Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Auto-CaseRec: A Novel Automated Recommender System Framework

Srijan Gupta

[redacted]

Supervisor: Dr. Joeran Beel

May 13, 2020

A Final Year Project submitted in partial fulfilment

of the requirements for the degree of

MAI (Computer Engineering)

Declaration

Abstract

Recommender Systems (RSs) are software tools and techniques that are used to produce *recommendations* for the users of a certain application in such a way that the recommendations generated are likely to be liked by the users. Popular examples of applications that use RSs include Amazon, Netflix, Spotify, and Youtube. To support the wide-spread use of RSs, a variety of open-source tool-kits have been developed. While research studies show that different algorithms work well for different recommendation scenarios and with varying data-set characteristics, it has also been pointed out that the same recommendation algorithms implemented from different tool-kits can produce significantly different results. Recommendation algorithms typically have *hyper-parameters* that can be used to change their behaviour. Tuning the hyper-parameters according to the recommendation scenario can effectively improve the performance of any such algorithm. Even so, RS tool-kits generally lack hyper-parameter optimization (HPO) methods.

On the other hand, the AutoML community has proposed many solutions to solve the problem of *Combined Algorithm Selection and Hyper-parameter optimization*. AutoML tools like Auto-sklearn [1] often use advanced HPO methods like Bayesian optimization to find the "best" algorithm and hyper-parameters for a machine learning problem. Inspired by the AutoML community, Auto-CaseRec, a novel Automated Recommender System framework is presented. Given a data set, Auto-CaseRec allows the usage of advanced HPO techniques to produce the "best" combination of algorithm and hyper-parameters. Experimentation with 5 tests each across 2 data sets in 2 recommendation scenarios: Item Recommendation and Rating Prediction show that Auto-CaseRec always outperforms the individual best recommendation algorithm with unmodified hyper-parameters. We hope that Auto-CaseRec will become a standard tool in the RS community.

Acknowledgements

I would like to thank my supervisor, Dr. Joeran Beel for his constant guidance and support throughout the execution of this research project. Dr. Joeran's expertise and feedback was crucial for the proper execution of this research and without it, this thesis would not have been possible.

I would also like to thank Dr. Mike Brady, the MAI Computer Engineering coordinator, my tutor, Dr. Nimah Harty and School of Engineering, Trinity College Dublin for being remarkably helpful and compassionate when I faced challenges during the course of this research.

I would like to thank my family and friends for helping me throughout this journey by providing emotional as well as physical support during tough times.

Finally, I would like to express my gratitude to all the unseen factors and influences in the world that led me towards conducting and completing this research work.

Contents

List of Figures

List of Tables

Abbreviations

RS	Recommender Systems
MF	Matrix Factorization
CF	Collaborative Filtering
CBF	Content-Based Filtering
RP	Rating Prediction
IR	Item Recommendation
HPO	Hyper-Parameter Optimization
SMBO	Sequential Model-Based Optimization
SMAC	Sequential Model-based Algorithm Configuration
GP	Gaussian Process
TPE	Tree Parzen Estimators

1 Introduction

1.1 Background

Recommender Systems (RSs) are software tools and techniques that suggest *items* to users that are likely to be of interest to them [5]. "Item" is the general term used to denote what an RS recommends to users. Recommender Systems play an important role in many applications like popular e-commerce websites such as Amazon and eBay, social media networks such as Facebook and LinkedIn, video streaming platforms such as Netflix and Youtube, music applications such as Spotify, Last.fm and Apple Music and even tourism businesses like TripAdvisor. In online environments like these, presenting the user with a list of recommendations that agrees with his/her preferences greatly helps in reducing the effort associated with choosing amongst the myriad of products and items that are offered and enhances user experience.

The algorithms used by RSs typically use feedback about user preferences (explicit or implicit) regarding items to generate recommendations. RS algorithms can be divided into non-personalized and personalized approaches, and the latter can be broadly divided into Content-Based Filtering (CBF) and Collaborative Filtering (CF) algorithms [6]. There exist several software tool-kits that provide functionality for implementing and evaluating RS algorithms on arbitrary data sets. Common implementations of a wide variety of RS algorithms (especially CF-based) inherently involve modifiable variables called **hyper-parameters** that can be used to change or *tune* their behaviour.

Different recommendation algorithms have been found to perform differently in different recommendation scenarios, with different user characteristics and even with different data set characteristics [7, 8, 9, 10, 11]. Even in the same recommendation scenario, different algorithms tend to perform differently in different applications (Example, news websites [12]). Additionally, the same algorithms implemented from different RS tool-kits have also been found to perform differently on the same data set [13]. Therefore, *algorithm selection* is a major challenge in the RS community. Additionally, studies have shown that tuning hyper-parameter values of CF algorithms using Hyper-Parameter Optimization (HPO) approaches can increase their performance on RS data sets [14, 15, 16, 17, 18]. Therefore, it can be said that the quality of the two tasks: **algorithm selection** and **hyper-parameter tuning** becomes a major factor in the performance of an RS algorithm for a given recommendation environment.

We combine the two tasks into one problem formulation: *Combined Algorithm Selection and Hyper-parameter optimization* (CASH) that involves automatically and simultaneously choosing a recommendation algorithm and setting its hyper-parameters to increase empirical performance. Despite the practical importance of solving the CASH problem, we are surprised to find just one open-source tool in the research literature, Lib-rec Auto [19, 20], that tackles it. Lib-rec Auto uses an exhaustive search method to evaluate all algorithms and hyper-parameters in a user given search space, but this can be inefficient when dealing with large hierarchical search spaces [4].

The CASH problem was first formulated in the research paper of one of the first advanced automated machine learning (AutoML) systems, Auto-WEKA[21, 22]. The AutoML community has been successful in developing many open-source tools to tackle the CASH problem [1, 21, 22, 23, 24, 25, 26, 27, 28]. Most of these AutoML tools augment existing machine-learning tools like scikit-learn [29] and WEKA [30] with advanced HPO techniques like Bayesian Optimization[31, 32], Hierarchical Task Networks [27] and evolutionary algorithms [24, 28] to tackle the CASH problem.

We focus our research on solving the CASH problem in the RS domain, with RS algorithms and their associated hyper-parameters. In the following sections of this introductory chapter, we formally define the CASH problem for the RS environment, state our research objectives, and present an overview of the structure of this report.

1.2 The Combined Algorithm Selection and Hyper-parameter Optimization (CASH) Problem

The CASH problem was formally defined by the authors of the AutoML framework Auto-WEKA in their introductory research paper [21] as the problem of automatically and simultaneously choosing a machine learning algorithm and its associated hyper-parameter values to increase empirical performance. The term has caught on and has been used in other research works also [1, 22, 25, 33], while another equivalent name is *Full Model Selection* (FMS). For our research purposes, we will use the CASH terminology.

We present a formulation of the CASH problem for the Recommender Systems environment, adapting it from the Auto-WEKA research paper [21]. We start with a Recommender System dataset $D = \{(user_1, item_1, feedback_1), \cdots, (user_n, item_n, feedback_n)\}$ that is split into K cross-validation folds of the form $fold^{(i)} = \{D^{(i)}_{train}, D^{(i)}_{test}\}$ where $K \geq 1$. Let $\mathcal{A} = \{A^{(1)}, \cdots, A^{(m)}\}$ be the set of recommendation algorithms and let the hyper-parameters of each algorithm $A^{(j)}$ be $\lambda \in \mathbf{\Lambda^{(j)}}$. The CASH problem refers to finding an optimal algorithm configuration $A^{(j)}_{\lambda}$, i.e., algorithm $A^{(j)}$ instantiated with hyper-parameters $\lambda \in \mathbf{\Lambda^{(j)}}$, that maximizes the cross-validation performance or equivalently, minimizes the cross-validation loss. Thus, the CASH problem can be written as:

$$\boldsymbol{A}^{*}_{\boldsymbol{\lambda}^{*}} \in \underset{A^{(j)} \in \mathcal{A}, \lambda \in \mathbf{\Lambda^{(j)}}}{\operatorname{argmin}} \frac{1}{K} \sum_{i=1}^{K} \mathcal{L}(A^{(j)}_{\lambda}, D^{(i)}_{train}, D^{(i)}_{test}),$$

where, The $\mathcal{L}(A^{(j)}_{\lambda}, D^{(i)}_{train}, D^{(i)}_{test})$ is the predictive error experienced on $D^{(i)}_{test}$ when algorithm $A^{(j)}$ instantiated with hyper-parameters λ is trained on $D^{(i)}_{train}$ .The results $\boldsymbol{A^*}$ and $\boldsymbol{\lambda^*}$ are the optimal algorithm and associated values of hyper-parameters that produce the minimum mean loss across the K folds of the data. We note that this problem can be re-formulated as a single combined hierarchical hyper-parameter optimization problem with a search space $\mathbf{\Lambda} = \mathbf{\Lambda^{(1)}} \cup \cdots \mathbf{\Lambda^{(m)}} \cup \{\lambda_r\}$, where $\{\lambda_r\}$ is a new root-level hyper-parameter that represents the choice between recommendation algorithms in $\mathcal{A}$. The hyper-parameters in each sub-space $\mathbf{\Lambda^{(j)}}$ are made conditional on $\{\lambda_r\}$ being instantiated to $A^{(j)}$.

In the following section, we outline the specific research objective undertaken in this research work to develop a solution for the CASH problem in a Recommender Systems setting.

1.3 Research Objectives

The research objective for this work was to develop a novel open-source automated Recommender System (Auto-RS) tool, i.e., a tool that can solve the CASH problem stated in Section 1.2 for any arbitrary Recommender System data set in a fully automated manner. To fulfill the stated research objective, the following tasks were undertaken in this course of this research.

- **Task 1**: To identify existing open-source RS tool-kits and analyse them in terms of the diversity of recommendation algorithms, diversity of evaluation methods, ease of use, and flexibility of operation.
- **Task 2**: To identify and perform an experimental analysis of existing open-source Hyper-Parameter Optimization (HPO) packages that provide advanced HPO methods like Bayesian Optimization and allow for the optimization of arbitrary functions, while also observing the flexibility for defining search spaces, parallelization capabilities and ease of use.
- **Task 3**: To experimentally analyse the feasibility of an Auto-RS tool by using a chosen HPO library to optimize the hyper-parameters of individual recommendation algorithms from a chosen RS tool-kit.
- **Task 4**: To construct an Auto-RS tool using functionality from the chosen HPO library and RS tool-kit such that it performs data processing (splitting into cross-validation folds) and optimization of supported evaluation metrics over a hierarchical search space of recommendation algorithms and their associated hyper-parameters in an automated manner with minimal user intervention.
- **Task 5**: To experimentally verify the benefits of the Auto-RS tool by evaluating it on gold-standard RS data sets.

We note that in this report, we focus on the development and evaluation of the end product, i.e., the **Auto-CaseRec** tool that resulted in the fulfillment of the objective stated above and choose to omit the intricacies of the experimentation involved in Tasks 1 -3. We instead provide a comprehensive literature survey on existing RS tool-kits and HPO packages and also present an analysis of our final choices for the same, later in our report.

Our ambitions for this research and the Auto-CaseRec tool are three-fold:

1. To reduce the human effort necessary for applying Recommender System techniques. We want to make it easier for both researchers and non-expert users of Recommender Systems to apply and evaluate RS algorithms in a robust manner.

2. To improve the performance of RS algorithms for any given task by producing tailored solutions.

3. To improve the fairness and reproducibility of Recommender Systems research. An Auto-RS tool for automatically tuning RS methods would facilitate fair comparisons between different methods and also help in producing robust baselines for evaluating novel recommendation methods.

1.4 Structure of the Thesis

The thesis is organised as follows.

Section 2 provides a literature review of the related work regarding the CASH problem. This includes research work in the Recommender Systems research community related to algorithm selection and hyper-parameter optimization, a survey of the various tools developed in the AutoML community to solve the CASH problem, a survey of existing Hyper-Parameter Optimization libraries and Recommender System tool-kits.

Section 3 introduces the Auto-CaseRec tool and provides in-depth information about the various components of the tool. These include the recommendation algorithm training and evaluation, the CASH optimization process, the search space used and finally, example usage of Auto-CaseRec's Python interface.

Section 4 provides information about the experimentation conducted to evaluate the Auto-CaseRec library. It starts with details about the experimental setup, the data set used, the baselines used and the hardware and software used in the experiments. After that, the results of the experiments, a detailed analysis and summary is provided.

Section 5 provides a conclusion to the research work presented in this report. Starting with a summary, we talk about the various challenges encountered during the execution of this research work, followed by the limitations of the research, and finally, the future work that is planned for the further development of the Auto-CaseRec library.

2 Related Work

2.1 Efforts in the Recommender Systems community

An investigation into the literature related to the CASH problem in the Recommender Systems (RS) community revealed that the CASH problem had not been directly formulated in the research work. Further, only one tool, Lib-rec Auto [19, 20] was found that automates experimentation of RS algorithms with a user-fed search space of algorithms and hyper-parameters. In the original research paper, there is no mention of minimization of the *loss* or maximization of the performance of the RS algorithms and instead, the focus is on automating RS experiments. Thus, Lib-rec Auto is not eligible to be called a direct solution to the CASH problem, although, it can be a useful tool in finding out good algorithms and hyper-parameter configurations. Lib-rec Auto uses an exhaustive experimentation methodology in which an experiment involving algorithm training and evaluation is conducted for each algorithm and hyper-parameter combination in the user-given search space.

On the other hand, research work related to the two sub-problems inherent in the CASH problem: algorithm selection and hyper-parameter tuning, was found. A survey of this research work is presented in the following sub-sections.

2.1.1 Algorithm Selection in Recommender Systems

A common approach in the literature for performing Algorithm Selection is *meta-learning*. Meta-learning seeks to induce a predictive model by associating past performances of algorithms to data set features that are known as *meta-features*. Studies have shown that the performance of Collaborative Filtering (CF) algorithms on gold-standard data sets like

MovieLens[1] and Last.fm [34] is correlated to features of the data sets [35, 36].

Cunha et. al [37] use classification algorithms to learn correlations between meta-features of 32 data sets and the best-performing CF algorithm for each. They demonstrate that CF algorithms predicted by their meta-learning approach significantly outperform the baseline majority voting algorithm on both rating prediction and item recommendation tasks. Cunha et. al [38] have also proposed a label-ranking approach that predicts a ranked list of CF algorithms instead of just predicting the best performing algorithm. They show this meta-learning approach to outperform the previously mentioned classification approach. The same authors propose a unique approach called CF4CF [39] in which they use a CF algorithm to predict ranked lists of CF algorithms and show this approach to outperform the previous label-ranking approach. Researchers have also proposed meta-learning algorithms that operate at granular levels like per-user or per-instance. For the MovieLens 10M[2] data set, Ekstrand and Riedl [9] show that different algorithms fail on different parts of the data set and that different algorithms perform well for different users. They also test a meta-learning approach in which they use a linear classifier and a meta-feature to choose between two CF algorithms for each user but this approach failed to beat the overall best algorithm for the data set. At a more granular level, Collins et. al [12] proposed a novel meta-learning approach that seeks to find the best algorithm for each instance in a data set. Their approach is not able to outperform the single best algorithm for the two evaluated data sets (meta-learner RMSEs: (0.973, 0.908), SVD++ RMSEs: (0.942, 0.887)).

While the literature shows that certain meta-features of RS data sets can be effectively used to design meta-learners for algorithm selection, meta-learning approaches require significant human and computational effort to design and extract meta-features and evaluate RS algorithms on an extensive library of data sets. For example, the meta-learning approaches proposed by Cunha et. al [37, 38, 39] use meta-features and performance results from 32-38 data sets to train their meta-learners.

[1]http://grouplens.org/node/73
[2]http://grouplens.org/node/73

2.1.2 hyper-parameter Tuning in Recommender Systems

In this sub-section, we examine the research literature related to hyper-parameter Tuning of RS algorithms.

Matuszyk et. al [15] conducted an experimental study in which the effects of tuning the hyper-parameters of a matrix factorization algorithm called BRISMF [40] was tested with 9 different hyper-parameter Optimization (HPO) algorithms, on the algorithm's performance on 4 different RS data sets: MovieLens 1M and 100k, and samples of 2000 users each from the Netflix and Flixster data sets. The study shows that Nelder-Mead [41], Simulated Annealing [42], Sequential Model-based Algorithm Configuration (SMAC) [32], Random Search [4], Genetic algorithm [43] and Particle Swarm Optimization [44] perform similarly, with Nelder-Mead and Simulated Annealing being marginally better. They recommend not using the Random Walk, Greedy Search, and Grid Search algorithms as they consistently perform worse and in the case of Grid Search, much worse than the other algorithms. Galuzzi et. al [16] use a Gaussian Process (GP) based Bayesian Optimization (BO) algorithm to find the optimal hyper-parameters of a Matrix Factorization algorithm for a rating prediction task on the MovieLens 100K data set. They experiment with two acquisition functions for the BO algorithm: Expected Improvement [45] and Thomson Sampling [46]. It is shown that both BO strategies are able to find near-optimal values that minimize the Root Mean-Squared Error (RMSE) within 30 evaluations. Dewancker et. al [17] use the Bayesian Optimization algorithm of the SigOpt tool [47] to tune the hyper-parameters of a CF algorithm called Alternating Least Squares (ALS). The algorithm is evaluated in a rating prediction task on the MovieLens data set (22 million ratings). The hyper-parameters found by Bayesian Optimization show 40% improvement in RMSE from the default hyper-parameters of ALS and also outperform the hyper-parameters found by Random Search. Another study [18] shows that tuning the regularization parameter of the Matrix Factorization algorithm increases the RMSE experienced on the MovieLens 100k data set by 6.43% from the RMSE obtained by using default parameters.

An innovative application of HPO for Recommender Systems was shown by Chan et. al [14]. They apply Random Search to tune the ALS-WR [48] and Stochastic Gradient Descent (SGD) [49] RS algorithms and evaluate them on an anonymized data-set of a UK retail business that contains 2 years of purchase records. The algorithms are trained on the 1st year data and then month-wise new data is introduced. They evaluate three approaches with the addition of new data from each month of the 2nd year to the original data set: no re-training and no HPO, re-training and no HPO, re-training and HPO. Separate experiments are conducted for data from offline stores and online stores. The comparison of the RMSE values and MAP@20 values for the rating prediction and item recommendation tasks respectively shows that re-training the algorithms with the addition of new data each month as well as tuning the hyper-parameters with Random Search every fourth month outperforms the other two approaches throughout the 2nd year in both online and offline settings, with the other approaches winning in very few circumstances. This research work demonstrates that regular application of HPO to tune RS algorithms can continuously increase their recommendation quality as opposed to static hyper-parameters in the dynamic environments where data is continuously collected. Such dynamic environments are easily found in applications that use Recommender Systems.

2.2 CASH Problem in AutoML

The CASH problem defined in Section ?? has been tackled successfully in the *automated machine learning* (AutoML) domain by a variety of open-source AutoML softwares [1, 21, 22, 23, 24, 25, 26, 27, 28, 50, 51]. Given a data set, AutoML softwares automatically find an appropriate machine learning pipeline and use that to make predictions. The *Application Programming Interfaces*(APIs) provided by these softwares enable users to apply machine learning without prior knowledge about the algorithms and hyper-parameters. A machine learning pipeline consists of a sequence of processes required to perform a machine learning task (e.g., classification). Finding appropriate pipelines consists of choosing appropriate machine learning and sometimes data processing algorithms, and choosing hyper-parameter values for the algorithms.

Different AutoML softwares employ different Hyper-Paramter Optimization (HPO) approaches to find the "best" choices for the given data set. A popular and promising approach is Bayesian Optimization [52, 53], specifically, the Sequential Model-Based Optimization framework (SMBO) [32]. Two popular AutoML tools, Auto-WEKA [21, 22] and Auto-sklearn [1] use the *Sequential Model-based Algorithm Configuration* (SMAC) [32] framework, a random-forest-based variant of SMBO to tune machine learning pipelines. Auto-WEKA and Auto-sklearn are based on popular machine-learning tools: the WEKA workbench [30, 54] and the scikit-learn Python library [29] respectively. Auto-sklearn brings two new innovations: *meta-learning* to identify promising pipelines from experiences with past data sets and use them to warm start the Bayesian Optimization, and *ensemble selection* [55] to boost performance by combining pipelines found by Bayesian Optimization into an *ensemble*. Hyperopt-sklearn [25] is also based on the scikit-learn library but uses Hyperopt[56], an existing open-source Python library, for describing search-spaces and implementing HPO algorithms. Hyperopt-sklearn supports all of Hyperopt's [56] HPO algorithms: *tree Parzen Estimators* (TPE) [31], another tree-based variant of SMBO, Random Search [4], Simulated Annealing [42] and a variant of TPE called Adaptive-TPE [57].

Tree-based methods such as SMAC and TPE have been shown to perform better than other Bayesian Optimization variants as well as Random Search in large categorical and conditional search spaces such as the ones involved in solving the CASH problem [31, 58, 59]. Additionally, among tree-based methods, Thornton et. al [21] found SMAC to outperform TPE and among model-free methods, Random Search has been shown to be much more efficient than Grid Search in problems where some hyper-parameters are bound to be more important than the others [4].

Another promising approach is that of MLPlan [27], which uses *Hierarchical Task Networks*, an AI planning technique to build pipelines and supports machine learning algorithms from both the WEKA and scikit-learn libraries. In [27], an evaluation across 20 machine learning data sets, ML-Plan, configured with WEKA and scikit-learn algorithms, was compared with Auto-WEKA and scikit-learn-based tools, Auto-sklearn and TPOT respectively. ML-Plan was found to outperform Auto-WEKA in 17/20 of the data sets, with significant improvements in at least 12 cases, and to be competitive with Auto-sklearn and TPOT, outperforming them in some cases. Another interesting approach is that of *genetic programming* (GP) and is applied by TPOT [24] and RECIPE [28]. In contrast to previously mentioned approaches, TPOT and RECIPE allow for an arbitrary number of data preprocessing algorithms in the pipeline and are, therefore, more flexible. TPOT is also based on the sci-kit learn package. A drawback of TPOT is that it can create *invalid* pipelines during the search which leads to wastage of computational resources. RECIPE's grammar-based GP approach prevents it from constructing invalid pipelines [28]. Finally, among simpler approaches, the AutoML module of the H2O library [23] uses Randomized Grid Search, i.e. Grid Search with random sampling, to find pipelines.

2.3 An Overview of Open-Source HPO tools

Quite a few open-source software tools that provide HPO algorithms for the minimization of arbitrary functions have been developed in the form of Python. [32, 56, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69]. The majority of these tools implement some form of Bayesian optimization as it is the most widely used approach for minimization of expensive black-box functions. SMAC v3 [69] provides an implementation of SMAC [32]. Hyperopt is another popular library that provides the TPE algorithm [31] along with Random Search [4], Simulated Annealing [42] and Adaptive-TPE [57]. Hyperopt is also the workhorse behind the AutoML framework Hyperopt-sklearn [25]. The HpBandSter tool provides a bandit-based algorithm called Hyperband [70] along with Random Search and a combination of Hyperband and Bayesian optimization called BOHB [61]. The MOE [62], Spearmint [64], GPyOpt [68] and the Bayesian Optimization [66] packages provide Gaussian Process (GP) based Bayesian optimization. The scikit-optimize package [63] provides GP-based as well as random-forest-based Bayesian optimization. Optuna [65] is a relatively new framework that provides multiple optimization strategies that include TPE (from Hyperopt), Random Search, Grid Search as well as CMA-ES [71], an evolutionary algorithm. Additionally, Optuna implements a pruning algorithm that terminates unpromising configurations early to save computation time. Amongst these tools, SMAC v3, Hyperopt, Spearmint, Optuna, scikit-optimize and GPyOpt also provide functionality for parallelizing optimization processes.

It is observed that most of these tools rely on some form of Bayesian Optimization for this functionality. This supports the claim that Bayesian Optimization is widely adopted for the optimization of expensive black-box functions.

[2] `https://github.com/automl/HpBandSter`

2.4 An Survey of Open-Source Recommender System tool-kits

In this sub-section, a survey of existing open-source RS tool-kits is provided. A variety of tool-kits have been developed by the RS community in programming languages like C#, Python, and Java. The different RS tool-kits have been assessed in terms of the following functionality: data processing tools for RS data-sets, diversity of RS algorithms, diversity of evaluation methods, and ease of use. The appropriate choice of an RS tool-kit was crucial for this research work, as the chosen tool-kit has been directly used to provide RS functionality for the Auto-CaseRec tool.

Popular open-source RS tool-kits are Mahout[3], LensKit [72], LibRec [73], MyMediaLite[4], EasyRec[5], Case Recommender[2], Surprise[74], Spotlight[75], LightFM[76], Crab[6] and recommenderlab[7]. Mahout is offered by the Apache Software Foundation and is built on Apache Spark. It only offers memory-based CF algorithms for rating prediction. EasyRec is a Java based library that provides CF techniques for rating prediction but is not being actively developed (Last update in 2013). MyMediaLite is a popular C library and offers many CF as well as Content-based techniques, along with appropriate evaluation metrics for both rating prediction and item recommendation. The last update to MyMediaLite was in 2015. LensKit was first released as a Java library but a Python successor has now been released. LensKit Python (LKPY) provides various CF algorithms for both rating prediction and item recommendation in addition to cross-validation procedures and various evaluation metrics. Crab is another Python library that provides CF algorithms for rating prediction and item recommendation but was last updated in 2012. LibRec is a Java library that provides implementations of a huge variety of recommendation algorithms for both the rating prediction and item recommendation scenarios. Additionally, it also provides various evaluation metrics and cross-validation strategies. LibRec's set of RS algorithms is by far the largest amongst existing RS tool-kits and contains many state-of-the-art algorithms. LightFM is a Python library consisting of various rating prediction as well as item prediction algorithms but only provides a basic train-test split

[3]`https://mahout.apache.org/`
[4]`http://www.mymedialite.net/`
[5]`http://easyrec.org/`
[6]`https://github.com/muricoca/crab`
[7]`https://github.com/mhahsler/recommenderlab`

function with no cross-validation. Recommenderlab is an R package and provides various CF algorithms for rating prediction and item recommendation along with cross-validation protocols. Spotlight is an actively-developed Python library based on PyTorch and provides implicit and explicit feedback matrix factorization along with *Sequence Models*. Spotlight contains limited functionality for data processing and no cross-validation protocols like K-Fold. Also, Spotlight uses custom data structures to manage data that make it less flexible. Surprise is a popular Python library that provides CF algorithms for the rating prediction scenario but does not support implicit feedback or content-based algorithms. It also provides Grid Search for hyper-parameter optimization, a unique feature amongst RS tool-kits. Surprise documentation mentions that the library will not receive any updates after September 2019 except bug fixes. Case Recommender is another Python library that provides various rating prediction and item recommendation algorithms and handles explicit and implicit feedback, provides various evaluation metrics as well as robust cross-validation strategies. The algorithms implemented by Case Recommender include various CF algorithms, content-based algorithms, and clustering-based algorithms. Additionally, Case Recommender allows the construction of *ensemble techniques* by combining various algorithms.

3 Auto-CaseRec

3.1 Introduction

To solve the CASH stated in Section 1.2, the **Auto-CaseRec**[1] Python library was developed. The current version of Auto-CaseRec provides an automated search process that involves *optimization* over a *hierarchical search-space* consisting of various Recommender System (RS) algorithms and their associated hyper-parameters. Using the Auto-CaseRec library, practitioners can search for the "best" recommendation algorithm and its associated hyper-parameters for any arbitrary recommendation data set and recommendation scenario such as Rating Prediction and Item Recommendation with a very few lines of Python code (See Example Usage in Section 3.7). Auto-CaseRec provides users with the flexibility to modify the optimization process as well as the recommendation algorithm training and evaluation process through a set of simple parameters.

Auto-CaseRec is essentially a wrapper around the Case Recommender [2] Python library, hence the name and augments it with Hyper-Parameter Optimization (HPO) functionality derived from another Python library, Hyperopt [56], to provide the automated and combined algorithm selection and hyper-parameter optimization of RS algorithms provided by Case Recommender. Hence, the name Auto-CaseRec. An introduction to the Case Recommender and Hyperopt library is given.

[1]`https://github.com/srijang97/Auto-CaseRec`

Case Recommender library

Case Recommender [2] is an open-source Python-based Recommender System toolkit and is available on the online Python Package Index (PyPI) [2] repository for use under the MIT license. Case Recommender provides various recommendation algorithms, data processing (includes 2 cross-validation strategies), and performance evaluation functions for the construction and evaluation of Recommender Systems. In addition to these, special features provided by the Case Recommender library include a framework to implement *ensemble*[3] algorithms and statistical tests such as the T-Test and Wilcoxon test. These features make Case Recommender an all-round package for practitioners that want to develop and robustly evaluate Recommender Systems. Auto-CaseRec uses Case Recommender's data processing functionality, RS algorithms, and evaluation functions.

Hyperopt library

Hyperopt [56] is a Python library that provides a framework for carrying out hyper-parameter optimization of arbitrary functions. To implement an optimization engine using Hyperopt, the following components need to be described.

1. A search space: A Python dictionary that describes the distributions of the various hyper-parameters that have to be used in the optimization process.

2. An objective function: The function whose value will be minimized by the optimization process. Hyperopt requires the objective function to be a Python function that accepts at least one hyper-parameter and returns at least a single value of the *loss*.

3. A trials database [optional]: An optional database to store the results generated by the objective function.

4. An optimization algorithm: The optimization algorithm that will be used to generate successive algorithm configurations in the optimization process.

In addition to this, Hyperopt also provides functionality for parallelizing the search process, i.e., to evaluate multiple configurations simultaneously and thereby reduce the computation

[2]`https://pypi.org`

[3]Ensemble methods are not included in the current version of the library, Case Recommender 1.1.0, but are planned to be included in future releases.

time. Auto-CaseRec uses Hyperopt's optimization framework to tackle the CASH problem for RS algorithms.

The various features of the Hyperopt and Case Recommender library will be presented in detail alongside Auto-CaseRec's functionality. In the following sections of this Chapter, first, an introduction to the different components of Auto-CaseRec is given followed by the detailed working of the different components, and finally, example usage of the Auto-CaseRec library.

3.2 Components of Auto-CaseRec

This section gives an overview of the functioning of the Auto-CaseRec library. Auto-CaseRec's workflow has been shown in Figure 3.1.

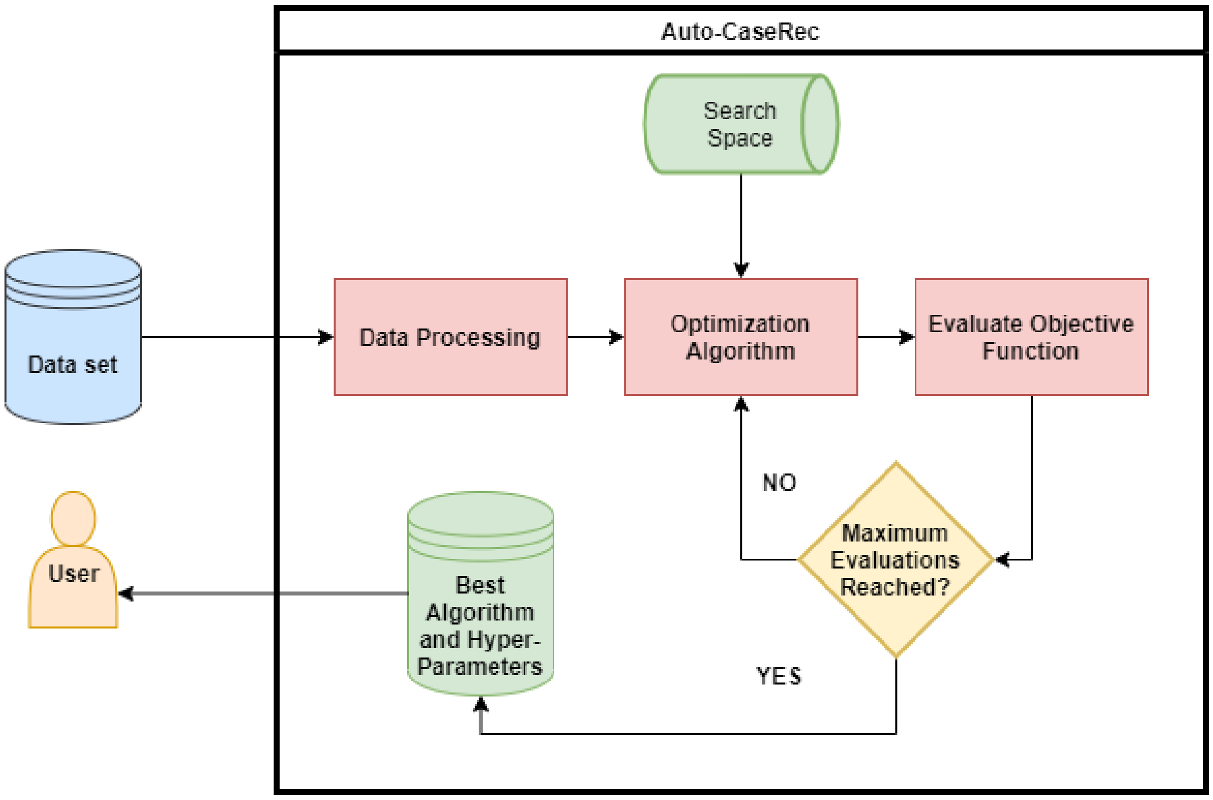

Auto-CaseRec
Search Space
Data set
Data Processing
Optimization Algorithm
Evaluate Objective Function
NO
Maximum Evaluations Reached?
User
Best Algorithm and Hyper-Parameters
YES

Figure 3.1: Workflow of Auto-CaseRec

Please note that in the subsequent matter of the report, the phrase 'an algorithm configuration' will be used as a substitute for writing 'an algorithm and its associated hyper-parameters'. To explain the working of Auto-CaseRec, it's workflow is divided into the following two components.

1. Data Processing: This component is responsible for splitting the recommendation data set into training and testing *folds* according to the cross-validation strategy dictated by the user.

2. Optimization Engine: The optimization engine is responsible for successively sampling a new algorithm configuration from the *search space* according to the optimization strategy, evaluating the *objective function* using the sampled configuration and recording the error rate, i.e., the *loss* until the maximum number of evaluations is reached. Finally, the algorithm configuration that produced the best loss value is returned to the user. The optimization engine can be further divided into the following components.

   (a) Search Space: The search space is a hierarchical space as described in the CASH problem (Section 1.2), with recommendation algorithms at the root-level and their associated hyper-parameters domains as their children.

   (b) Objective function: The objective function is responsible for generating a loss value for the algorithm configuration provided by the optimization algorithm. This involves training the recommendation algorithm on the training data and evaluating its predictions on the test data.

   (c) Optimization Algorithm: The optimization algorithm is responsible for generating new algorithm configurations. The generation strategy may or may not be based on the loss values encountered for previous configurations.

Now, the working of Auto-CaseRec is presented formally. Starting with a recommender system dataset D, Auto-CaseRec creates K cross-validation folds of the form $fold^{(i)} = \{D^{(i)}_{train}, D^{(i)}_{test}\}$ where $K \geq 1$ and is specified by the user. Let $\mathcal{A} = \{A^{(1)}, \cdots, A^{(1)}\}$ be the set of recommendation algorithms and let the hyper-parameters of each algorithm $A^{(j)}$ be $\lambda \in \mathbf{\Lambda}^{(\mathbf{j})}$. Finally, let $\mathcal{L}(A^{(j)}_{\lambda}, D^{(i)}_{train}, D^{(i)}_{test})$ be the loss experienced on $D^{(i)}_{test}$ when algorithm A is trained on $D^{(i)}_{train}$ with parameters λ. Auto-CaseRec's optimization process aims to compute

$$\boldsymbol{A}^{*}_{\boldsymbol{\lambda}^{*}} \in \underset{A^{(j)} \in \mathcal{A}, \lambda \in \mathbf{\Lambda}^{(\mathbf{j})}}{\operatorname{argmin}} \frac{1}{K} \sum_{i=1}^{K} \mathcal{L}(A^{(j)}_{\lambda}, D^{(i)}_{train}, D^{(i)}_{test})$$

The results $\boldsymbol{A}^{*}$ and $\boldsymbol{\lambda}^{*}$ are the recommendation algorithm and associated values of hyper-parameters that produce the minimum mean loss across the K folds of the data within the maximum number of evaluations allowed.

In the following sub-sections, the Data Processing component and the various components of Auto-CaseRec's optimization engine are presented in detail in separate sections.

3.3 Data Processing

The data processing component of Auto-CaseRec involves reading the recommendation data set supplied by the user and splitting it into training and testing sets.

Auto-CaseRec implements Case Recommender's data processing functions and cross-validation strategies to divide recommendation data sets into training and testing sets. Case Recommender supports files with delimiter-separated values. Such files contain data in a two-dimensional array, where, each row is a data record and the columns are fields. Each field value is separated by a delimiter, for example, comma (",") is used as the delimiter for CSV (comma-separated value) files. Case Recommender accepts recommendation data sets that contain records in the form of user-item-feedback triplets. A recommendation data set D can be represented as a matrix $\boldsymbol{D_{p\times3}}$ with the row-vector $D^{(i)} = [user_i, item_i, feedback_i]$. The three values $user_i$, $item_i$ and $feedback_i$ correspond to the *user-ID*, *item-ID* and *feedback value* at row i of the dataset. Each user may be associated with one or more items and each item may be associated with one or more users.

Case Recommender provides two cross-validation strategies to split the data into training and test sets.

1. Shuffle Split: A cross-validation strategy in which a fixed number of training and testing set folds are generated, where for each fold, the data points are allocated randomly to the train and test sets. The testing sets may or may not be mutually disjoint. The training and testing set sizes are specified by the user beforehand.

2. K-Fold Cross Validation: A cross-validation strategy in which the data is divided into K samples following which K folds of training and testing sets are generated. In each fold, a different sample is used as the testing set while the other K-1 samples are compiled into the training set. The number of folds, K, is specified beforehand.

After the data is read, Auto-CaseRec uses Case Recommender's cross-validation strategies to arrange the data into K training and testing sets called folds, where each fold is of the type $fold^{(i)} = \{D^{(i)}_{train}, D^{(i)}_{test}\}$. If a cross-validation strategy is chosen, $K > 1$, else, $K = 1$ The training sets of the different folds are not disjoint, i.e., $D^{(1)}_{train} \cap \cdots \cap D^{(K)}_{train} \neq \emptyset$. The testing sets are disjoint in case K-Fold Cross Validation is chosen.

The input data is processed and the training and testing files are stored on the local drive, and the files are imported during the algorithm training process. One reason for this is to reduce the memory overhead from storing data in program variables. The parameters for the cross-validation strategies, the input file delimiter, and the local directory path for storing the different folds can be specified by the user.

The cross-validation functionality provides a robust estimate of algorithm performance by training and testing the concerned recommendation algorithm on different permutations of the same data set. This robust evaluation of performance is crucial as a poor evaluation of the performance of an algorithm will directly propagate a bias to the optimization process as well, thereby reducing the quality of solutions produced by the optimization algorithm. The single or multiple data folds created are written to the user's local drive for further use.

3.4 Objective Function

In mathematical optimization, the function to be maximized or minimized is called the objective function. Hyperopt requires the objective function to be defined as any Python function that accepts the algorithm configuration as a parameter and at a minimum, returns a single value that is the result of computing the function value at the configuration supplied. The value returned is termed as the loss of the function and it is this loss which is to be minimized by the optimization process. If needed, the objective function can be designed to return auxiliary information that may be of value to the user. For example, a researcher may require the store of the function computation time, the configuration supplied to the function, etc. for further analysis. The information returned by the objective function can be stored for further use in a Trials database object which is passed to the Hyperopt optimization function beforehand. The Trials object is an in-built class object of Hyperopt that stores information returned by the objective function in the form of a Python dictionary.

The objective function implemented in the Auto-CaseRec library accepts a Python dictionary as a parameter, which contains an algorithm configuration $A_{\lambda}^{(j)}$ and returns a Python dictionary that contains information about the iteration, which is then stored in a Trials object inside the optimization engine.

Now, the working of Auto-CaseRec's obejctive function is explained. The objective function is responsible for computing the loss value for a given algorithm $A^{(j)} \in \mathcal{A}$ and associated hyper-parameter values $\lambda \in \boldsymbol{\Lambda^{(j)}}$. For explanation, we consider the simple case where the number of folds, K is equal to 1 and $fold^{(1)} = \{D_{train}, D_{test}\}$. First, The algorithm $A^{(j)}$ instantiated with parameters λ is trained on the training data set D_{train}. Simply put, training a recommendation algorithm means *learning* an approximation $\hat{f}(user_i, item_i)$ of the actual function $f(user_i, item_i) = feedback_i$, where $user_i$, $item_i$ and $feedback_i$ belong to the i$^{\text{th}}$ row-vector $D_{train}^{(i)}$ of D_{train}. The actual function f is not known in mathematical form but the data set D contains known values of f for a subset of user-item pairings. The recommendation algorithm uses the known function values in D_{train} to find the approximated function $\hat{f}$.

After the training process has ended, the function value of $\hat{f}$ is then computed for each user-item pair in the testing data set D_{test} and the output values are stored as the predictions $\hat{y} = \{\hat{f}(user_1, item_1), \cdots, \hat{f}(user_q, item_q)\}$, where q is the number of rows in D_{test}. The ground-truth feedback values $\boldsymbol{y}$ for user-item pairs in D_{test} are known beforehand. The loss is computed using a function $\mathcal{L}(y, \hat{y})$ that is different for different evaluation metrics.

If the user opts to use cross-validation, the recommendation algorithm is trained on all the K folds and the loss is calculated separately for each fold. The final loss is calculated by averaging the losses for all the K folds.

$$loss = \frac{1}{K} \sum_{i=1}^{K} \mathcal{L}(y_i, \hat{y}_i),$$

where y_i and $\hat{y}_i$ are the ground-truth feedback values and predicted feedback values for $fold_i$. The final loss value is returned to the optimization algorithm. There are different evaluation metrics that can be used in the objective function, depending on whether the recommendation scenario is that of rating prediction or item recommendation.

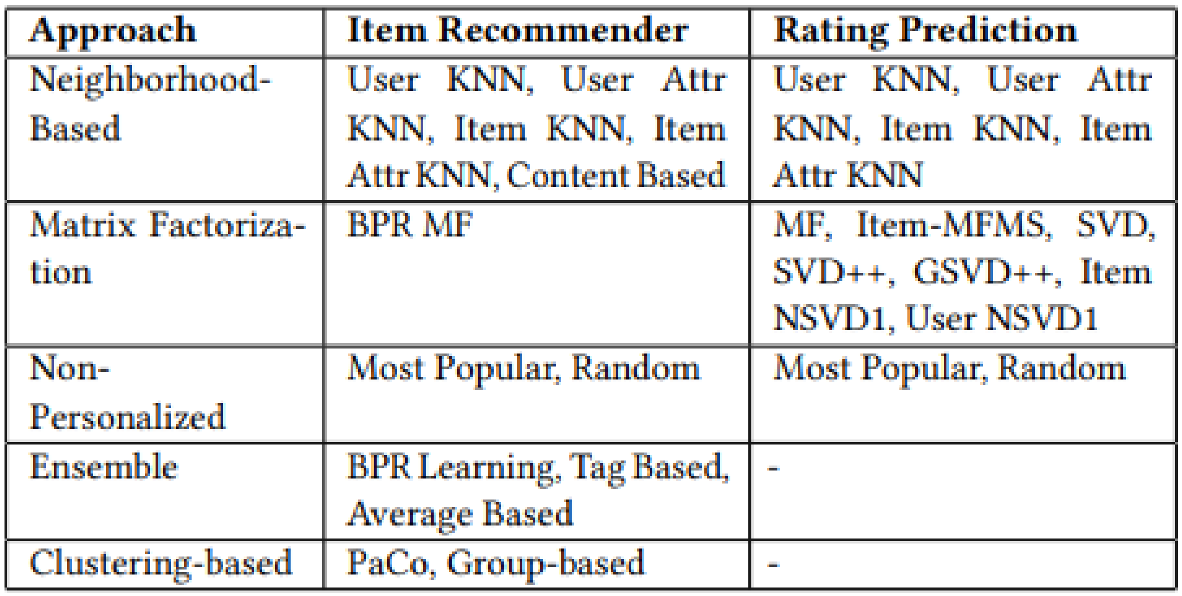
Auto-CaseRec also implements an **early-stopping** mechanism for the cross-validation process. If early stopping is selected by the user, Auto-CaseRec prematurely terminates the cross-validation process after the computation of the loss of the i^{th} fold if the minimum loss value associated with the first i folds is not less than the best loss experienced in the optimization process. The value i can also be set by the user, with $i \leq K$. The advantage of early-stopping is the significant reduction in computation time for configurations that do not seem to be promising based on the evaluation of the first i folds. The limitation is that in the process of reducing computation time, the robustness of the performance evaluation of the recommendation algorithm is compromised by only evaluating it on a fraction of the data set folds. The parameter i is therefore representative of the trade-off between computation time and robustness of evaluation. For higher values of i, the evaluation is more robust as the number of folds used for evaluation is higher, while for lower values, the computation time is reduced by a factor of $\frac{i}{K}$.

It is worth mentioning that the quality of predictions $\hat{y}$ and hence the loss $\mathcal{L}(y, \hat{y})$ is dependent on the hyper-parameters λ that the recommendation algorithm A is instantiated with. This is because the values of the hyper-parameters are used directly in the training process of the algorithms. Therein is the need to find good hyper-parameter configurations.

Now, the various recommendation algorithms and evaluation functions provided by Case Recommender that have been used in the Auto-CaseRec library are described below.

3.4.1 Recommendation Algorithms

Case Recommender provides implementations of a variety of recommendation algorithms, for use in two different recommendation scenarios: Rating Prediction and Item Recommendation. In the Rating Prediction scenario, the respective recommendation algorithms use *explicit feedback* such as ratings given by the users, to predict unknown ratings. The predicted ratings are considered estimates of user preferences. In the Item Recommendation scenario, usually, direct measures of user preferences such as ratings are not available, and *implicit feedback* such as user purchase history, view history or item play-counts are used. The Item Recommendation algorithms assume that the implicit feedback is representative of the user's preferences and predict a ranked list of preferred items for each user. The algorithms provided by Case Recommender including Neighborhood-based algorithms, Matrix Factorization algorithms as well



Approach	Item Recommender	Rating Prediction
Neighborhood-Based	User KNN, User Attr KNN, Item KNN, Item Attr KNN, Content Based	User KNN, User Attr KNN, Item KNN, Item Attr KNN
Matrix Factorization	BPR MF	MF, Item-MFMS, SVD, SVD++, GSVD++, Item NSVD1, User NSVD1
Non-Personalized	Most Popular, Random	Most Popular, Random
Ensemble	BPR Learning, Tag Based, Average Based	-
Clustering-based	PaCo, Group-based	-

Figure 3.2: Recommendation Algorithms in the Case Recommender library [2]

Table 3.1: Recommendation Algorithms in Auto-CaseRec

Algorithm	Type	Rating Prediction	Item Recommendation
ItemKNN	Neighborhood-based	✓	✓
UserKNN	Neighborhood-based	✓	✓
Matrix Factorization	Matrix Factorization	✓	
SVD	Matrix Factorization	✓	
Most-Popular	Non-Personalized	✓	✓
Random Recommender	Non-Personalized	✓	✓
BPR-MF	Matrix Factorization		✓

as clustering-based algorithms. The algorithms provided by the Case Recommender library are shown in 3.2.

Each algorithm can be implemented using a simple Python statement, that follows a generic structure, with the addition or removal of a few specialized hyper-parameters that are unique to each algorithm. Currently, Auto-CaseRec implements only a subset of the total set of recommendation algorithms in CaseRec, which have been described in Table 3.1.

The algorithms that have not been included require *metadata* files in addition to the main recommendation data set containing user feedback for items. Metadata is usually additional information regarding the users or items, for example, in a movie rating scenario, metadata could describe movie genres and titles. While metadata can potentially enrich configurations, the Auto-CaseRec library is still in the preliminary experimentation stage which just involves basic versions of recommendation data sets without any metadata. These algorithms will surely be included in the future.

3.4.2 Evaluation Functions

Auto-CaseRec implements the evaluation functions for all the different metrics provided by Case Recommender. The user can choose any of the metrics as the minimization objective of Auto-CaseRec's optimization process, depending on the recommendation scenario. The various metrics are described in this section.

Lets consider the simple case where the data set D is split into only one data fold, $fold^{(1)} = \{D_{train}, D_{test}\}$. Let the testing data set D_{test} be of length q. To evaluate the performance of recommendation algorithms in the rating prediction and item recommendation tasks, Case Recommender provides computation of the following metrics:

- Rating Prediction: Let the predictions made by the rating prediction algorithm for D_{test} be $\hat{y} \in \mathbb{R}^q$ and the ground-truth feedback values for D_{test} be $y \in \mathbb{R}^q$, hence, both are vectors of length q. The evaluation metrics for rating prediction algorithms are as follows.

    1. MAE (Mean Squared Error): This is the mean of the absolute differences between the predicted ratings and the ground truth ratings in the test dataset.

    $$MAE = \frac{1}{q}\sum_{i=1}^{q} |y_i - \hat{y}_i|$$

    2. RMSE (Root Mean Squared Error): This is the root of the mean of the squared differences between the predicted ratings and ground truth ratings in the test dataset.

$$RMSE = \sqrt{\frac{1}{q}\sum_{i=1}^{q}(y_i - \hat{y}_i)^2},$$

- Item Recommendation: The item recommendation algorithms produce a ranked list of items $predicted\_items_{u,N} = \{item^{(1)}, \cdots, item^{(N)}\}$ of a pre-specified length N for each user $u \in \mathcal{U}$, the set of all users. The predicted ranked lists can be evaluated using the following metrics:

  1. Precision@K: For a user $u \in \mathcal{U}$, the $precision@K_u$ is the fraction of relevant items in the top K items of the ranked list predicted by the algorithm.

  $$relevant\_items_{u,K} = test\_items_u \cap predicted\_items_{u,K},$$
  $$precision@K_u = \frac{n(relevant\_items_{u,K})}{K},$$

  where, $test\_items_u$ is the set of all items that the user u actually interacted with and $predicted\_items_{u,K}$ is the set of first K items in the predicted rank list for the user u.

  2. Recall@K: For a user $u \in \mathcal{U}$, the $recall@K_u$ is the number of relevant items in the top K predictions divided by the maximum possible number of relevant items for the user.

  $$recall@K_u = \frac{n(relevant\_items_{u,K})}{n(test\_items_u)}$$

  3. MAP@K (Mean Average Precision @K): The $MAP@K$ is the mean of the $AP@K_u$ (Average precision @K) across all the users $u \in \mathcal{U}$. The $AP@K_u$ is the mean of the $precision@K_u$ across all relevant items in the predicted ranked list.

  $$AP@K_u = \frac{1}{n(relevant\_items_{u,K})}\sum_{k=1}^{K} precision@K_u \times rel_u(k)$$

where, $rel_u(k)$ is 1 if the k^{th} item in the ranked list is relevant to the user u, otherwise 0. Then, the $MAP@K$ is calculated as

$$MAP@K = \frac{1}{|\mathcal{U}|} \sum_{u \in \mathcal{U}} AP@K_u$$

4. NDCG@K (Normalized Discounted Cumulative Gain @K): The $CG@K$ (cumulative gain) is the sum of the relevance scores of the top K items in the predicted ranked list for a given user $u \in \mathcal{U}$.

   $$CG@K_u = \sum_{k=1}^{K} rel_u(k)$$

   The $DCG@K$ (discounted cumulative gain) penalizes the relevance with decreasing rank, i.e., if the item is lower in the ranked list.

   $$DCG@K_u = \sum_{k=1}^{K} \frac{rel_u(k)}{\log_2(k+1)}$$

   The $DCG@K$ divided by the $iDCG@K$ (ideal Discounted Cumulative Gain) gives the $NDCG@K$ (normalized DCG). The $iDCG@K$ is the $DCG@K$ of the predicted ranked list sorted in the ideal order (more relevant items have higher ranks). For example, if the relevance scores of the ranked list are [0, 1, 3, 2, 5], the ideal order would be [5,3,2,1,0].

   $$NDCG@K_u = \frac{DCG@K_u}{iDCG@K_u}$$

In the item recommendation scenario, evaluation metrics such as precision and recall are positive descriptors of performance, in contrast with metrics like RMSE in the rating prediction scenario. Hence, the optimization goal changes from minimization to maximization in the item recommendation task. However, Hyperopt only supports minimization of loss and hence, in the item recommendation scenario, the final loss value returned to the optimization algorithm in each iteration is multiplied by a negative sign $(-)$. The negation of the loss converts the maximization problem to a minimization problem.

Case Recommender contains an evaluation class for each of the item recommendation and rating prediction scenarios. The evaluation class objects provide simple function calls to run the evaluation and calculate the values of one or even multiple evaluation metrics that have been passed as a function parameter.

3.5 Search Space

The search space defines the domains of the hyper-parameters that are to be optimized. Auto-CaseRec uses Hyperopt's functionality to define the search space. The hyper-parameters are treated as variables that have a fixed type, for example, continuous, discrete, or categorical, and a domain of values that they can take. The search space is a crucial part of the optimization process and hyper-parameter domains should be such that there is a reasonable probability of finding good solutions in them. The definition of these domains is mostly guided by prior knowledge or intuition about how the concerned algorithm performs with certain hyper-parameter values. A poorly defined search space will not yield good results, no matter how good the optimization algorithm is. In addition to continuous, categorical, and discrete domains, Hyperopt allows the user to define probability-based categorical domains, normally-distributed domains as well as exponentially-distributed domains. Hyperopt also allows users to define more complex search spaces, such as hierarchical search spaces or variables with domains as deterministic expressions.

For combined algorithm selection and hyper-parameter optimization, Auto-CaseRec uses a hierarchical search space $\mathbf{\Lambda} = \mathbf{\Lambda^{(1)}} \cup \cdots \mathbf{\Lambda^{(m)}} \cup \{\lambda_r\}$, where $\{\lambda_r\}$ is a root-level hyper-parameter that represents the choice between recommendation algorithms in $\mathcal{A}$. The hyper-parameters in each sub-space $\mathbf{\Lambda^{(j)}}$ are made conditional on $\{\lambda_r\}$ being instantiated to $A^{(j)}$. The root level of the search space consists of the set of recommendation algorithms $\mathcal{A}$. During sampling, the optimization algorithm is required to choose an algorithm from the root level, followed by specific values for all the hyper-parameters associated with that algorithm. This conditional model of the search space reduces optimization time as the optimization algorithm does not have to learn by trial-and-error that one algorithm's hyper-parameters do not affect the performance of another algorithm.

Table 3.2: Rating Prediction Search Space

Algorithm	Hyperparamter	Type	Domain
ItemKNN	k_neighbors	Discrete	$\{1, \cdots, 100\}$
	similarity_metric	Categorical	$['cosine', 'euclidean', 'correlation']$
	as_similar_first	Categorical	$[True, False]$
Matrix Factorization	factors	Discrete	$\{10, \cdots, 200\}$
	learn_rate	Continuous	$[0.001, 0.1]$
	delta	Continuous	$[0.001, 0.1]$
SVD	factors	Discrete	$\{10, \cdots, 200\}$
Random Recommender	uniform	Categorical	$[True, False]$
UserKNN	k_neighbors	Discrete	$\{1, \cdots, 100\}$
	similarity_metric	Categorical	$['cosine', 'euclidean', 'correlation']$
	as_similar_first	Categorical	$[True, False]$
Most Popular	-	-	-

Each optimization process executes for either the rating prediction scenario or the item recommendation scenario. This is because both, the set of algorithms $\mathcal{A}$ and the set of evaluation metrics are different for the two scenarios. Hence, Auto-CaseRec also defines different search spaces for the optimization process of each recommendation scenario. The user is required to specify the scenario type for the optimization process as well as the evaluation metric which will be computed by the objective function at the time of initialising the optimization engine. The search spaces for both the recommendation scenarios are shown in Table 3.2 and Table 3.3 respectively.

The domains for the different hyper-parameters were defined based on a mixture of previous knowledge derived from experimentation and by reviewing the original research papers of some of the recommendation algorithm, and intuition.

The search spaces can be further optimized down based on experiments conducted with

Table 3.3: Item Recommendation Search Space

Algorithm	Hyperparamter	Type	Domain
ItemKNN	k_neighbors	Discrete	$\{5, \cdots, 50\}$
	similarity_metric	Categorical	$['cosine','euclidean','correlation']$
	as_similar_first	Categorical	$[True, False]$
BPR-MF	factors	Discrete	$\{10, \cdots, 200\}$
	learn_rate	Continuous	$[0.001, 0.1]$
UserKNN	k_neighbors	Discrete	$\{5, \cdots, 50\}$
	similarity_metric	Categorical	$['cosine','euclidean','correlation']$
	as_similar_first	Categorical	$[True, False]$
Random Recommender	-	-	-
Most Popular	-	-	-

the Auto-CaseRec library on many diverse recommendation data sets, and it is one of the goals for the next version of the Auto-CaseRec library. Further experimentation will also be analyzed to find correlations between features of the data set (example, number of users, number of items, sparsity value, data set type, and so on) and hyper-parameter domains that produce well-performing configurations. The relationship between data set features and hyper-parameter domains can then be exploited to make a personalized search space generation function according to the recommendation data set and scenario. We leave this for future work.

3.6 Optimization

As mentioned previously, the goal of Auto-CaseRec's search process is to compute

$$\boldsymbol{A}^*, \boldsymbol{\lambda}^* \in \underset{A^{(j)} \in \mathcal{A}, \lambda \in \boldsymbol{\Lambda}^{(j)}}{\operatorname{argmin}} \frac{1}{K} \sum_{i=1}^{K} \mathcal{L}(A_\lambda^{(j)}, D_{train}^{(i)}, D_{test}^{(i)})$$

The algorithm $\boldsymbol{A^*}$ and associated hyper-parameter values $\boldsymbol{\lambda^*}$ that minimize the mean loss are found using an optimization algorithm. The optimization algorithm executes as an iterative process. In each iteration, the optimization algorithm samples an algorithm configuration $A_\lambda^{(j)}$ from the search space and executes the objective function with the sampled configuration.

The loss value computed by the objective function for the configuration $A_\lambda^{(j)}$ may or may not be used by the optimization algorithm to decide the next configuration to sample. Auto-CaseRec implements the TPE, Random Search, and Simulated Annealing algorithms from the Hyperopt library. These are presented in detail below.

3.6.1 Optimization Algorithms

Hyperopt provides four optimization algorithms, namely, Tree of Parzen Estimators (TPE), Adaptive-TPE (ATPE), Simulated Annealing, and Random Search. The TPE and ATPE algorithms are based on Sequential Model-based Optimization (SMBO) approach, while Random Search and Simulated annealing are model-free approaches, i.e., they do not use a probabilistic model to decide which configurations to sample. Hyperopt also allows users to use multiple optimization algorithms for a single optimization process by providing a weight of participation for each algorithm. The various optimization algorithms are described below. The ATPE algorithm is not described as it has not been included in the current version of Auto-CaseRec.

Random Search

Random Search is a simple model-free, gradient-free optimization algorithm. The random search algorithm randomly samples configurations from the search space and evaluates the objective function on these configurations.

Simulated Annealing

Simulated Annealing (SA) is a probabilistic method for finding the global minima of an objective function that may possess several local minima. SA is modelled after annealing, a metallurgical method in which, a material is heated to a specific temperature and then slowly cooled. At high temperatures, the atoms of the material tend to migrate from the crystal lattice. This diffusion process has the effect of reducing the impurities in the material, thereby reducing its overall energy. Then, it is slowly cooled back to the fully solid state and exhibits more favorable properties like increased ductility as a result of the annealing process.

In SA, the objective function is analogous to the metal energy, which is to be minimized. A random configuration is taken as the starting point of the process. At each iteration, a random configuration sampled from the neighborhood of the current configuration and is evaluated. If it produces a better result than the current configuration, it is accepted as the new current configuration. If it is worse, it is still accepted but with a probability that is directly proportional to the temperature and inversely proportional to the difference in the performance of the solutions. This probabilistic acceptance of worse solutions has the effect of rescuing the system from local minima. The temperature is analogous to the temperate in the annealing process. It is slowly lowered so that the probability of accepting worse solutions is lowered and the algorithm converges.

Tree Parzen Estimators (TPE)

TPE is a variant of a Bayesian Optimization framework, Sequential Model-Based Optimization (SMBO). For the explanation of SMBO and TPE, the Auto-WEKA research paper [21] is referred to by the author. SMBO is a Bayesian Optimization framework that can work with both categorical and continuous hyper-parameters and is known to be able to exploit hierarchical structures in the hyper-parameter space. The algorithm for SMBO is shown in Algorithm 1.

Algorithm 1 Sequential Model-Based Optimization

```
1: Initialise model $\mathcal{M}_L$; History $\mathcal{H} \leftarrow \emptyset$
2: repeat
3:     $\boldsymbol{\lambda} \leftarrow$ candidate configuration from $\mathcal{M}_L$
4:     Compute $y = \mathcal{L}(A_\lambda, D^{(i)}_{train}, D^{(i)}_{valid}$
5:     $\mathcal{H} \leftarrow \mathcal{H} \cup \{(\lambda, y)\}$
6:     Update $\mathcal{M}_L$ given $\mathcal{H}$
7: until The optimization budget has been exhausted
8: return $\boldsymbol{\lambda}$ from $\mathcal{H}$ with minimal $y$
```

SMBO first builds a $\mathcal{M}_L$ that approximates the relation between the loss function $\mathcal{L}$ with the hyper-parameter settings λ. Then, it executes the following loop: uses the model to generate a promising configuration (line 3), evaluate the loss y of the configuration (line 4) and use

(y, λ) to update the model $\mathcal{M}_L$. To select the next hyper-parameter configuration using model $\mathcal{M}_L$, SMBO uses an *acquisition function*. The most popular acquisition function is *Expected Improvement*. Let $y(\lambda)$ denote the error of thhe hyper-parameter configuration λ. Then, the positive improvement is defined by

$$I_{y_{min}}(\lambda) = \max\{y_{min} - y(\lambda), 0\}$$

As $y(\lambda)$ is not known beforehand, its expectation over the model $\mathcal{M}_L$ is calculated as follows.

$$E_{\mathcal{M}_L}[I_{y_{min}}(\lambda)] = \int_{-\infty}^{y_{min}} \max\{y_{min} - y(\lambda), 0\} * p_{\mathcal{M}_L}(y|\lambda)dy$$

The next hyper-parameter configuration is the value that maximizes the Expected Improvement (EI). Different SMBO variants use different kinds of models. Some examples are Gaussian Processes (GP), random-forest based models, and TPE models. Random-forest based models are used by SMAC [32], the optimization framework of Auto-sklearn [1]. Here, the modelling strategy employed by TPE is explained.

Instead of directly modelling $p(y|\lambda)$, TPE models $p(y)$ and $p(\lambda|y)$. This comes directly from the Bayes' Rule:

$$p(y|\lambda) = \frac{p(\lambda|y) * p(y)}{p(\lambda)}$$

Then, $p(\lambda|y)$ is modelled using two density estimates, depending on whether y is greater than or less than a threshold value y^* as follows.

$$p(\lambda|y) = \begin{cases} l(\lambda), \text{if } y < y^* \\ g(\lambda), \text{if } y \geq y^* \end{cases}$$

The density estimates $l(.)$ and $g(.)$ are learned using previous hyper-parameters values that produced a loss below and above the threshold respectively. Bergstra et. al [31] show that by substituting this into the Expected Improvement formula, EI takes the form:

$$E_{\mathcal{M}_L}[I_{y_{min}}(\lambda)] \;\alpha\; (\gamma + \frac{g(\lambda)}{l(\lambda)} * (1-\gamma))^{-1}$$

TPE then maximizes EI by generating random configurations and picking λ that minimizes $\frac{g(\lambda)}{l(\lambda)}$. We can intuitively see that this would try to generate configurations that have a high probability under $l(.)$ and a low probability under $g(.)$.

3.6.2 Discussion

In practice, the optimization process is limited by the computational budget set forth by the user. Therefore, the optimal algorithm $\boldsymbol{A^*}$ and associated hyper-parameter values $\boldsymbol{\lambda^*}$ are not guaranteed to be the global minima of the loss function and are simply the configuration with the minimum associated mean loss that is found by Auto-CaseRec in the allowed number of evaluations or period of time. The output of the optimization process is the best algorithm configuration $\boldsymbol{A^*_{\lambda^*}}$. Auto-CaseRec is also designed to store information about each iteration of the optimization process. This information includes the configuration used in that iteration, the computation time of the objective function as well as the loss experienced in that iteration. This information can be used to perform further analysis of the optimization process.

An alternative approach to designing the optimization process would be to run the optimization process separately for each recommendation algorithm. This approach would use a different search space for each recommendation algorithm instead of the single conditional search space used by Auto-CaseRec. For each recommendation algorithm, the optimization process would run for a fixed number of iterations. The best performing configurations of each algorithm would be compared and the best among them would be presented as the optimal configuration $\boldsymbol{A^*_{\lambda^*}}$. Although this approach would theoretically achieve optimal results, I argue that it suffers from two problems. The argument is based on the assumption that different recommendation algorithms produce different results on a given dataset and the performance rankings also vary across different datasets.

1. Poor resource allocation: Logically, allocating an equal amount of iterations to the best

performing and worst performing algorithms would be a wastage of resources. Instead, the system should be designed to allocate resources according to algorithm performance itself. Setting unequal iteration budgets for the different algorithms would require prior knowledge about the dataset and algorithm performance that is not always readily available. Even with prior knowledge, the allocation of resources would have to be done manually.

2. Sub-optimal performance: In a limited computational budget, such a system could fail in allocating enough iterations for the best algorithm in the given recommendation scenario. This would prevent the optimization process to find better solutions by testing more configurations for the best algorithm and would hence increase the chances of the final solution being sub-optimal.

Auto-CaseRec's optimization process is designed to avoid both problems. By making the choice of recommendation algorithm as a hyper-parameter in itself, the resource allocation task is handled by the optimization algorithm in real-time. As the optimization process proceeds, the optimization algorithm is able to exploit the fact that one algorithm performs better than another and as a result, generates more configurations that include the better performing algorithm. The result is the increased probability of finding the best solution for the recommendation scenario.

Auto-CaseRec's optimization process was implemented using the Hyperopt Python library. Auto-CaseRec implements Hyperopt's various optimization strategies as well as the functionality for defining the search space and storing results from the optimization process. Hyperopt's features have been described in section **??** of this report.

3.7 Example Usage

Auto-CaseRec provides the **AutoEstimator** class for running an automatic configuration search for an arbitrary recommendation data set. The AutoEstimator class object can be implemented from the **estimator** module of the Auto-CaseRec library. During initialisation, AutoEstimator internally reads the data set file whose local path is provided by the user, splits the data into K folds, and stores the data folds on the user's local drive. By calling the *fit()* method of the AutoEstimator class object, the configuration search is initiated. Example usage is shown below.

```
#Example configuration search using Auto-CaseRec for the rating
   prediction scenario
#Import the AutoEstimator module
from auto_caserec.estimator import AutoEstimator

#Define the settings for the search process
kwargs = {'datapath': 'datasets/ml-latest-small/ratings.csv',
                         'predictor': 'rating',
                         'eval_metric': 'RMSE',
                         'early_stop': True,
                         'early_stop_split': 2,
                         'algo': 'tpe',
                         'max_evals': 50,
                         'cross_validate':True,
                         'cross_validation_strategy': 'kfold',
                         'n_splits': 5,
                         'sep_read': ',',
                         'sep_write': '\t',
                         'write_results': True
                         }

#Initialise the AutoEstimator object
myEstimator = AutoEstimator(**kwargs)

#Start the search process
best, trials = myEstimator.fit()
```

In each iteration, AutoEstimator prints out the loss for each data fold, the training and prediction time for each data fold, as well as the mean loss at the end of the iteration and the overall best loss. When the search process ends, the fit method returns the best algorithm configuration in the form of a data dictionary, as well as the Trials object that contains information about each iteration. The data fields stored in the Trials object are the *loss*, *iteration count*, *computation time*, *configuration* and *status* associated with the search iteration.

```
#Run the search process
best, trials = myEstimator.fit()

#Get the results as a Python dictionary from the Trials object
results = trials.results

#Print the results
print(results)
```

Each algorithm brings certain benefits and biases to a search problem. In some cases, it may be useful to use a mix of optimization strategies for the optimization process.

```
#Import different search algorithms and necessary packages
from auto_caserec.estimator import AutoEstimator
from hyperopt import anneal, rand, tpe, mix
from functools import partial

#Define the mixed optimization strategy
#Uses Random Search 20% of the time, Simulated Annealing 10% of the time
   and TPE 70% of the time
mix_algo = partial(mix.suggest, p_suggest=[
           (0.1, anneal.suggest),
           (0.2, rand.suggest),
           (0.7, tpe.suggest)])

#Define the settings for the search process
kwargs = {'datapath': 'datasets/ml-latest-small/ratings.csv',
                         'predictor': 'rating',
                         'eval_metric': 'RMSE',
                            \textbf{'algo': mix_algo}
                            }

#Initialise the AutoEstimator object
myEstimator = AutoEstimator(**kwargs)

#Start the search process
best, trials = myEstimator.fit()
```

4 Evaluation

This chapter of the report describes the experimental setup used to evaluate the Auto-CaseRec library, followed by the results obtained from the experiments, the interpretation of these results, and the conclusions formed regarding the performance of Auto-CaseRec library, including the associated time complexity.

4.1 Experimental Setup

This section describes the experimental setup used to evaluate Auto-CaseRec's optimization process when applied to two different data sets, the *Movielens 100K* and *Last.fm* recommender system data sets. Two experiments were conducted for each data set, one for the rating prediction scenario and one for the item recommendation scenario. The following subsections describe in detail the design of the experiments, the data sets used, the baselines used, and the hardware and software used to conduct the experiments and analyze the results.

4.1.1 Design of Experiments

To evaluate Auto-CaseRec's search process, 4 experiments were conducted. The experiments were conducted across two datasets: MovieLens-100K and Last.fm, and two recommendation scenarios: Item Recommendation and Rating Prediction. Each experiment was constituted of 5 *tests*, where in each test, Auto-CaseRec's search process was executed for 50 iterations for the data set D and recommendation scenario R, using the optimization algorithm A_{opt} for the search process. All experiments used 5-fold cross-validation to compute the loss for a given configuration with the early-stopping mechanism invoked after the 3rd fold. For a given configuration sampled by the optimization algorithm, early-stopping exits the cross-validation process after the 3rd fold if the best loss achieved in the 3 folds is not better than the overall best loss experienced in the search process. Early-stopping was used to reduce computation time for configurations that did not produce satisfactory results until the 3rd fold and cross-validation was used to ensure that the sampled configurations were evaluated robustly. The search process for each test randomly samples the first few configurations from the search space. Multiple tests thus demonstrate how the optimization algorithms behave with different starting configurations which is useful for analyzing the variability in the quality of solutions produced by Auto-CaseRec.

As Auto-CaseRec requires a single evaluation metric to be used as the loss function, the *RMSE* metric was used for the rating prediction experiments and the *MAP@N*[1] metric was used for the item recommendation experiments, where N was set equal to 10.

For each test, information from all search iterations such as computation time, the loss experienced and configuration used, was stored to a CSV (Comma-delimited value) file on the local drive, to be analyzed later.

4.1.2 Datasets

The experiments were conducted across two datasets: the MovieLens-100K data set and the Last.fm data set. This section describes the features of both data sets.

Movielens-100K

The MovieLens 100-K data set was released in 1998 by GroupLens[2], a research lab in the Department of Computer Science and Engineering at the University of Minnesota. The data set is a collection of 100,000 movie ratings in the form of integers from 1-5, given to 1,682 movies by 943 users. Each user in the data set has rated a minimum of 20 movies. The data set's density is 6.304%. The density is the percentage of known feedback values amongst all possible feedback values. It is a stable data set, i.e., it will not be updated and can, therefore, serve as a benchmark data set for research purposes. The movie ratings are a form of explicit feedback given by the users and so, rating prediction algorithms can be directly applied to the Movielens-100K data set. For using the data set in the implicit feedback scenario, the ratings are used by the algorithms with the assumption that the ratings are a form of implicit positive

[1]As the *MAP@N* metric is a positive indicator of performance and the search process inherently performs minimization, the loss function actually used was $-MAP@N$.

[2]`https://grouplens.org/`

feedback, i.e., they are positive representatives of user preferences. In addition to the rating data, the MovieLens-100K data set also contains metadata for users (age, gender, etc.) and movies (genres), but it was not used for our experiments.

Last.fm

The Last.fm data set was released in 2011 in the framework of the 2nd International Workshop on Information Heterogeneity and Fusion in Recommender Systems[34]. The data set contains music artist listening information in the form of 92,834 listening-counts for a set of 1892 users and 17,632 artists, from the Last.fm online music system[3]. The data set's density is 0.278%, which makes it more than 20 times sparser than the MovieLens-100K data set. The listening-counts are a form of implicit feedback given by the user as they do not directly convey the user preferences but can be assumed to be representative of positive user feedback. The data set is used in the explicit feedback scenario by normalizing the listening-counts to real numbers between 1-5 and assuming that higher listening counts indicate higher user preference for the related artist. The dataset also contains metadata like user-user friend relations and artist tags, but these have not been used in our experiments.

4.1.3 Baselines

In research settings, the baselines are the benchmark systems against whose performance, the performance of novel systems is evaluated. The baseline methods should usually be state-of-the-art in a particular research domain or the best-performing methods. Comparing the performance of a novel system against such baselines as described above is crucial to establish a robust estimate of the improvement or deterioration of the state-of-the-art systems that the novel system brings.

Auto-CaseRec aims to aid non-expert users of recommender systems to research scientists alike. It is safe to assume that all users will not have enough prior knowledge to intuitively choose the best performing recommendation algorithm and optimal parameters for their recommendation problem. A natural approach such users could take is to test the different recommendation algorithms applicable in their recommendation scenario, evaluate them with cross-validation, and choose the best performing algorithm as their preferred solution. Keeping

[3]http://www.lastfm.com

this in mind, the baseline chosen for the evaluation of Auto-CaseRec in a given recommendation scenario is the recommendation algorithm that produces the lowest cross-validation loss when executed with the default, unmodified hyperparameters.

Another baseline that was considered is Grid Search. Grid Search is a model-free algorithm that performs an exhaustive search over a *grid* of hyper-parameter values. It takes as input only a *discrete* set of numbers as the search space for each hyper-parameter. Grid Search evaluates every possible combination of hyper-parameter values in the search space and returns the best performing configuration. For N hyper-parameters with K discrete values for each, the total number of evaluations in Grid Search become K^N. For example, 3 hyper-parameters with search-spaces of 20 discrete values each would make the total number of Grid Search evaluations equal to $20^3 = 8000$. Grid Search suffers from what is known as the *curse of dimensionality* as the number of evaluations and hence the computation time exponentially increases with the number of hyper-parameters. In high dimensional and categorical search spaces with continuous ranges of hyper-parameters, Grid Search thus becomes a highly inappropriate and inefficient tool. The search-space used by Auto-CaseRec for the rating prediction scenario (Figure 3.2) has 15 hyper-parameters, some of which take hundreds of discrete values. Grid Search over such a search space would require function evaluations in the order of 10000s as compared to the 50 evaluations we allow in our experiments for Random Search and TPE. In [15], it has been shown that Grid Search performs significantly worse than 8 other HPO approaches including Random Search for optimizing three hyper-parameters of a Matrix Factorization algorithm with a space of 20 discrete values for each. Another major drawback of Grid Search is that for a maximum of B function evaluations in a search space of N hyper-parameters, the maximum number of values evaluated for a given hyper-parameter is $B^{1/N}$ as compared to B values evaluated by other HPO approaches like Random Search and TPE. In environments where certain hyper-parameters affect the performance of the system more than certain others, which is generally the case, Grid Search will inevitably be inefficient [3, 4]. This is demonstrated in Figure 4.1. Grid Search is able to explore just 3 values for the important parameter while Random Search evaluates 9 different values in 9 evaluations and thus, finds the minima of the function. Due to the theoretical inefficiency of Grid Search and the lack of computation resources, we could not implement Grid Search as a baseline in our research. However, it seemed appropriate to present this analysis as Grid Search is a standard method for optimizing hyper-parameters in the absence of advanced techniques.

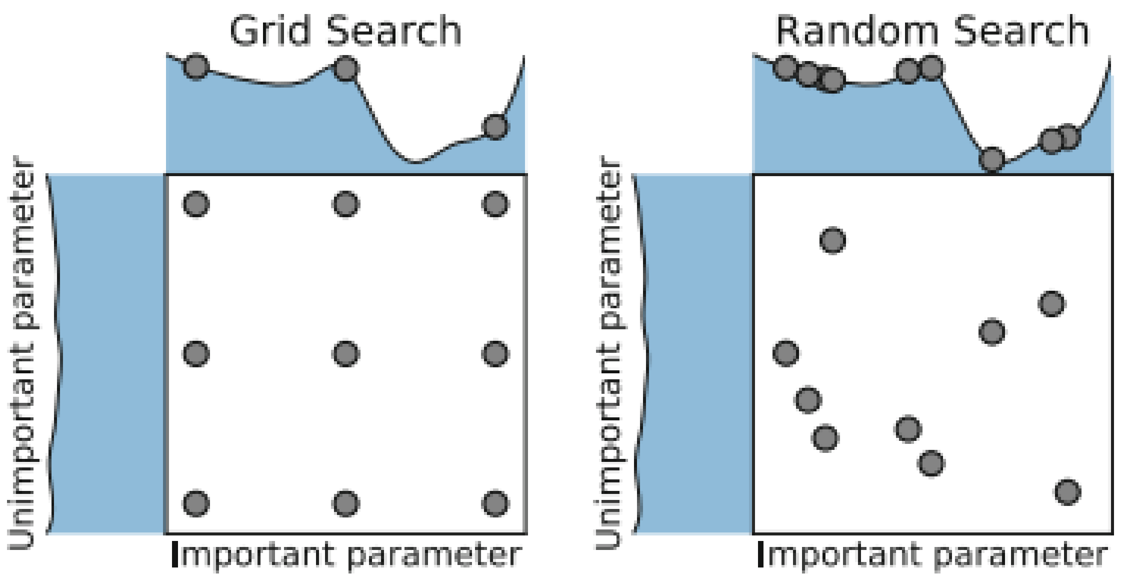


Figure 4.1: Comparison of grid search and random search for minimizing a function with one important and one unimportant parameter. Figure taken from Hutter et. al [3] and based on Figure 1 of Bergstra and Bengio [4]

4.1.4 Software and Hardware

Python Programming Language

Python is a high-level programming language that supports dynamic programming and object-oriented functionality. Python is known for its high code readability and the support of a myriad of libraries that provide all kinds of functionality. The libraries used for processing the experimental results saved on the local drive and visualizing them very the Pandas library and the matplotlib library respectively.

Pandas [77, 78] is a powerful python library that provides extensive data-analysis tools and data structures for storing and manipulating data.

Matplotlib [79] is a python library used for data visualization. Data visualization is the graphical representation of data using formats like charts and graphs. It enables users to see the historical trends in the data, patterns, and anomalies. Matplotlib was used to make the bar-plots and line-plots that will be shown with the results.

Jupyter Notebook

The Jupyter Notebook[4] application is a client-server-based application provided by Project Jupyter that enables developers to develop code in a web-browser based environment. A Jupyter notebook is a document-style coding script provided by the Jupyter Notebook application inside which users can run code in modular *cells* and simultaneously visualize graphs and contents of variables along with the output of the code. The cell structure allows for easy compartmentalized execution and debugging of code.

The preliminary testing of Auto-CaseRec was done using Jupyter Notebooks. Jupyter Notebooks run on the local machine and use local computing resources. The Item Recommendation experiment for the Last.fm data set was conducted using Jupyter Notebooks on the local machine, i.e., tests for both Auto-CaseRec as well as the baselines were conducted on the local machine. The hardware of the local Desktop machine included 12 GB RAM with an Intel Core i5-760 processor.

Google Colab

Google Colab[5] provides *colab notebooks*, that are similar to Jupyter Notebooks in their modular cell-based programming interface. With colab notebooks, anyone with a web browser can use Google's cloud computing capabilities to execute programs. The experiments for both the scenarios on the MovieLens-100K data set as well as the Rating Prediction experiment on the Last.fm data set was conducted on a Google Colab notebook with 25GB RAM and 100GB disk memory. Computation on Google Colab proved to be much faster than the local machine.

4.2 Results

In this section, experimental results for the evaluation of the Auto-CaseRec framework are presented. As described before, experiments are conducted for both the item recommendation and rating prediction scenarios. As both scenarios contain different evaluation metrics, their results are presented in different sub-sections, and within each sub-section, results for the two data sets: MovieLens 100K and Last.fm are given. Following the results, a detailed

[4]`https://jupyter.org/index.html`
[5]`https://colab.research.google.com/`

analysis is presented in the form of a discussion. The results are presented using 3 types of graphics: tables, line-plots, and bar-plots, and each is used once for each data set in each recommendation scenario. A brief overview of the notations used in the graphics is presented below to aid the reader in understanding them.

Tables

Tables display quantitative information about the tests in each experiment. The fields in the tables are to be interpreted in the following manner: $Loss_{ACR}$ is the best loss achieved by Auto-CaseRec, $Loss_{base}$ is the best loss achieved by the recommendation algorithms with default parameters (baselines), $Algo_{ACR}$ is the algorithm found by Auto-CaseRec that produced the best loss, t_{ACR} is the time taken by Auto-CaseRec to complete 50 search iterations, t_{base} is the cumulative time taken to train and evaluate the baseline algorithms, $Iter_{best}$ is the iteration in which the best configuration was found by Auto-CaseRec and finally, $t_{best-iter}$ is the time taken by Auto-CaseRec to find the best configuration. The percentage improvement (if any) is also shown in curly brackets alongside the values in the $Loss_{ACR}$ field. Also, the ★ symbol next to a value in the $Algo_{ACR}$ field means that the best algorithm found by Auto-CaseRec was different from the best performing baseline algorithm. Note: We omit the best performing baseline from the tables due to lack of space and request the reader to refer to the legends of the result plots for the names.

Line-Plots

The line-plots visualize the *best loss* achieved by Auto-CaseRec at the i^{th} iteration for all the 5 tests in a given experiment. Horizontal black dashed lines in the plots show the loss achieved by the best performing baseline algorithm. Vertical dashed lines represent the first iteration at which Auto-CaseRec outperforms the baselines. A color-coded vertical line is shown for each test in a given experiment.

Bar-Plots

The bar-plots visualize the run-times for different stages of each of the 5 tests, specifically, the time at which Auto-CaseRec first outperforms the best baseline, the time at which Auto-CaseRec achieves the best loss for that test and the time at which Auto-CaseRec completes the test. The notations for these are *t-good*, *t-best* and *t-full*. The cumulative time taken to

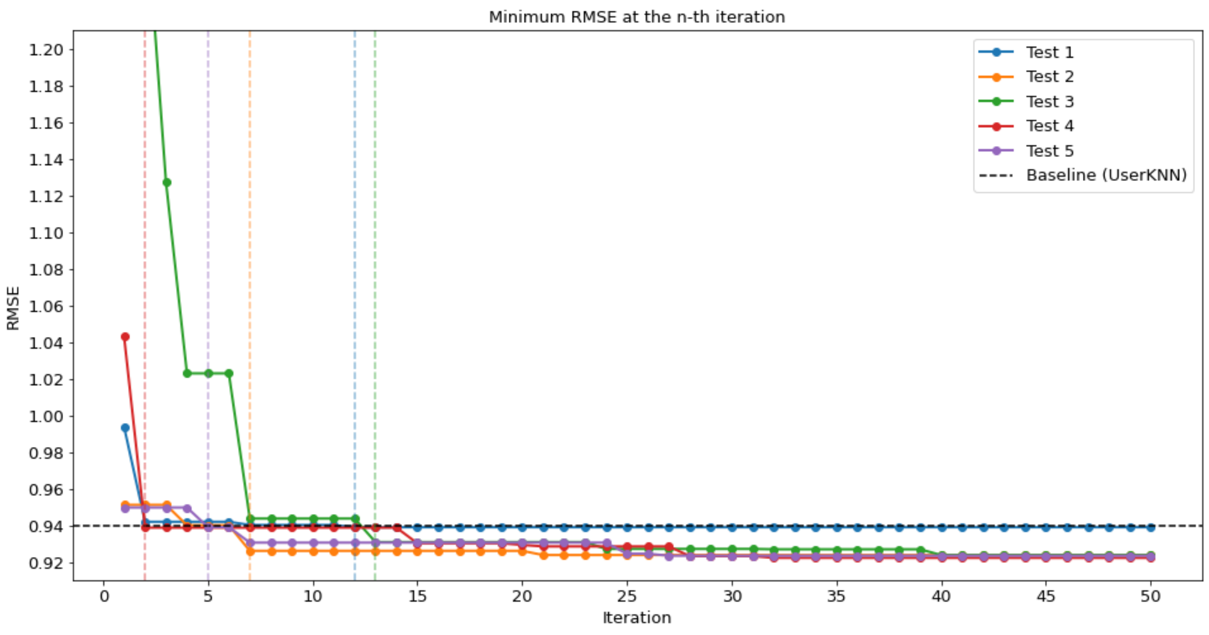


Figure 4.2: Convergence of Auto-CaseRec for the MovieLens 100K data set in the Rating Prediction scenario

train and evaluate the baseline algorithms is shown as a black dashed line and is represented by *t-base*. The number of iterations taken to achieve a given stage is shown at the top of the respective bars in the plot.

4.2.1 Rating Prediction Scenario

For the rating prediction scenario, the evaluation metric of choice was the RMSE. For both the data sets, 5 tests were conducted in which Auto-CaseRec was allowed a total of 50 iterations in each to find the best algorithm configuration using the Tree Parzen Estimators (TPE) optimization algorithm.

Table 4.1: MovieLens 100K Rating Prediction Results

Test	$Loss_{ACR}$	$Loss_{base}$	$Algo_{ACR}$	t_{ACR}	t_{base}	$Iter_{best}$	$t_{best-iter}$
1	0.9388 (+0.11%)	0.9399	UserKNN	00:24:31	00:03:40	12th	00:08:00
2	0.9237 (+1.72%)	0.9399	Matrix Factorization★	00:29:43	00:03:40	21st	00:10:33
3	0.9236 (+1.72%)	0.9399	Matrix Factorization★	00:30:27	00:03:40	40th	00:24:29
4	0.9221 (+1.88%)	0.9399	Matrix Factorization★	00:35:23	00:03:40	32nd	00:24:12
5	0.9233 (+1.67%)	0.9399	Matrix Factorization★	00:33:28	00:03:40	27th	00:19:25

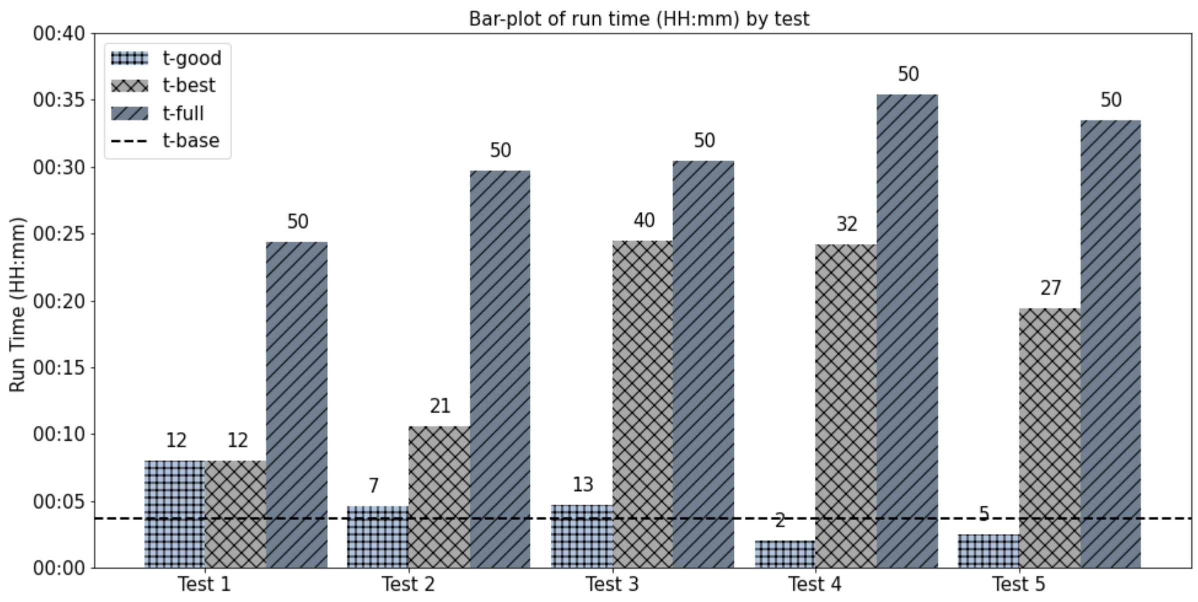


Figure 4.3: Run-time analysis for the MovieLens 100K data set in the Rating Prediction scenario

MovieLens 100K

The results achieved by Auto-CaseRec (TPE) for the rating prediction scenario on the MovieLens 100K data set are shown in Figure 4.2, Figure 4.3 and Table 4.1. As shown in Table 4.1, Auto-CaseRec (TPE) outperforms the best performing baseline algorithm UserKNN in all the 5 tests, with a maximum improvement of 1.88% and mean improvement of 1.42% in RMSE value. In 4 out of 5 tests, the best algorithm produced by Auto-CaseRec (Matrix Factorization) is different from the best performing baseline (UserKNN). It is also worth noting that Matrix Factorization as a baseline, i.e., with default hyper-parameters, produced a loss of 0.9695. Auto-CaseRec on the other hand is able to find hyper-parameter configurations that improve the performance of Matrix Factorization by 4.72% (RMSE: 0.9237) in Test2, 4.73% (0.9236) in Test3, 4.89% (0.9221) in Test4 and 4.76% (0.9233) in Test5. Auto-CaseRec also finds a better hyper-parameter configuration for the UserKNN algorithm in Test1 that results in a minor improvement of 0.11% in RMSE.

As shown in the Table 4.1, the best loss is achieved by Auto-CaseRec 2 out of 5 times in the first half of the tests (Test 1 and 2), and 3 times in the second half of the tests for Test 3, 4 and 5. From the same plot, it is observed that Auto-CaseRec (TPE) outperforms the best performing baseline (UserKNN) after a maximum of 13 iterations and a minimum of 2 iterations in all 5 tests. Further iterations find even better-performing configurations in all tests except Test 1 (Figure 4.3). The total computation time for Auto-CaseRec's tests is between 6 to 10 times that of the cumulative computation time for the baseline algorithms. Figure 4.3 also shows that *t-good* is less than *t-base* for Tests 4 and 5, almost equal to *t-base* for Tests 2 and 3, and approximately double of *t-base* for Test 1.

Last.fm

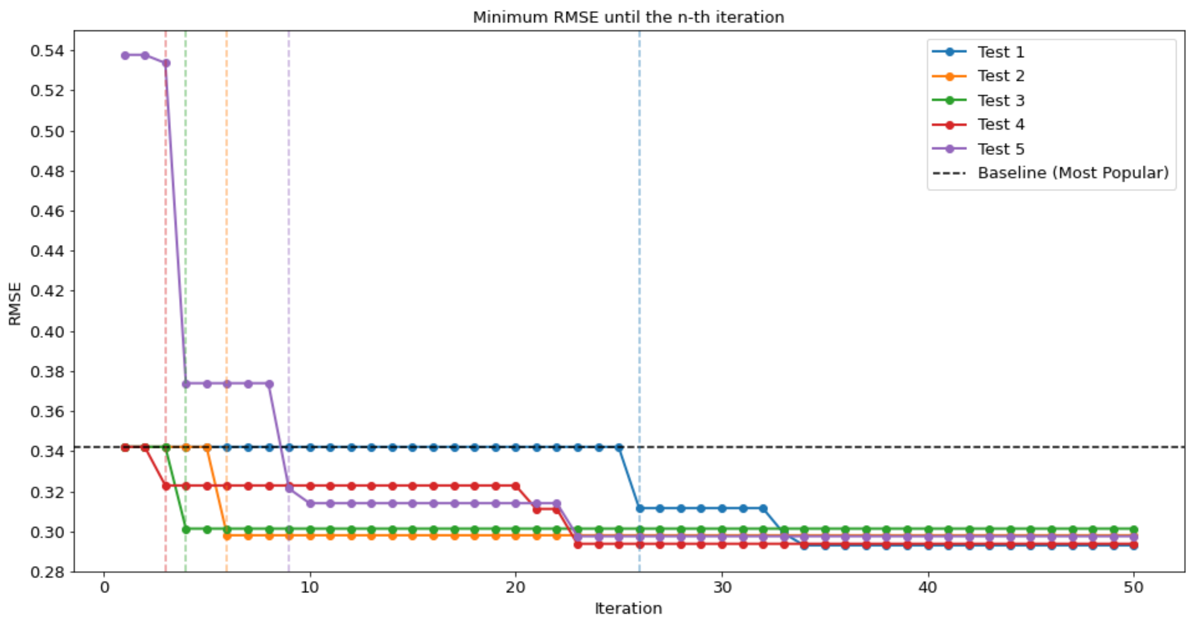


Figure 4.4: Convergence of Auto-CaseRec for the Last.fm data set in the Rating Prediction scenario

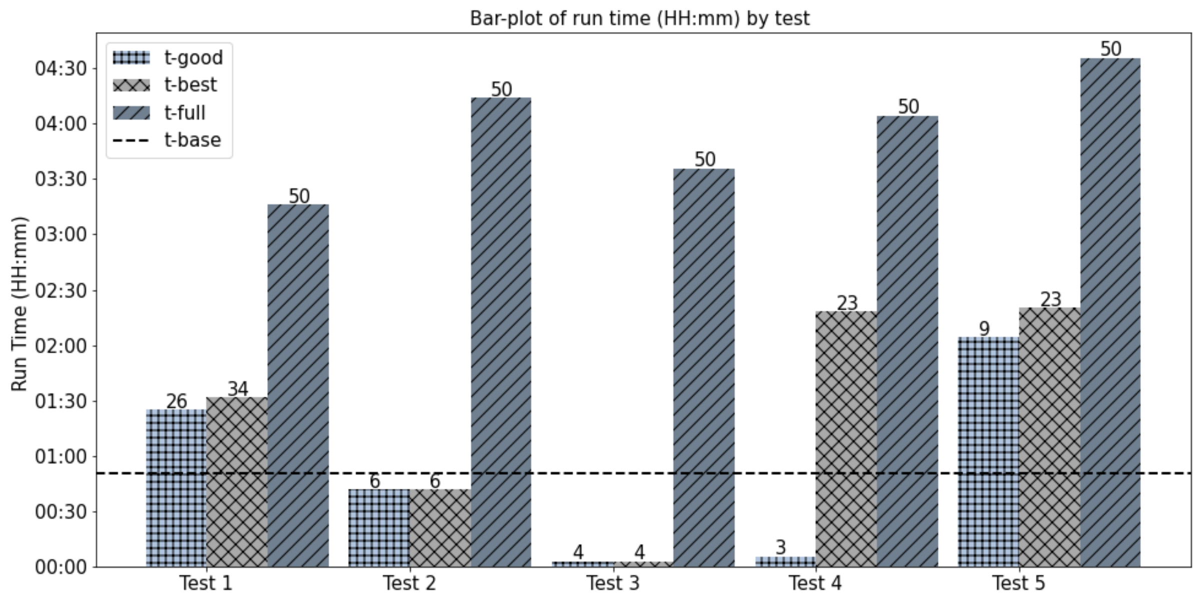


Figure 4.5: Run-time analysis for the Last.fm data set in the Rating Prediction scenario

The results achieved by Auto-CaseRec (TPE) for the rating prediction on the Last.fm data set are shown in Figure 4.4 and Table 4.2. Auto-CaseRec (TPE) outperforms the best performing baseline (Most Popular) in all the 5 tests with a maximum improvement of 14.36% and a mean improvement of 13.26% in RMSE. Table 4.2 shows that in all 5 tests, the best performing algorithm found by Auto-CaseRec (Matrix Factorization) is different from the best performing baseline (Most Popular). With default hyper-parameters, the Matrix Factorization baseline produced a cross-validation RMSE of 0.3464 whereas with the best hyper-parameters found by Auto-CaseRec, the Matrix Factorization algorithm showed an improvement of 15.41% (RMSE: 0.2930) in Test1, 13.97% (0.2980) in Test2, 13.01% (0.3013) in Test3, 15.18% (0.2938) in Test4 and 14.08% (0.2976) in Test5.

Table 4.2 shows that Auto-CaseRec achieves the best loss 4 out of 5 times in the first half of the test and once in the second half of the test for Test 1. Figure 4.5 shows that Auto-CaseRec outperforms the best performing baseline (Most Popular) within 10 iterations in 4 out of 5 tests, with the minimum being 3 iterations in Test 4 and maximum being 26 iterations in Test 1. The total computation time for Auto-CaseRec is between 4 to 6 times that of the cumulative time taken to train and evaluate the baseline algorithms. Figure 4.5 also shows that *t-good* is less than *t-base* for 3 out of 5 tests and greater than *t-base* for Test 1 and Test 5.

4.2.2 Item Recommendation Scenario

For the Item Recommendation scenario, the evaluation metric of choice was the MAP@10 (Mean Average Precision at Rank 10). The MAP@10 is representative of the average quality of the top 10 recommendations produced for all the users of a data set. For both the MovieLens 100K and Last.fm data sets, 5 tests were conducted in which Auto-CaseRec was allowed 50 iterations. The results of the tests have been visualized in the same way as the Rating

Table 4.2: Last.fm Rating Prediction Results

Test	$Loss_{ACR}$	$Loss_{base}$	$Algo_{ACR}$	t_{ACR}	t_{base}	$Iter_{best}$	$t_{best-iter}$
1	0.2930 (+14.36%)	0.3421	Matrix Factorization★	03:16:15	00:50:20	34th	01:31:30
2	0.2980 (+12.89%)	0.3421	Matrix Factorization★	04:13:50	00:50:20	6th	00:41:39
3	0.3013 (+11.91%)	0.3421	Matrix Factorization★	03:35:51	00:50:20	4th	00:02:40
4	0.2938 (+14.12%)	0.3421	Matrix Factorization★	04:04:25	00:50:20	23rd	02:18:06
5	0.2976 (+13.02%)	0.3421	Matrix Factorization★	04:35:36	00:50:20	23rd	02:20:08

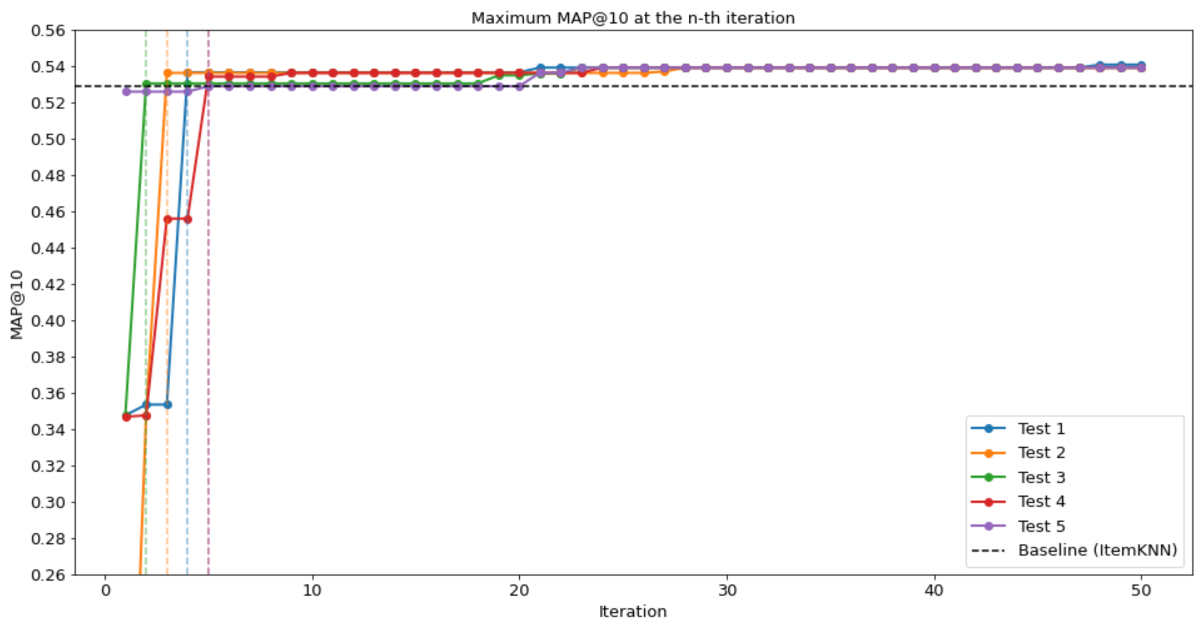


Figure 4.6: Convergence of Auto-CaseRec for the MovieLens 100K data set in the Item Recommendation scenario

Table 4.3: MovieLens 100K Item Recommendation Results

Test	$Loss_{ACR}$	$Loss_{base}$	$Algo_{ACR}$	t_{ACR}	t_{base}	$Iter_{best}$	$t_{best-iter}$
1	0.5408 (+2.28%)	0.5288	BPRMF★	01:31:29	00:12:54	48th	01:16:24
2	0.5392 (+1.97%)	0.5288	UserKNN★	01:09:21	00:12:54	28th	00:39:53
3	0.5392 (+1.97%)	0.5288	UserKNN★	01:11:29	00:12:54	23rd	00:35:49
4	0.5392 (+1.97%)	0.5288	UserKNN★	01:22:31	00:12:54	24th	00:49:29
5	0.5392 (+1.97%)	0.5288	UserKNN★	01:11:22	00:12:54	23rd	00:38:44

Prediction experiments, except that the *maximum MAP@10* is recorded for the baselines and Auto-CaseRec. This is because *MAP* is a positive indicator of performance.

MovieLens 100K

The results achieved by Auto-CaseRec (TPE) for the MovieLens 100K data set in the Item Recommendation scenario are shown in Figure 4.6 and Table 4.3. It is observed that Auto-CaseRec (TPE) outperforms the best baseline algorithm (ItemKNN, MAP@10: 0.5288) in all 5 tests with a maximum improvement of 2.28% (MAP@10: 0.5408) and mean improvement of 2.02% (mean MAP@10: 0.5395). From Table 4.3 we also observe that the best algorithms found by Auto-CaseRec (BPRMF, UserKNN) are different from the best performing baseline algorithm (ItemKNN) for all 5 tests. The baseline MAP@10 of the BPRMF and UserKNN algorithms (with default hyper-parameters) was found to be 0.5002 and 0.5252 respectively,

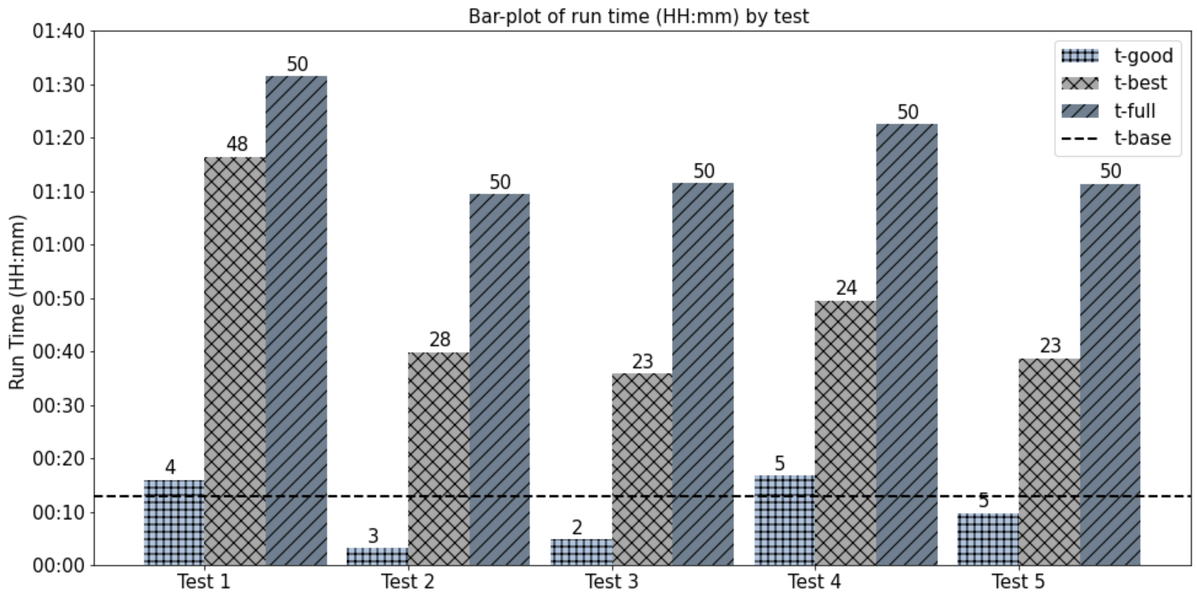


Figure 4.7: Run-time analysis for the MovieLens 100K data set in the Item Recommendation scenario

Table 4.4: Last.fm Item Recommendation Results

Test	$Loss_{ACR}$	$Loss_{base}$	$Algo_{ACR}$	t_{ACR}	t_{base}	$Iter_{best}$	$t_{best-iter}$
1	0.3654 (+0.6%)	0.3633	UserKNN	14:18:39	04:46:00	23rd	05:48:58
2	0.3656 (+0.63%)	0.3633	UserKNN	20:55:03	04:46:00	35th	14:40:04
3	0.3656 (+0.63%)	0.3633	UserKNN	21:08:26	04:46:00	46th	19:19:34
4	0.372313 (+2.48%)	0.3633	UserKNN	18:15:57	04:46:00	45th	15:57:41
5	0.3656 (+0.63%)	0.3633	UserKNN	19:26:20	04:46:00	22nd	11:19:21

whereas, with the hyper-parameter configuration found by Auto-CaseRec, BPRMF performs 8.11% better (MAP@10: 0.5408) as seen in Test 1 and UserKNN performs 2.66% better (MAP@10:0.5392) as seen in Tests 2-5.

Table 4.3 shows that for Auto-CaseRec achieves the best loss in the first half of the test for 3 out of 5 tests and in the second half of the test for Tests 1 and 2. Figure 4.7 shows that Auto-CaseRec outperforms the best baseline algorithm in a maximum of 5 iterations as seen in Tests 4 and 5, and a minimum of 2 iterations as seen in Test 3. The overall computation time for Auto-CaseRec is observed to vary between 5 to 8 times that of the cumulative computation time of the baseline algorithms. Figure 4.7 also shows that *t-good* is less than *t-base* for 3 out of 5 tests and marginally greater than *t-base* for Test 1 and Test 4.

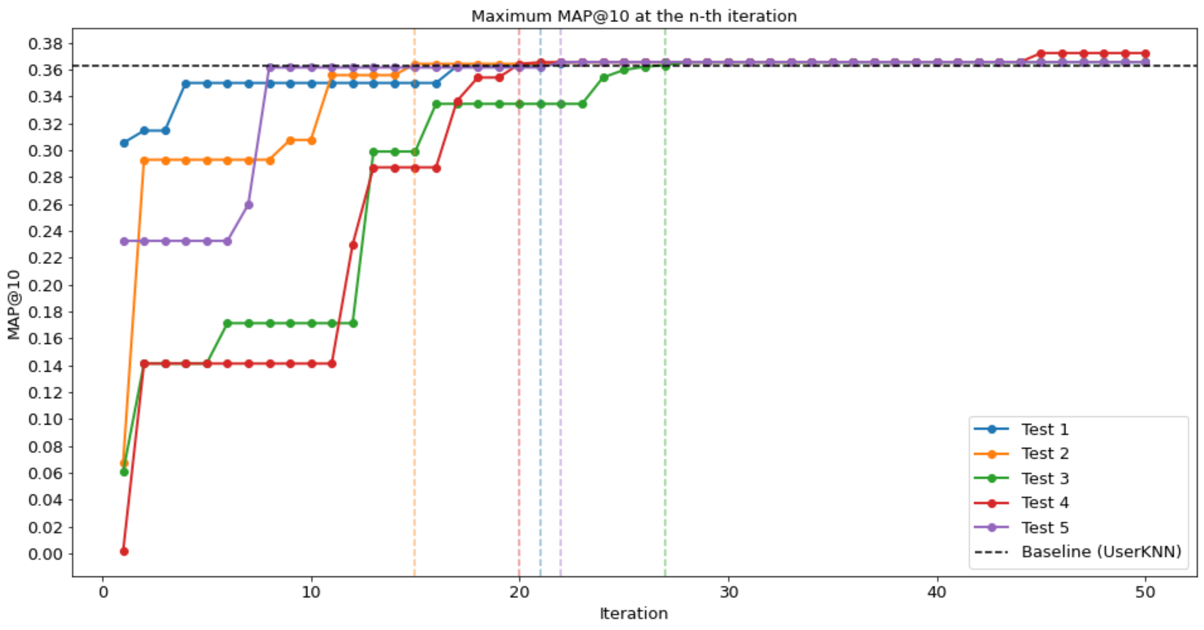


Figure 4.8: Convergence of Auto-CaseRec for the Last.fm data set in the Item Recommendation scenario

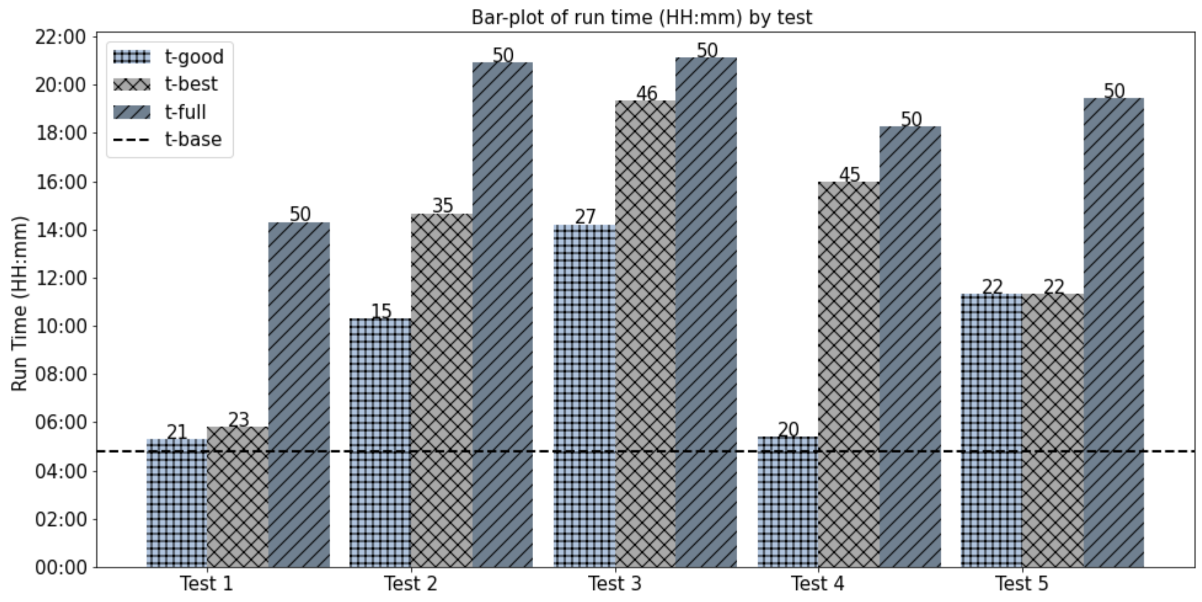


Figure 4.9: Run-time analysis for the Last.fm data set in the Item Recommendation scenario

Last.fm

The results achieved by Auto-CaseRec (TPE) for the Last.fm data set in the Item Recommendation scenario are shown in Figure 4.8 and Table 4.4. It is observed that Auto-CaseRec (TPE) outperforms the best baseline algorithm (UserKNN, MAP@10: 0.3633) in all 5 tests, with a maximum improvement of 2.48% (MAP@10: 0.3723) in Test 4 and improvements of around 0.6% (MAP@10: 0.3654 for Test 1 and 0.3656 for Tests 2,3 and 5) in the other tests. In this experiment, the best algorithms found by Auto-CaseRec (Table 4.4) were the same as the best performing baseline algorithm (UserKNN) and so, the improvement in MAP@10 across all 5 tests is solely attributed to better performing hyper-parameter configurations for the UserKNN algorithm.

Table 4.4 shows that Auto-CaseRec achieved the best loss in the first half of the test for 2 out of 5 tests and in the second half of the test for Tests 2, 3 and 4. As shown in Figure 4.9, Auto-CaseRec outperforms the best baseline algorithm (UserKNN) in a maximum of 27 iterations in Test 3 and a minimum of 15 iterations in Test 2. The overall computation time for Auto-CaseRec is significantly more than the previous experiments (Table 4.4). The computation time for Auto-CaseRec varies between 3-5 times that of the cumulative computation time for the baseline algorithms. It is also observed that the computation time of Test 1 (14:18:39) is significantly more than the other 4 tests (18:15:57 - 21:08:26). Figure 4.9 also shows that *t-good* is always greater than *t-base*, with significant differences in Tests 2, 3 and 5 and marginal differences in Tests 1 and 4.

4.3 Discussion

This section of the report presents a detailed analysis of the results shown in the previous section. We start with a brief recap of the experimental setup. The Auto-CaseRec library was evaluated on two popular Recommender System data sets: MovieLens 100K that contains 100,000 ratings given by 943 users to 1,682 movies, and Last.fm that contains 92,834 listen-counts for 17,632 artists by 1,892 users. For both the data sets, experiments consisting of 5 tests each were conducted for the two recommendation scenarios: Rating Prediction and Item Recommendation. Auto-CaseRec was executed for 50 iterations in each test with the tree Parzen Estimators (TPE) optimization algorithm. The recommendation algorithms instantiated with their default unmodified hyper-parameters were taken as the baselines. The evaluation metrics used were the RMSE for Rating Prediction and MAP@10 for Item Recommendation, and the evaluation for each iteration of Auto-CaseRec and the baselines was performed using 5-Fold Cross-Validation. Auto-CaseRec's performance was then compared against the best performing baseline.

4.3.1 Performance Evaluation

The results of the 4 experiments (two recommendation scenarios and two data sets) have been collated into a table for the benefit of the reader (Table 4.5). The table shows the best algorithms produced by Auto-CaseRec out of the 5 test results for each experiment and the best, mean, and worst loss values achieved in each experiment. It also shows the percentage improvement alongside the absolute loss values and shows the loss value for the best performing baseline in each experiment in a separate column.

Table 4.5: Auto-CaseRec loss comparison

Experiment	$Algo_{ACR}^{best}$	$Loss_{ACR}^{best}$	$Loss_{ACR}^{mean}$	$Loss_{ACR}^{worst}$	$Loss_{base}$
RP, ML-100K	MF★	0.9221 (+1.88%)	0.9263 (+1.44%)	0.9388 (+0.11%)	0.9399
RP, Last.fm	MF★	0.2930 (+14.36%)	0.2967 (+13.25%)	0.3013 (+11.91%)	0.3421
IR, ML-100K	BPRMF★	0.5408 (+2.28%)	0.5395 (+2.02%)	0.5392 (+1.97%)	0.5288
IR, Last.fm	UserKNN	0.3723 (+2.48%)	0.3669 (+0.99%)	0.3654 (+0.6%)	0.3633

From Table 4.5, we observe that Auto-CaseRec outperformed the baseline algorithms in all 5 tests in 4 out of 4 experiments. In the Rating Prediction (RP) experiments, for the Last.fm data set, Auto-CaseRec consistently produced large improvements (average improvement of 13.35% in RMSE), while for the MovieLens-100K data set, the improvements were moderately significant (average of 1.44%). For the Item Recommendation (IR) experiments, the improvements were moderately significant for both the ML-100K data set (average improvement of 2.02%) and the Last.fm data set (average of 0.99%) Also, in 3/4 experiments, the best performing algorithm produced by Auto-CaseRec was found to be different from the best performing baseline algorithm. In 2/4 experiments, the worst algorithm produced by Auto-CaseRec was also different than the best performing baseline algorithm (Table 4.2 and Table 4.3). We can make the following inferences from these results:

1. Finding the best algorithm and a set of optimal hyper-parameters using Bayesian Op-

timization (TPE) consistently increases recommendation performance as compared to the performance of the best individual algorithm with default hyper-parameters, while the degree of improvement can vary according to the data set and recommendation scenario.

2. The actual best algorithm(s) for a data set is likely to be different from the best baseline algorithm.

3. Hyper-parameter tuning can be used to find the limits of the separate algorithms and thereby, produce an accurate ranking of algorithms for a given recommendation task.

Table 4.6: Effects of Hyper-Parameter Tuning

Scenario	*Dataset*	*Algorithm*	$Loss_{ACR}^{best}$	$Loss_{ACR}^{mean}$	$Loss_{default}$
RP	ML-100K	MF★	0.9221 (+4.89%)	0.9232 (+4.78%)	0.9695
RP	ML-100K	UserKNN	0.9388 (+0.11%)	0.9388 (+0.11%)	0.9399
RP	Last.fm	MF★	0.2930 (+15.41%)	0.2967 (+14.33%)	0.3464
IR	ML-100K	BPRMF★	0.5408 (+8.11%)	0.5408 (+8.11%)	0.5002
IR	ML-100K	UserKNN★	0.5392 (+2.66%)	0.5392 (+2.66%)	0.5252
IR	Last.fm	UserKNN	0.3723 (+2.48%)	0.3669 (+0.99%)	0.3633

We also analyze the effects of tuning the hyper-parameters of the algorithms. The results for the same are shown in Table 4.6. In this table, $Loss_{ACR}^{best}$ refers to the best loss achieved amongst the tests that produced the associated algorithm as a result and $Loss_{ACR}^{mean}$ is the mean of all the losses achieved by such tests. $Loss_{default}$ refers to the loss achieved by the associated algorithm when executed as a baseline, i.e., with default hyper-parameters. The motive is to observe the improvement in performance (if any) of individual algorithms with tuned hyper-parameters as compared to their performance with the default hyper-parameters. We only compare the best algorithms produced by the tests in a given experiment. In 2/4 experiments, the test results contained two different algorithms so we perform the analysis for both.

From Table 4.6, we observe that through hyper-parameter tuning, Auto-CaseRec always increases the performance of the analysed algorithms. In some cases, the effect of Hyper-Parameter tuning is very pronounced. For example, in the Rating Prediction scenario for the Last.fm data set, the best baseline was the Most Popular algorithm and Matrix Factorization

(MF) was ranked second. With hyper-parameter tuning, Auto-CaseRec was able to improve the RMSE of MF by a mean value of 14.33%. Another example is the Item Recommendation scenario for the MovieLens data set, in which, the best baseline was UserKNN, followed by ItemKNN and then BPRMF. With hyper-parameter tuning, Auto-CaseRec improved the MAP@10 of BPRMF by 8.11% and it was produced as the best performing algorithm.

From these results, we can infer that evaluating the recommendation algorithms with their default configurations individually for any of the two scenarios can be inconclusive with regards to finding the best available algorithm for a given data set. This is because without exploring the hyper-parameter space, it is difficult to know the performance limits of any given algorithm and thus the ranking of algorithms produced by such experimentation cannot be guaranteed to be accurate. Hence, if a user decides to only tune the recommendation algorithm that produced the best performance with default hyper-parameters, chances are that his solution would still be far from the best possible solution that can be achieved in his computational budget. A user may do this if he is bounded by a limited computational budget or, relies on manual tuning or Grid Search due to lack of advanced Hyper-parameter optimization (HPO) algorithms (which is generally the case in the Recommender System tool-kits). In such a scenario, a tool like Auto-CaseRec that automates optimization with advanced HPO techniques would greatly help the user by producing good algorithms and associated hyper-parameter configurations within a limited computational budget. Even in scenarios where the user has a reasonable computational budget, Auto-CaseRec would still be highly efficient than manual tuning or Grid Search due to the size of the search space involved in Combined Algorithm Selection and Hyper-Parameter Optimization.

4.3.2 Run-Time Analysis

The experimental results show that the run-time of the experiments varies greatly with the recommendation scenario as well as the data set.

It is observed that the Item Recommendation experiments consume a lot more time than the Rating Prediction experiments. One possible explanation for this is that the worst-case time complexity of calculating the MAP@N metric is $O(|\mathcal{U}| * N^2)$. This is because the calculation involves N computations to calculate the *precision@N* at each N, followed by a maximum of N computations for calculating the *AP@N* and this is performed for all the users $u \in \mathcal{U}$.

In contrast, the time complexity for calculating the RMSE metric in the Rating Prediction scenario is $O(|D_{test}|)$ as one computation is performed for each user-item pair in the test set. In our experimentation, $|\mathcal{U}|$ is equal to 943 and 1892 for the MovieLens-100K and Last.fm data sets respectively. The value of N is 10 and the $|D_{test}|$ takes a maximum value of 20000. We can observe that the evaluation time taken in the Item Recommendation scenario is conclusively bound to be much more than the evaluation time taken in the Rating Prediction scenario.

It is also observed that the experiments involving the Last.fm data set took considerably longer than those involving the MovieLens 100K data set. This is because of the recommendation algorithms scale according to the number of users and the number of items. While algorithms like Matrix Factorization scale well, the Neighbourhood-based methods UserKNN and ItemKNN scale very badly [80, 81, 82]. The training time complexity for the UserKNN and ItemKNN algorithms is $O(|\mathcal{U}|^2 * p)$ and $O(|\mathcal{I}|^2 * q)$ respectively, where, $\mathcal{U}$ and $\mathcal{I}$ are the set of users and items respectively and p and q are the maximum number of ratings per user and the maximum number of ratings per item respectively [82]. The prediction time complexities for both are $O(|\mathcal{I} * k|)$ [82]. The Last.fm data set has 1,892 users and 17,632 artists as compared to the MovieLens-100K data set that contains 943 users and 1,682 movies. As the number of users and items in the Last.fm data set are approximately double and 10 times respectively, as compared to the users and items in the MovieLens-100K data set, the time complexities of the UserKNN and ItemKNN methods are much worse and they significantly increase the run-times. The ItemKNN was found to take the longest times as the number of items is greater than the number of users in the data sets.

Auto-CaseRec was allowed a total of 50 iterations which translates to 50 rounds of algorithm training and evaluation. The run-time of Auto-CaseRec scales linearly with the number of iterations and in the worst case is $O(n_{iter} * t_{worst})$, where t_{worst} is the time required for training and evaluation by the recommendation algorithm with the worst time complexity. The baselines, on the other hand, consisted of 5 algorithms for the Item Recommendation scenario and 6 algorithms for the Rating Prediction scenario. This translates to 5 and 6 rounds for the two scenarios as compared to the 50 rounds for Auto-CaseRec. As a result, the total time taken by Auto-CaseRec is always a few times more than that of computing the baselines.

Even so, it is also observed that the time taken by Auto-CaseRec to outperform the best baseline algorithm is not much. For the Rating Prediction scenario, t_{good} was less than t_{base} for 2 out of 5 and 3 out of 5 tests for the MovieLens-100K and Last.fm data set respectively. For the Item Recommendation scenario, t_{good} was less than t_{base} 3 out of 5 times and 0 out of 5 times for the MovieLens-100K and Last.fm data sets respectively. This translates to 8 wins out of 20 for Auto-CaseRec. Additionally, in the tests that t_{good} was more than t_{base}, the difference was marginal in 6 out of 12 cases. Combining the two, we can effectively say that in 14 out of 20 cases, the time taken by Auto-CaseRec to outperform the baselines was better than or marginally worse than the cumulative time taken to compute the baselines. Therefore, even within very small computational budgets, Auto-CaseRec can be expected to deliver better algorithm configurations as compared to the default configurations.

Additionally, in 11 out of 20 tests across the 4 experiments, Auto-CaseRec found the best performing configuration in the first half of the test itself, i.e., within 25 iterations. Of the remaining 9 tests, the best performing configurations occurred 5 times in the iteration range 26-35, 2 times in the range 36-45, and 2 times in the range 46-50. We cannot draw hard conclusions about the run-time from this data as the individual iteration run-times greatly vary with the associated algorithms but we can infer that in a majority of the tests performed, the effective computation time taken by Auto-CaseRec to find the best configuration is much less than the overall computation time reported.

4.4 Summary

The Auto-CaseRec library was evaluated on two data sets: the MovieLens-100K data set and the Last.fm data set. Although Auto-CaseRec generally produced improvements on both data sets, the improvements were not always significant. For example, the Item Recommendation experiment for the Last.fm data set produced a mean improvement of 0.99%. In such cases, it can be argued that it is not worth investing computational resources into solving the CASH problem. However, the end-results of hyper-parameter optimization are rarely known and the user can be expected to at least experiment with the different baseline algorithms to find out the best performing algorithm for his scenario. The run-time analysis has shown that the time taken by Auto-CaseRec to outperform the baselines is less than the time taken to evaluate all baselines in some cases (6 out of 20) and only marginally worse in some more (8 out of 20). Therefore, the author recommends using Auto-CaseRec instead of manual algorithm selection when prior knowledge about the algorithm hyper-parameters is limited. It has also been shown that the algorithms found by Auto-CaseRec to produce the best results are often different than the best performing baseline (3 out of 4 experiments). Additionally, the effect of hyper-parameter tuning shows that often, algorithms perform significantly better with hyper-parameter tuning. Therefore, we can say that performing manual algorithm selection without hyper-parameter tuning is unlikely to produce the actual best algorithm for the user's scenario.

The results produced in these experiments are not generalizable to all Recommender System scenarios as it is a broad domain with applications varying greatly in the user-base as well as the items offered. The results presented may also contain an inherent bias in the performance of the baselines (default hyper-parameters). This is possible because the data sets used are gold-standard data sets in the RS domain and are very often used to evaluate systems. Therefore, it is possible that the default hyper-parameters of the Case Recommender library may already be tuned to some extent for the MovieLens-100K or the Last.fm data set. To get a more accurate picture of Auto-CaseRec vs the algorithms with default hyper-parameters, experiments need to be performed with many more RS data sets, both standard and non-standard. Also, evaluating a stronger baseline like Grid Search would make the evaluations more robust. This is left for future work.

5 Conclusion

This chapter provides a conclusion to the research work by presenting a *summary* of the scientific developments made through this research work, discussing the *challenges* faced during the execution of the research tasks outlined in Section 1.3, recognising the *limitations* inherent in the research and finally, presenting the ideas that the author has for the *future work* to further develop the Auto-CaseRec tool.

5.1 Summary

This research work presented a novel automated Recommender System Library, Auto-CaseRec. The research literature in the Recommender Systems community makes two observations through various experimental studies: different recommendation algorithms perform differently in different recommendation scenarios and even with changing data set characteristics in the same scenario and, hyper-parameter optimization of recommendation algorithms improves the performance of recommendation algorithms. To exploit the benefits of both *algorithm selection* and *hyper-parameter optimization*, both problems were reformulated as one single problem, the Combined Algorithm Selection and Hyper-parameter optimization problem (CASH). The Auto-CaseRec library was proposed as the first automated tool for the RS community that uses advanced HPO techniques to solve the CASH problem over a combined hierarchical search space of algorithms and hyper-parameters. Auto-CaseRec uses advanced HPO techniques like Bayesian Optimization to find the "best" algorithm and hyper-parameters for a given recommendation scenario and RS data set. Auto-CaseRec is based on the Case Recommender Python library and supports the optimization of various evaluation metrics in both the Rating Prediction and Item Recommendation recommendation scenarios. The experimental results showed that Auto-CaseRec outperformed the individual recommendation algorithms of Case Recommender with default hyper-parameters consistently across 5 out of 5 tests in both recommendation scenarios for the two gold-standard RS data sets: MovieLens-100K and Last.fm. The author aims to further develop the Auto-CaseRec library and publish it for open-source use very soon. Hopefully, Auto-CaseRec will serve as a useful tool for the RS community and fulfill the ambitions that were stated alongside the research objective in Section 1.3.

5.2 Challenges

This section presents the various technical and analytical challenges faced during the execution of the research tasks that were presented as the means of fulfilling the research objective stated in Section 1.3. Examining the challenges in the context of the research tasks was deemed appropriate by the author as they are fundamentally related to the tasks during which they were encountered in the first place. The tasks and their associated challenges are given below.

- **Task 1: To identify existing open-source RS tool-kits and analyse them in terms of the diversity of recommendation algorithms, diversity of evaluation methods, ease of use and flexibility of operation.**

  The analysis of the RS tool-kits showed that the most appropriate libraries offered unique advantages but had certain disadvantages as well. Initially, the well-known Surprise library was considered for its diverse set of algorithms and Grid Search functionality but it was discovered that Surprise is not under active development since 2019. Next, the Spotlight library was chosen because of its unique set of algorithms and it was carried forward into Task 3. The initial experimentation in Task 3 revealed that Spotlight's in-built data processing methods were inflexible and thus inadequate for the robust evaluation of RS algorithms. Finally, the relatively new Case Recommender library was chosen for use in Auto-CaseRec owing to its diverse set of algorithms, adequate evaluation metrics, and robust cross-validation strategies.

- **Task 2: To identify and perform an experimental analysis of existing open-source Hyper-Parameter Optimization (HPO) packages that provide advanced HPO methods like Bayesian Optimization and allow for the opti-**

mization of arbitrary functions, while also observing the flexibility for defining search spaces, parallelization capabilities and ease of use.

The SMAC v3 package was considered as the first choice as the SMAC algorithm [32] had proven to be a state-of-the-art AutoML method and the was the work-horse behind the Auto-WEKA [21] and Auto-sklearn [1] libraries. However, the tool was limited to the Linux environment and produced bugs during experimentation that prevented the optimization process from terminating. When the bugs couldn't be solved even after considerable efforts and no support could be found on the internet, it was deemed appropriate to use another library, specifically, Hyperopt [25]. The advantages of Hyperopt turned out to be numerous: multiple HPO algorithms, flexible interface for search-spaces, and support for parallelization as well as different Operating Systems.

- **Task 3: To experimentally analyse the feasibility of an Auto-RS tool by using a chosen HPO library to optimize the hyper-parameters of individual recommendation algorithms from a chosen RS tool-kit.**

  The experimentation for this task was performed using Hyperopt and Spotlight and showed improvements across 4 data sets (Book-Crossing, Jester, Last.fm, and MovieLens-100K). However, Spotlight presented limitations and it was decided to use Case Recommender instead. Due to time constraints, the development of Auto-CaseRec had to be initiated based on the results achieved with Spotlight. This presented a certain amount of uncertainty but was overcome by the fast development of an initial prototype of Auto-CaseRec that solved the CASH problem for the rating prediction scenario and produced good results for the MovieLens-100K and Last.fm data sets.

- **Task 4: To construct an Auto-RS tool using functionality from the chosen HPO library and RS tool-kit such that it performs data processing (splitting into cross-validation folds) and optimization of supported evaluation metrics over a hierarchical search space of recommendation algorithms and their associated hyper-parameters in an automated manner with minimal user intervention.**

  During the further development of the Auto-CaseRec tool, a major challenge presented itself. An accident to the author's laptop resulted in the partial loss of the source code and a major loss of written and collected material, and experimental results. An extension was granted to the author by Trinity College to ensure the completion of the research. The author was able to redo the research work, develop the Auto-CaseRec tool, and perform experimentation on another Desktop machine.

- **Task 5: To experimentally verify the benefits of the Auto-RS tool by evaluating it on gold-standard RS data sets.**

  This task presented an analytical as well as logistical challenge. The experiments had to be designed robustly so that the results would be statistically significant while there was also a time constraint. To ensure the robust validation of the Auto-CaseRec tool, the experiments were conducted in sets of 5 tests, with 5-Fold cross-validation in each iteration, and across two data sets. This design of the experiments proved to take up a lot of time and often, the experiments would terminate abruptly due to external factors. Experimentation was simultaneously conducted on Google Colab as well as the Desktop machine to ensure their completion.

5.3 Future Work and Limitations

In this section, an organised view of the limitations associated with this research work and the Auto-CaseRec library is presented, along with future work that is to be undertaken to tackle these limitations. The limitations are as follows.

1. **Run-time**: The Auto-CaseRec library uses Bayesian Optimization (TPE) to find the "best" algorithm configurations for a given data set which is an iterative optimization process. Finding a configuration that produces significant improvements can take much more time than simply manually testing all the default algorithms and may also vary across data sets. When computational budgets are low, users may simply choose to trade-off the improvement in performance for the much less time consumed.

2. **Evaluation**: The evaluation of Auto-CaseRec was performed on only two data sets. Also, the data sets used are standard in the RS community and Case Recommender algorithms can be expected to be pre-tuned for at least one of them. A more robust

evaluation of Auto-CaseRec on data sets from diverse applications is required to form a scientific opinion about its general advantages and disadvantages.

3. **Baseline**: A stronger baseline like Grid Search needs to be considered. Such a baseline would enable a more concrete comparison with Auto-CaseRec in terms of run-time as well as performance.

4. **Scalability**: It was seen that the run-times of Auto-CaseRec were significantly more as the number of Users and Items increased from the MovieLens-100K data set to the Last.fm data sets. These data sets contained 100,000 and 92,000 instances respectively. Much larger data sets are common in the RS research community, for example, the MovieLens 1M, 10M, and 20M data sets, and the BookCrossing data set with around 1M instances. It was found that the reason is that some RS algorithms scale very badly with an increasing number of users and items. This directly translates to scalability issues for Auto-CaseRec that performs multiple algorithm evaluations for all algorithms in the search space.

Future work is aimed at making improvements that counter-act the effects of most of the limitations presented above, although, it is questionable as to whether issues like scalability and run-time can be completely removed. The future work is as follows.

1. **Parallelization**: The Hyperopt library [56] provides a simple parallelization interface based on MongoDB and Apache Spark and can be readily included into Auto-CaseRec without much effort. However, parallelization presents a trade-off between run-time and guided exploration of the search space as more parallel configurations involve more random sampling. Implementing and evaluating Auto-CaseRec with various degrees of parallelization is a crucial part of the future development of Auto-CaseRec.

2. **Pruning**: Pruning techniques can be used to narrow the search space down to configurations that are more promising than others before running the search process. For example, meta-learning can use past experiences to determine promising configurations based on data set features and has been used by Auto-sklearn [1]. Other techniques include bandit-based methods such as Successive Halving [83] and Hyperband [70] that remove configurations that deliver bad results based on their cross-validation perfor-

mance during run-time. A simple strategy called *early-stopping* has already been implemented in Auto-CaseRec but it only terminates bad configurations pre-maturely and does not permanently discard them.

3. **Evaluation**: To establish robust estimates of the advantages and disadvantages of using Auto-CaseRec in terms of both performance and run-time, further experimentation is required on multiple RS data sets. The results from these experiments would also feature in future research paper submissions to research Journals and/ or conferences.

4. **HPO methods**: Several HPO approaches exist in the literature (Section 2) and some methods might produce better performance or better convergence than Hyperopt. Further research is required to add more HPO methods using other libraries or implement them from scratch to complement or even completely replace Hyperopt.

Bibliography

[1] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in neural information processing systems*, pages 2962–2970, 2015.

[2] Arthur da Costa, Eduardo Fressato, Fernando Neto, Marcelo Manzato, and Ricardo Campello. Case recommender: a flexible and extensible python framework for recommender systems. In *Proceedings of the 12th ACM Conference on Recommender Systems*, pages 494–495, 2018.

[3] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated Machine Learning*. Springer, 2019.

[4] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(Feb):281–305, 2012.

[5] Francesco Ricci, Lior Rokach, and Bracha Shapira. Introduction to recommender systems handbook. In *Recommender systems handbook*, pages 1–35. Springer, 2011.

[6] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE transactions on knowledge and data engineering*, 17(6):734–749, 2005.

[7] Joeran Beel, Corinna Breitinger, Stefan Langer, Andreas Lommatzsch, and Bela Gipp. Towards reproducibility in recommender-systems research. *User modeling and user-adapted interaction*, 26(1):69–101, 2016.

[8] Joeran Beel, Stefan Langer, Andreas Nürnberger, and Marcel Genzmehr. The impact of demographics (age and gender) and other user-characteristics on evaluating recommender

systems. In *International Conference on Theory and Practice of Digital Libraries*, pages 396–400. Springer, 2013.

[9] Michael Ekstrand and John Riedl. When recommenders fail: predicting recommender failure for algorithm selection and combination. In *Proceedings of the sixth ACM conference on Recommender systems*, pages 233–236, 2012.

[10] Michael D Ekstrand and Maria Soledad Pera. The demographics of cool. *Poster Proceedings at ACM RecSys. ACM, Como, Italy*, 2017.

[11] Il Im and Alexander Hars. Does a one-size recommendation system fit all? the effectiveness of collaborative filtering based recommendation systems across different domains and search modes. *ACM Transactions on Information Systems (TOIS)*, 26(1):4–es, 2007.

[12] Andrew Collins, Dominika Tkaczyk, and Joeran Beel. A novel approach to recommendation algorithm selection using meta-learning. In *AICS*, pages 210–219, 2018.

[13] Alan Said and Alejandro Bellogín. Comparative recommender system evaluation: benchmarking recommendation frameworks. In *Proceedings of the 8th ACM Conference on Recommender systems*, pages 129–136, 2014.

[14] Simon Chan, Philip Treleaven, and Licia Capra. Continuous hyperparameter optimization for large-scale recommender systems. In *2013 IEEE International Conference on Big Data*, pages 350–358. IEEE, 2013.

[15] Pawel Matuszyk, Renê Tatua Castillo, Daniel Kottke, and Myra Spiliopoulou. A comparative study on hyperparameter optimization for recommender systems. In *Workshop on Recommender Systems and Big Data Analytics (RS-BDA'16).—2016.—S*, pages 13–21, 2016.

[16] Bruno Giovanni Galuzzi, Ilaria Giordani, A Candelieri, Riccardo Perego, and Francesco Archetti. Bayesian optimization for recommender system. In *World Congress on Global Optimization*, pages 751–760. Springer, 2019.

[17] Ian Dewancker, Michael McCourt, and Scott Clark. Bayesian optimization for machine learning: A practical guidebook. *arXiv preprint arXiv:1612.04858*, 2016.

[18] Gess Fathan, Teguh Bharata Adji, and Ridi Ferdiana. Impact of matrix factorization and regularization hyperparameter on a recommender system for movies. *Proceeding of the Electrical Engineering Computer Science and Informatics*, 5(5):113–116, 2018.

[19] Masoud Mansoury and Robin Burke. Algorithm selection with librec-auto. In *AMIR@ ECIR*, pages 11–17, 2019.

[20] Masoud Mansoury, Robin Burke, Aldo Ordonez-Gauger, and Xavier Sepulveda. Automating recommender systems experimentation with librec-auto. In *Proceedings of the 12th ACM Conference on Recommender Systems*, pages 500–501, 2018.

[21] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855, 2013.

[22] Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *The Journal of Machine Learning Research*, 18(1):826–830, 2017.

[23] H2O.ai. *H2O AutoML*, June 2017. URL `http://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html`. H2O version 3.30.0.2.

[24] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Automated Machine Learning*, pages 151–160. Springer, 2019.

[25] Brent Komer, James Bergstra, and Chris Eliasmith. Hyperopt-sklearn: automatic hyperparameter configuration for scikit-learn. In *ICML workshop on AutoML*, volume 9. Citeseer, 2014.

[26] Axel A de Romblay, Nicholas Cherel, Mohamed Maskani, and Henri Gergard. *MLBox*. URL `https://mlbox.readthedocs.io/en/latest/#`.

[27] Felix Mohr, Marcel Wever, and Eyke Hüllermeier. Ml-plan: Automated machine learning via hierarchical planning. *Machine Learning*, 107(8-10):1495–1515, 2018.

[28] Alex GC de Sá, Walter José GS Pinto, Luiz Otavio VB Oliveira, and Gisele L Pappa. Recipe: a grammar-based framework for automatically evolving classification pipelines. In *European Conference on Genetic Programming*, pages 246–261. Springer, 2017.

[29] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[30] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[31] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[32] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[33] Hugo Jair Escalante, Manuel Montes, and Luis Enrique Sucar. Particle swarm model selection. *Journal of Machine Learning Research*, 10(Feb):405–440, 2009.

[34] Iván Cantador, Peter Brusilovsky, and Tsvi Kuflik. 2nd workshop on information heterogeneity and fusion in recommender systems (hetrec 2011). In *Proceedings of the 5th ACM conference on Recommender systems*, RecSys 2011, New York, NY, USA, 2011. ACM.

[35] Gediminas Adomavicius and Jingjing Zhang. Impact of data characteristics on recommender systems performance. *ACM Transactions on Management Information Systems (TMIS)*, 3(1):1–17, 2012.

[36] Josephine Griffith, Colm O'Riordan, and Humphrey Sorensen. Investigations into user rating information and predictive accuracy in a collaborative filtering domain. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 937–942, 2012.

[37] Tiago Cunha, Carlos Soares, and André CPLF de Carvalho. Selecting collaborative filtering algorithms using metalearning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 393–409. Springer, 2016.

[38] Tiago Cunha, Carlos Soares, and André CPLF de Carvalho. A label ranking approach for selecting rankings of collaborative filtering algorithms. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1393–1395, 2018.

[39] Tiago Cunha, Carlos Soares, and André CPLF de Carvalho. Cf4cf: recommending collaborative filtering algorithms using collaborative filtering. In *Proceedings of the 12th ACM Conference on Recommender Systems*, pages 357–361, 2018.

[40] Gábor Takács, István Pilászy, Bottyán Németh, and Domonkos Tikk. Scalable collaborative filtering approaches for large recommender systems. *Journal of machine learning research*, 10(Mar):623–656, 2009.

[41] John A Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[42] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[43] JH Holland. Adaptation in natural and artificial systems. 2a ediçao, 1993.

[44] Russell Eberhart and James Kennedy. A new optimizer using particle swarm theory. In *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pages 39–43. Ieee, 1995.

[45] Jonas Mockus. *Bayesian approach to global optimization: theory and applications*, volume 37. Springer Science & Business Media, 2012.

[46] Kirthevasan Kandasamy, Akshay Krishnamurthy, Jeff Schneider, and Barnabás Póczos. Parallelised bayesian optimisation via thompson sampling. In *International Conference on Artificial Intelligence and Statistics*, pages 133–142, 2018.

[47] Ian Dewancker and Michaeland Clark Scott McCourt. *Bayesian Optimization Primer*. URL `https://app.sigopt.com/static/pdf/SigOpt_Bayesian_Optimization_Primer.pdf`.

[48] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *International conference on algorithmic applications in management*, pages 337–348. Springer, 2008.

[49] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 69–77, 2011.

[50] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, KDD '19, page 1946–1956, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362016. doi: 10.1145/3292500.3330648. URL https://doi.org/10.1145/3292500.3330648.

[51] Salesforce. *TransmogrifAI*. URL https://transmogrif.ai/.

[52] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

[53] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[54] Eibe Frank and A Mark. Hall, and ian h. witten (2016). the weka workbench. online appendix for" data mining: Practical machine learning tools and techniques, 2016.

[55] Rich Caruana, Alexandru Niculescu-Mizil, Geoff Crew, and Alex Ksikes. Ensemble selection from libraries of models. In *Proceedings of the twenty-first international conference on Machine learning*, page 18, 2004.

[56] James Bergstra, Dan Yamins, and David D Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in science conference*, pages 13–20. Citeseer, 2013.

[57] Electric Brain. *Learning to Optimize*. URL https://www.electricbrain.io/post/learning-to-optimize.

[58] Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger Hoos, and Kevin Leyton-Brown. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, volume 10, page 3, 2013.

[59] James Bergstra, Daniel Yamins, and David Daniel Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. 2013.

[60] José Jiménez and Josep Ginebra. pygpgo: Bayesian optimization for python. *Journal of Open Source Software*, 2(19):431, 2017.

[61] Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. *arXiv preprint arXiv:1807.01774*, 2018.

[62] Yelp. Moe: Metric optimization engine. URL `http://github.com/Yelp/MOE`.

[63] Tim Head, Louppe MechCoder, Iaroslav Shcherbatyi, et al. scikit-optimize/scikit-optimize: v0. 5.2, 2018.

[64] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[65] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2623–2631, 2019.

[66] Fernando Nogueira. Bayesian Optimization: Open source constrained global optimization tool for Python, 2014. URL `https://github.com/fmfn/BayesianOptimization`.

[67] Aaron Klein, Stefan Falkner, Numair Mansur, and Frank Hutter. Robo: A flexible and robust bayesian optimization framework in python. In *NIPS 2017 Bayesian Optimization Workshop*, 2017.

[68] The GPyOpt authors. Gpyopt: A bayesian optimization framework in python, 2016. URL `http://github.com/SheffieldML/GPyOpt`.

[69] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, Stefan Falkner, André Biedenkapp, and Frank Hutter. Smac v3: Algorithm configuration in python, 2017. URL `https://github.com/automl/SMAC3`.

[70] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[71] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[72] Michael D Ekstrand, Michael Ludwig, Joseph A Konstan, and John T Riedl. Rethinking the recommender research ecosystem: reproducibility, openness, and lenskit. In *Proceedings of the fifth ACM conference on Recommender systems*, pages 133–140, 2011.

[73] Guibing Guo, Jie Zhang, Zhu Sun, and Neil Yorke-Smith. Librec: A java library for recommender systems. In *UMAP Workshops*, volume 4, 2015.

[74] Nicolas Hug. Surprise, a Python library for recommender systems. `http://surpriselib.com`, 2017.

[75] Maciej Kula. Spotlight. `https://github.com/maciejkula/spotlight`, 2017.

[76] Maciej Kula. Metadata embeddings for user and item cold-start recommendations. In Toine Bogers and Marijn Koolen, editors, *Proceedings of the 2nd Workshop on New Trends on Content-Based Recommender Systems co-located with 9th ACM Conference on Recommender Systems (RecSys 2015), Vienna, Austria, September 16-20, 2015.*, volume 1448 of *CEUR Workshop Proceedings*, pages 14–21. CEUR-WS.org, 2015. URL `http://ceur-ws.org/Vol-1448/paper4.pdf`.

[77] The pandas development team. pandas-dev/pandas: Pandas, February 2020. URL `https://doi.org/10.5281/zenodo.3509134`.

[78] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. doi: 10.25080/Majora-92bf1922-00a.

[79] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[80] Laurent Candillier, Frank Meyer, and Marc Boullé. Comparing state-of-the-art collaborative filtering systems. In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, pages 548–562. Springer, 2007.

[81] Christian Desrosiers and George Karypis. A comprehensive survey of neighborhood-based recommendation methods. In *Recommender systems handbook*, pages 107–144. Springer, 2011.

[82] Francesco Ricci. Recommender systems: Models and techniques., 2014.

[83] Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*, pages 1238–1246, 2013.