit(items)
```

- Simulation cycle

```
# sample users
current_users = sim.sample_users(0.1).cache()
# history of interactions
log = sim.get_log(current_users)
# getting recommendations for sampled users from the recommender system
recs = rs_model.predict(
    log=log, k=5, users=current_users, items=items, filter_seen_items=False
)
# getting responses to recommended items from the response function
true_resp = sim.sample_responses(
    recs_df=recs,
    user_features=current_users,
    item_features=items,
    action_models=response_model,
)
# update user interaction history
sim.update_log(true_resp, iteration=i)
# measure the quality of the recommender system
metrics.append(calc_metric(true_resp))
# refitting the recommender model
rs_model._clear_cache()
train_log = sim.log
rs_model.fit(
    log=train_log.select("user_idx", "item_idx", "response").withColumnRenamed(
        "response", "relevance"
    )
)
```



Case studies: Long-term RS performance evaluation

The speed of core simulation steps with respect to the number of users

The time for each step of the simulation:

- sampling visiting users,
- generating responses to recommendations,
- updating users' history with received responses,
- metric calculation

and the total time of all steps was averaged over 5 runs for each sample size.

The total number of users is 50 million.