The LKPY Package for Recommender Systems Experiments*

Michael D. Ekstrand

January 11, 2020

Abstract

Since 2010, we have built and maintained LensKit, an open-source toolkit for building, researching, and learning about recommender systems. We have successfully used the software in a wide range of recommender systems experiments, to support education in traditional classroom and online settings, and as the algorithmic backend for user-facing recommendation services in movies and books. This experience, along with community feedback, has surfaced a number of challenges with LensKit's design and environmental choices. In response to these challenges, we are developing a new set of tools that leverage the PyData stack to enable the kinds of research experiments and educational experiences that we have been able to deliver with LensKit, along with new experimental structures that the existing code makes difficult. The result is a set of research tools that should significantly increase research velocity and provide much smoother integration with other software such as Keras while maintaining the same level of reproducibility as a LensKit experiment. In this paper, we reflect on the LensKit project, particularly on our experience using it for offline evaluation experiments, and describe the next-generation LKPY tools for enabling new offline evaluations and experiments with flexible, open-ended designs and well-tested evaluation primitives.

1 Introduction

LensKit [Ekstrand et al., 2011] is an open-source toolkit that provides a variety of features in support of research, education, and deployment of recommender systems. It provides tools and infrastructure support for managing data and algorithm configurations, implementations of several collaborative filtering algorithms, and an evaluation suite for conducting offline experiments.

Based on our experience researching and teaching with LensKit, and the experience reports we hear directly and indirectly from others, we have come to believe that LensKit's current design and technology choices are not a good match for the current and future needs of the recommender systems research community. Further, in our examination of the software landscape to

*This material is based upon work supported by the National Science Foundation under Grant No. IIS 17-51278. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

determine what existing tools might support the kinds of offline evaluation experiments we have been running with LensKit and plan to run in coming years, we came to the conclusion that there is a need for high-quality, well-tested support code for recommender systems experiments in the PyData environment.

To meet that need, we have developed LKPY, a "spiritual successor" to LensKit written in Python. This project brings the LensKit's focus on reproducible research supported by well-tested code to a more widely-used and easier-to-learn computational environment.

In this paper, we reflect on some of the successes and failures of the LensKit project and present the goals and design of the LKPY software. We invite the community to provide feedback on how this software does or does not meet their needs.

2 LensKit in Use

In its 8 years of development, we and others have successfully used LensKit across all its intended application contexts.

2.1 Research

LensKit has supported a good number of research projects spanning a range of research questions and methodologies; a complete list of known published papers, theses, and dissertations using LensKit is available from the project web site[1].

We have used LensKit ourselves both for offline evaluations exploring various aspects of algorithm and user behavior [Ekstrand and Riedl, 2012, Ekstrand et al., 2018a, Kluver et al., 2012] and studying the evaluation process itself [Ekstrand and Mahant, 2017]. It has also been used to study specialized recommendation problems, including cold start [Kluver et al., 2014] and reversible machine learning [Cao and Yang, 2015]. Its algorithms have been used to recommend books [Pera and Ng, 2017], tourist destinations [Pessemier et al., 2016], videos [Solvang, 2017], and a number of other item types.

One of LensKit's primary objectives is to promote reproducible, reliable research. To that end we have been modestly successful; in our own work, it has enabled us to publish complete reproducer code for recent papers [Ekstrand and Mahant, 2017, Ekstrand et al., 2018a], and most recently to provide reproducer code during the review process [Ekstrand et al., 2018b].

2.2 Education

We have used LensKit to support recommender systems education in multiple settings. At the University of Minnesota, Boise State University, and Texas State University, we and our collaborators have used it as the basis for assignments in graduate classes on recommender systems. It also forms the basis for the assignments in the *Recommender Systems* MOOC [Konstan et al., 2015].

Response to the software in this setting has been mixed. Its standardized APIs and integration with the Java ecosystem have made things such as automated grading in the MOOC environment

[1] `http://lenskit.org/research`

relatively easy to implement, but students have complained about needing to work in Java and about the system's complexity.

2.3 Production

LensKit powers the MovieLens movie recommender system [Harper and Konstan, 2015], the BookLens book recommender [Kluver et al., 2014], and the Confer conference session recommender [Zhang et al., 2016]. This work, particularly in MovieLens, has enabled new user-centered research on recommender system capabilities and user response.

3 Reflections on LensKit

Our direct experience using LensKit in this range of applications, and what we hear from other users and prospective users, has given us a somewhat different perspective on the software than we had when we developed the early versions of LensKit. We think some of our decisions and goals hit the mark and we were successful in achieving them; some other design decisions have not held up well in the light of experience.

3.1 What We Got Right

There are several things that we think we did well in the original LensKit software; some of these we carry forward into LKPY, while others were good ideas in LensKit's contexts but are not as important today.

Testing

In the LensKit development process, we have a strong focus on code testing [Ekstrand, 2016]. This has served us well, and helped ensure the reliability of the LensKit code. It is by no means perfect, and there have been bugs that slipped through, but the effort we spent on testing was a wise investment.

The Java Platform

When we began work on LensKit, there were three possible platforms we seriously considered: Java, Python, and C++. At that time, the Python data science ecosystem was not what it is today; NumPy and SciPy existed, but Pandas did not. Pure Python does not have the performance needed for efficient recommender experiments. Java enabled us to achieve strong computational performance in a widely-taught programming language with well-established standard practices, making it significantly easier for new users (particularly students) to adapt and contribute to it than we likely would have seen with a corresponding C++ code base.

Modular Algorithms

LensKit was built around the idea of modular algorithms where individual components can be replaced and reconfigured. In the item-item collaborative filter, for example, users can change the similarity function, the neighborhood weighting function, input rating vector normalizations, item neighborhood normalizations, and the strategy for mapping input data to rating vectors. This configurability has been useful for exploring a range of configuration options for algorithms, at the expense of larger hyperparameter search spaces.

3.2 What Doesn't Work

Other aspects of LensKit's design and development do not seem to have met our needs or those of the community as well.

Opinionated Evaluation

While LensKit's algorithms were highly configurable, its offline evaluation tools are much more opinionated. You can specify a few types of data splitting strategies, and recommendation candidate strategies, and it has a range of evaluation metrics, but the overall evaluation process and methods for aggregating metric results are fixed. Metrics are also limited in their interface.

As we expanded our research into the recommender evaluation process itself, we repeatedly ran in to limits of this evaluation strategy and had to write new evaluation code in a fairly heavy framework, or just have the evaluator dump intermediate files that we would reprocess in R or Python, in order to carry out our research. Too often the answer we would have to give to questions on the mailing list or StackOverflow is "we're sorry, LensKit can't do that yet".

One of our goals was to make it difficult to do an evaluation wrong: we wanted the defaults to embody best practices for offline evaluation. However, best practices have been sufficiently unknown and fast-moving that we now believe this approach has held our research back more than it has helped the field. It is particularly apparent that a different approach is necessary to support next-generation offline evaluation strategies, such as counterfactual evaluation [Bottou et al., 2013], and carry out the evaluation research needed to advance our understanding of effective, robust, and externally valid offline evaluations.

Indirect Configuration

LensKit is built on the dependency injection principle, using a dependency injection container [Ekstrand and Ludwig, 2016] to instantiate and connect recommender components. This method gave us quite a few useful capabilities, such as the automatically detecting components that could be shared between multiple experiment runs in a parameter tuning experiment, but resulted in a system where it is difficult to configure an algorithm, and difficult to understand an algorithm's configuration. It was also difficult to document how to use the system.

We believe this largely stems from the role of inversion of control in working with LensKit code — users never write code that assembles a LensKit algorithm, they simply ask LensKit to instantiate one and LensKit calls their custom components.

Implicit Features

Beyond indirect configuration, LensKit has a lot of implicit behavior in its algorithms and evaluator. This has at least two downsides: first, it is less clear from reading a configuration precisely what LensKit will do, making it more difficult to review code and experiment scripts; second, if documentation slipped behind the code, understanding the behavior of LensKit experiment scripts required reading the LensKit source code itself.

Living in an Island

LensKit has its own data structures and data access paradigms. Part of this is due to lack of standardized, high-quality scientific data tooling that is not connected to a larger framework such as Spark (while Spark does seem to expose data structures as a separate library, documentation is nonexistent).

This makes it difficult, however, to make LensKit interoperate with other software and data sets. While LensKit is flexible in the data it accepts, users must write data adapters. Using other tools such as Spark is difficult. It's unclear what, given the Java commitment, we could have done differently here, but it is definitely a liability for the future of recommender systems research.

4 Toolkit Desiderata

To address these weaknesses and power the next generation of recommender systems research, both in our own research group and elsewhere, there are several things that we desire from our recommender systems research software:

Build on Standard Tools There are now standard tools, such as Pandas and the surrounding PyData ecosystem [McKinney, 2018], that are widely adopted for data science and machine learning research. Building on these packages maximizes interoperability with other software packages and enables code reuse across different types of research. We also find Jupyter notebooks to be a valuable means of promoting reproducibility, and would like an experiment workflow where the final analysis consists of loading the recommender results into a Jupyter notebook and computing desired metrics over them. Small experiments could even be driven entirely from Jupyter.

Leverage Existing Software Modern recommender systems are usually machine learning systems; a recommender research toolkit should not try to reinvent that wheel. Scikit-Learn provides many machine learning algorithms suitable for recommender systems research, and Keras, PyTorch, and TensorFlow all provide neural network functionality to Python. Recommender research tooling should work seamlessly with algorithms implemented in these kinds of toolkits.

Expose the Data Pipeline LensKit hides the data pipeline: it controls data splitting, algorithm training, recommendation, and evaluation. Outputs of each stage can be examined, but not manipulated, and the pipeline itself cannot be easily changed. Putting the user in control of the pipeline, and providing functions to implement standard versions of each of its stages,

will have two benefits: the actual pipeline used is clearly documented in the experiment code, and the pipeline can be modified as research needs demand.

Explicit is Better than Implicit Related to exposing the data pipeline, we wish for our new tools to follow the Python philosophy of favoring explicit denotations of desired operation. This will make it easier to review experiment designs and improve the reliability and rigor of reproducible research.

We hope that this will also make the LKPY code itself easier to read and understand.

Simple Interfaces Interfaces to individual software components should be as simple as possible, so that it is easy to document, test, and reimplement them.

Easy-to-Use Development Environment In order for prospective users and contributors, particularly students, to use LKPY and participate in its development, we want the development tools to be as standard and easy-to-use as possible.

Notably absent from this list is LensKit's (in)famous algorithm configurability. That configurability was useful for exploring the space of algorithm configurations, but its particular design is more suitable to heuristic techniques such as k-NN; machine learning approaches seem better served by a different design. Connecting with existing flexible optimization software will provide a great deal of configurability for new algorithms. We think it is more important for the recommender-specific software to focus on flexibility in the experiment design.

5 The LKPY Software Package

To that end, we have developed a successor to LensKit, LKPY. LKPY is a new Python package for recommender systems experiments, particularly offline evaluations and similar studies, that we hope will also be useful in educational settings. LKPY is capable of running many of the kinds of experiments we have run with LensKit in a more flexible and forward-looking fashion. It is published under the MIT license.

5.1 LKPY Facilities

LKPY provides several modules for aiding recommender experiments:

Data Preparation The `crossfold` module provides functions to split data for cross-validation. It supports rating-based, user-based, and item-based splitting strategies, with configurable data holdout settings.

Algorithm APIs The `algorithms` module defines Python interfaces for training models, generating predictions, and generating recommendations. These interfaces are minimal, defined in terms of Pandas data structures, and can be readily implemented on top of any desired machine learning toolkit such as Scikit-Learn or Pandas. The algorithm API is based on the patterns used by Scikit-Learn [Buitinck et al., 2013], where a `fit` method estimates model parameters (trains the model) in-place. This makes the API familiar to users with experience using other packages in the SciKit ecosystem.

Evaluation Metrics The `metrics` package provides classical top-N and prediction accuracy metrics. Metrics are functions that operate directly over Pandas series objects, and thus can be applied to any data of an appropriate shape. There is therefore no limit to how the user reprocesses recommendations and predictions before computing metrics. The `topn` package also provides a `RecListAnalysis` class to facilitate evaluation of top-N recommendation lists, merging test data with recommendations and accounting for unrecommended items.

Classical CF Algorithms LKPY provides implementations of nonpersonalized, k-NN, and several matrix factorization collaborative filters. We expect to expand the set of algorithms provided, but being a source of algorithms is not LKPY's primary objective. These algorithms are provided in part to give LensKit users a migration path to LKPY that keeps the algorithms as consistent as possible; LKPY's algorithm implementations are based on LensKit's and should generally be the same. They are less configurable, however, selecting configuration options such as item-item similarity functions that we have found to work well across a range of data sets. We use Numba to accelerate inner algorithm computations when NumPy, SciPy, or Pandas operations are inadequate.

API Bridges LKPY provides implementations of its algorithm APIs using Implicit[2] (Implicit ALS and BPR with GPU acceleration) and HPFREC[3] (providing Hierarchical Poisson Factorization [Gopalan et al., 2013]).

Batch Utilities To ease writing evaluation scripts, LKPY provides utility functions for computing predictions or recommendations for many users in batch, and for evaluating multiple algorithms on multiple data sets. These batch routines are capable of running directly, but can also save experiment descriptions to disk to enable large experiments to be parallelized through external systems such as SLURM's job arrays.

Since components connect with standard Pandas data structures, they can be used together or individually. It is trivial to use alternative algorithms with LKPY's data splitting and metrics, or to use an entirely different data preparation strategy with LensKit's algorithms and batch functions.

One way in which our desire to favor explicit code over implicit behavior manifests is in data processing: instead of telling LensKit how to transform data, in LKPY, the user just directly writes their data transformations using standard Pandas operations.

5.2 LKPY Example Code

Figure 1 shows an example of using LKPY to compute the nDCG of a k-NN algorithm with 5-fold cross-validation on the MovieLens 100K data set. It is somewhat verbose, but ever step of the evaluation process is clear in the code, so the experiment structure can be checked and it is self-documenting when the evaluation code is published.

[2] `https://github.com/benfred/implicit`

[3] `https://github.com/david-cortes/hpfrec`

```
import pandas as pd
from lenskit import batch, topn, datasets, util
from lenskit import crossfold as xf
from lenskit.algorithms import knn, Recommender

ml = datasets.ML100K('ml-100k')
ratings = ml.ratings

algo = knn.ItemItem(30)

def eval(train, test):
    # ensure the recommender is a top-N recommender
    rec = Recommender.adapt(algo)
    # cloning the model ensures forward compatibility
    # with things like parallelism
    rec = util.clone(rec)
    rec.fit(train)
    users = test.user.unique()
    recs = batch.recommend(algo, model, users, 100)
    return recs

# compute evaluation
splits = list(xf.partition_users(ratings, 5, xf.SampleFrac(0.2))
recs = pd.concat((eval(p.train, p.test)
                  for p in splits),
                 ignore_index=True)

# integrate test data; since splits are disjoint users
# for a single data set, this is sufficient
all_test = pd.concat(p.test for p in splits)
# compute metric results
rla = topn.RecListAnalysis()
rla.add_metric(topn.ndcg)
metrics = rla.compute(recs, all_test)
```

Figure 1: Example Simple Evaluation

5.3 Dependencies and Environment

LKPY leverages Pandas [McKinney and Others, 2010], numpy [Oliphant, 2006], scipy [Oliphant, 2007], and PyTables/HDF5, along with several other Python modules. We use Numba [Lam et al., 2015] for native-code acceleration, as it enables efficient, parallelized NumPy-based code without complex build toolchains; JobLib provides parallelism for higher-level batch operations and other settings where Numba parallelism is not effective.

We regularly test LKPY with recent Python versions on Windows, Linux, and macOS. Our ongoing criteria are to support the three major operating systems, and the version of Python 3 available in the most most recent release of major Linux distributions (Ubuntu LTS, RHEL/CentOS with EPEL, and Debian). We focus primarily on Anaconda-based Python installations, but also test the code with vanilla Python on all supported platforms. We provide binaries via Anaconda Cloud for supported Python versions and operating systems[4], along with source distributions on the Python Package Index[5].

5.4 Future Directions

LKPY provides a clean, minimal core that can be extended however our needs and those of the research community require. We are welcome to contributions of additional capabilities, algorithms, and shims to external packages, either as pull requests to the main LKPY project or as add-on packages distributed separately.

6 Comparison to Existing Packages

Before embarking on this project, we re-examined the landscape of existing software to determine if the needs we saw would be met by one of the other software packages. We were unable to find existing tooling that supports our goals of flexible recommender systems experiments that leverage the PyData ecosystem.

Two of the most obvious contenders for current Python recommender systems are surprise [Hug, 2017] and PyRecLab [Sepulveda and Parra, 2017]. While Surprise uses numpy and scipy, neither of these packages leverage the PyData ecosystem; instead, each has its own classes for representing data sets. PyRecLab's evaluation facilities are also more automatic; we want to write out the evaluation steps in our own code so that we can experiment with them more readily. Further, we do not want to require students to learn C++ to be able to participate in research that requires extending LKPY.

Our approach is perhaps most similar to that of mrec[6], in that we focus on discrete steps enabled by separate tools. Again, though, mrec does not leverage contemporary Python data science tools, and does not seem to be under active maintenance. We also focus on Python as the scripting language for experiment control instead of mrec's command-line orientation.

It is our sense that many Python-based recommender systems research projects roll their own evaluation procedure directly in Python tools while building the recommender in scikit-learn or one of the deep learning frameworks. It is our goal to integrate with such workflows, enabling them to leverage common, well-tested implementations of metrics and other experimental sup-

[4] `https://anaconda.org/lenskit/lenskit`

[5] `https://pypi.org/project/lenskit/`

[6] `https://github.com/Mendeley/mrec`

port code while continuing to use their existing data flows for the recommendation process.

7 Conclusion

We have learned many lessons developing, maintaining, and researching with the LensKit software. Based on these lessons, we came to the conclusion that, in its current form, it is not meeting our needs or the needs of the recommender systems community well, and that resources would be better spent on tools that improve research being done with the widely-used Python packages that drive much of modern data science.

Where LensKit focused on providing building blocks for recommender *algorithms*, LKPY provides building blocks for recommender *experiments*. Built on the PyData stack and oragnized around clear, explicit data processing pipelines with a minimum of custom concepts, LKPY provides a solid foundation for new experimentation and concepts at all stages of the offline recommender system evaluation lifecycle. We expect it to meet our own research needs in offline evaluation and simulation of recommender systems much better than the current LensKit code going forward, and hope that others in the research community find it useful as well. We welcome contributions on GitHub, and invite feedback from the community to guide future development.

Bibliography

L. Bottou, J. Peters, J. Quionero-Candela, D. X. Charles, D. M. Chickering, E. Portugaly, D. Ray, P. Simard, and E. Snelson. Counterfactual Reasoning and Learning Systems: The Example of Computational Advertising. *Journal of Machine Learning Research*, 14(1):3207–3260, 2013. ISSN 1532-4435. URL `http://www.jmlr.org/papers/volume14/bottou13a/bottou13a.pdf`.

L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. Vanderplas, A. Joly, B. Holt, and G. Varoquaux. API Design for Machine Learning Software: Experiences from the scikit-learn Project. Sept. 2013. URL `http://arxiv.org/abs/1309.0238`.

Y. Cao and J. Yang. Towards Making Systems Forget with Machine Unlearning. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*. IEEE, May 2015. URL `http://www.ieee-security.org/TC/SP2015/papers-archived/6949a463.pdf`.

M. Ekstrand. Testing Recommenders, Feb. 2016. URL `https://buildingrecommenders.wordpress.com/2016/02/04/testing-recommenders/`.

M. Ekstrand and J. Riedl. When Recommenders Fail: Predicting Recommender Failure for Algorithm Selection and Combination. In *Proceedings of the Sixth ACM Conference on Recommender Systems*, RecSys '12, pages 233–236, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1270-7. doi: 10.1145/2365952.2366002. URL `http://doi.acm.org/10.1145/2365952.2366002`.

M. Ekstrand, M. Ludwig, J. A. Konstan, and J. Riedl. Rethinking the Recommender Research Ecosystem: Reproducibility, Openness, and LensKit. In *Proceedings of the 5th ACM Conference on Recommender Systems*, pages 133–140. ACM, 2011. ISBN 978-1-4503-0683-6. doi: 10.1145/2043932.2043958. URL `http://doi.acm.org/10.1145/2043932.2043958`.

M. D. Ekstrand and M. Ludwig. Dependency Injection with Static Analysis and Context-Aware Policy. *Journal of Object Technology*, 15(1):1:1, Feb. 2016. ISSN 1660-1769. doi: 10.5381/jot.2016.15.5.a1. URL http://www.jot.fm/contents/issue_2016_01/article1.html.

M. D. Ekstrand and V. Mahant. Sturgeon and the Cool Kids: Problems with Top-N Recommender Evaluation. In *Proceedings of the 30th Florida Artificial Intelligence Research Society Conference*. AAAI Press, May 2017. URL https://aaai.org/ocs/index.php/FLAIRS/FLAIRS17/paper/viewPaper/15534.

M. D. Ekstrand, M. Tian, I. M. Azpiazu, J. D. Ekstrand, O. Anuyah, D. McNeill, Pera, and M. Soledad. All The Cool Kids, How Do They Fit In?: Popularity and Demographic Biases in Recommender Evaluation and Effectiveness. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*, volume 81 of *PMLR*, pages 172–186, Feb. 2018a. URL http://proceedings.mlr.press/v81/ekstrand18b.html.

M. D. Ekstrand, M. Tian, M. R. Imran Kazi, H. Mehrpouyan, and D. Kluver. Exploring Author Gender in Book Rating and Recommendation. In *Proceedings of the Twelfth ACM Conference on Recommender Systems*. ACM, 2018b.

P. Gopalan, J. M. Hofman, and D. M. Blei. Scalable Recommendation with Poisson Factorization. *arXiv:1311.1704 [cs, stat]*, Nov. 2013. URL http://arxiv.org/abs/1311.1704.

F. M. Harper and J. A. Konstan. The MovieLens Datasets: History and Context. *ACM Transactions on Interactive Intelligent Systems*, 5(4):19:1–19:19, Dec. 2015. ISSN 2160-6455. doi: 10.1145/2827872. URL http://doi.acm.org/10.1145/2827872.

N. Hug. *Surprise, a Python Library for Recommender Systems*. 2017. URL http://surpriselib.com.

D. Kluver, T. T. Nguyen, M. Ekstrand, S. Sen, and J. Riedl. How Many Bits per Rating? In *Proceedings of the sixth ACM conference on Recommender systems*, RecSys '12, pages 99–106, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1270-7. doi: 10.1145/2365952.2365974. URL http://doi.acm.org/10.1145/2365952.2365974.

D. Kluver, M. Ludwig, R. T. Davies, J. A. Konstan, and J. T. Riedl. *BookLens*. GroupLens Research, University of Minnesota, 2014. URL https://booklens.umn.edu/.

J. A. Konstan, J. D. Walker, D. C. Brooks, K. Brown, and M. D. Ekstrand. Teaching Recommender Systems at Large Scale: Evaluation and Lessons Learned from a Hybrid MOOC. *ACM Transactions on Computer-Human Interaction*, 22(2):10:1–10:23, Apr. 2015. ISSN 1073-0516. doi: 10.1145/2728171. URL http://doi.acm.org/10.1145/2728171.

S. K. Lam, A. Pitrou, and S. Seibert. Numba: A LLVM-based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-4005-2. doi: 10.1145/2833157.2833162. URL http://doi.acm.org/10.1145/2833157.2833162.

W. McKinney. *Python for Data Analysis: Data Wrangling with pandas, NumPy, and IPython*. O'Reilly, 2018. ISBN 978-1-4919-5766-0. URL http://shop.oreilly.com/product/0636920023784.do.

W. McKinney and Others. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010. URL http://conference.scipy.org/proceedings/scipy2010/pdfs/mckinney.pdf.

T. E. Oliphant. *A Guide to NumPy*. Trelgol Publishing, 2006.

T. E. Oliphant. Python for Scientific Computing. *Computing in Science Engineering*, 9(3):10–20, May 2007. ISSN 1521-9615. doi: 10.1109/MCSE.2007.58. URL http://dx.doi.org/10.1109/MCSE.2007.58.

M. S. Pera and Y.-K. Ng. Recommending books to be exchanged online in the absence of wish lists. *Journal of the Association for Information Science and Technology*, Nov. 2017. ISSN 2330-1643. doi: 10.1002/asi.23978. URL http://dx.doi.org/10.1002/asi.23978.

T. D. Pessemier, J. Dhondt, and L. Martens. Hybrid Group Recommendations for a Travel Service. *Multimedia Tools and Applications*, 75(5):1–25, Jan. 2016. ISSN 1380-7501, 1573-7721. doi: 10.1007/s11042-016-3265-x. URL http://link.springer.com/article/10.1007/s11042-016-3265-x.

G. Sepulveda and D. Parra. pyRecLab: A Software Library for Quick Prototyping of Recommender Systems. *arXiv preprint arXiv:1706.06291*, 2017. URL https://arxiv.org/abs/1706.06291.

M. L. Solvang. Video Recommendation Systems: Finding a Suitable Recommendation Approach for an Application Without Sufficient Data. Master's thesis, 2017. URL http://hdl.handle.net/10852/59239.

A. X. Zhang, A. Bhardwaj, and D. Karger. Confer: A Conference Recommendation and Meetup Tool. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion*, CSCW '16 Companion, pages 118–121, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3950-6. doi: 10.1145/2818052.2874340. URL http://doi.acm.org/10.1145/2818052.2874340.