RecServer Documentation

Release 0.10.1

Carlos A. Segura Cerna
Michael D. Ekstrand

Nov 21, 2020

OVERVIEW

The recommendation server for Lenskit is a web server that exposes LensKit's recommendation and rating prediction capabilities.

The recommendation server makes it much easier to use LensKit in deployed applications and interactive research where end users are actively involved, extending the reach and research impact of LensKit. The user of the recommendation server can be anyone who knows how to build a web application, such as a software developer or a researcher. They need to create an application, which will itself have end users, and the recommendation server will generate recommendations for them. The recommendation server is able to handle multiple concurrent requests and load and reload of recommendation models. It also provides recommendations on demand based on the latest ratings end users have provided.

CHAPTER

ONE

USER GUIDE

To start using the recommendation server we need to follow some steps:

1) First, you need to create a python virtualenv or an anaconda environment.
2) Install the python dependencies by executing pip or conda.
   - pip install -r requirements.txt
   - conda env update –file environment.yaml
3) Create your model files by using util/train_save_model.py script. You need to have the ratings in your database or in a file. You can configure the database connection, folder paths and file names on util/train_save_model_config.json
4) Configure your database connection and default algorithm for the recommendation server in config.cfg
5) Start gunicorn. For instance, you can start it with 4 workers using the default port by running: gunicorn -w 4 wsgi:app
6) Place the model files inside the lkweb/models folder manually or upload them by using the endpoint '/algorithms/<algo>/modelfile'

You can find more information about how to create the model files in the Training & quality check process

You can easily create a docker image for the recommendation server by using the Dockerfile at the root of the project. For more information read the Deployment guide.

1.1 Functional testing

To make sure everything works fine, you can execute the functional tests by running pytest. We use pytest-docker, so all functional tests will be executed in a docker environment. The whole docker machines (recommendation server and database) will be created for you. You only need to place your items.csv and ratings.csv in the test_db folder and the whole process is automatically executed for you.

CHAPTER

TWO

CONFIGURATION

The main configuration file is config.cfg in the root folder. The things you can configure in this file are:

- DEFAULT_ALGORITHM. It specifies the default algorithm to be used in the recommendation server.
- DB_CONNECTION_*Extending the recommendation server.
- It specifies the different parts of the database connection string.
- RATING_TABLE_TABLE_NAME. It specifies the rating table name.
- RATING_TABLE_USER_COLUMN_NAME. It specifies the user column name of the rating table.
- RATING_TABLE_ITEM_COLUMN_NAME. It specifies the item column name of the rating table.
- RATING_TABLE_RATING_COLUMN_NAME. It specifies the rating column name of the rating table.

2.1 How to extend the recommendation server

It is easy to add a new endpoint in the recommendation server that uses the current model files. We used python decorators to make the methods extensibles. The two files that need to be modified to add new endpoints are app.py and model_manager.py. For an example, check the section “Extending the recommendation server” of Examples

CHAPTER

THREE

DEPLOYMENT

All the packages necessary to run the recommendation server can be installed by using the requirements.txt file or the environment.yaml file, depending if you use pip or conda. Those packages can be easily installed by running either of these commands:

- pip install -r requirements.txt
- conda env update –file environment.yaml

3.1 Docker-compose configuration

If you just want to deploy the recommendation server and a Postgres database in Docker using data files located in test_db/ then follow these steps:

1) Move to the tests folder: cd tests
2) Run the command "docker-compose up".
3) Update the configuration values from util/train_save_model_config.json to reflect your desired values.
4) Remember to set these keys to true in the util/train_save_model_config.json
    - "create_models": true
    - "upload_models": true
5) Run the util/train_save_model.py script to create the model files and upload them to the recommendation server.

3.2 Docker configuration

If you want to use your own database, then you can build only the Dockerfile from lkweb folder. The steps to setup the recommendation server using Docker are:

1) Update the database configuration from config.cfg
2) Move to the lkweb folder: cd lkweb
3) Build the rec server image from the Dockerfile: docker build -t rec-server .
4) Update the values from util/train_save_model_config.json to reflect your desired values.
5) Remember to set these keys to true in the util/train_save_model_config.json
    - "create_models": true
    - "upload_models": true

6) Run the util/train_save_model.py script to create the model files and upload them to the recommendation server.

3.3 No-Docker configuration

The steps to setup the recommendation server without Docker are:

1) Update the database configuration from config.cfg
2) Install the python packages by using the requirements.txt file or the environment.yaml file.
3) Start the recommendation server with 4 workers in gunicorn: gunicorn -w 4 wsgi:app
4) Update the values from util/train_save_model_config.json to reflect your desired values.
5) Remember to set these keys to true in the util/train_save_model_config.json
    - "create_models": true
    - "upload_models": true
6) Run the util/train_save_model.py script to create the model files and upload them to the recommendation server.

It is recommended to create a virtualenv or anaconda environment before step 2.

CHAPTER

FOUR

TRAINING & QUALITY CHECK PROCESS

Before trying to run the train_save_model.py script, we need to install some packages that the script needs to run without issues. The best option to run it is to create a virtualenv or anaconda environment and install the packages from requirements.txt or environment.yaml, depending which one you choose.

4.1 How to create the model files

The script train_save_model.py located in the util folder is used to create the model files. Before running the script, you would like to update the get_algo_class() or get_topn_algo_class() methods to use your desired parameters in each algorithm.

You can simply call it like this: python train_save_model.py

You need to modify the file train_save_model_config.json to reflect your configuration. You can configure the database connection string, the recommendation server url, how to create the models, the algorithms to create models for, and other parameters.

The configuration for the script is defined in train_save_model_config.json. The different keys are:

- data_folder_path. The folder where the data file is.
- models_folder_path. The folder where the models will be saved.
- ratings_file_name. The name of the rating file.
- db_connection. It defines the different parts of the sql connection string.
- create_models. A flag used to indicate if new models will be created in the process.
- create_top_n_models. It is a boolean value (True or False) that specifies if the script will create the models using the topn algorithm.
- create_memory_optimized_models. It is a boolean value (True or False) that specifies if the models will be created with memory maps.
- from_data_files. It is a boolean value (True or False) that specifies if the data comes fron a file or a database.
- upload_models. A flag used to indicate if the models located in /models folder will be uploaded to the recommendation server.
- rec_server_base_url. The url of the recommendation server.
- algorithms. The algorithms to be used in the process. It is an array of strings.

4.2 How to extend the algorithms

If you want to create custom algorithms, simply you can extend the existing ones from Lenskit or create new ones by extending the base classes of Predictor or Recommender. Then, you should modify the script train_save_model.py to import the new algorithm and then update the get_algo_class() or get_topn_algo_class() methods to include the new algorithm in the logic with your desired parameters. Finally, just execute the script train_save_model.py

CHAPTER FIVE

ENDPOINTS

GET | POST /recommendations

Get *num_recs* recommendations from the default configured algorithm for the *user_id* sent.

Example requests:

```
GET /recommendations/?user_id=1&num_recs=5
Host: example.com
Accept: application/json, text/javascript
Content-Length: 20
```

```
POST /recommendations
Host: example.com
Accept: application/json, text/javascript
Content-Length: 26

Data: { "user_id" : 1, "num_recs" : 5 }
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 178

{
"recommendations": [
        {
            "item": 356.0,
            "score": 341.0
        },
        {
            "item": 296.0,
            "score": 324.0
        },
        {
            "item": 318.0,
            "score": 311.0
        },
        {
            "item": 593.0,
            "score": 304.0
        },
        {
            "item": 260.0,
            "score": 291.0
```

(continues on next page)

(continued from previous page)

```
        }
    ]
}
```

Query Parameters

- **user_id** (*int*) – user id to get recommendations for
- **num_recs** (*int*) – number of recommendations to return

JSON Parameters

- **user_id** (*int*) – user id to get recommendations for.
- **num_recs** (*int*) – number of recommendations to return.

GET | POST /algorithms/(string: *algo*)**/recommendations**

Get *num_recs* recommendations using the *algorithm* and *user_id* sent.

Example requests:

```
GET /algorithms/popular/recommendations?user_id=1&num_recs=5
Host: example.com
Accept: application/json, text/javascript
Content-Length: 20
```

```
POST /algorithms/popular/recommendations
Host: example.com
Accept: application/json, text/javascript
Content-Length: 26

Data: { "user_id" : 1, "num_recs" : 5 }
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 178

{
"recommendations": [
        {
            "item": 356.0,
            "score": 341.0
        },
        {
            "item": 296.0,
            "score": 324.0
        },
        {
            "item": 318.0,
            "score": 311.0
        },
        {
            "item": 593.0,
            "score": 304.0
        },
```

(continues on next page)

(continued from previous page)

```
        {
            "item": 260.0,
            "score": 291.0
        }
    ]
}
```

Query Parameters

- **user_id** (*int*) – user id to get recommendations for
- **num_recs** (*int*) – number of recommendations to return

JSON Parameters

- **user_id** (*int*) – user id to get recommendations for.
- **num_recs** (*int*) – number of recommendations to return.

GET | POST /algorithms/(string: *algo*)/predictions

Get predictions using the *algorithm*, *user_id* and *items* sent.

Example requests:

```
GET /algorithms/bias/predictions?user_id=1&items=5,102,203,304,400
Host: example.com
Accept: application/json, text/javascript
Content-Length: 33
```

```
POST /algorithms/bias/predictions
Host: example.com
Accept: application/json, text/javascript
Content-Length: 39

Data: { "user_id" : 1, "items" : 5,102,203,304,400 }
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 212

{
"predictions": [
        {
            "item": 5,
            "score": 3.268
        },
        {
            "item": 102,
            "score": 2.591
        },
        {
            "item": 203,
            "score": 3.304
        },
        {
```

(continues on next page)

(continued from previous page)

```
            "item": 304,
            "score": 3.333
        },
        {
            "item": 400,
            "score": 3.544
        }
    ]
}
```

Query Parameters

- **user_id** (*int*) – user id to get predictions for
- **items** (*list_of_ints*) – items to get predictions for

JSON Parameters

- **user_id** (*int*) – user id to get predictions for
- **items** (*list_of_ints*) – items to get predictions for

GET /algorithms/(string: *algo***)/info**

Get the model file information from the *algorithm* sent.

Example requests:

```
GET /algorithms/popular/info
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 105

{
    "model": {
        "creation_date": "2020-08-28 18:38:42",
        "size": 200.826,
        "updated_date": "2020-08-21 18:32:55"
    }
}
```

PUT /algorithms/(string: *algo***)/modelfile**

Update the model file for the *algorithm* and *file* sent.

Example requests:

```
PUT /algorithms/popular/modelfile
Host: example.com
Content-Length: 103863987
Content-Type: multipart/form-data;
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 15

{ 'result' : 200 }
```

CHAPTER

SIX

EXAMPLES

This notebook shows some examples of how to use the recommendation server.

6.1 Setup

```
[1]: import requests
     import json
```

Let's define some variable we are going to use in the examples

```
[2]: rec_server_base_url = "http://127.0.0.1:5000/"
     rec_algos = ["popular"]
     pred_algos = ["bias","itemitem","useruser","biasedmf","implicitmf","funksvd","tf_bpr"]
     user_id = 1
     nr_recs = 10
     items = "10,20,30,40,50"
```

6.2 Recommendations

```
[3]: for algo in rec_algos:
         url = f'{rec_server_base_url}/algorithms/{algo}/recommendations?user_id={user_id}&
     ↪num_recs={nr_recs}'
         r = requests.get(url)
         print(json.dumps(r.json(), indent=2))
```

```
{
  "recommendations": [
    {
      "item": 356.0,
      "score": 341.0
    },
    {
      "item": 318.0,
      "score": 311.0
    },
    {
      "item": 593.0,
      "score": 304.0
    },
    {
```

(continues on next page)

(continued from previous page)

```
      "item": 260.0,
      "score": 291.0
    },
    {
      "item": 480.0,
      "score": 274.0
    },
    {
      "item": 2571.0,
      "score": 259.0
    },
    {
      "item": 1.0,
      "score": 247.0
    },
    {
      "item": 527.0,
      "score": 244.0
    },
    {
      "item": 589.0,
      "score": 237.0
    },
    {
      "item": 1196.0,
      "score": 234.0
    }
  ]
}
```

6.3 Predictions

[4]:
```
for algo in pred_algos:
    print(f"Algorithm: {algo}")
    url = f'{rec_server_base_url}/algorithms/{algo}/predictions?user_id={user_id}&
↪items={items}'
    r = requests.get(url)
    print(json.dumps(r.json(), indent=2))
    print()
    print('-----------------------------')
    print()
```

```
Algorithm: bias
{
  "predictions": [
    {
      "item": 10,
      "score": 3.4674757437886936
    },
    {
      "item": 20,
      "score": 2.5551176101190847
    },
    {
```

(continues on next page)

(continued from previous page)

```
      "item": 30,
      "score": 4.066656071657546
    },
    {
      "item": 40,
      "score": 3.933322738324213
    },
    {
      "item": 50,
      "score": 4.387302837826703
    }
  ]
}

-----------------------------

Algorithm: itemitem
{
  "predictions": [
    {
      "item": 10,
      "score": 3.231073190470284
    },
    {
      "item": 20,
      "score": 2.9305180210428925
    },
    {
      "item": 30,
      "score": 4.073950231233046
    },
    {
      "item": 40,
      "score": 3.986381186580402
    },
    {
      "item": 50,
      "score": 4.580127129467687
    }
  ]
}

-----------------------------

Algorithm: useruser
{
  "predictions": [
    {
      "item": 10,
      "score": 3.4913655291780747
    },
    {
      "item": 20,
      "score": 3.516813277855599
    },
    {
      "item": 30,
```

(continues on next page)

(continued from previous page)

```
      "score": 4.204543778213467
    },
    {
      "item": 40,
      "score": 4.672953331143783
    },
    {
      "item": 50,
      "score": 4.4831597852166025
    }
  ]
}

-----------------------------

Algorithm: biasedmf
{
  "predictions": [
    {
      "item": 10,
      "score": 3.2654584201076498
    },
    {
      "item": 20,
      "score": 3.20231997852298
    },
    {
      "item": 30,
      "score": 4.129488327175242
    },
    {
      "item": 40,
      "score": 3.6586054841117126
    },
    {
      "item": 50,
      "score": 4.600072996787447
    }
  ]
}

-----------------------------

Algorithm: implicitmf
{
  "predictions": [
    {
      "item": 10,
      "score": 0.7739809836712266
    },
    {
      "item": 20,
      "score": -0.14057957568765025
    },
    {
      "item": 30,
      "score": 0.30112599732114315
```

(continues on next page)

(continued from previous page)

```
    },
    {
      "item": 40,
      "score": 0.150151834788069
    },
    {
      "item": 50,
      "score": 0.9581236325369329
    }
  ]
}

-----------------------------

Algorithm: funksvd
{
  "predictions": [
    {
      "item": 10,
      "score": 2.763367031201388
    },
    {
      "item": 20,
      "score": 2.16933683430268
    },
    {
      "item": 30,
      "score": 3.238171076963463
    },
    {
      "item": 40,
      "score": 3.1055437883876067
    },
    {
      "item": 50,
      "score": 3.635343170703555
    }
  ]
}

-----------------------------

Algorithm: tf_bpr
{
  "predictions": [
    {
      "item": 10,
      "score": 4.359846115112305
    },
    {
      "item": 20,
      "score": 1.9790164232254028
    },
    {
      "item": 30,
      "score": 1.851624608039856
    },
```

(continues on next page)

(continued from previous page)

```
    {
      "item": 40,
      "score": 1.1883563995361328
    },
    {
      "item": 50,
      "score": 6.511105537414551
    }
  ]
}

------------------------------
```

6.4 Get model information

[5]:
```
for algo in pred_algos:
    print(f"Algorithm: {algo}")
    url = f'{rec_server_base_url}/algorithms/{algo}/info'
    r = requests.get(url)
    print(json.dumps(r.json(), indent=2))
    print()
    print('------------------------------')
    print()
```

```
Algorithm: bias
{
  "model": {
    "creation_date": "2020-10-19 23:42:14",
    "size": 739.388,
    "updated_date": "2020-10-19 23:42:14"
  }
}

------------------------------

Algorithm: itemitem
{
  "model": {
    "creation_date": "2020-10-21 16:47:08",
    "size": 212330.473,
    "updated_date": "2020-10-21 16:47:08"
  }
}

------------------------------

Algorithm: useruser
{
  "model": {
    "creation_date": "2020-10-15 21:16:28",
    "size": 688.368,
    "updated_date": "2020-08-21 18:33:27"
  }
```

(continues on next page)

(continued from previous page)

```
}

-----------------------------

Algorithm: biasedmf
{
  "model": {
    "creation_date": "2020-10-15 21:16:28",
    "size": 3973.614,
    "updated_date": "2020-10-08 21:10:51"
  }
}

-----------------------------

Algorithm: implicitmf
{
  "model": {
    "creation_date": "2020-10-15 21:16:28",
    "size": 1507.69,
    "updated_date": "2020-08-21 18:33:45"
  }
}

-----------------------------

Algorithm: funksvd
{
  "model": {
    "creation_date": "2020-10-15 21:16:28",
    "size": 1522.498,
    "updated_date": "2020-08-21 18:33:49"
  }
}

-----------------------------

Algorithm: tf_bpr
{
  "model": {
    "creation_date": "2020-10-27 17:25:43",
    "size": 19515.322,
    "updated_date": "2020-10-27 17:25:43"
  }
}

-----------------------------
```

6.5 Upload a model file

[6]:
```
current_algo = rec_algos[0]
url = f'{rec_server_base_url}/algorithms/{current_algo}/modelfile'
model_name = current_algo + ".bpk"
files = {
    'file': open(model_name, 'rb')
}
response = requests.put(url, files=files)
print(json.dumps(r.json(), indent=2))
```

```
{
  "model": {
    "creation_date": "2020-10-27 17:25:43",
    "size": 19515.322,
    "updated_date": "2020-10-27 17:25:43"
  }
}
```

6.6 Extending the recommendation server

An example of extending the recommendation server:

Create a new custom endpoint where we return the worst predictions for the algo sent in the request.

We created a new method called `get_worst_predictions_from_model` in lkweb/model_manager.py and another endpoint called `get_worst_predictions` with the python decorator: @models.model_method("worst_predictions", Predictor, models.get_worst_predictions_from_model, models.get_preds_params) in app.py

And that's it, we have a new endpoint that returns the worst predictions for an algorithm. The code is shown below:

[7]:
```
def get_worst_predictions_from_model(self, model, *args):
    user, items = None, None
    try:
        user, items, ratings = args[0][0], args[0][1], args[0][2]
        results = []
        df_preds = model.predict_for_user(user, items, ratings)
        for index, value in df_preds.iteritems():
            if not math.isnan(value):
                results.append({'item': index, 'score': value})
        results = sorted(results, key = lambda i: i['score'])
        return results
    except:
        logging.error(f"Unexpected preds error for user: {user}, with items: {items}.␣
↪Error: {sys.exc_info()[0]}")
        raise
```

[8]:
```
def get_worst_predictions(results):
    """
    Get worst predictions using the algorithm, user id and items sent.
    Args:
        algo: algorithm to be used.
        user_id: user id to get predictions for.
        items: items to get predictions for.
```

(continues on next page)

(continued from previous page)

```
    Returns:
        A list of predictions with items and scores.
    """
    return jsonify({"predictions": results})
```

Let's test the new endpoint.

```
[9]: for algo in pred_algos:
         print(f"Algorithm: {algo}")
         url = f'{rec_server_base_url}/algorithms/{algo}/worst_predictions?user_id={user_
     ↪id}&items={items}'
         r = requests.get(url)
         print(json.dumps(r.json(), indent=2))
         print()
         print('------------------------------')
         print()
```

```
Algorithm: bias
{
  "predictions": [
    {
      "item": 20,
      "score": 2.5551176101190847
    },
    {
      "item": 10,
      "score": 3.4674757437886936
    },
    {
      "item": 40,
      "score": 3.933322738324213
    },
    {
      "item": 30,
      "score": 4.066656071657546
    },
    {
      "item": 50,
      "score": 4.387302837826703
    }
  ]
}

------------------------------

Algorithm: itemitem
{
  "predictions": [
    {
      "item": 20,
      "score": 2.9305180210428925
    },
    {
      "item": 10,
      "score": 3.231073190470284
    },
    {
```

(continues on next page)

(continued from previous page)

```
      "item": 40,
      "score": 3.986381186580402
    },
    {
      "item": 30,
      "score": 4.073950231233046
    },
    {
      "item": 50,
      "score": 4.580127129467687
    }
  ]
}

-----------------------------

Algorithm: useruser
{
  "predictions": [
    {
      "item": 10,
      "score": 3.4913655291780747
    },
    {
      "item": 20,
      "score": 3.516813277855599
    },
    {
      "item": 30,
      "score": 4.204543778213467
    },
    {
      "item": 50,
      "score": 4.4831597852166025
    },
    {
      "item": 40,
      "score": 4.672953331143783
    }
  ]
}

-----------------------------

Algorithm: biasedmf
{
  "predictions": [
    {
      "item": 20,
      "score": 3.20231997852298
    },
    {
      "item": 10,
      "score": 3.2654584201076498
    },
    {
      "item": 40,
```

(continues on next page)

(continued from previous page)

```
      "score": 3.6586054841117126
    },
    {
      "item": 30,
      "score": 4.129488327175242
    },
    {
      "item": 50,
      "score": 4.600072996787447
    }
  ]
}

-----------------------------

Algorithm: implicitmf
{
  "predictions": [
    {
      "item": 20,
      "score": -0.14057957568765025
    },
    {
      "item": 40,
      "score": 0.150151834788069
    },
    {
      "item": 30,
      "score": 0.30112599732114315
    },
    {
      "item": 10,
      "score": 0.7739809836712266
    },
    {
      "item": 50,
      "score": 0.9581236325369329
    }
  ]
}

-----------------------------

Algorithm: funksvd
{
  "predictions": [
    {
      "item": 20,
      "score": 2.16933683430268
    },
    {
      "item": 10,
      "score": 2.763367031201388
    },
    {
      "item": 40,
      "score": 3.1055437883876067
```

(continues on next page)

(continued from previous page)

```
    },
    {
      "item": 30,
      "score": 3.238171076963463
    },
    {
      "item": 50,
      "score": 3.635343170703555
    }
  ]
}

------------------------------

Algorithm: tf_bpr
{
  "predictions": [
    {
      "item": 40,
      "score": 1.1883563995361328
    },
    {
      "item": 30,
      "score": 1.851624608039856
    },
    {
      "item": 20,
      "score": 1.9790164232254028
    },
    {
      "item": 10,
      "score": 4.359846115112305
    },
    {
      "item": 50,
      "score": 6.511105537414551
    }
  ]
}

------------------------------
```

[ ]:

```

```

CHAPTER

SEVEN

ACCURACY OF MODELS USING LENSKIT

This notebook shows how to test the recommendations accuracy of models using lenskit.

7.1 Setup

```
[1]: from lenskit.datasets import MovieLens
from lenskit import batch, topn, util
from lenskit import crossfold as xf
from lenskit.algorithms import Recommender, als, item_knn, basic
import lenskit.metrics.predict as pm
import pandas as pd
```

```
/Users/carlos/anaconda3/lib/python3.7/site-packages/fastparquet/encoding.py:222:␣
↪NumbaDeprecationWarning: The 'numba.jitclass' decorator has moved to 'numba.
↪experimental.jitclass' to better reflect the experimental nature of the␣
↪functionality. Please update your imports to accommodate this change and see http://
↪numba.pydata.org/numba-doc/latest/reference/deprecation.html#change-of-jitclass-
↪location for the time frame.
  Numpy8 = numba.jitclass(spec8)(NumpyIO)
/Users/carlos/anaconda3/lib/python3.7/site-packages/fastparquet/encoding.py:224:␣
↪NumbaDeprecationWarning: The 'numba.jitclass' decorator has moved to 'numba.
↪experimental.jitclass' to better reflect the experimental nature of the␣
↪functionality. Please update your imports to accommodate this change and see http://
↪numba.pydata.org/numba-doc/latest/reference/deprecation.html#change-of-jitclass-
↪location for the time frame.
  Numpy32 = numba.jitclass(spec32)(NumpyIO)
/Users/carlos/anaconda3/lib/python3.7/site-packages/fastparquet/dataframe.py:5:␣
↪FutureWarning: pandas.core.index is deprecated and will be removed in a future␣
↪version.  The public classes are available in the top-level namespace.
  from pandas.core.index import CategoricalIndex, RangeIndex, Index, MultiIndex
```

7.2 Load data

```
[3]: mlens = MovieLens('data/ml-latest-small')
ratings = mlens.ratings
ratings.head()
```

```
[3]:    user  item  rating   timestamp
0      1    31     2.5  1260759144
1      1  1029     3.0  1260759179
```

(continues on next page)

(continued from previous page)

```
2      1  1061      3.0  1260759182
3      1  1129      2.0  1260759185
4      1  1172      4.0  1260759205
```

7.3 Define algorithms

[4]:
```
biasedmf = als.BiasedMF(50)
bias = basic.Bias()
itemitem = item_knn.ItemItem(20)
```

7.4 Evaluate recommendations

[5]:
```
def create_recs(name, algo, train, test):
    fittable = util.clone(algo)
    fittable = Recommender.adapt(fittable)
    fittable.fit(train)
    users = test.user.unique()
    # now we run the recommender
    recs = batch.recommend(fittable, users, 100)
    # add the algorithm name for analyzability
    recs['Algorithm'] = name
    return recs
```

We loop over the data to generate recommendations for the defined algorithms.

[6]:
```
all_recs = []
test_data = []
for train, test in xf.partition_users(ratings[['user', 'item', 'rating']], 5, xf.
↪SampleFrac(0.2)):
    test_data.append(test)
    all_recs.append(create_recs('ItemItem', itemitem, train, test))
    all_recs.append(create_recs('BiasedMF', biasedmf, train, test))
    all_recs.append(create_recs('Bias', bias, train, test))
```

We create a single data frame with the recommendations

[7]:
```
all_recs = pd.concat(all_recs, ignore_index=True)
all_recs.head()
```

[7]:
```
     item     score  user  rank Algorithm
0    3171  5.366279     9     1  ItemItem
1  104283  5.279667     9     2  ItemItem
2   27803  5.105468     9     3  ItemItem
3    4338  5.037831     9     4  ItemItem
4   86000  4.991602     9     5  ItemItem
```

We also concatenate the test data

[8]:
```
test_data = pd.concat(test_data, ignore_index=True)
```

Let's analyse the recommendation lists

```
[9]: rla = topn.RecListAnalysis()
     rla.add_metric(topn.ndcg)
     results = rla.compute(all_recs, test_data)
     results.head()
```

```
[9]:                 nrecs  ndcg
     Algorithm user
     Bias      1     100.0   0.0
               2     100.0   0.0
               3     100.0   0.0
               4     100.0   0.0
               5     100.0   0.0
```

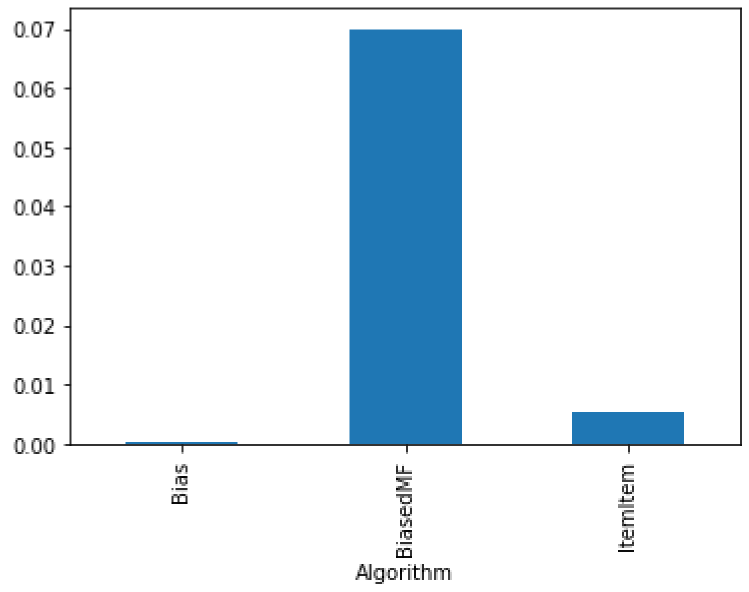
Let's see the nDCG mean value for each algorithm

```
[10]: results.groupby('Algorithm').ndcg.mean()
```

```
[10]: Algorithm
      Bias        0.000309
      BiasedMF    0.069957
      ItemItem    0.005367
      Name: ndcg, dtype: float64
```

```
[11]: results.groupby('Algorithm').ndcg.mean().plot.bar()
```

```
[11]: <matplotlib.axes._subplots.AxesSubplot at 0x12335b110>
```



7.5 Evaluate prediction accuracy

[12]:
```
def evaluate_predictions(name, algo, train, test):
    algo_cloned = util.clone(algo)
    algo_cloned.fit(train)
    return test.assign(preds=algo_cloned.predict(test), algo=name)
```

[13]:
```
preds_bias = pd.concat(evaluate_predictions('bias', bias, train, test) for (train,␣
↪test) in xf.partition_users(ratings, 5, xf.SampleFrac(0.2)))
preds_biasedmf = pd.concat(evaluate_predictions('biasedmf', biasedmf, train, test)␣
↪for (train, test) in xf.partition_users(ratings, 5, xf.SampleFrac(0.2)))
preds_itemitem = pd.concat(evaluate_predictions('itemitem', itemitem, train, test)␣
↪for (train, test) in xf.partition_users(ratings, 5, xf.SampleFrac(0.2)))
```

7.5.1 Bias

[14]:
```
print(f'MAE: {pm.mae(preds_bias.preds, preds_bias.rating)}')
print(f'RMSE: {pm.rmse(preds_bias.preds, preds_bias.rating)}')
```

```
MAE: 0.6950106667260073
RMSE: 0.9066546007561017
```

7.5.2 BiasedMF

[15]:
```
print(f'MAE: {pm.mae(preds_biasedmf.preds, preds_biasedmf.rating)}')
print(f'RMSE: {pm.rmse(preds_biasedmf.preds, preds_biasedmf.rating)}')
```

```
MAE: 0.6818618886318303
RMSE: 0.8911595961607526
```

7.5.3 ItemItem

[16]:
```
print(f'MAE: {pm.mae(preds_itemitem.preds, preds_itemitem.rating)}')
print(f'RMSE: {pm.rmse(preds_itemitem.preds, preds_itemitem.rating)}')
```

```
MAE: 0.6640965754633255
RMSE: 0.8730680515165724
```

[ ]:
```

```

CHAPTER

EIGHT

PERFORMANCE TESTING RESULTS

We provide benchmarks for the prediction and recommendation endpoints for different algorithms using our reference machine of four workers sending 1000 requests to the recommendation server. The results for each algorithm are detailed below.

8.1 Setup

[2]:
```
import json
import pickle
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
```

[3]:
```
class ConfigReader:
    def get_value(self, key):
        with open('config.json') as json_data_file:
            data = json.load(json_data_file)
        return data[key]
```

[20]:
```
results_folder = "results/"

def print_stats_from_file(file_name):
    obj = pickle.load(open(results_folder + file_name, "rb"))
    times = obj['times']
    time_taken_all = obj['time_taken_all']
    num_requests = obj['num_requests']
    print(f'Number of requests: {num_requests}')
    print(f'Total response time: {round(time_taken_all, 6)}')
    print(f'Throughput (requests per second): {round(num_requests / time_taken_all,␣
↪6)}')
    print(f'Peak response time: {round(max(times), 6)}')
    print(f'Mean response time: {round(np.mean(times), 6)}')
    print(f'Median response time: {round(np.median(times), 6)}')
    print(f'99 percentile: {round(np.quantile(times, 0.99), 6)}')

def plot_numbers(file_name):
    obj = pickle.load(open(results_folder + file_name, "rb"))
    resp_time_per_request = obj['times']
    plt.plot(resp_time_per_request)
    plt.show()
```

(continues on next page)

(continued from previous page)

```
def hist_numbers(file_name):
    obj = pickle.load(open(results_folder + file_name, "rb"))
    resp_time_per_request = obj['times']
    plt.hist(resp_time_per_request, bins='auto')
    plt.title('Number of requests over time')
    plt.xlabel('Time')
    plt.ylabel('Requests')
    plt.show()

def create_df(file_name, algo, num_workers):
    obj = pickle.load(open(results_folder + file_name, "rb"))
    df = pd.DataFrame(obj['times'])
    df['Workers'] = num_workers
    df['Algorithm'] = algo
    df.rename(columns={0:'Time'}, inplace=True)
    return df
```

8.1.1 Get config values

[6]:
```
reader = ConfigReader()
n_rand_users = num_requests = reader.get_value("num_requests")
base_url = reader.get_value("rec_server_base_url")
n_recs = reader.get_value("n_recs")
items = reader.get_value("items")
pred_algos = reader.get_value("pred_algos")
rec_algos = reader.get_value("rec_algos")
lk_recserver_algos = reader.get_value('lk_recserver_algos')
linear_speedup_algos = reader.get_value("linear_speedup_algos")
workers_config = reader.get_value("workers_config")
```

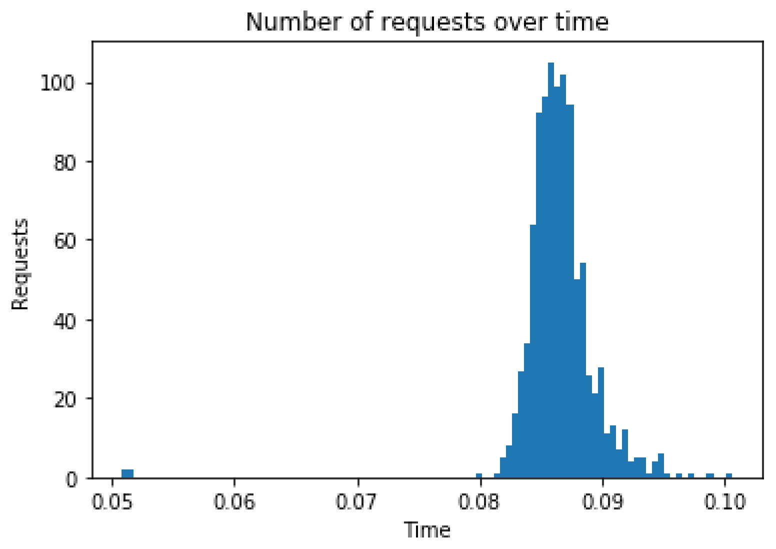
8.2 Show results

8.2.1 Predict and recommend endpoints from server for canonical config

Predictions for different algorithms

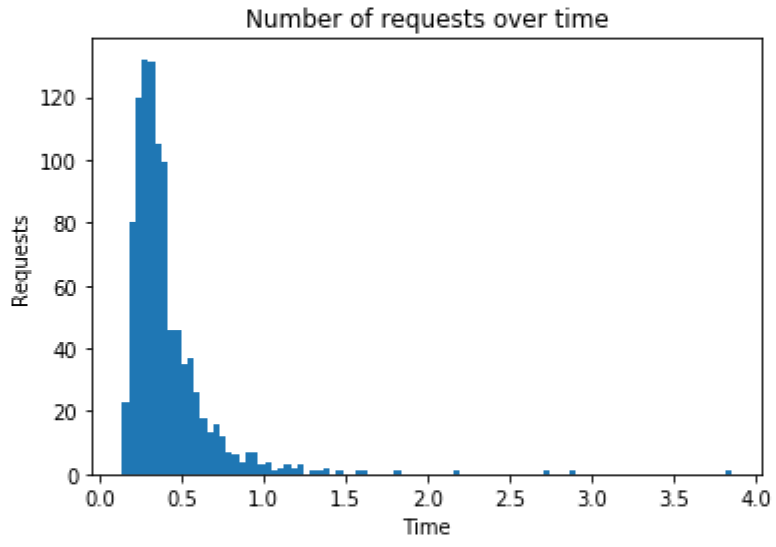
[5]:
```
for algo in pred_algos:
    print(f'Algorithm: {algo}')
    file_name = f'preds_{algo}_workers_4_num_req_{num_requests}.pickle'
    print_stats_from_file(file_name)
    hist_numbers(file_name)
    print('----------------------')
    print('')
```

```
Algorithm: bias
Number of requests: 1000
Total response time: 10.96
Throughput (requests per second): 91.242
Peak response time: 0.101
Mean response time: 0.087
99 percentile: 0.095
```



```
---------------------

Algorithm: itemitem
Number of requests: 1000
Total response time: 51.418
Throughput (requests per second): 19.448
Peak response time: 3.846
Mean response time: 0.407
99 percentile: 1.342
```



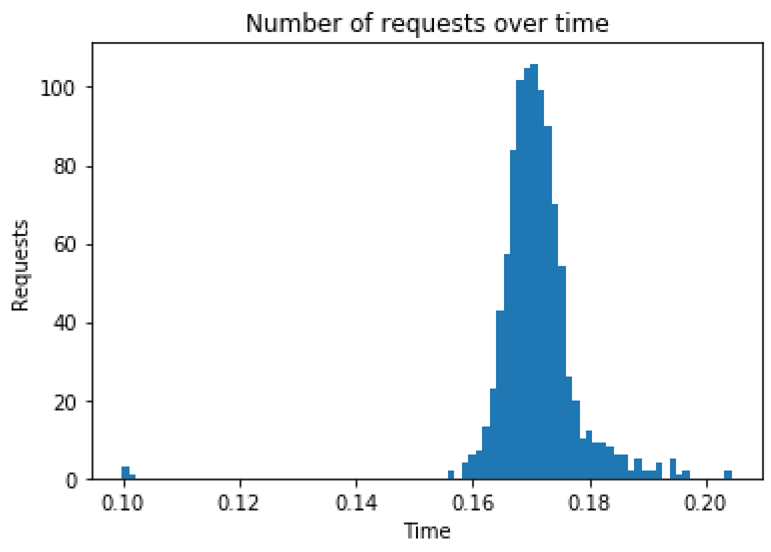
```
---------------------

Algorithm: useruser
Number of requests: 1000
Total response time: 21.567
Throughput (requests per second): 46.367
Peak response time: 0.204
Mean response time: 0.171
```

(continues on next page)

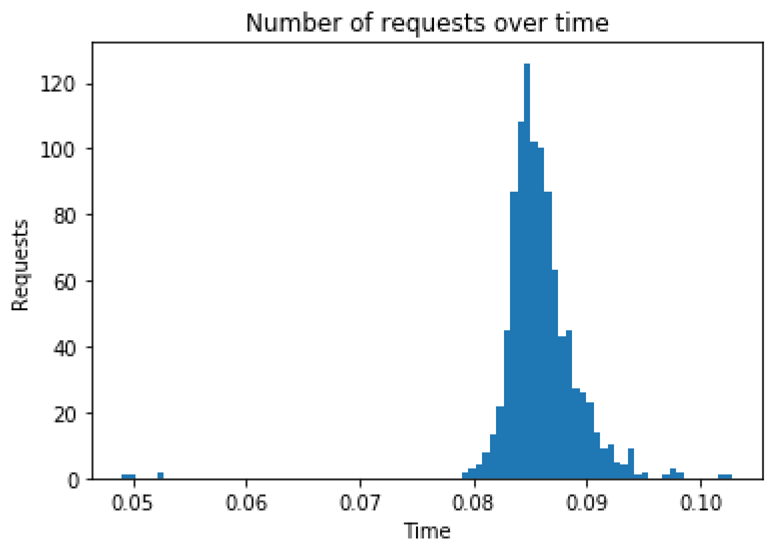
(continued from previous page)

```
99 percentile: 0.192
```



```
---------------------

Algorithm: biasedmf
Number of requests: 1000
Total response time: 10.873
Throughput (requests per second): 91.972
Peak response time: 0.103
Mean response time: 0.086
99 percentile: 0.094
```



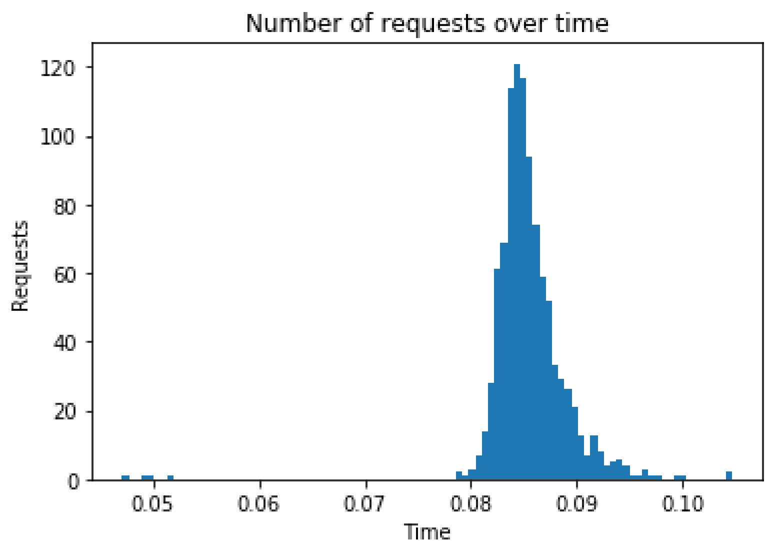
```
---------------------

Algorithm: implicitmf
Number of requests: 1000
Total response time: 10.826
Throughput (requests per second): 92.373
```

(continues on next page)

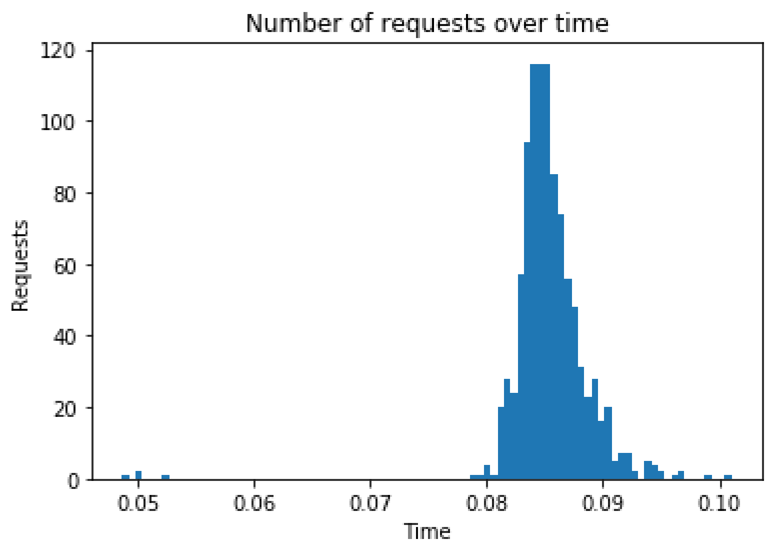
(continued from previous page)

```
Peak response time: 0.105
Mean response time: 0.086
99 percentile: 0.095
```



```
---------------------

Algorithm: funksvd
Number of requests: 1000
Total response time: 10.826
Throughput (requests per second): 92.368
Peak response time: 0.101
Mean response time: 0.085
99 percentile: 0.094
```



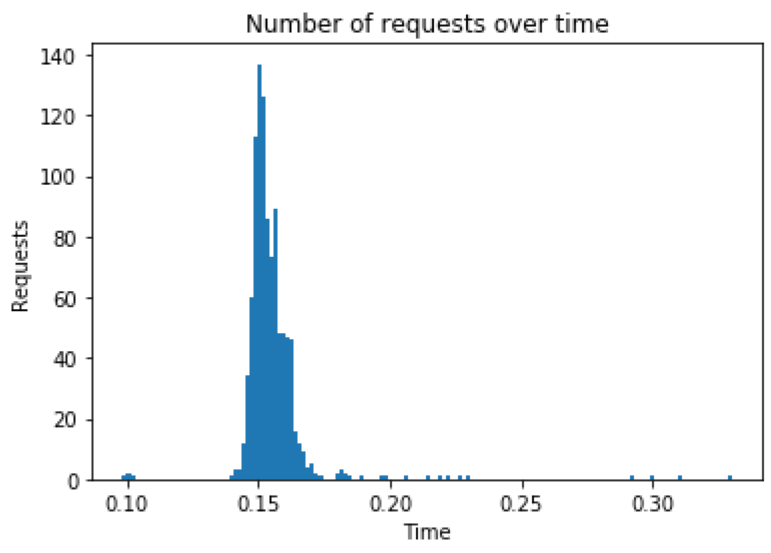
```
---------------------

Algorithm: tf_bpr
Number of requests: 1000
```

(continues on next page)

(continued from previous page)

```
Total response time: 19.537
Throughput (requests per second): 51.186
Peak response time: 0.33
Mean response time: 0.155
99 percentile: 0.199
```

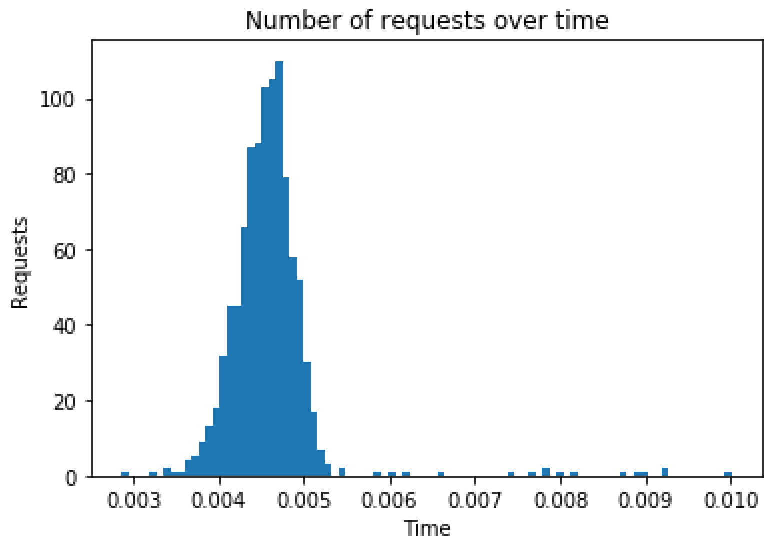


```
----------------------
```

Recommendations

```
[6]: for algo_rec in rec_algos:
         print(f'Algorithm: {algo_rec}')
         file_name = f'recs_{algo_rec}_workers_4_num_req_{num_requests}.pickle'
         print_stats_from_file(file_name)
         hist_numbers(file_name)
         print('----------------------')
         print('')
```

```
Algorithm: popular
Number of requests: 1000
Total response time: 0.682
Throughput (requests per second): 1467.233
Peak response time: 0.01
Mean response time: 0.005
99 percentile: 0.008
```



```
----------------------
```

8.2.2 Lenskit

[9]:
```
def create_df_lk(file_name, algo, type_result):
    obj = pickle.load(open(results_folder + file_name, "rb"))
    df = pd.DataFrame(obj['times'])
    df['Type'] = type_result
    df['Algorithm'] = algo
    df.rename(columns={0:'Time'}, inplace=True)
    return df
```

[21]:
```
for lk_recserver_algo in lk_recserver_algos:
    print(f'Algo: {lk_recserver_algo}')
    print('-------------------')
    print('Lenskit performance:')
    file_name = f'lkpy_{lk_recserver_algo}_num_req_{num_requests}.pickle'
    print_stats_from_file(file_name)
#     hist_numbers(file_name)
    print('-------------------')
    print('Recommendation server performance:')
    file_name = f'preds_{lk_recserver_algo}_against_lkpy_workers_4_num_req_{num_
↪requests}.pickle'
    print_stats_from_file(file_name)
#     hist_numbers(file_name)

    # boxplot
    df = pd.DataFrame({'Times': [], 'Type': [], 'Algorithm': []})
    file_name = f'lkpy_{lk_recserver_algo}_num_req_{num_requests}.pickle'
    df = df.append(create_df_lk(file_name, lk_recserver_algo, 'Lenskit'), ignore_
↪index=True)
    file_name = f'preds_{lk_recserver_algo}_against_lkpy_workers_4_num_req_{num_
↪requests}.pickle'
    df = df.append(create_df_lk(file_name, lk_recserver_algo, 'Recommendation Server
↪'), ignore_index=True)
```

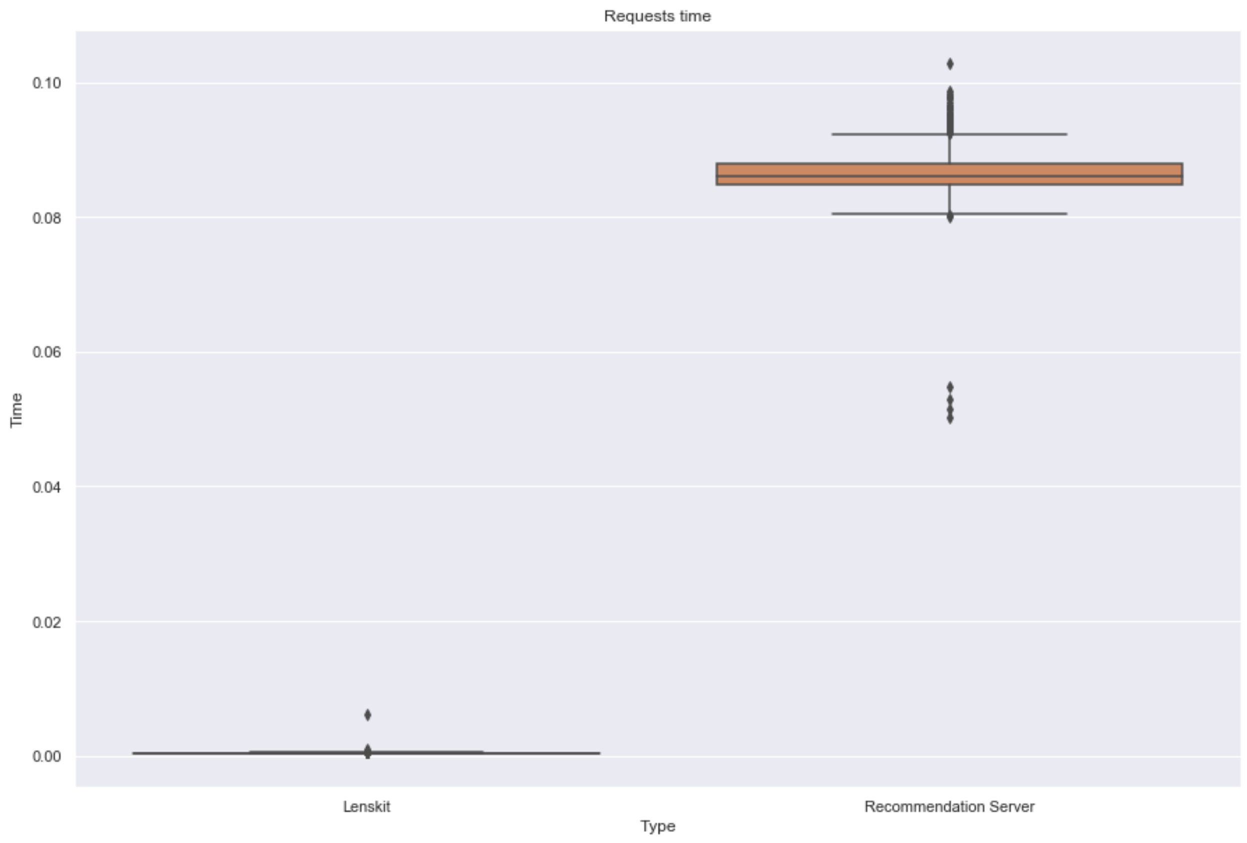
(continues on next page)

(continued from previous page)

```
    plt.title('Requests time')
    ax = sns.boxplot(x="Type", y="Time", data=df)
    plt.show()

    print('******************************************************')
```

```
Algo: bias
------------------
Lenskit performance:
Number of requests: 1000
Total response time: 0.494293
Throughput (requests per second): 2023.090986
Peak response time: 0.006024
Mean response time: 0.000424
Median response time: 0.000417
99 percentile: 0.000445
------------------
Recommendation server performance:
Number of requests: 1000
Total response time: 10.940877
Throughput (requests per second): 91.400348
Peak response time: 0.10275
Mean response time: 0.086518
Median response time: 0.086157
99 percentile: 0.096198
```

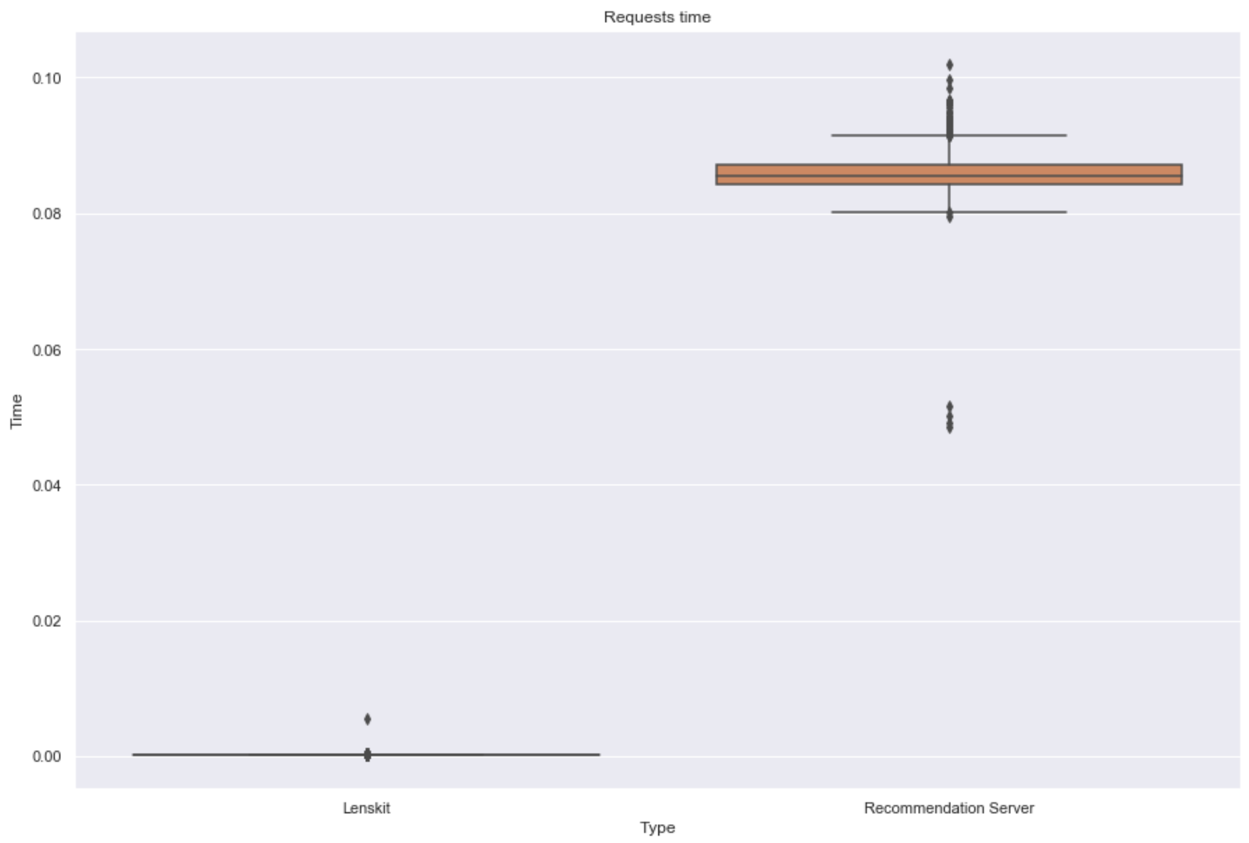


```
******************************************************
Algo: biasedmf
```

(continues on next page)

(continued from previous page)

```
------------------
Lenskit performance:
Number of requests: 1000
Total response time: 0.341478
Throughput (requests per second): 2928.449627
Peak response time: 0.005458
Mean response time: 0.000272
Median response time: 0.000265
99 percentile: 0.000295
------------------
Recommendation server performance:
Number of requests: 1000
Total response time: 10.87211
Throughput (requests per second): 91.978464
Peak response time: 0.101846
Mean response time: 0.085913
Median response time: 0.085471
99 percentile: 0.095898
```

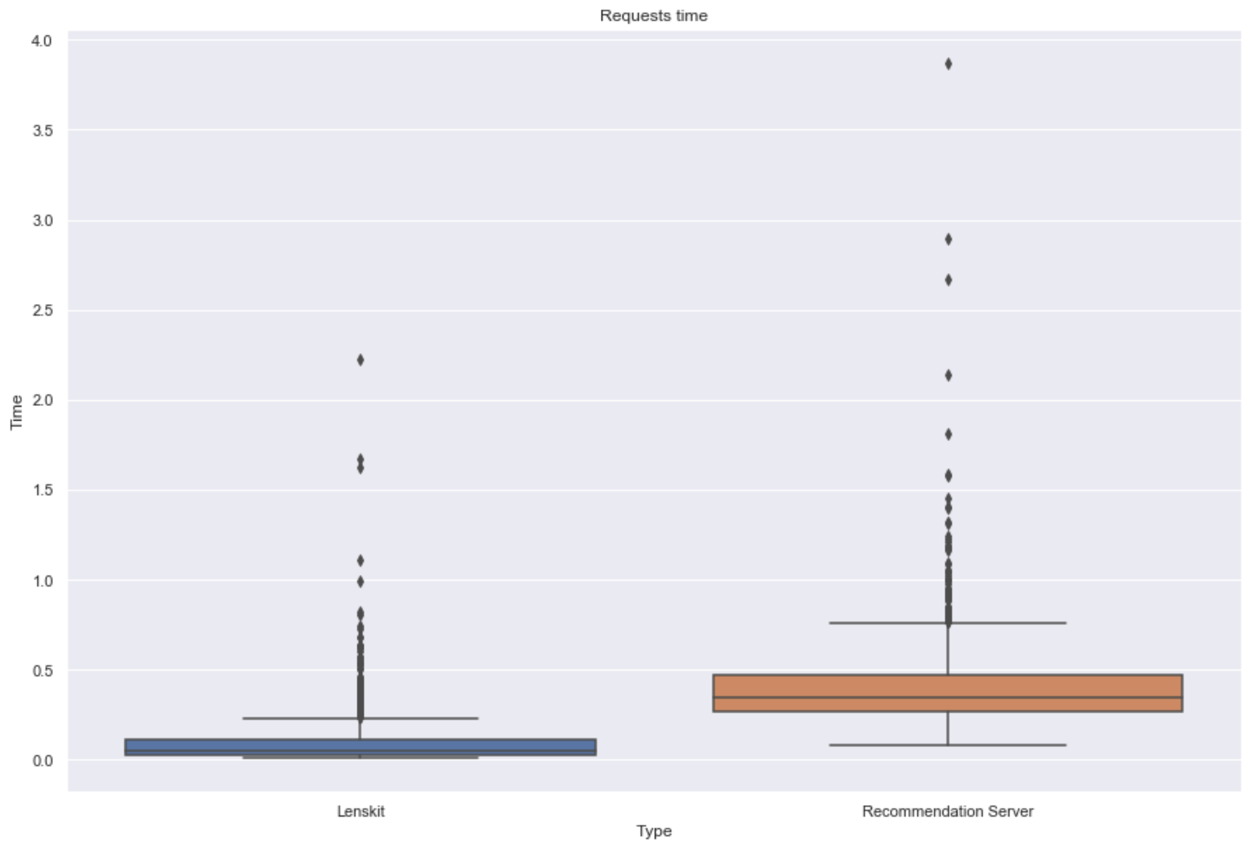


```
*******************************************************
Algo: itemitem
------------------
Lenskit performance:
Number of requests: 1000
Total response time: 101.676548
Throughput (requests per second): 9.83511
Peak response time: 2.22426
Mean response time: 0.101508
```

(continues on next page)

(continued from previous page)

```
Median response time: 0.050559
99 percentile: 0.679383
------------------
Recommendation server performance:
Number of requests: 1000
Total response time: 52.110227
Throughput (requests per second): 19.190091
Peak response time: 3.86636
Mean response time: 0.412121
Median response time: 0.342963
99 percentile: 1.32436
```

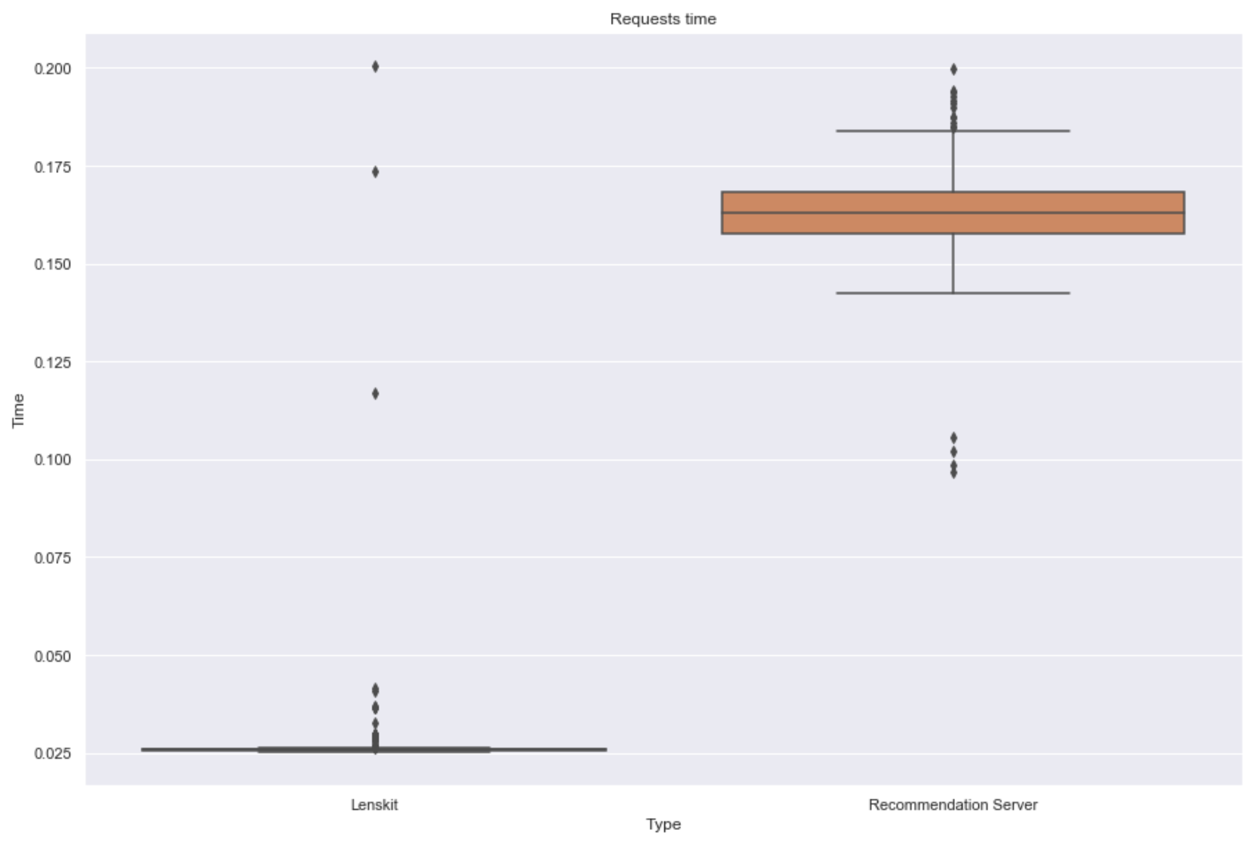


```
*******************************************************
Algo: tf_bpr
------------------
Lenskit performance:
Number of requests: 1000
Total response time: 26.68567
Throughput (requests per second): 37.473296
Peak response time: 0.200347
Mean response time: 0.02661
Median response time: 0.025739
99 percentile: 0.029865
------------------
Recommendation server performance:
Number of requests: 1000
Total response time: 20.59358
```

(continues on next page)

(continued from previous page)

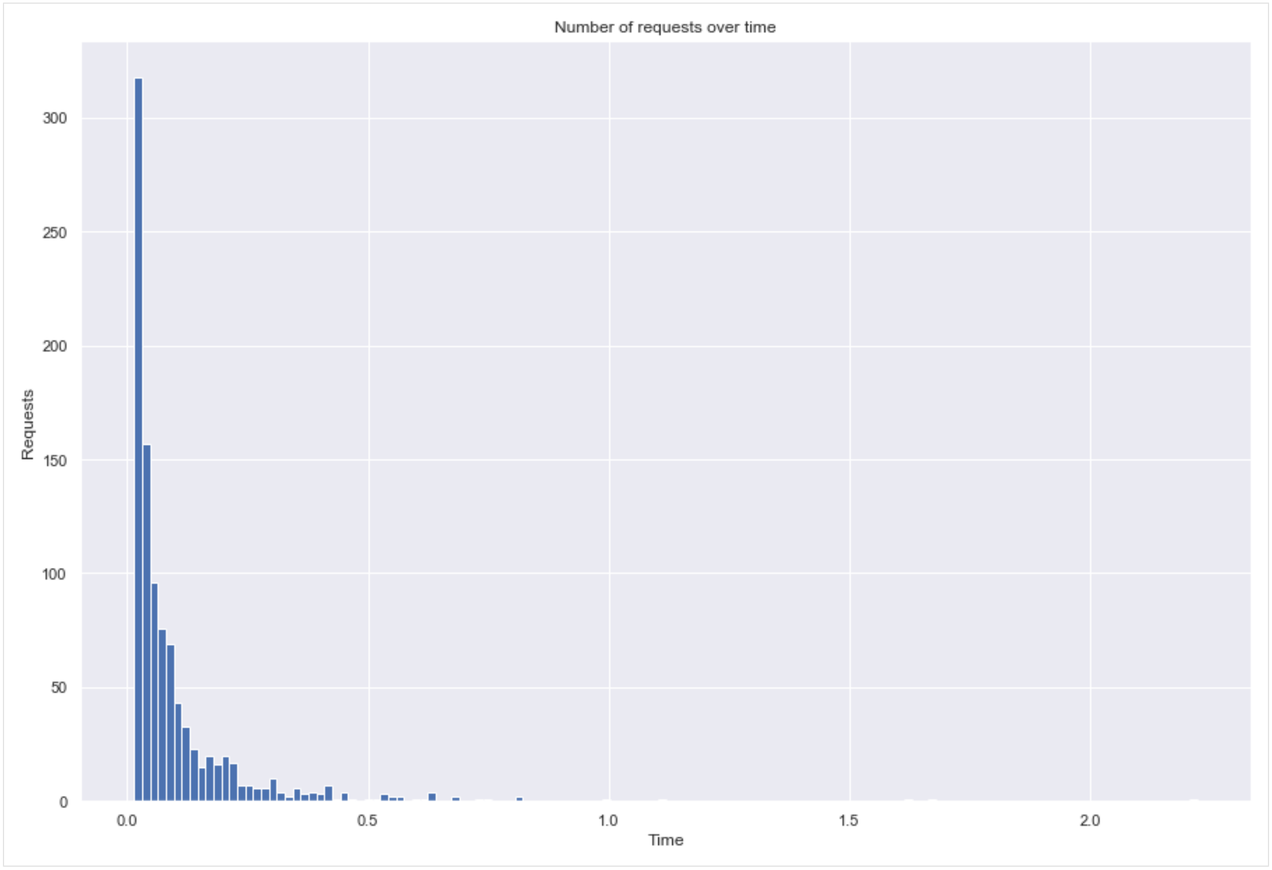
```
Throughput (requests per second): 48.558822
Peak response time: 0.199767
Mean response time: 0.163299
Median response time: 0.162994
99 percentile: 0.184859
```



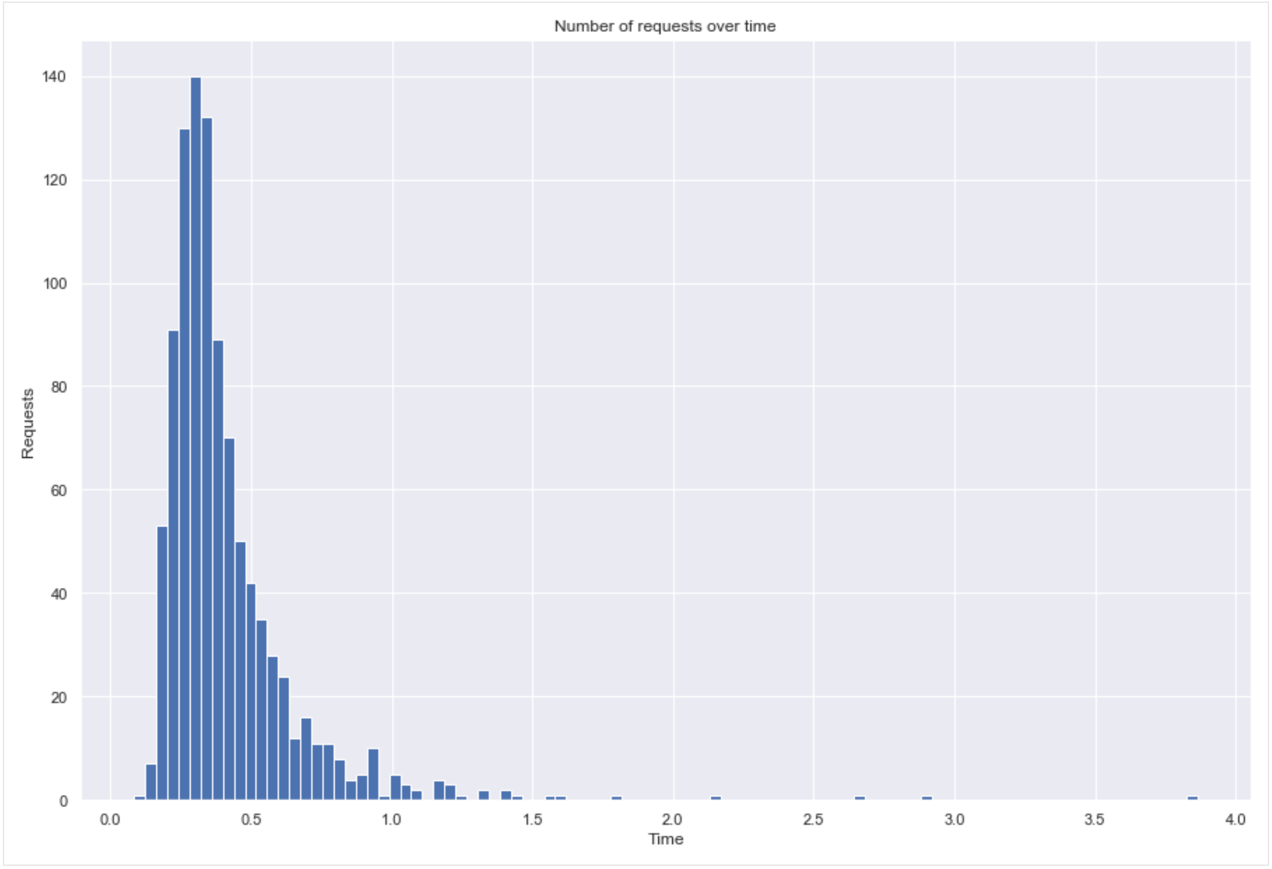
```
*******************************************************
```

Let’s see in more detail the ItemItem response time.

```
[17]: file_name = f'lkpy_itemitem_num_req_{num_requests}.pickle'
      hist_numbers(file_name)
```



```
[16]: file_name = f'preds_itemitem_against_lkpy_workers_4_num_req_{num_requests}.pickle'
      hist_numbers(file_name)
```



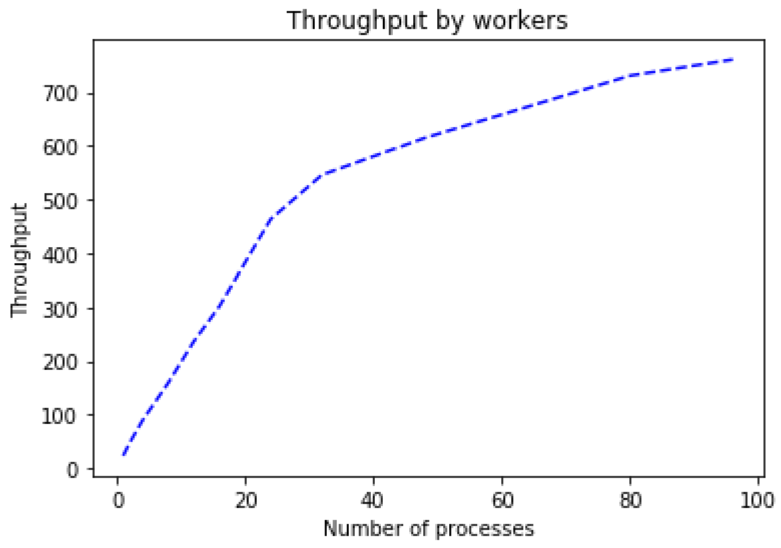
8.2.3 Speedup Tests

Throughput by number of workers

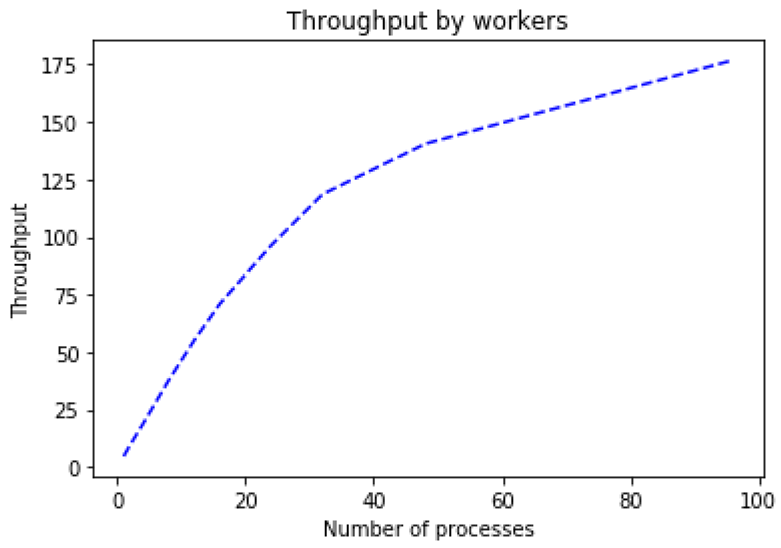
```
[8]: for algo in linear_speedup_algos:
         print(f'Algo: {algo}')
         throughput_file_name_workers = f'{results_folder}throughput_single_multiple_
     ↪workers_algo_{algo}.csv'
         throughputs_workers_from_file = np.genfromtxt(throughput_file_name_workers,␣
     ↪delimiter=',')
         y_pos = np.arange(len(throughputs_workers_from_file))
     #    plt.bar(y_pos, throughputs_workers_from_file, align='center', alpha=0.5)
         plt.plot(workers_config, throughputs_workers_from_file, 'b--')
     #    plt.xticks(y_pos, workers_config)
         plt.ylabel('Throughput')
         plt.xlabel('Number of processes')
         plt.title('Throughput by workers')

         plt.show()
         print('*******************************************************')
```

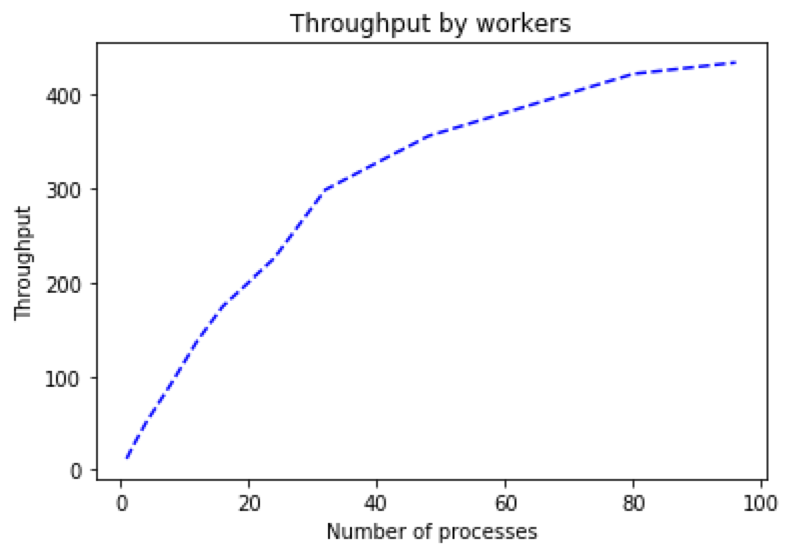
```
Algo: biasedmf
```



```
*************************************************************
Algo: itemitem
```



```
*************************************************************
Algo: tf_bpr
```



```
*******************************************************
```

Plots by number of workers

```
[7]: df = pd.read_csv(f'{results_folder}throughput_single_multiple_workers_algo_tf_bpr.csv
     ↪', header=None)
     df['Workers'] = workers_config
     df.rename(columns={0:'Throughput'}, inplace=True)
     df
```

```
[7]:     Throughput  Workers
     0    11.880536        1
     1    25.505847        2
     2    50.071827        4
     3    92.440922        8
     4   137.375141       12
     5   174.451148       16
     6   225.043659       24
     7   298.307484       32
     8   355.671852       48
     9   422.071596       80
     10  434.162142       96
```

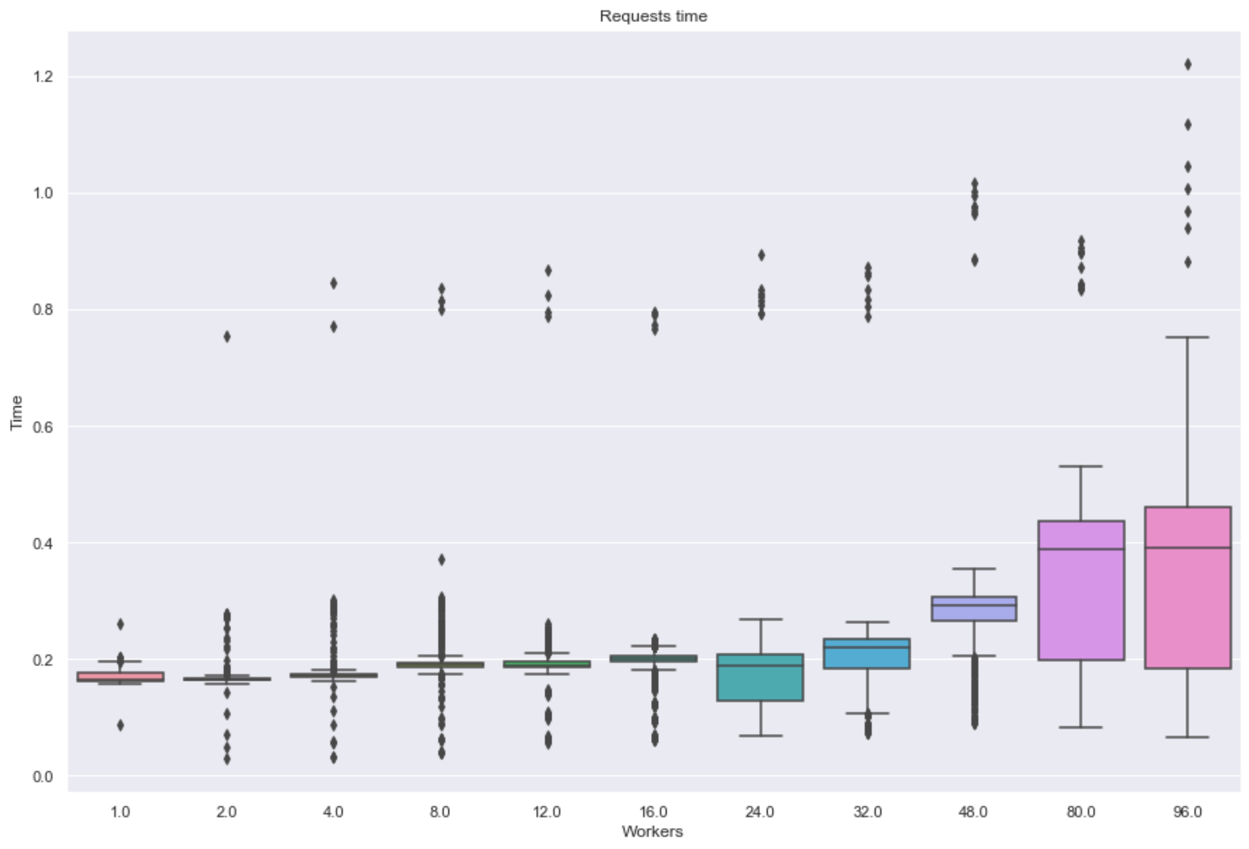
```
[8]: sns.set(rc={'figure.figsize':(15, 10)})
     for algo in linear_speedup_algos:
         df = pd.DataFrame({'Times': [], 'Workers': [], 'Algorithm': []})
         print(f'Algorithm: {algo}')
         for w in workers_config:
             file_name = f'linear_speedup_preds_{algo}_workers_{w}_num_req_1000.pickle'
             df_temp = create_df(file_name, algo, w)
             df = df.append(df_temp, ignore_index=True)
         plt.title('Requests time')
         ax = sns.boxplot(x="Workers", y="Time", data=df)
         plt.show()
         print('---------------------')
         print('')
```

(continues on next page)

(continued from previous page)

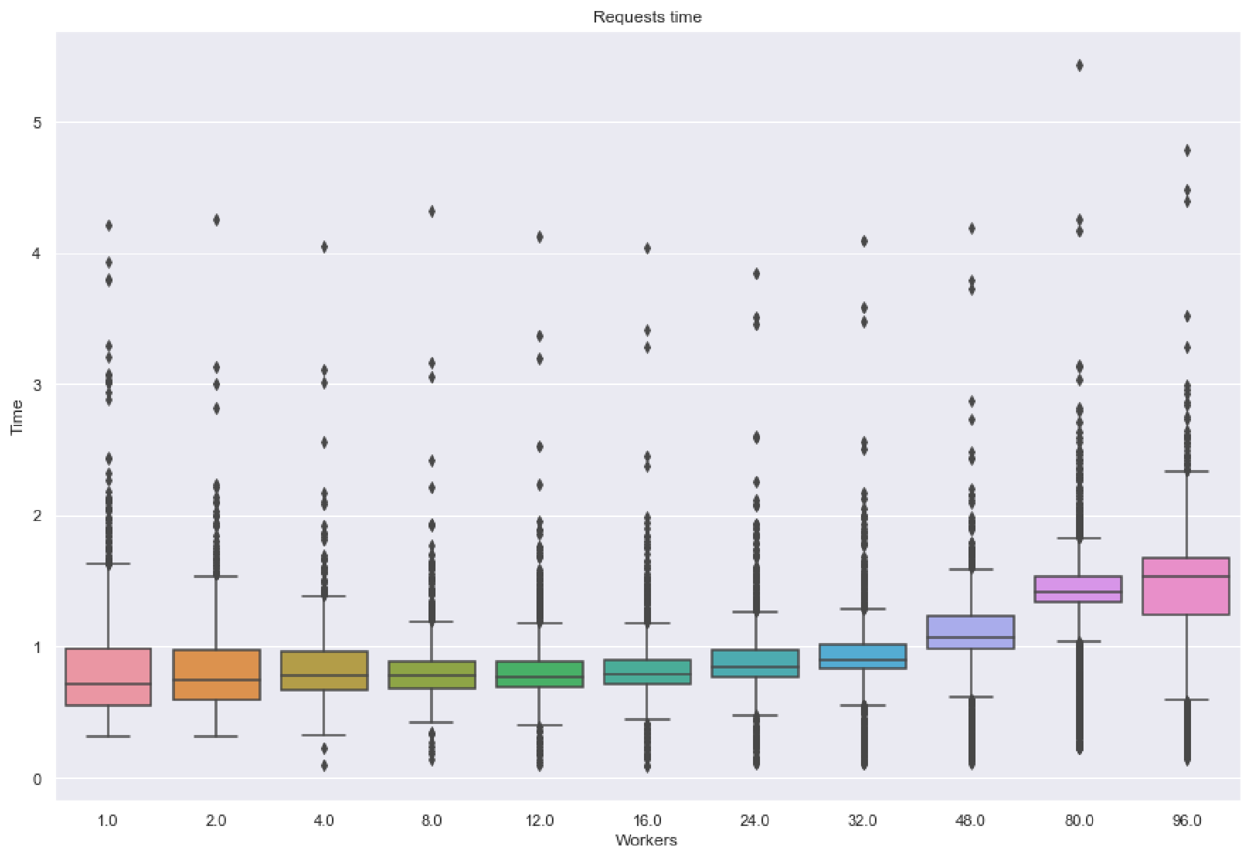
```
#     break
```

```
Algorithm: biasedmf
```



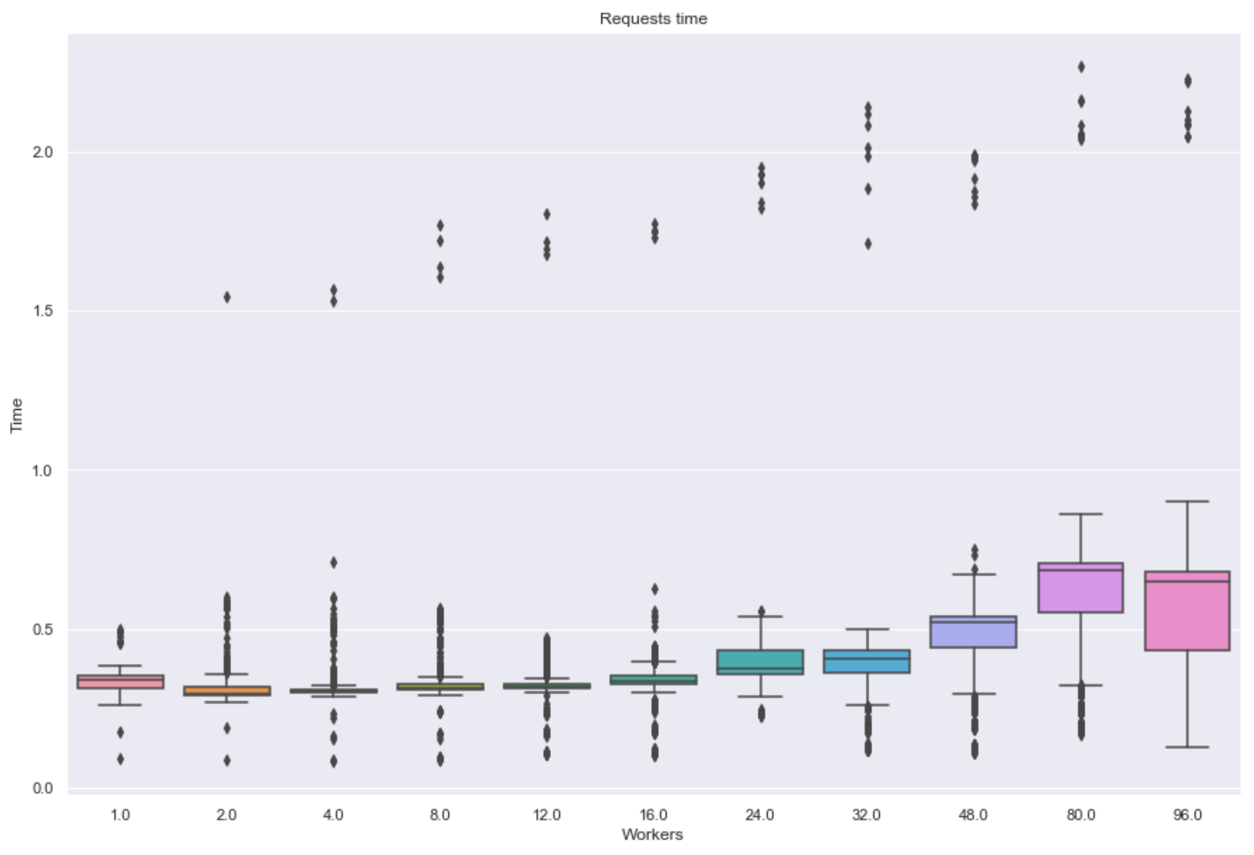
```
----------------------

Algorithm: itemitem
```



```
---------------------

Algorithm: tf_bpr
```



```
----------------------
```

[13]:
```
for w in workers_config:
    print(f"Worker: {w}, Mean Time: {df[df['Workers'] == w]['Time'].mean()}")
```

```
Worker: 1, Mean Time: 0.16933080368420633
Worker: 2, Mean Time: 0.1678498732119333
Worker: 4, Mean Time: 0.1769776633818401
Worker: 8, Mean Time: 0.19603982038088724
Worker: 12, Mean Time: 0.19488702520968218
Worker: 16, Mean Time: 0.20030749574633955
Worker: 24, Mean Time: 0.18364940203385777
Worker: 32, Mean Time: 0.21081128295401869
Worker: 48, Mean Time: 0.2691085608160647
Worker: 80, Mean Time: 0.3322938989320246
Worker: 96, Mean Time: 0.3337401035259827
```

[ ]:

HTTP ROUTING TABLE