Auto-Surprise: An Automated Recommender-System (AutoRecSys) Library with Tree of Parzens Estimator (TPE) Optimization

Rohan Anand

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Intelligent Systems)

Supervisor: Joeran Beel

September 2020

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Rohan Anand

September 6, 2020

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

____

Rohan Anand

September 6, 2020

Acknowledgments

I would like to thank Professor Joeran Beel for his continuous mentorship and support towards completing my dissertation. He provided valuable suggestions towards the project as well as encouragement to submit a paper to a conference. Without his guidance, I would not have been able to get my poster paper accepted for the ACM RecSys 20 conference.

I would also like to thank the ADAPT Center for providing access to their high performance compute cluster for running my long term experiments.

Finally, I would like to thank my family and friends for their continuous mental support during these uncertain times.

Rohan Anand

University of Dublin, Trinity College
September 2020

Auto-Surprise: An Automated Recommender-System (AutoRecSys) Library with Tree of Parzens Estimator (TPE) Optimization

Rohan Anand, Master of Science in Computer Science
University of Dublin, Trinity College, 2020

Supervisor: Joeran Beel

Finding the optimal algorithm and hyperparameters for modelling has been a challenge. I introduce Auto-Surprise, an Automated Recommender System library. Auto-Surprise is an extension of the Surprise recommender system library and eases the algorithm selection and configuration process. It uses a parallel Sequential Model Based Optimization approach together with Tree of Parzens Estimator's for finding the best algorithm configurations. Compared to an out-of-the-box Surprise library, AutoSurprise performs upto 4% better in terms of RMSE when evaluated with MovieLens, Book Crossing and Jester datasets. It may also result in the selection of an algorithm with significantly lower runtime. Compared to Surprise's grid search, Auto-Surprise performs equally well or slightly better in terms of RMSE, and is notably faster in finding the optimum hyperparameters. Auto-Surprise is designed to be easy to use, the entire optimization process can be executed in just one line of code. As such, a user can create an well performing recommendation model without having any knowledge in machine learning.

Contents

Acknowledgments **iii**

Abstract **iv**

List of Tables **vii**

List of Figures **viii**

Chapter 1 Introduction **1**

1.1 Research Problem . . . 2

1.2 Research Question . . . 2

1.3 Research Goal and Objectives . . . 3

Chapter 2 Background **4**

Chapter 3 Related Work **7**

3.1 Recommender System Frameworks and Libraries . . . 7

3.1.1 Surprise . . . 9

3.1.2 LibRec-Auto . . . 10

3.2 Optimization in AutoML . . . 11

3.2.1 The CASH Problem . . . 11

3.2.2 Sequential Model Based Optimization . . . 12

3.3 Auto Machine Learning Frameworks . . . 13

3.3.1 AutoWEKA . . . 13

3.3.2 AutoSklearn . . . 14

3.3.3 Auto-Keras . . . 15

3.3.4 Tree-based Pipeline Optimization Tool (TPOT) 16

Chapter 4 Auto-Surprise **17**

4.1 Optimization Strategy 18

4.2 Hyperparameter Optimization 19

4.3 Usage 22

4.3.1 Code Guide 22

Chapter 5 Evaluation **26**

5.1 Methodology 26

5.2 Datasets 27

5.3 Metrics 27

5.3.1 Root Mean Square Error 27

5.3.2 Mean Absolute Error 28

5.3.3 Time 28

Chapter 6 Results **29**

6.1 Comparison for different Datasets 29

6.1.1 Movielens 100k dataset 29

6.1.2 Book Crossing Dataset 30

6.1.3 Jester 2 Dataset 31

6.1.4 Comparison with Random Search 32

6.2 Discussion 33

Chapter 7 Conclusion **37**

Chapter 8 Limitations and Future Work **39**

Summary **41**

Bibliography **42**

Appendices **46**

Appendix A Search Space of Auto-Surprise **47**

List of Tables

6.1 Comparison of Auto-Surprise with other Surprise algorithms and Grid Search for the Movielens 100k dataset. 30
6.2 Comparison of Auto-Surprise with other Surprise algorithms and Grid Search for the Book Crossing dataset. 31
6.3 Comparison of Auto-Surprise with other Surprise algorithms and Grid Search for the Jester 2 dataset. 32
6.4 Summary of Performance improvement in percentage compared to the best default algorithm 35

List of Figures

3.1 Auto-Sklearn's approach to AutoML using Bayesian Optimization and Meta-learning. Meta-learning is used to jump start the Bayesian optimizer by providing well fitted initial parameters. 15

4.1 An Overview of the internal working of Auto-Surprise 19

4.2 The inner working of the SMBO process in Auto-Surprise. The domain search space is defined and loaded into the TPE model. A candidate configuration is then selected for evaluation based on the history of configurations. If no history is present, the configuration is randomized. Once evaluated, the loss and configuration are recorded in trials and loaded to the TPE model for finding the next configuration. Once the time budget is exceeded, the history is sorted based on the loss and the configuration with the lowest loss is returned 20

6.1 Mean Validation loss for Auto-Surprise with TPE 33

6.2 Mean Validation loss for Auto-Surprise with ATPE 34

6.3 Mean Validation loss for Random Algorithm 34

6.4 Optimization of algorithms in Auto-Surprise over iterations 36

Listings

3.1 Pseudo Code for SMBO algorithm 12
4.1 Sample Search Space for the KNN Baseline Algorithm 21
4.2 Example usage of Auto-Surprise 23
4.3 Example of using ATPE as an hyperparameter optimization method. . 25
A.1 Search space for Co Clustering algorithm. 47
A.2 Search Space for Similarity Options. Common for any algorithm that uses “sim_options” parameter. 48
A.3 Search space for Baseline Only algorithm. This search space is also used for any algorithm that uses the “bsl_options” parameter. 48
A.4 Search space for KNN Basic, KNN With Means and KNN with Z-Score algorithms. 49
A.5 Search space for KNN Baseline algorithm. 49
A.6 Search space for SVD algorithm. 49
A.7 Search space for SVD++ algorithm. 50
A.8 Search space for NMF algorithm. 50

Chapter 1

Introduction

Developing recommender systems has always been a challenge. Particularly, identifying the best algorithm out of a myriad of possible algorithms as well as the most optimal hyperparameters for a given scenario is difficult. Even the collective intuition of experienced data scientists seems to have trouble to identify the ideal algorithm and hyperparameters. Even minor variations in the implementation and parameters in these models may lead to significantly different performances. This is not a new problem. The machine learning and other communities also faces a similar challenge and tackled it quite successfully with sophisticated Automated Machine Learning (AutoML) solutions.

The goal of most AutoML solutions is to automate the entire machine learning pipeline, from data pre-processing to building an optimized model. AutoML has been particularly impactful in algorithm selection and hyperparameter optimization (HPO). AutoML applies HPO techniques beyond standard grid search and random search. This applies to hyperparameter tuning as well as algorithms selection. Typical AutoML methods include Bayesian Optimization [1], Sequential Model Based Optimization citehutter2011sequential, and Hierarchical Planning [2]. Some more advanced solutions use meta-learning to "warm-start" the optimization process, i.e. they predict a set of algorithms and parameters that seem promising for the given task.

AutoML solutions are primarily in the form of software packages such as H20[3], TPOT [4], AutoWEKA [5], AutoSklearn [6], AutoKeras [7], and MLPlan [2]. Even some cloud service providers such as Google Cloud Platform, Amazon Web Services,

and Azure have also integrated AutoML into their existing cloud machine learning pipelines.

The recommender-systems community seems to have fallen behind the advances of the (automated) machine learning community in this regard. While there are many recommender systems libraries such as [8], LibRec [9], Surprise [10], CaseRec [11], and Lenskit [12], there is – to the best of my knowledge – only one Automated Recommender System Library, namely LibRec-Auto [9]. LibRec-Auto extends the Librec library by adding automated algorithm selection and configuration functionality, though this functionality is limited. LibRec-Auto iterates over parameter spaces in one scripted experiment. This experiment is completely setup by the user, including setting all the possible configuration options. This is useful for experienced data scientists who wish to experiment with different configurations and analyse the models, but an average user would not be able to take advantage of this functionality. LibRec-Auto is also not as advanced as typical AutoML solutions – it uses grid search for its optimization, which is not ideal.

1.1 Research Problem

Currently, there is Auto-Recommender System software library available which automates the algorithm selection and hyperparameter tuning process using advanced AutoML techniques as well as easy to use for user's with minimal prior experience in machine learning and data science.

1.2 Research Question

Can we incorporate existing Auto Machine Learning techniques to be used to build an Auto Recommender System library for algorithm selection and hyperparameter tuning? Can this system work better, in terms of time or metrics, then existing solutions like Grid Search?

1.3 Research Goal and Objectives

The goal of my research is to re-purpose existing Auto Machine Learning techniques to be used for automating recommender systems. The objectives are as follows -

- To automate algorithm selection and hyperparameter tuning for recommender systems by creating an AutoRecSys library.
- Should optimally utilize time and resources while evaluating configurations. Instead of evaluating all possible configurations, this AutoRecSys solution should adapt while evaluating parameters.
- To be easy to use. The user should not need to have any background knowledge on the algorithms and their parameters. Software must also be documented well for this purpose.

Chapter 2

Background

Over the past few decades, the importance of data, and manipulating this data to provide insights and build machine learning models has become more important. Machine learning libraries such as Scikit-learn [13], Tensorflow [14] and Keras [15] provide a large set of tools that facilitate analysis, evaluation, and construction new machine learning models. The general workflow for machine learning is fairly straightforward -

1. Gather data from your data sources.
2. Clean your data and pre-process it. This could be as simple as removing empty data or be as complex as generating new features from insights gained from existing data.
3. Analyze and understand which model works best for your dataset.
4. Train the final model and evaluate it.

While this may seem simple, there are a lot of variables and possible permutation and combinations in each step, making the problem much more complex. As machine learning continues to become more prominent, many new developers have been entering the stream. It was realized early on that there is a large segment of developers who are not data scientists and may not be able to produce good models easily. Thus, AutoML solutions became more prominent. These solutions, such as AutoWEKA [5] and AutoSklearn [6], extended their corresponding machine learning library with automated pipelines. These pipelines automated many steps in the machine learning

workflow such as data pre-processing, algorithm selection, hyperparameter tuning and ensemble modelling.

One of the key areas where AutoML solutions have made the most headway is in Step 3 of the machine learning workflow - solving the algorithm selection and hyperparameter optimization problem. It has been shown before that variations in algorithms and their hyperparameters can significantly affect performance of the model in different scenarios [16]. As such, selecting the optimal algorithm and hyperparameters is crucial to the effectiveness of the machine learning model. This is not an easy process, and involves having a thorough understanding of machine learning algorithms and multiple trial and errors. This is a problem where even the “intuition” of an experienced data scientist may not lead to the best result [17].

One of the early solutions to this problem was Grid Search [18] - where every single configuration was evaluated to find the best configuration. The downside of this solution is that it requires that you need to run every single possible configuration, which is not always feasible in terms of time and resources. Another alternative is Random Search, which randomly tries all kind of configurations within a limited search space until the given time budget is over. The problem with Random Search [18] is that results may not be consistent due to its random nature. Instead of Grid Search and Random Search, other optimization techniques were identified.

One of the first prominent technique used to solve the hyper-parameter tuning problem was Bayesian Optimization [1]. Bayesian optimization is based on Bayes probability theorem. Unlike Grid Search and Random Search, the prior results are taken into account when predicting hyperparameters in Bayesian optimization. This model created using the history of result's is referred to as a ”surrogate” function [19]i. It is much easier to optimize for this surrogate function than the actual objective function. As Bayesian optimization is an informed search method, it is generally much more efficient in finding the optimal hyperparameters for a model than Grid Search. Bayesian optimization is often used together with Sequential Model Based Optimization [20].

The field of Recommendation Systems can be considered as a sub-field of Machine Learning. While machine learning is broadly used for any model that performs classification, regression, and prediction - Recommender Systems focus primarily on user modelling with regards to items. Recommender systems are designed to provide item recommendations for user's. This can be done either explicitly, by knowing the user's

previous preferences, or implicitly determined.

One common technique used by to create recommendation models is Collaborative Filtering [21]. In this technique, the dataset provides the relation between multiple user's and item's. This dataset is then searched to find the smallest subset of user's who have similar preferences. With this subset, a ranking of predictions can be made as to what item a user would have a positive relation with. There are multiple ways this ranking similarity can be calculated such as pearson or cosine similarity [21].

Similar to machine learning community, there are a variety of recommender system libraries available. Some popular ones are Mahout [8], LibRec [9], Surprise [10], CaseRec [11], and Lenskit [12]. These libraries implement several popular recommender system algorithms based on matrix factorization, KNN, clustering, and others. Some, such as Mahout, offer support for execution on distributed systems for high performance and scalability. These libraries also include tools for managing datasets, cross validation, as well as evaluating with metrics.

Unlike the machine learning community, there has not been much headway into automating the Recommender System workflow for ease of use to new users. Some libraries such as Surprise [10] do provide modules for optimal parameter search. However, this parameter search is often done via Grid Search, with the user specifying the exact domain space to search. Grid Search may not be an optimal solution to the problem of algorithm selection and hyperparameter tuning.

Chapter 3

Related Work

3.1 Recommender System Frameworks and Libraries

There are a variety of recommender system libraries available. These libraries implement several popular recommender system algorithms based on matrix factorization, KNN, clustering, and others. Some may also offer support for execution on distributed systems for high performance and scalability. These libraries also include tools for managing datasets, cross validation, as well as evaluating with metrics. Some popular libraries [1] are -

- Apache Mahout [8]. This is more of a general purpose distributed framework for implementing any algorithm, but includes built in recommendation engine support for easy distributed implementation using Apache Spark [22] as a backend.
- LibRec [9], an expansive Java library. This library offers an impressive over 70 recommendation algorithms and highly configurable workflow. Due to this large number of possible configurations as well as the underwhelming documentation, this library is hard to use for any beginner.
- CaseRec [11] is a simple RecSys Python library with a number of algorithms that support rating prediction, item recommendations, and clustering. It also

[1] A full list can be found at `http://recommender-systems.com/research-develop/software-libraries/`

supports use for different similarity metrics such as cosine and similarity. A platform for creating ensembles is also available for developers to use.

- WEKA [23] is a Java library for building recommender systems. It provides a large variety of algorithms to choose from for rating predictions and recommendation modelling. It provides a user interface for easy experimentation with models as well as programmable implementations using the built JAVA API.

- LensKit [24] for Python markets itself as a next generation solution for recommender system experiments. It is based on a previous Java version of LensKit but improves on it's predecessors short comings. It supports use cases for implicit as well as explicit ratings. It also provides an interface for tensorflow based models.

- EasyRec [25] is a simple recommender system library designed with the intention of having a very low barrier for entry for new users of machine learning and recommender systems. EasyRec has a user interface and as such requires no programming knowledge to use. One downside of this is that it offers less control to the user. The EasyRec project is also no longer actively developed.

- RecDb [26] is a database only implementation of a recommender system. It uses a custom implementation of the PostgreSQL [27] database with built-in support for generating recommendations. The advantage of this system is that recommendation operations can be done in a unified approach with other SQL operations (select, join, etc.). This leads to better performance than similar solutions applied separately on top of the database. The downside of this solution is that the entire application would need to be designed around using PostgreSQL database, which means that if the application use's another database, a complete migration of the database would need to be made.

- Surprise [10] is a popular python library. I discuss more about this library in Section 3.1.1.

Most of these libraries are generally created with the goal of being easy to use, and achieve this to varying degrees of success. However, it is necessary to have some background information of machine learning in general as well as the algorithms being used to effectively use these libraries. Other than experienced data scientists, most average

developers and students would not have this information. This is where automating the recommender system workflow can make an impact by making it easy for anyone to build recommendation models.

3.1.1 Surprise

Surprise [10] is a popular python library which specializes in analyzing and building recommendation models. It provides a variety of algorithms which are ready to use. These include -

- **Basic Algorithms:** Normal Predictor (Random) and Baseline Only
- **KNN based algorithms:** KNN Basic, KNN with Means, KNN with Z-Score and KNN with Baseline
- **Matrix Factorization Algorithms:** SVD, SVD++, NMF
- **Others:** Normal predictor, Co-Clustering, Slope One

Some algorithms in Surprise can use similarity or baseline estimates or even both. The baseline estimate module offers 2 options for computing baselines - Stochastic Gradient Descent (SGD) and Alternating Least Squares (ALS) [28]. The similarities module is used to compute similarities between user's and items. This can be used by many algorithms to predict a rating. The available similarity measures can be computed in multiple ways using Surprise:

- Cosine similarity
- Mean Squared Distance (MSD)
- Pearson correlation
- Pearson Baseline correlation coefficient - computed using baselines instead of means

Most recommender systems libraries such as Lenskit [24], CaseRec [11], WEKA [23], LibRec [9] etc., do not provide any built in hyperparameter optimization. Surprise is

unique in that it provides a built-in module for optimizing your model. This can be done using either Grid Search or Random Search. Unfortunately, these methods for parameter search are not the most efficient. In the case of Grid Search, a large number of iterations will need to be done to comprehensively search the entire domain space. This is both computationally expensive as well as time consuming. Random Search can help in reducing time consumption but may not cover the search space efficiently enough and the results may be sub-optimal.

Surprise also provides various tools for evaluating and comparing algorithms. Cross-validation is built-in with multiple possible configurations. A Dataset module is also provided to load built-in datasets as well as handle custom datasets. One downside of Surprise is that it does not support content-based or implicit rating. This is by design and deemed out of scope for the Surprise project and hence will probably not be implemented in the future.

3.1.2 LibRec-Auto

LibRec-Auto [29], to the best of my knowledge, is the only example of an AutoRecSys like library available. It is built as a wrapper around the popular Java software - LibRec [9]. LibRec is a vast library, with over 70 algorithms available for use for collaborative filtering as well as taking into account implicit and explicit ratings. LibRecAuto extends the LibRec recommender system library to add algorithm selection and configuration capabilities. Librec-Auto retains the benefits of using the LibRec library while enabling easier implementation of large scale experiments as well as being scalable.

However, the configuration of algorithms and search space must be done completely by the user. While this makes it very flexible and gives the user complete control over the experiment, it means the user must spend time identifying a configuration space, and hence would require the user to have domain knowledge of LibRec, it's algorithms, as well as deep knowledge about the available hyperparameters for each algorithm. Another drawback of LibRec-Auto is that it uses Grid Search to find the best algorithm configuration. This is inefficient and means that each configuration needs to be evaluated to identify the best solution.

As such, while LibRec-Auto is great for those who have domain knowledge and

want to run a complete experiment on their dataset. However, for those looking for ready to use solutions, LibRec-Auto is less than ideal.

3.2 Optimization in AutoML

Auto Machine Learning attempts to solve the problem of automating the workflow of building machine learning models. The two main problem's AutoML [5] attempts to solve are -

1. No single ML method is best for all problems

2. The performance of some algorithms can be greatly impacted by hyperparameter optimization.

These problems are not related just to machine learning but to recommender systems and other similar optimization problems as well. Both these problems can be handled by the Combined Algorithm Selection and Hyperparameter Optimization (CASH) problem.

3.2.1 The CASH Problem

Auto Machine Learning can be formalized as the Combined Algorithm Selection and Hyperparameter Optimization (CASH) problem [5]. The problems of AutoML can be handled efficiently as single, structured optimization problem with CASH. Given a set of algorithms $\mathcal{A} = \{A^1, ..., A^k\}$ with associated hyperparameters spaces $\Lambda^1, ..., \Lambda^k$, the CASH problem can be mathematically defined as

$$A^*\lambda^* \in \underset{A^{(j)}\in\mathcal{A},\lambda\in\Lambda^{(j)}}{\operatorname{argmin}} \frac{1}{k}\sum_{i=1}^{k}\mathcal{L}\left(A_\lambda^{(j)}, \mathcal{D}_{\text{train}}^{(i)}, \mathcal{D}_{\text{valid}}^{(i)}\right) \tag{3.1}$$

The CASH problem can be tackled in multiple ways. A solution that performs well is Bayesian Optimization, and in particular the use of Sequential Model Based Optimization (SMBO) has proven to be an effective solution [20]. Most AutoML solutions have used SMBO effectively to solve the CASH problem.

```
initialise model $M_\Lambda : H \leftarrow \phi$;
while optimization budget has not been exceeded do
    $\lambda \leftarrow$ candidate configuration from $M$
    Compute $loss = \Lambda(A_\lambda, D(i)_{train}, D(i)_{valid})$
    $H \leftarrow H \cup \{(\lambda, loss)\}$
    Update $M$ given $H$
end while
return $\lambda$ from $H$ with minimal $loss$
```

Listing 3.1: Pseudo Code for SMBO algorithm

3.2.2 Sequential Model Based Optimization

Sequential Model Based Optimization (SMBO) [20] is an optimization approach where multiple trials are run on the surrogate function sequentially. With each trial, the history of the result is stored. This history is then used to attempt to determine optimal hyperparameters to minimize the result of the surrogate function.

A simple pseudo code for SMBO is given in Listing 3.1. The first step of SMBO is to initialize a model based on the search history. Next we start training in a loop until the given optimization budget is not exceeded. In the beginning of each loop, a candidate configuration is taken from the initial model. This configuration is based on the previous history results. In the beginning, a random configuration is selected. The loss for this configuration is then calculated and saved in the history. The model is then updated with the history. Once the time budget has been exceeded, the configuration with the best loss is returned.

The type of surrogate function being used is important, as it directly models the objective function. Common choices which have shown success are Gaussian processes [30], Random Forrest Regression [31], and Tree of Parzens Estimator (TPE) [32].

Tree of Parzens Estimator (TPE)

The Tree Structured or Tree of Parzens Estimator (TPE) [32] is an approach used for SMBO. TPE replaces the generative approach of finding configurations in a search space by using a tree structure of with a set of non parametric distributions. The choices for the parameter distributions are replaced based on the type of distribution. Uniform

distributions are replaced with truncated Gaussian mixtures, log uniform distributions are replaced with exponential Gaussian mixtures, and categorical parameters are re-weighted. This categorical re-weighting is based on the density functions formed by the observations with minimal loss, and the remaining observations.

In experiments, TPE was found to perform significantly better than Grid Search and Random Search - by 20% to 30% [32]. TPE also performed better than Gaussian processes. As such, TPE can be used as a good surrogate function for SMBO processes.

3.3 Auto Machine Learning Frameworks

3.3.1 AutoWEKA

AutoWEKA [5] is one of the first AutoML solutions. AutoWEKA is a package around the popular Java WEKA [23] machine learning package. It employs techniques that many other AutoML libraries now use. AutoWEKA's approach combines a highly parametric machine learning framework with a Bayesian optimization method for instantiating this framework well over a given dataset within a specified budget. Two important problems in AutoML are (1) no single ML method is best for all problems (2) some algorithms rely heavily on hyper parameter optimization. Both these problems are handled efficiently as single, structured optimization problem with CASH. AutoWEKA handles CASH by using WEKA for it's algorithms, and tree-based Bayesian optimization methods.

AutoWEKA uses sequential model-based Algorithm Configuration (SMAC) [33] for it's hyperparameter optimization. SMAC supports various surrogate functions to use as models based on the objective function. This includes Gaussian processes as well as Random Forrest [31]. AutoWEKA uses the random forrest component of SMAC as it deals well with the higher dimensionality that is expected from most input datasets. SMAC implements various mechanics to improve robustness such that consistent results may be expected. These mechanics include diversifying configurations by randomization, as well as evaluating configurations in multiple folds against previous configurations.

In their testing, the authors found that AutoWEKA performed better than Grid Search and random search in most cases. In all cases, AutoWEKA was much more

time efficient as well, showing that even extensive Grid Search may not always give the best results. In some cases, AutoWEKA performed over 10% better than their baselines. This shows that AutoML can be a viable solution over manually building a model with extensive experimentation.

3.3.2 AutoSklearn

AutoSklearn [6] is an AutoML library for the astronomically popular Scikit-learn machine learning library. It also takes inspiration from AutoWEKA and uses it's CASH problem approach, as well as using SMAC [33] with Random Search for it's hyperparameter tuning. AutoSklearn also adds a couple of new methods to increase efficiency and robustness to their AutoML solution.

The first improvement from Auto-Sklearn is adding a meta-learning step to "warm start" the Bayesian optimization process. Information about the performance of algorithms based on meta-features of a dataset are collected. These meta-features are based on information that is easy to compute to reduce overhead, such as statistics of number of classes and features, as well as skewness and entropy. Expensive to compute features like landmarking are avoided. This is used to select initial algorithm hyperparameters and configurations. Meta-learning is complementary to Bayesian Optimization, where meta-learning provides an initial model that can perform well and with bayesian optimization this can be even further optimized. This application of meta-learning is based on other implementations such as ML-SMBO [34].

The second improvement they make is including an automated ensemble construction step. The authors of Auto-Sklearn note that while Bayesian optimization for hyperparameters is data efficient in finding the best hyperparameters, all the models that were trained are lost. These models may perform almost as well as the best model. Discarding these models would be a waste, and instead Auto-Sklearn employs a post processing step for construction of an ensemble of models. This also makes the AutoML process more robust as it makes the final model less prone to overfitting.

The authors of Auto-Sklearn found that their meta learning approach greatly improves initial results. Overall using meta-learning as well as ensembles showed the best performance. Auto-Sklearn was also found to perform favourably when compared to AutoML solutions, to the point that it is now considered a baseline to validate other

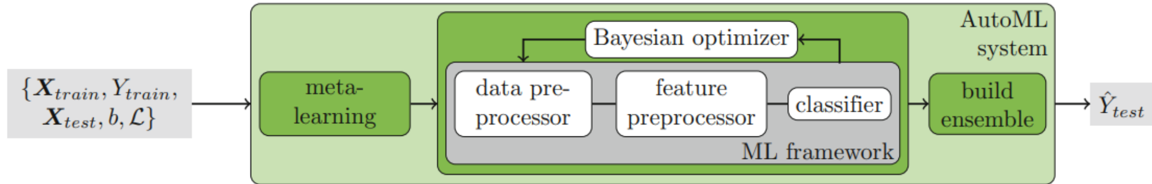

Figure 3.1: Auto-Sklearn's approach to AutoML using Bayesian Optimization and Meta-learning. Meta-learning is used to jump start the Bayesian optimizer by providing well fitted initial parameters.

AutoML solutions.

3.3.3 Auto-Keras

Auto-Keras [7] is unique in the world of AutoML solutions as it's goal is to deep tune neural networks. Previous solutions to the Neural Architecture Search (NAS) problem such as deep reinforcement learning, gradient based approaches, and evolutionary algorithms require a large number of trials to reach a good performance. Many of these solutions also require to start the entire training procedure from scratch. All of this means that these NAS solutions have a high time complexity and are inefficient.

Auto-Keras uses Network Morphism guided by Bayesian Optimization. Network Morphism is a method which morphs a neural network architecture while keeping it's functionality [35]. This means that adding new neural network layers and other morphing operations are possible, only require a few training iterations to achieve good results as we don't need to train from scratch. One major drawback of network morphism is that it requires either a large number of training examples [36] or are not efficient in exploring a large search space [35]. Bayesian optimization has been used for global optimization in machine learning where the training time is also large, similar to the NAS problem. Auto-Keras uses Bayesian optimization in an novel way with Network Morphism guided by Bayesain Optimization.

Auto-Keras is also designed to have an easy to use programming interface as well as being locally deployable. Auto-Keras makes efficient use of all local resources, using the CPU for searching the domain space, RAM for storing the network graphs, GPU for training the model, and storing the models in persistent storage.

3.3.4 Tree-based Pipeline Optimization Tool (TPOT)

TPOT [37] is another wrapper around the popular python machine learning package Scikit-learn [13]. TPOT relies on using machine learning operatiors in the form of Genetic Programming (GP) [38] primitives of machine learning algorithms from Scikit-learn. These primitives are broadly divided into 3 categories - Supervised Classification, Feature Preprocessing, and Feature Selection operators. These operators are parameterized based on the algorithm being used.

These operators are combined together to form a machine learning pipeline. This is done by creating GP trees from the operator's GP primitives. In this pipeline, multiple copies of the dataset is sent to the pipeline. Each dataset is then modified consecutively by each GP operator and finally combined into a single dataset, which is then used to make classifications. Using this flow, arbitrary machine learning pipelines can be created from the GP tree. This provides flexibility in representing the machine learning pipelines. TPOT uses DEAP [39], a GP python package, to automatically construct and optimize these trees.

The authors benchmarked TPOT with 150 supervised classification datasets. It was compared against 30 replicas of a Random Forrest algorithm with 500 trees for each dataset. It was found that in the majority of benchmarks, TPOT had a large performance improvement of 10% to 60% in terms of median accuracy over Random Forrest. In only a few cases, TPOT had performed by 2 to 4% worse.

Chapter 4

Auto-Surprise

With the goal of building a state of the art Auto Recommender System library, I introduce Auto-Surprise [40][1] an open source [2] AutoRecSys library that uses parallel SMBO approach with Tree of Parzens Estimators for hyperparameter tuning. Auto-Surprise is named after the Python Surprise [10] library, around which it is built. The reason why I chose the Surprise library to automate and not the other libraries mentioned is due to a couple of reasons -

- Surprise is one of the most popular recommender system library. There is a large user base for this software who could benefit from it being automated with Auto-Surprise.
- It provides a good mix of types of algorithms to optimize - 11 in total. The number of algorithms is not overwhelming, but still cover most use cases. These algorithms are based on K Nearest Neigbours, Matrix Factorization, Clustering, and others. This means that we can build more complex search spaces to evaluate and hence Auto-Surprise could be more flexible in it's configuration to match different datasets.
- Surprise provides other comprehensive features such as dataset management and cross-validation tools. This reduces the number of external dependencies for

[1] A poster paper for Auto-Surprise was accepted and is to be presented at the ACM RecSys 20 conference

[2] The Auto-Surprise project is available on Github at `https://github.com/BeelGroup/Auto-Surprise` and is open for contributions

Auto-Surprise as well as provides an avenue for further automation, such as configuring cross validation iterators.

- Surprise already provides a Grid Search mechanism, making it easier to evaluate Grid Search against Auto-Surprise and see more fair results.

It uses a sequential model-based optimization approach for the algorithm selection and configuration, is open-source and brings the advances of AutoML to the recommender-system community. Auto-Surprise offers all 11 algorithms that Surprise has implemented. To use Auto-Surprise, a user needs to import the auto-surprise package and pass data to the trainer method. Auto-Surprise then automatically identifies the best performing algorithm and hyperparameters out of all 11 Surprise algorithms. As such, almost no prior knowledge is needed.

4.1 Optimization Strategy

The overall optimization strategy of Auto-Surprise is similar to AutoWEKA [5]. Auto-Surprise uses a parallel SMBO approach to solving the global optimization problem. Auto-Surprise first evaluate a baseline score for the given dataset using random predictor. This sets the minimum loss that each algorithm must achieve. Each algorithm is then optimized in parallel until a user defined time limit or a maximum evaluations limit is reached. If any of the algorithms perform worse than the baseline after a number of evaluations, then that algorithm is not optimized any further and its related processes end, returning the best configuration and loss. Once this process is completed, the best performing algorithm with optimized hyperparameters is returned along with a dictionary of the performance of all the algorithm.

Auto-Surprise is designed in such a way as to allow for multiple implementations of optimization strategies. While the only one currently available is SMBO, other's were also experimented with. One such strategy was a leaderboard strategy. In it, all algorithms would be evaluated for an initial time period. The top performing algorithms would be then taken for another round of evaluations with a longer time period. This step would continue until there is only one algorithm left. This strategy was later on omitted from Auto-Surprise due to inconsistencies in results. However,

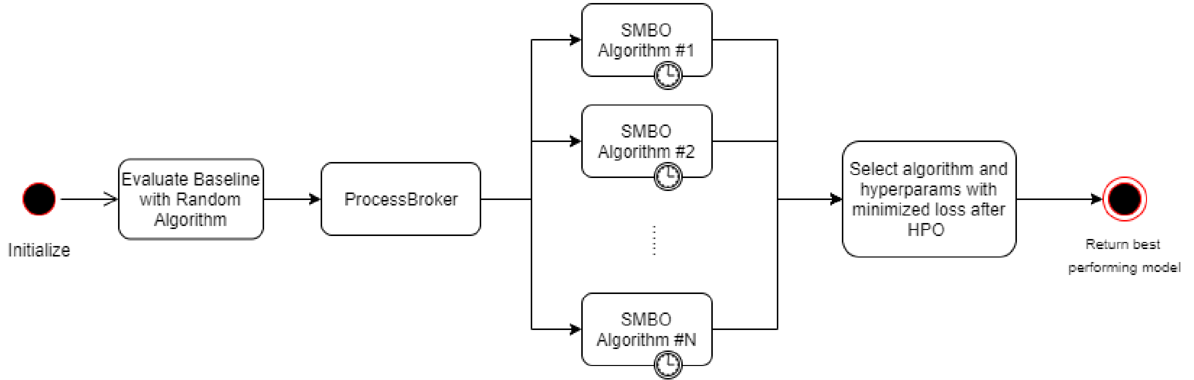

Figure 4.1: An Overview of the internal working of Auto-Surprise

designing different strategies for Auto-Surprise is an option that can be easily added in the future.

4.2 Hyperparameter Optimization

Auto-Surprise can use three hyper-parameter optimization methods as implemented by Hyperopt [41] - Tree of Parzens Estimator (TPE) [32], Adaptive TPE (ATPE) [42] as well as Random Search. The main reason TPE was chosen over other SMBO approaches such as SMAC [33] was because of it's easier implementation. However, Auto-Surprise is designed to be further extendable and a custom SMBO approach that takes the same inputs and outputs as the current implementation could be used later.

The user may set whichever hyper-parameter optimization they prefer, although it is recommended to use TPE as currently it's implementation is more stable than ATPE. Hyper-parameter optimization is done on all available algorithms. A search space is defined for each algorithm. These search spaces are specified for each and every type of algorithm. Numerical parameter are given either a uniform or log uniform distribution with a minimum and a maximum limit to search over. The type of distribution to be used is based on the significance of the hyper-parameter. For example, most learning rate and regularization hyper-parameters are searched over a log uniform distribution, while parameters such as number of factors and k value are searched over a uniform distribution. Other hyper-parameters which take categorical values are given an array search space with all possible options.

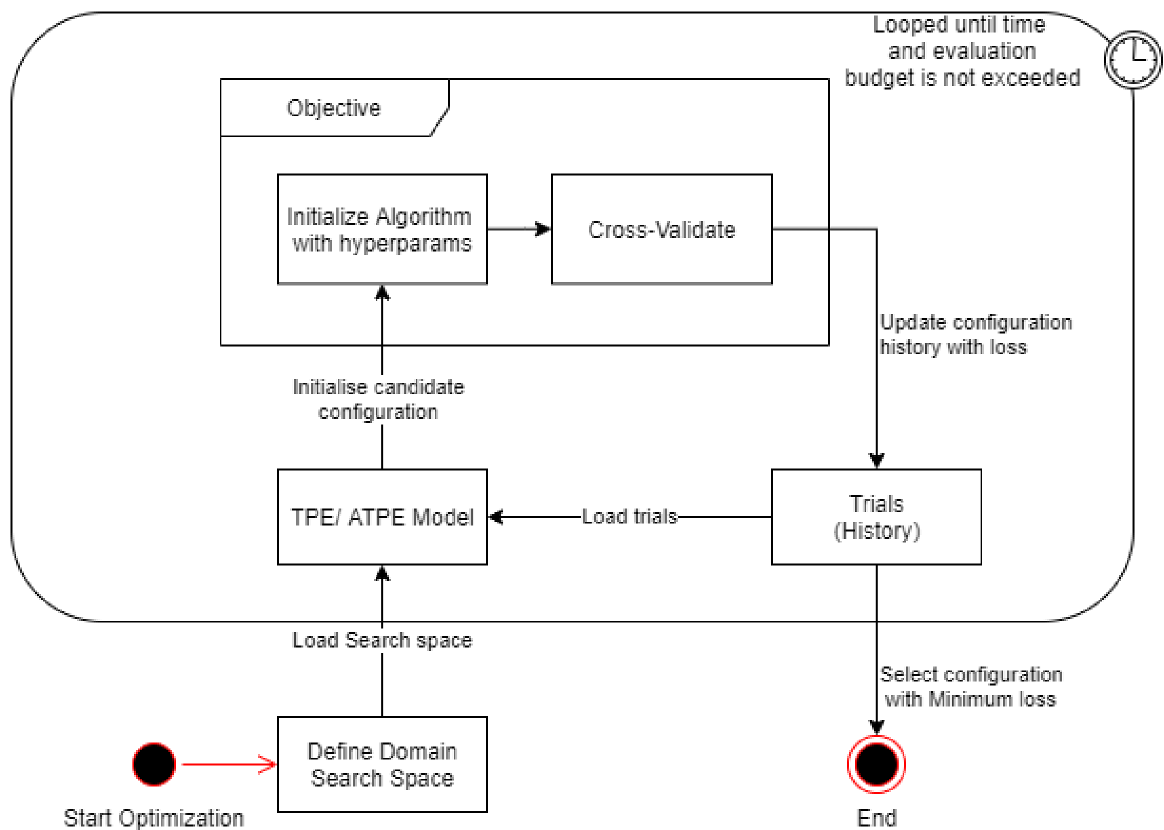

Figure 4.2: The inner working of the SMBO process in Auto-Surprise. The domain search space is defined and loaded into the TPE model. A candidate configuration is then selected for evaluation based on the history of configurations. If no history is present, the configuration is randomized. Once evaluated, the loss and configuration are recorded in trials and loaded to the TPE model for finding the next configuration. Once the time budget is exceeded, the history is sorted based on the loss and the configuration with the lowest loss is returned

```
{
    k: Uniform(1, 500),
    min_k: Uniform(1, 50),
    sim_options: {
        name: ["cosine", "msd", "pearson", "pearson_baseline"],
        user_based: [False, True],
        min_support: Uniform(1, 100),
        shrinkage:  Uniform(1, 300),
    },
    bsl_options: [
        {
            method: "als",
            reg_i: Uniform(1, 100),
            reg_u: Uniform(1, 100,
            n_epochs: Uniform(5, 200)
        },
        {
            method: "sgd",
            reg: LogUniform(0.0001, 0.1),
            learning_rate: LogUniform(0.0001, 0.1),
        }
    ]
}
```

Listing 4.1: Sample Search Space for the KNN Baseline Algorithm

An example of a search space used is given in Listing 4.1. This example is the search space for the KNN Baseline Algorithm. Notice the use of similarity options and baseline options used to supplement the KNN algorithm. A search space for these options is defined as well. In the case of baseline options, you can see that depending on which baseline method is selected, different hyper-parameters are being optimized as well.

4.3 Usage

One of the goals of Auto-Surprise is to be easy to use for user's who have no prior experience in machine learning or recommender systems, as well as providing some amount of flexibility in configuring automation for more advanced users[3]. There are 4 basic steps to using Auto-Surprise -

1. **Install and Import Auto-Surprise:** Auto-Surprise is provided as a Python package available from PyPi. As such installing it is easy with 'pip'. Once installed, simply import Auto-Surprise like any other python package.

2. **Load the Dataset:** As Auto-Surprise uses Surprise for it's underlying algorithms, it requires the dataset to be loaded in the form of a Surprise 'Dataset', which may be created using a standard Pandas Dataframe. The data must also include columns for 'user', 'item', and 'rating'.

3. **Instantiate Auto-Surprise Engine:** The user can configure Auto-Surprise to be verbose or enable debugging here. This will also check if the current python environment meets all the required dependencies.

4. **Starting the training process:** With one line of code, the user may start the entire algorithm evaluation and hyperparameter tuning process. The user may also configure which metric to minimize, which algorithms to run, the time limit, as well as the maximum evaluations to run. Once evaluation is complete. The user will have access to the best algorithm and hyperparameters as well as a dictionary of all the configurations tested.

4.3.1 Code Guide

Auto-Surprise also provides a number of parameters available for the user to configure to control the training process and output.

- Parameters for `auto_surprise.Engine` module

[3]For full documentation, see `https://auto-surprise.readthedocs.io/en/stable/`

```
# Import required libraries
from surprise import Dataset
from auto_surprise.engine import Engine

# Load the dataset
data = Dataset.load_builtin(‘ml-100k")

# Intitialize auto surprise engine
engine = Engine(verbose=True)

# Start the trainer
best_algo, best_params, best_score, tasks = engine.train(
    data=data,
    target_metric=‘test_rmse",
    cpu_time_limit=60 * 60, # Run for 1 hour
    max_evals=100
)
```

Listing 4.2: Example usage of Auto-Surprise

  - `verbose`: Accepts Boolean values. By default set to `True`. This controls the verbosity level of Auto-Surprise. Setting it to `False` will disable most console outputs.
  - `algorithms`: Accepts array of strings. This parameter controls which recommender algorithms should be evaluated to be optimized. By default includes all algorithms.
  - `random_state`: Accepts `numpy.random.RandomState` values. This controls the random state set when optimizing and evaluating algorithms. Using this parameter, experiments can be reproduced by using the same seed.

- Parameters for `auto_surprise.Engine.train` method
  - `data`: The data as an instance of `surprise.dataset.DatasetAutoFolds`. This is the data based on which the algorithms will be modeled.
  - `target_metric`: The metric which Auto-Surprise will seek to minimize. Currently acceptable values are `test_rmse` and `test_mae`
  - `cpu_time_limit`: The maximum time allotted to Auto-Surprise to complete it's search over the domain space. The time is accepted as an integer in the form of number of seconds. This value should generally be at least an hour or more for most datasets.
  - `max_evals`: The maximum number of evaluations each algorithm gets for hyperparameter optimization.
  - `hpo_algo`: Auto-Surprise uses Hyperopt for hyperparameter tuning. By default, it's set to use TPE, but you can change this to any algorithm supported by hyperopt, such as Adaptive TPE or Random search. An example of this is given in Listing 4.3. If a user has their own optimization method and it is compatible with Hyperopt.

```
import hyperopt

...

best_algo, best_params, best_score, tasks = engine.train(
    data=data,
    target_metric='test_rmse',
    cpu_time_limit=60*60*2,
    max_evals=100,
    hpo_algo=hyperopt.atpe.suggest
)
```

Listing 4.3: Example of using ATPE as an hyperparameter optimization method.

Chapter 5

Evaluation

5.1 Methodology

To evaluate how well Auto-Surprise does we decided to run it against multiple datasets. Separate configurations of Auto-Surprise with Adaptive TPE and TPE as the hyper-parameter optimization algorithm were evaluated. In each configuration, the target metric to minimize was set to RMSE with a maximum evaluation time of 2 hours. All algorithms were set to be evaluated. Auto-Surprise's random HPO method was not evaluated as this method is not meant to be used for the end user and our focus is on more effective and robust HPO techniques.

These configurations of Auto-Surprise were compared against all eleven algorithms in surprise with a) the algorithms' default parameters and b) the algorithms' being optimized with Grid Search, as implemented by Surprise. These algorithms were cross-validated with multiprocessing enabled. For Grid Search, the search space was limited so as to use a reasonable amount of time and system resources. The resultant search space is a subset of the one used by Auto-Surprise with a smaller range.

All experiments were run on a cloud instance as a node. Each node for evaluating Auto-Surprise is allocated 8 CPU cores and 128 GB of system memory. Nodes for evaluating Grid Search required more memory - upto 512 GB. All systems were running on Linux with python Python 3.6.

5.2 Datasets

One of the goals of Auto-Surprise is that it should perform well for any kind of collaborative filtering dataset. As it is impossible to evaluate all datasets, 3 popular datasets for recommender systems was selected -

1. Movielens 100k Dataset [43]: A collection of user ratings of movies. This dataset is considered to be the de-facto gold-standard dataset in the recommender system community [44]. Rating scale is from 1 to 5
2. Jester 2 Dataset [45]: A dataset of jokes and their user ratings. Rating scale is from -10 to 10, where negative ratings mean that the user did not enjoy the joke at all.
3. Book Crossing Dataset [46]: A collaborative filtering dataset of book ratings. Books are rated by users on a scale of 0 to 10.

The entire Jester and Book Crossing datasets consist of more than a million rows each and they are both dense datasets. As such loading the datasets into memory and attempting to cross validate for a single algorithm requires an absurd amount of memory (over 624 GB in my experiments), not to mention an incredible amount of time for a single evaluation when we would need to do multiple evaluations. Due to these memory and time constraints the entire Jester and Book Crossing dataset were not used. Instead, a fixed random sample of 100,000 rows was selected.

5.3 Metrics

The goal of Auto-Surprise is to evaluate a search space of algorithms and their configurations efficiently and determine the best performing configuration in terms of loss. As such, the following metrics were considered.

5.3.1 Root Mean Square Error

The Root Mean Square Error (RMSE) is the standard deviation of the prediction errors in an experiment. It indicates how well of a fit the predictions are to the actual results.

As such, a lower value of RMSE indicates that the predictions are a better fit. Since my evaluation of Auto-Surprise the target metric is set to RMSE, this is one of the more important metrics in my experiments.

The basic notation for RMSE is given in equation 5.1

$$RMSE = \sqrt{\overline{(f - o)^2}} \tag{5.1}$$

Here, f is the forecasted data and o is the observed data. This can be further denoted as equation 5.2

$$RMSE = \{\sum_{i=1}^{N}(f_i - o_i)^2/N\}^{1/2} \tag{5.2}$$

Where N is the total sample size. Equation 5.2 is how Auto-Surprise calculates RMSE.

5.3.2 Mean Absolute Error

Although RMSE is a suitable metric, it is useful to have a secondary loss metric to . As such another suitable metric is the Mean Absolute Error (MAE). The absolute error is the amount of error in the forecast versus observed results. The MAE is the combined mean of the absolute error of multiple forecasts and observations. Like RMSE, a lower MAE is desirable from a model. The MAE can be denoted and calculated as show in equation 5.3

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|f_i - o_i| \tag{5.3}$$

5.3.3 Time

Time is as important a resource as compute power and memory. Since one of the goals of Auto-Surprise is to be time efficient and hence CPU execution time taken for each configuration was also a key metric. The time is measured from the start of the optimization process for Auto-Surprise and grid search. For default configurations of Surprise, the cross validation time is measured.

Chapter 6

Results

6.1 Comparison for different Datasets

The results for the Movielens 100k dataset, Book-Crossing dataset and Jester dataset are shown in tables 6.1.1, 6.1.2, 6.1.3 respectively. Results in bold indicate the best performing algorithm in it's default configuration. A summary of all results is provided in Table 6.2. A comparison of Auto-Surprise's optimization over time is also made with Random Search.

6.1.1 Movielens 100k dataset

The best default algorithm in Surprise for this dataset was SVD++ with an RMSE of 0.9196. Auto-Surprise was able to perform best with adaptive TPE with an RMSE of 0.9116. This is a small - but statistically significant difference (2 tailed p value ¡ 0.05) in RMSE of 0.86%. Grid Search was beaten by a small margin with an RMSE of 0.9139. However, to achieve this result Grid Search took just over 27 hours, while Auto-Surprise performed slightly better in just 2 hours, over 13 times faster.

It is also worth noting that the final selected algorithm was not SVD++ (the best performing by default) but rather KNN Baseline and NMF in Auto-Surprise TPE and ATPE configurations respectively. When comparing RMSE of these default algorithms, Auto-Surprise showed an improvement of 1.8% and 5.6% respectively. This selection of different algorithms for both configurations is interesting and is discussed further in Section 6.2.

Algorithm	**RMSE**	**MAE**	**Time (HH:MM:SS)**
Normal Predictor	1.5195	1.2200	00:00:01
SVD	0.9364	0.7385	00:00:23
SVD++	**0.9196**	**0.7216**	**00:14:23**
NMF	0.9651	0.7592	00:00:25
Slope One	0.9450	0.7425	00:00:15
KNN Basic	0.9791	0.7738	00:00:18
KNN with Means	0.9510	0.7490	00:00:19
KNN with Z-score	0.9517	0.7470	00:00:21
KNN Baseline	0.9299	0.7329	00:00:22
Co-clustering	0.9678	0.7581	00:00:08
Baseline Only	0.9433	0.7479	00:00:01
GridSearch	0.9139	0.7167	27:02:48
Auto-Surprise (TPE)	0.9136	0.7280	02:00:01
Auto-Surprise (ATPE)	0.9116	0.7244	02:00:02

Table 6.1: Comparison of Auto-Surprise with other Surprise algorithms and Grid Search for the Movielens 100k dataset.

The only metric for which Auto-Surprise did not perform as well was in terms of MAE. Compared to SVD++, Auto-Surprise performed 0.33 - 0.88% worse, while Grid Search performed 0.67% better. However, when compared to the actual selected algorithm, there is is still an improvement of 0.66% when compared to KNN Baseline and Auto-Surprise (TPE), and 4.58% when compared with NMF and Auto-Surprise (ATPE).

6.1.2 Book Crossing Dataset

For the book crossing dataset, we see that the best performing default algorithm configuration was SVD with an RMSE of 3.5586. Both NMF and Slope One algorithms failed to model this dataset. We see a better performance with Auto-Surprise here where with a minimum RMSE of 3.5586 with ATPE - a 1.11% difference. This time the selected algorithm was KNN Baseline and SVD in TPE and ATPE configurations respectively.

Once again, Grid Search took much longer to evaluate the search space - over

Algorithm	**RMSE**	**MAE**	**Time (HH:MM:SS)**
Normal Predictor	4.8960	3.866	00:00:01
SVD	**3.5586**	**3.013**	**00:00:11**
SVD++	3.5842	2.991	00:01:48
NMF	–	–	–
Slope One	–	–	–
KNN Basic	3.9108	3.562	00:00:38
KNN with Means	3.8574	3.301	00:00:35
KNN with Z-score	3.8526	3.292	00:00:37
KNN Baseline	3.6181	3.101	00:00:36
Co-clustering	4.0168	3.409	00:00:19
Baseline Only	3.5760	3.095	00:00:02
GridSearch	3.5467	2.9554	48:29:46
Auto-Surprise (TPE)	3.5221	2.8871	02:00:58
Auto-Surprise (ATPE)	3.5190	2.8739	02:00:06

Table 6.2: Comparison of Auto-Surprise with other Surprise algorithms and Grid Search for the Book Crossing dataset.

48 hours, 24 times longer than the time allotted to Auto-Surprise. Grid Search also produced a model with significantly worse performance in RMSE when compared to Auto-Surprise. Auto-Surprise performed 0.67% and 0.78% better than Grid Search in terms of RMSE in it's TPE and ATPE configurations respectively.

Unlike the experiment of the Movielens dataset, this time there is an improvement in MAE as well. When compared to SVD, the MAE is reduced by 4.41% using TPE and 4.61% with ATPE. Auto-Surprise also performed better than Grid Search in this regards as well - 3.73% better with TPE and 4.21% better than ATPE.

6.1.3 Jester 2 Dataset

The best performing default configuration was the Baseline Only algorithm with an RMSE of 4.849. Similar to the results for the Book-Crossing dataset, NMF algorithm failed to model this dataset. In this case, Auto-Surprise managed a much better improvement in RMSE to 4.6489, a difference of 4.12%. In both TPE and ATPE configurations, KNN baseline algorithm was selected.

Algorithm	RMSE	MAE	Time (HH:MM:SS)
Normal Predictor	7.277	5.886	00:00:01
SVD	4.905	3.97	00:00:13
SVD++	5.102	4.055	00:00:29
NMF	–	–	–
Slope One	5.189	3.945	00:00:02
KNN Basic	5.078	4.034	00:02:14
KNN with Means	5.124	3.955	00:02:16
KNN with Z-score	5.219	3.955	00:02:20
KNN Baseline	4.898	3.896	00:02:14
Co-clustering	5.153	3.917	00:00:12
Baseline Only	**4.849**	**3.934**	**00:00:01**
GridSearch	4.7409	3.8147	80:52:35
Auto-Surprise (TPE)	4.6489	3.6837	02:00:10
Auto-Surprise (ATPE)	4.6555	3.6906	02:00:01

Table 6.3: Comparison of Auto-Surprise with other Surprise algorithms and Grid Search for the Jester 2 dataset.

Auto-Surprise outperformed Grid Search by a significant margin in terms of both RMSE and time. The difference between the RMSE in Grid Search and Auto-Surprise is 1.94% with TPE and 1.80% with ATPE. With regards to time, Grid Search took more than 80 hours, approximately 40.5 times longer than Auto-Surprise.

Auto-Surprise also outperforms in terms of MAE. With an MAE of 3.6837 and 3.6906 with TPE and ATPE respectively, it performs 6.79% (TPE) and 6.61% (ATPE) better than Baseline Only in it's default configuration. It also perform 3.53% (TPE) and 3.25% (ATPE) better than Grid Search.

6.1.4 Comparison with Random Search

Figure 6.1 and 6.2 show the validation loss for the best performing algorithm in Auto-Surprise. The high level of fluctuation in loss between each iteration show's how large of a difference altering parameters can have on a model. This fluctuation is also especially high as these algorithms are highly parameterized. As more iterations are done, this fluctuation in validation loss, as well as the mean validation loss. As such running

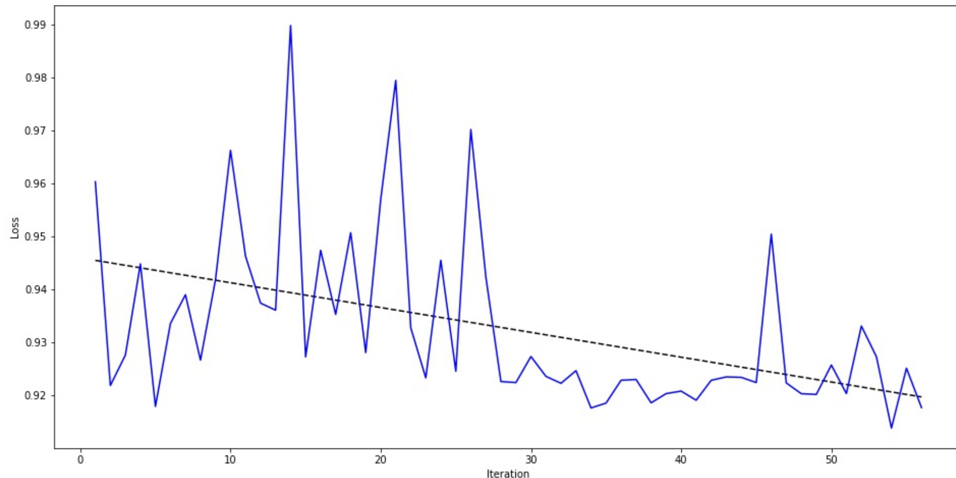

Figure 6.1: Mean Validation loss for Auto-Surprise with TPE

Auto-Surprise for longer may yield better results as well. This is unlike Random Search (Figure 6.3) where the slope of the mean validation loss isn't nearly as steep, i.e., increasing the number of iterations will not lead to improved results in average validation loss.

6.2 Discussion

To complete these experiments, it took approximately 169 hours or 7 days worth of compute time. In my experiments, it is clear that Grid Search is not the ideal solution for automating algorithm selection and hyperparameter tuning. In terms of RMSE, Auto-Surprise achieved anywhere from 0.65% to 4.12% better performance when compared to the next best algorithm in its default configuration while Grid Search only performed 0.33% to 2.22% better. Even though the target metric was RMSE, Auto-Surprise also optimizes at upto 6.36% better in terms of. A higher improvement would probably be seen in MAE if the target metric was MAE.

In all 3 datasets, Auto-Surprise outperforms Grid Search and in 2 out of 3 datasets in terms of MAE. It is notable how time efficient Auto-Surprise is when compared to Grid Search. While resulting in similar and sometimes worse RMSE and MAE performance

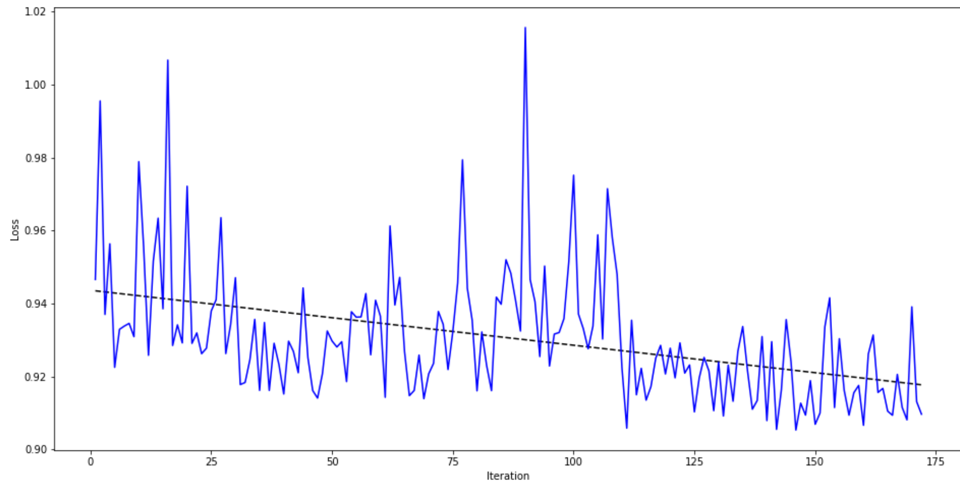

Figure 6.2: Mean Validation loss for Auto-Surprise with ATPE

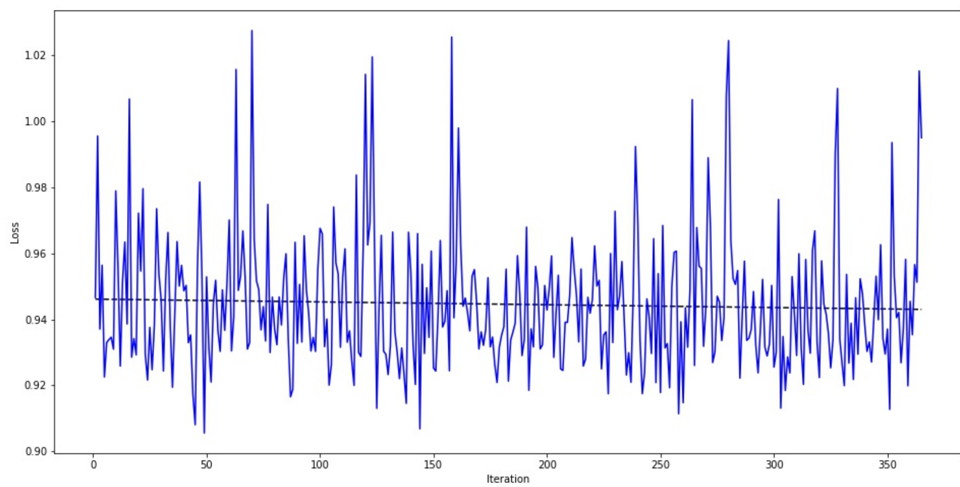

Figure 6.3: Mean Validation loss for Random Algorithm

	Grid Search		Auto-Surprise (TPE)		Auto-Surprise (ATPE)	
Dataset	**RMSE**	**MAE**	**RMSE**	**MAE**	**RMSE**	**MAE**
Movielens 100k	0.61	0.67	0.65	-0.88	0.86	-0.38
Book Crossing	0.33	1.99	1.02	4.17	1.11	4.61
Jester 2	2.22	3.03	4.12	6.36	3.99	6.18

Table 6.4: Summary of Performance improvement in percentage compared to the best default algorithm

than Auto-Surprise, Grid Search also took anywhere from 13 to 40 times longer. It is possible that confining the search space may have hurt the RMSE performance of Grid Search to a degree. However, if the search space was as large as the one defined for Auto-Surprise, Grid Search would take an unreasonable amount of time . Thus, Auto-Surprise can be considered as a good alternative for Grid Search.

In some datasets, it is observed that Auto-Surprise with it's different configurations can lead to selecting different algorithms for its final model. This can be because hyperparameter tuning for a long period of time could mean that the difference in loss of two different models is neglegible. Figure 6.4 is a snapshot of the optimization of 3 well performing algorithms for the MovieLens dataset. KNN Baseline does perform the best, but the KNN with Means algorithm also performs fairly similarly. As such, if the training period was much longer, it is possible that we could see KNN with Means performing better. As such it is possible that for different seeds, the results may be different.

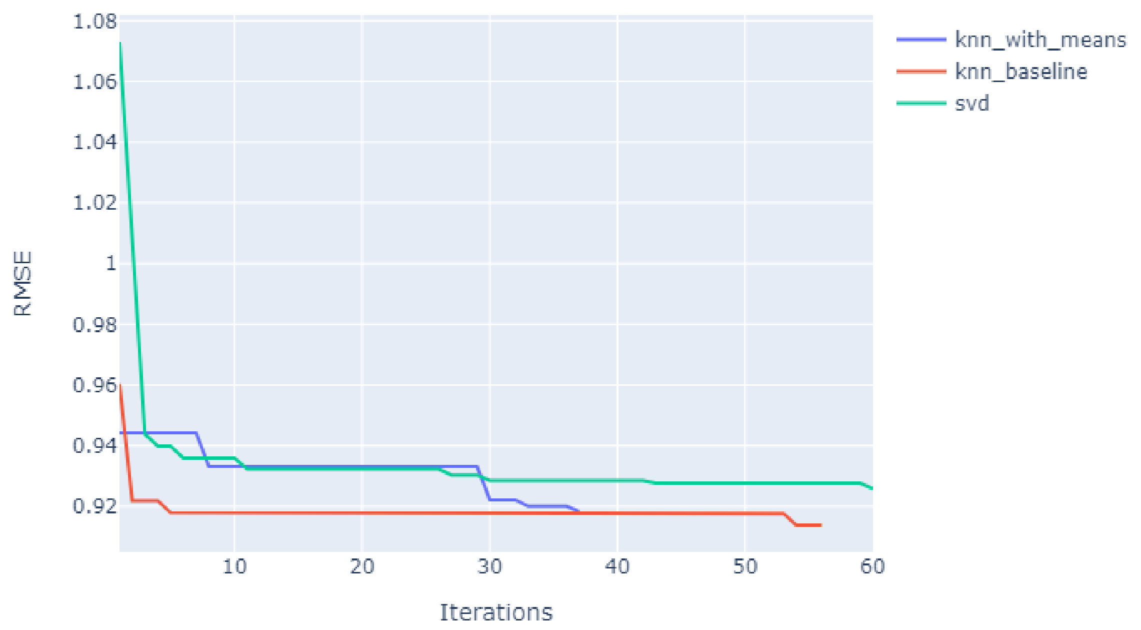

Figure 6.4: Optimization of algorithms in Auto-Surprise over iterations

Chapter 7

Conclusion

In these experiments, Auto-Surprise performed anywhere from 0.8 to 4% better in terms of RMSE and upto 6.36% better in terms of MAE when compared to the best result from a default algorithm configuration of Auto-Surprise. The actual evaluation time of the combined default Surprise algorithms is lower than Auto-Surprise in these experiments. However, Auto-Surprise still outperforms Gridsearch in regards to time by a huge margin - while Auto-Surprise got good results in 2 hours, Grid Search could take anywhere from 24 to 80 hours - 12 to 40 times longer. Auto-Surprise also outperformed Grid Search in terms of RMSE in all 3 test datasets and MAE in 2 of the 3 test datasets.

It is also worth noting that the selected algorithm may have a much lower runtime compared to the default algorithm as shown for the Movielens dataset test where the selected algorithm NMF only has a runtime of 25 seconds compared to the 15 minutes runtime for SVD++, the best performing default algorithm. And, of course, Auto-Surprise eases the entire process of algorithm selection and hyperparameter optimization by automating it in a single line of code.

Auto-Surprise is designed to be easy to use for even a novice user. Without knowing much about machine learning or recommender systems, a user can create a well performing recommender model with Auto-Surprise. The entire optimization process can be done in one line of code and can be easily configured by user's looking to have a little more control on the optimization process. Also, Auto-Surprise is designed to be modular, such that new optimization strategies can be easily included as an alternative

to existing strategies. The open source nature of Auto-Surprise also encourages these improvements.

Auto-Surprise will be useful for those wishing to experiment with recommender system's while having minimum experience. Individuals and Organizations that are looking for adding any recommendation feature to a product could use Auto-Surprise to quickly create a well performing model with minimal effort, experience, and most importantly, time. Student's of machine learning or recommender systems could also use Auto-Surprise to understand the effects that hyperparameters can have on a model.

There has been some interest being generated for Auto-Surprise and Auto Recommender Systems in general as of late. The Auto-Surprise project has (as of version v0.1.6) reached over 4,000 downloads and counting [1]. Auto-Surprise has also been accepted for the ACM RecSys 20 conference as a demo paper. I hope to further enhance Auto-Surprise with the help of open source contributors.

[1] For download statistics, please see `https://pepy.tech/project/Auto-Surprise`

Chapter 8

Limitations and Future Work

Just like any other framework, Auto-Surprise is not without its limitations. One major downside of Auto-Surprise is that it currently only supports modelling for datasets with explicit ratings. As such, it cannot evaluate models for implicit ratings or content-based filtering. This is mainly because the underlying Surprise [10] library does not support them, stating that it is out of scope for their project. As such Auto-Surprise will also probably not support implicit ratings or content-based filtering anytime in the near future without adding other libraries that support these features.

Another challenge for Auto-Surprise is resource utilization. This has been a challenge for AutoML frameworks as well. As Auto-Surprise run's multiple SMBO's in parallel, multiple copies of the original dataset are created. This means that for large dataset's, Auto-Surprise would require large amount's of RAM. This is also a problem that Grid Search as well as other AutoML solutions face. Auto-Keras [7] solved this problem by offloading some data to persistent storage devices. Such a solution would require a rework of how data is loaded into Auto-Surprise and each SMBO node. Another solution would be to allow Auto-Surprise to run on a cluster, with each node of the cluster running each algorithm. This solution would not be feasible for the average user, but could be useful for training large models for an organization.

Auto-Surprise attempts to optimize multiple algorithms but the final result is only one models, a lot of computation is wasted in the end. It would be helpful to make use of those models that performed close to the best performing algorithm. Just like Auto-Sklearn [6], Auto-Surprise could also make use of automated ensemble modelling.

This would mean that some of the better performing models are not wasted and more effective use of computing power. Another advantage would be that this would reduce the chance of over-fitting, which Auto-Surprise can be susceptible to for longer evaluations. It would also be useful to explore other hyper-parameter optimization methods such as SMAC [5], neural network search, or genetic programming.

One area of the machine learning pipeline that Auto-Surprise currently does not automate is data pre-processing and feature engineering. This is an area where most AutoML systems have made progress - from simple statistical meta-data to engineering completely new features automatically. This is an important step which could greatly improve performance without increasing the time complexity too much. Re-purposing these techniques for Auto-Surprise would be greatly beneficial. Time complexity could also be reduced by reducing the number of iterations needed by jump starting the optimizer with good starting parameters. Meta-learning has been used by Auto-Sklearn [6] and ML-Plan [34] to solve this and a similar technique could be used for Auto-Surprise.

Due to the open source nature of Auto-Surprise, these goals are hoped to be addressed by the combined effort of passionate open source collaborators. To that end the Auto-Surprise project has been designed in a modular fashion to allow for ease of development of new features. Workflows for creating a new feature, reporting bugs, testing, and publishing have been setup on the GitHub page [1] for further encouragement and ease of development.

[1] `https://github.com/BeelGroup/Auto-Surprise`

Summary

Selecting the best algorithm to model for a problem as well as optimize it's hyperparameters has been a problem in machine learning and recommender systems. One solution was Grid Search, in which every single configuration is evaluated. This however, is inefficient. The machine learning community solved this problem with some success using Auto-ML solutions. These employ sophisticated feature pre-processing, algorithm selection and hyperparameter tuning to automate the entire machine learning workflow. However, the recommender system community has been slow to adopt these features to automate their workflows.

I introduce Auto-Surprise, an Automated Recommender System Library. Auto-Surprise is an extension of the Surprise recommender system library. It eases the algorithm selection and hyperparameter tuning process in the recommender system workflow. This automation can be done with low code and with minimal experience in machine learning and recommender systems. Compared to out-of-the-box Surprise algorithm configurations, Auto-Surprise performs anywhere from 0.8 to 4.0% better in terms of RMSE in 3 test datasets. It is also notably faster (13 to 40 times) compared to Grid Search at finding the optimal hyperparameters.

Bibliography

[1] J. Mockus, V. Tiesis, and A. Zilinskas, "The application of bayesian methods for seeking the extremum," *Towards global optimization*, vol. 2, no. 117-129, p. 2, 1978.

[2] F. Mohr, M. Wever, and E. Hüllermeier, "Ml-plan: Automated machine learning via hierarchical planning," *Machine Learning*, vol. 107, no. 8-10, pp. 1495–1515, 2018.

[3] A. Candel, V. Parmar, E. LeDell, and A. Arora, "Deep learning with h2o," *H2O. ai Inc*, 2016.

[4] R. S. Olson and J. H. Moore, "Tpot: A tree-based pipeline optimization tool for automating machine learning," in *Automated Machine Learning*, pp. 151–160, Springer, 2019.

[5] C. Thornton, F. Hutter, H. H. Hoos, and K. 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*, pp. 847–855, 2013.

[6] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Advances in neural information processing systems*, pp. 2962–2970, 2015.

[7] H. Jin, Q. Song, and X. Hu, "Auto-keras: An efficient neural architecture search system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1946–1956, 2019.

[8] D. Lyubimov and A. Palumbo, *Apache Mahout: Beyond MapReduce.* CreateSpace Independent Publishing Platform, 2016.

[9] G. Guo, J. Zhang, Z. Sun, and N. Yorke-Smith, "Librec: A java library for recommender systems.," in *UMAP Workshops*, vol. 4, 2015.

[10] N. Hug, "Surprise, a python library for recommender systems," *URL: http://surpriselib. com*, 2017.

[11] A. da Costa, E. Fressato, F. Neto, M. Manzato, and R. Campello, "Case recommender: a flexible and extensible python framework for recommender systems," in *Proceedings of the 12th ACM Conference on Recommender Systems*, pp. 494–495, 2018.

[12] M. D. Ekstrand, M. Ludwig, J. A. Konstan, and J. T. Riedl, "Rethinking the recommender research ecosystem: reproducibility, openness, and lenskit," in *Proceedings of the fifth ACM conference on Recommender systems*, pp. 133–140, 2011.

[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[14] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

[15] A. Gulli and S. Pal, *Deep learning with Keras.* Packt Publishing Ltd, 2017.

[16] J. Beel, C. Breitinger, S. Langer, A. Lommatzsch, and B. Gipp, "Towards reproducibility in recommender-systems research," *User modeling and user-adapted interaction*, vol. 26, no. 1, pp. 69–101, 2016.

[17] C. A. Gomez-Uribe and N. Hunt, "The netflix recommender system: Algorithms, business value, and innovation," *ACM Transactions on Management Information Systems (TMIS)*, vol. 6, no. 4, pp. 1–19, 2015.

[18] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *The Journal of Machine Learning Research*, vol. 13, no. 1, pp. 281–305, 2012.

[19] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, pp. 2951–2959, 2012.

[20] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *International conference on learning and intelligent optimization*, pp. 507–523, Springer, 2011.

[21] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in *Proceedings of the 10th international conference on World Wide Web*, pp. 285–295, 2001.

[22] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, p. 56–65, Oct. 2016.

[23] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[24] M. D. Ekstrand, "The LKPY package for recommender systems experiments: Next-generation tools and lessons learned from the lenskit project," *CoRR*, vol. abs/1809.03125, 2018.

[25] E. Husremović, "Easyrec."

[26] M. Sarwat, R. Moraffah, M. F. Mokbel, and J. L. Avery, "Database system support for personalized recommendation applications," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pp. 1320–1331, IEEE, 2017.

[27] K. Douglas and S. Douglas, *PostgreSQL: a comprehensive guide to building, programming, and administering PostgresSQL databases*. SAMS publishing, 2003.

[28] Y. Koren, "Factor in the neighbors: Scalable and accurate collaborative filtering," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 4, no. 1, pp. 1–24, 2010.

[29] M. Mansoury and R. Burke, "Algorithm selection with librec-auto.," in *AMIR@ ECIR*, pp. 11–17, 2019.

[30] C. E. Rasmussen, "Gaussian processes in machine learning," in *Summer School on Machine Learning*, pp. 63–71, Springer, 2003.

[31] A. Liaw, M. Wiener, *et al.*, "Classification and regression by randomforest," *R news*, vol. 2, no. 3, pp. 18–22, 2002.

[32] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in neural information processing systems*, pp. 2546–2554, 2011.

[33] A. Asuncion and D. Newman, "Uci machine learning repository," 2007.

[34] M. Feurer, J. T. Springenberg, and F. Hutter, "Initializing bayesian hyperparameter optimization via meta-learning," in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[35] T. Elsken, J.-H. Metzen, and F. Hutter, "Simple and efficient architecture search for convolutional neural networks," *arXiv preprint arXiv:1711.04528*, 2017.

[36] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Efficient architecture search by network transformation," *arXiv preprint arXiv:1707.04873*, 2017.

[37] R. S. Olson and J. H. Moore, "Tpot: A tree-based pipeline optimization tool for automating machine learning," in *Workshop on automatic machine learning*, pp. 66–74, 2016.

[38] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic programming*. Springer, 1998.

[39] F.-A. Fortin, F.-M. De Rainville, M.-A. G. Gardner, M. Parizeau, and C. Gagné, "Deap: Evolutionary algorithms made easy," *The Journal of Machine Learning Research*, vol. 13, no. 1, pp. 2171–2175, 2012.

[40] R. Anand and J. Beel, "Auto-surprise: An automated recommender-system (autorecsys) library with tree of parzens estimator (tpe) optimization," *To be published in ACM Conference: RecSys '20, Fourteenth ACM Conference on Recommender Systems*, 2020.

[41] J. Bergstra, D. Yamins, and D. D. Cox, "Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms," in *Proceedings of the 12th Python in science conference*, pp. 13–20, Citeseer, 2013.

[42] Electricbrain, "Hypermax." `https://github.com/electricbrainio/hypermax`, 2019.

[43] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *Acm transactions on interactive intelligent systems (tiis)*, vol. 5, no. 4, pp. 1–19, 2015.

[44] J. Beel and V. Brunel, "Data pruning in recommender systems research: Best-practice or malpractice," *ACM RecSys*, 2019.

[45] K. Goldberg, T. Roeder, D. Gupta, and C. Perkins, "Eigentaste: A constant time collaborative filtering algorithm," *information retrieval*, vol. 4, no. 2, pp. 133–151, 2001.

[46] C.-N. Ziegler, S. M. McNee, J. A. Konstan, and G. Lausen, "Improving recommendation lists through topic diversification," in *Proceedings of the 14th international conference on World Wide Web*, pp. 22–32, 2005.

Appendix A

Search Space of Auto-Surprise

The search space of Auto-Surprise dictates the limits of the ranges which Auto-Surprise will check for hyperparameter optimization. In the following listing's, the search space for any algorithm that use's hyperparameters is defined. Normal Predictor and Slope One algorithm do not use any hyperparameters and as such do not require a search space defined. "Uniform" and "LogUniform" refer to the type of distribution defined, as well as their lower and upper limits. "Choice" refers to a parameter whose value can be taken from one of the given options.

```
{
    "n_cltr_u": Unform(1, 1000),
    "n_cltr_i": Unform(1, 100),
    "n_epochs": Unform(5, 200),
}
```

Listing A.1: Search space for Co Clustering algorithm.

```
{
    "name": Choice[
        "cosine", "msd", "pearson", "pearson_baseline"
    ],
    "user_based": Choice[false, true],
    "min_support": Uniform(1, 100),
    "shrinkage":  Uniform(1, 300),
}
```

Listing A.2: Search Space for Similarity Options. Common for any algorithm that uses "sim_options" parameter.

```
Choice(
    [
        {
            "method": "als",
            "reg_i": Uniform(1, 100),
            "reg_u": Uniform(1, 100),
            "n_epochs": Unform(5, 200),
        },
        {
            "method": "sgd",
            "reg": LogUniform(0.0001, 0.1),
            "learning_rate": LogUniform(0.0001, 0.1),
        },
    ],
)
```

Listing A.3: Search space for Baseline Only algorithm. This search space is also used for any algorithm that uses the "bsl_options" parameter.

```
{
    "k": Unform(1, 500),
    "min_k": Unform(1, 50),
    "sim_options": SIMILARITY_OPTIONS_SPACE,
}
```

Listing A.4: Search space for KNN Basic, KNN With Means and KNN with Z-Score algorithms.

```
{
    "k": Unform(1, 500),
    "min_k": Unform(1, 50),
    "sim_options": SIMILARITY_OPTIONS_SPACE
    "bsl_options": BSL_OPTIONS_SPACE
}
```

Listing A.5: Search space for KNN Baseline algorithm.

```
{
    "n_factors": Unform(1, 200),
    "n_epochs": Unform(1, 200),
    "lr_bu": LogUniform(0.0001, 0.1),
    "lr_bi": LogUniform(0.0001, 0.1),
    "lr_pu": LogUniform(0.0001, 0.1),
    "lr_qi": LogUniform(0.0001, 0.1),
    "reg_bu": LogUniform(0.0001, 0.1),
    "reg_bi": LogUniform(0.0001, 0.1),
    "reg_pu": LogUniform(0.0001, 0.1),
    "reg_qi": LogUniform(0.0001, 0.1),
}
```

Listing A.6: Search space for SVD algorithm.

```
{
    "n_factors": Unform(1, 200),
    "n_epochs": Unform(1, 200),
    "lr_bu": LogUniform(0.0001, 0.1),
    "lr_bi": LogUniform(0.0001, 0.1),
    "lr_pu": LogUniform(0.0001, 0.1),
    "lr_qi": LogUniform(0.0001, 0.1),
    "reg_bu": LogUniform(0.0001, 0.1),
    "reg_bi": LogUniform(0.0001, 0.1),
    "reg_pu": LogUniform(0.0001, 0.1),
    "reg_qi": LogUniform(0.0001, 0.1),
    "lr_yj": LogUniform(0.0001, 0.1),
    "reg_yj": LogUniform(0.0001, 0.1),
}
```

Listing A.7: Search space for SVD++ algorithm.

```
{
    "n_factors": Unform(1, 500),
    "n_epochs": Unform(5, 200),
    "lr_bu": LogUniform(0.0001, 0.1),
    "lr_bi": LogUniform(0.0001, 0.1),
    "reg_bu": LogUniform(0.0001, 0.1),
    "reg_bi": LogUniform(0.0001, 0.1),
    "reg_pu": LogUniform(0.0001, 0.1),
    "reg_qi": LogUniform(0.0001, 0.1),
    "biased": Choice([False, True]),
}
```

Listing A.8: Search space for NMF algorithm.