Bayesian Tuning and Bandits: An Extensible, Open Source Library for AutoML

by

Laura Gustafson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

© Massachusetts Institute of Technology 2018. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 25, 2018

Certified by . . . .
Kalyan Veeramachaneni
Principal Research Scientist
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chairman, Masters of Engineering Thesis Committee

Bayesian Tuning and Bandits: An Extensible, Open Source Library for AutoML

by

Laura Gustafson

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2018, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

The goal of this thesis is to build an extensible and open source library that handles the problems of tuning the hyperparameters of a machine learning pipeline, selecting between multiple pipelines, and recommending a pipeline. We devise a library that users can integrate into their existing datascience workflows and experts can contribute to by writing methods to solve these search problems. Extending upon the existing library, our goals are twofold: one that the library naturally fits within a user's existing workflow, so that integration does not require a lot of overhead, and two that the three search problems are broken down into small and modular pieces to allow contributors to have maximal flexibility.

We establish the abstractions for each of the solutions to these search problems, showcasing how both a user would use the library and a contributor could override the API. We discuss the creation of a recommender system, that proposes machine learning pipelines for a new dataset, trained on an existing matrix of known scores of pipelines on datasets. We show how using such a system can lead to performance gains.

We discuss how we can evaluate the quality of different solutions to these types of search problems, and how we can measurably compare them to each other.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Principal Research Scientist

Acknowledgments

I would like to thank Kalyan for his guidance through out the project. His willingness to talk through ideas, make suggestions, and assist when needed were crucial to the success of the project.

Additionally, I would like to thank my parents, MaryEllen and Paul, for their constant support throughout my life. Without their continued love and encouragement, I never would have gotten to where I am today. I would also like to thank my older brother, Kevin, for his support.

Finally, I would like to thank Kali, Anne, and Brittany for their support over the past year. While I may now have another BTB, I would like to thank the members of Burton Third for having been there through thick and thin and ensuring I never was lonely.

Contributions

We would like to acknowledge the contributions of Carles Sala and Micah Smith in the design of the API, the open-source release, and general development of the BTB library.

We acknowledge generous funding support from Xylem and National science foundation for this project.

Contents

1 Introduction **19**

1.1 Overview of Problems 20

1.2 Tuning 20

1.3 Selection 22

1.4 Recommendation 24

1.5 BTB 26

1.6 Thesis organization 27

2 Tuners and Selectors **29**

2.1 Tuner 29

2.1.1 API Changes 29

2.1.2 User API 30

2.1.3 Contributor API 31

2.2 Selector 35

2.2.1 User API 35

2.2.2 Contributor API 36

3 Recommender **39**

3.1 Overview of recommender problem 39

3.1.1 Related Work 40

3.1.2 Notation 40

3.2 Methods 41

3.3 Design 45

3.4 User API . . . 45
3.5 Contributor API . . . 46
3.6 Evaluation . . . 49
3.6.1 Data . . . 49
3.6.2 Storing the results . . . 51
3.6.3 Evaluation of Results . . . 52
3.7 Results . . . 54
3.7.1 Performance Graphs . . . 56
3.7.2 Improvement Upon the Results . . . 57

4 Current usage of BTB **59**
4.1 DeepMining . . . 60
4.2 MIT TA2 System . . . 62
4.3 ATM . . . 65

5 Open Source Preparation **67**
5.1 Increased Functionality . . . 67
5.1.1 Handling Categorical hyperparameters . . . 67
5.1.2 Abstraction of Logic for Hyperparameter Transformation . . . 69
5.2 Testing . . . 70
5.3 Examples . . . 71

6 Getting contributions from experts **73**
6.1 Evaluating methodological contributions . . . 76
6.1.1 Setting benchmarks . . . 77
6.1.2 Evaluation Metrics . . . 79
6.1.3 Results of an experiment . . . 81
6.2 Automation . . . 83
6.3 Workflow for creating and merging a new method . . . 85

7 Conclusion **87**
7.1 Key Findings and Results . . . 87

7.2 Contributions 87

A Tables **89**

List of Figures

1-1 Diagram showcasing how a tuner works. Hyperparameter sets tried previously and their scores are passed to the tuner, which proposes a new set of hyperparameters. A new pipeline is run with the hyperparameters, the resulting score is added to the data, and the tuning continues. . . 21

1-2 Gaussian kernel density estimator of performance on `hutsof99_logis_1` for three different classification techniques: logistic regression, multi-layer perceptron and random forest. The performance for each of the classifiers is aggregated over tuning a variety of different hyperparameter search spaces. 22

1-3 Diagram showing how a typical selector system works. Data pertaining to past scores for each of the choices is passed to the selector which outputs a choice to make. The score that results from that choice is added to the data, and the process continues. 23

1-4 Diagram showing data used in a recommender system. We know the performances of some solutions on existing problems, along with the performance of a few solutions on a new problem. We can use the known problem data to make recommendations for the new problem. 25

1-5 Diagram showing how data flows to and from a recommender system. At each iteration, the matrix is passed to the recommender. The recommender's proposal is tried on the new row in an experiment, and the matrix is updated. 25

2-1 A Python code snippet illustrating how to use a tuner's user API. The code illustrates how to use a tuner to tune a `number of estimators` hyperparameter between 10 and 500 and `maximum depth` of a tree between 3 and 20 for a random forest. The code snippet is given training data X, y and testing data X_{test} and Y_{test}. The tuner does not require the user to store any scoring data on their end. The user only has to call two methods after initialization, `add` to add new data to the tuner and `propose` to receive a new recommendation from the tuner. 32
2-2 Interaction between the user API and developer-overridden methods for the tuner class `add` method. 33
2-3 Interaction between the user API and developer-overridden methods for the tuner class `propose` method 33
2-4 Relationship between six different tuners. Includes the developer API functions overridden to implement the tuner. 35
2-5 Python code snippet illustrating how to use a selector alongside a tuner to choose whether to next tune a random forest or an support vector machine (SVM) classifier. The `n_estimators` is a hyperparameter for a random forest and has the range $[10, 500]$. The `c` is a hyperparameter for SVM and has the range $[0.01, 10.0]$. The code snippet is supplied with training data X, y and testing data X_{test} and Y_{test}. `TUNER_NUM_ITER` specifies the number of iterations of tuner when a particular pipeline (in this case only a classifier) and `SELECTOR_NUM_ITER` specifies the number of iterations for the selector. 36
2-6 Interaction between user-facing API and developer overridden methods for selector class select method 38

3-1 Python pseudo-code snippet illustrating how to use a recommender's user API 46
3-2 Interaction between user API and developer-overridden methods for recommender class add method 48

3-3 Interaction between user API and developer-overridden methods for recommender class propose method 48
3-4 `pbest` at different iterations for matrix factorization and uniform recommenders on *visualizing_ethanol_1* dataset 56

4-1 An illustration of how data flows through a system that uses a tuner, such as DeepMining or TA2, which shows how score and hyperparameter information is shared within the system. 59
4-2 Python pseudo-code snippet illustrating how DeepMining uses BTB to score hyperparameter combinations in parallel. 61
4-3 Python pseudo-code snippet illustrating how MIT TA2 uses BTB to yield the best possible pipeline over a series of iterations. 63
4-4 LSTM Pipeline Data Flow Diagram 64
4-5 Diagram showing how information including scores, hyperparamters, and choices flows through ATM. 65

6-1 Histogram of a search problem with a difficult distribution. While it is easy to perform well on this search space, it is hard to model the distribution to consistently outperform a uniform tuner. This search problem is searching the KNN hyperparameters described in Table 6.2 on *housing_1* dataset. 83
6-2 Comparison of the best ranking hyperparameter combination given a MLP pipeline on `chscase_health_1` dataset. The grid for this search problem had 1000 different hyperparameter sets. 84
6-3 Comparison of the best mean f1 score yielded by a pipeline tuned for a given the KNN pipeline and the `chscase_health_1` dataset. 84

List of Tables

3.1 Potential scores for three datasets A, B, C on 5 pipelines 1,2,3,4,5 . . 43
3.2 Score rankings corresponding to table 3.1 44
3.3 List of datasets used in evaluation of matrix factorization-based recommender. 50
3.4 Storage of Results of recommender Evaluation 52
3.5 Maximum p_best, Number of Wins and C.I. Lower Bound Results from Recommender Evaluation on 30 Datasets; Each Experiment Ran 20 Times 54
3.6 Performance Increase Percentage Results from Recommender Evaluation on 30 Datasets; Each Experiment Ran 20 Times 54
3.7 Statistical Analysis of Results from Recommender Evaluation on 30 Datasets; Each Experiment Ran 20 Times 55

4.1 Table of the LSTM Text pipeline's tunable hyperparameters, ranges, and description. 65

5.1 Example showing how the values of a categorical hyperparameter can be mapped to a numerical representation and back 68

6.1 List of datasets used in tuner evaluation 77
6.2 The model type and tunable hyperparameters for the search problems used in tuner evaluation 78

A.1 User API for selectors 90
A.2 Developer API for selectors 90

A.3 User API for tuners 91
A.4 Developer API for tuners 92
A.5 User API for recommenders 93
A.6 Developer API for recommenders 93
A.7 API of Hyperparameter BTB Class 94
A.8 Results from tuner evaluation on 20 20 search spaces; Each experiment was run 20 times 95
A.9 Statistical analysis of results from tuner evaluation on 20 search spaces; Each experiment was run 20 times 95

Chapter 1

Introduction

In recent years, the success of machine learning (ML) has lead to an increased demand for ML techniques in fields such as computer vision and natural language processing. This in turn has led to a demand for systems that can help automate aspects of the process that involve humans, such as choosing a machine learning model or selecting its hyperparameters. These systems have become a part of the rapidly growing field of automated machine learning (AutoML) [17], which seeks to reduce the need for human interaction to create and train machine learning models.

In this thesis, we create a general purpose open source library, Bayesian Tuning and Bandits (BTB), that enables the following search problems within AutoML: tuning the hyperparameters of a machine learning pipeline, selecting between multiple pipelines, and recommending a pipeline.

We envision the people who will interact with BTB will fall into two categories: users and expert contributors. Users are the people who will integrate BTB into their existing data science workflows, but will not need in-depth knowledge of the field of AutoML in order to use the library. Expert contributors are AutoML or ML experts who will contribute to the BTB library by writing their own novel methods for solving the tuning, recommendation, or selection problems.

Our goals for the library are twofold: First, that the library fits within a user's existing workflow naturally enough that integration does not require a lot of overhead, and second, that the three search problems are broken down into small and modular

pieces to give expert contributors maximal flexibility. As there is debate about different methods, we have taken steps to ensure that the library is flexible enough to easily swap out different methods. While the field of AutoML is growing rapidly, no existing libraries cleanly and concisely deliver their capabilities to users while simultaneously bridging the gap between experts in AutoML and data scientists solving real world problems using AutoML.

1.1 Overview of Problems

The problems we choose to address in the BTB library are:

1. Tuning the hyperparameters of a pipeline to maximize a score.
2. Selecting among a series of options in order to maximize the score of the option.
3. Leveraging the performance of different pipelines on previous datasets to recommend a pipeline for a new dataset.

1.2 Tuning

Tuning tackles the simplest of all search problems. It involves a fixed pipeline where only the hyperparameters are tuned to maximize the score. For example, take tuning a random forest machine learning model. We could choose to search for the two hyperparameters: `max_depth`, the maximum depth allowed for any tree in the forest, and `n_estimators`, the number of trees in the forest. We can set the ranges for search at $[4, 8]$, and $[7, 20]$ respectively. For a pipeline P that has k hyperparameters, $\alpha_1...\alpha_k$, and a function f that scores P with the specified hyperparameters, we formally define the tuning problem as:

$$\begin{aligned} Given: \ & P(\alpha_1...\alpha_k), f(\cdot) \\ Find: \ & argmax_{\alpha_1...\alpha_k}\langle f(P(\alpha_1...\alpha_k)\rangle \end{aligned} \tag{1.1}$$

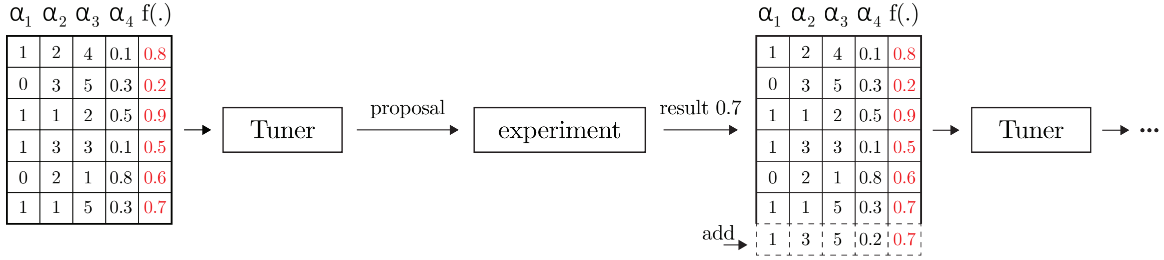

Figure 1-1: Diagram showcasing how a tuner works. Hyperparameter sets tried previously and their scores are passed to the tuner, which proposes a new set of hyperparameters. A new pipeline is run with the hyperparameters, the resulting score is added to the data, and the tuning continues.

The scoring function typically calculates a desired metric *via* cross-validation. This task can usually be accomplished by what is known as a black box optimization. In this approach, :

– A meta-model is formed that identifies the functional (or probabilistic) relationship between the hyperparameters and the pipeline scores.

– Scores are predicted for a series of candidate hyperparameter sets using the meta-model.

– A specific hyperparameter set is chosen based on these predictions.

The AutoML community has developed a number of methods for meta-modeling, including a Gaussian process-based regression and a tree-structured parzen estimator [3]. This type of search also has many other applications, including tuning the hyperparameters of a compiler to maximize speed [24].

Figure 1-1 shows a typical integration with a tuner. The user has a pipeline that they want to tune, which requires a series of hyperparameter values to be specified in order to run. At each time step, the user has a matrix of hyperparameter sets that were tested before, along with their associated scores. The user passes this to the tuner, which creates a meta-model and then proposes a new hyperparameter set to *try*. The user sets these hyperparameters for the pipeline, fits the pipeline and records the resulting score. The newly tried hyperparameters set and score are added to the

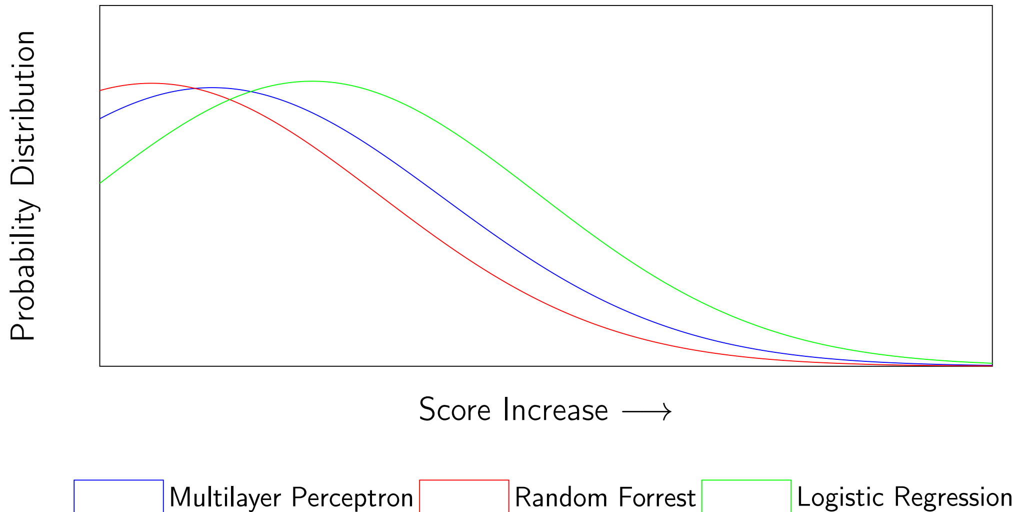

Figure 1-2: Gaussian kernel density estimator of performance on `hutsof99_logis_1` for three different classification techniques: logistic regression, multilayer perceptron and random forest. The performance for each of the classifiers is aggregated over tuning a variety of different hyperparameter search spaces.

matrix for the next iteration. The tuning process continues until a fixed number of iterations are finished or a desired score is achieved.

1.3 Selection

Selection is the next level of search. Here, we have multiple possible pipelines, each with their own hyperparameters that can be tuned. This search problem is complex because we do not know *apriori* which pipeline will yield the best score when fully tuned. This problem essentially involves allocating resources for tuning different machine learning pipelines.

Formally, for a set of m pipelines (or choices) P_i that each has k_i hyperparameters, we define the selection problem as:

$$\begin{aligned} Given: \;\; & P_i(\alpha_i ... \alpha_{k_i}) | i = 1...m, f(\cdot) \\ Find: \;\; & argmax_{\ i;\ \alpha_1 ... \alpha_{k_i}} f(P_i(\alpha_1 ... \alpha_{k_i})) \end{aligned} \tag{1.2}$$

At each time step, the selector recommends the next pipeline to tune for a previously determined number of iterations. The best score achieved as a result is fed back to the

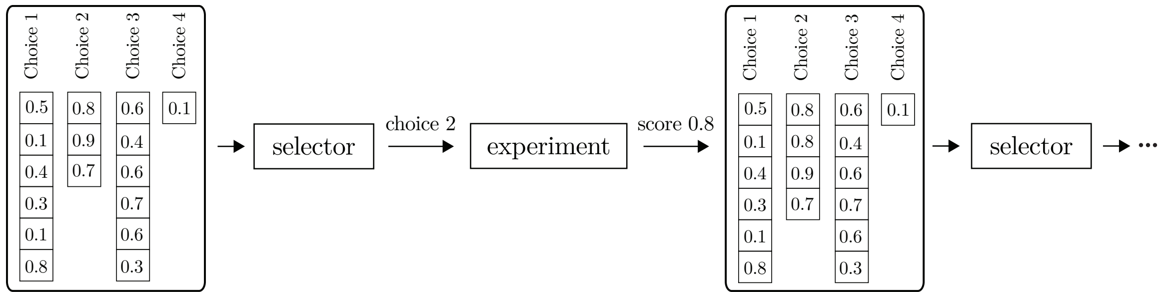

Figure 1-3: Diagram showing how a typical selector system works. Data pertaining to past scores for each of the choices is passed to the selector which outputs a choice to make. The score that results from that choice is added to the data, and the process continues.

selector, which refits and proposes the which pipeline to tune next. After a specific choice is made, the score comes from an unknown probability distribution. Figure 1-2 shows an example score distribution for three pipeline choices. In this case, the choices are between training a multilayer perceptron, a random forest, or a logistic regression model to make predictions on the dataset. The figure shows the distribution of the scores of the model given a variety of different hyperparameter combinations. Suppose that, on the first iteration, we choose a poor hyperparameters for the logistic regression model (the score would be far on the left of Figure 1-2). This would lead to an inference that perhaps logistic regression is a bad choice. If we were using a naive or greedy approach rather than a selector, we would likely rank this choice poorly. However, we may try logistic regression with a good selection algorithm and are likely to get a better result given the distribution. After enough iterations, the selector will likely converge on logistic regression, the most likely to give the best score (when properly tuned).

To aid in selection, the scores achieved for each choice are typically converted into a set of rewards. The selection problem uses a meta-model to estimate the distribution of a reward for a specific choice, and then chooses based on that estimation. Alternately, it can also use a simple *explore-exploit* heuristic to make a choice [30].

Figure 1-3 shows how a selector is put into practice. At each iteration, a data structure stores a mapping of each of the possible choices available to the selector and a historical list of scores received for that choice. This is passed to the selector,

which selects an option from those choices. The pipeline corresponding to that choice is picked and a preset number of tuning iterations are run. At the end of the run, a score update is received. We add the score to our list of scores for that choice, and continue the process. While this process is straightforward, we design a much more user-friendly API for selection.

1.4 Recommendation

The highest level of search is recommendation [25]. In this level, we have scores for a set of pipelines with fixed hyperparameters that have been tried on a number of datasets (Not all pipelines may have been tried on all datasets). Given this information and a new dataset, we aim to pick the candidate pipeline that will give the best score. The best pipeline selected with a recommender can be further improved by tuning its hyperparameters.

Thus recommendation proposes pipelines to new datasets by leveraging their previous performances to *similar* datasets. Formally, for a set of m pipelines $\{P_i|_{\alpha_1...\alpha_{k_i}}|i = 1...m\}$ with fixed k_i hyperparameters and a series of n known datasets $\{d_1...d_n\}$, we define the recommendation problem as:

$$\begin{aligned} Given:\ & P_1...P_m,\ d_1...d_n, f(\cdot) \\ Let: A \in R^{n\times m}\ & s.t.\ a_{ij} = f(d_j, P_i|\alpha_1...\alpha_{k_i}) \\ Find:\ & argmax_i\ f(d_{j+1}, P_i|\alpha_1...\alpha_{k_i})|A \end{aligned} \tag{1.3}$$

These problems usually have the same goal, and share some underlying structural elements. One example that illustrates this is that of movie recommendations. Suppose there is a matrix that contains different people, along with their ratings of a series of different movies. We can use this matrix to recommend a new movie to a user given a few of their existing movie ratings. Because we are trying to achieve the same objective —- find a movie that the user would rate highly —- and there is a similarity between the solutions to the problems —- people with similar taste tend to rate movies similarly —- we can leverage other users' ratings to predict how the

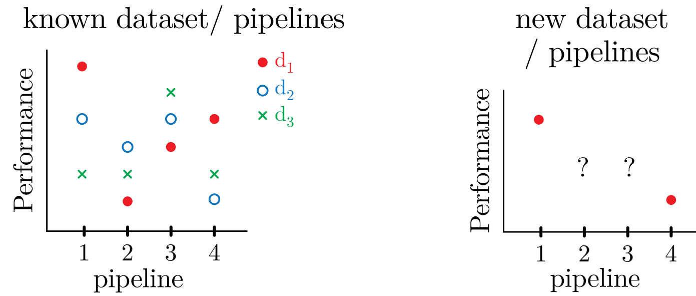

Figure 1-4: Diagram showing data used in a recommender system. We know the performances of some solutions on existing problems, along with the performance of a few solutions on a new problem. We can use the known problem data to make recommendations for the new problem.

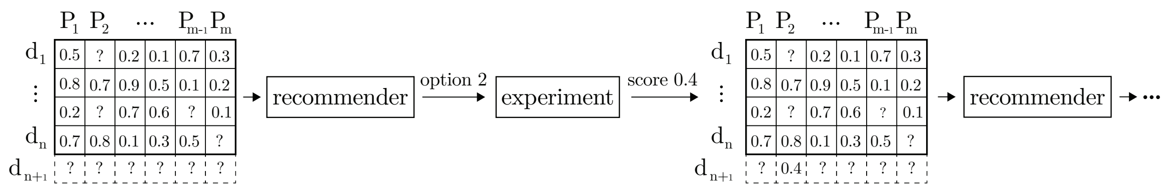

Figure 1-5: Diagram showing how data flows to and from a recommender system. At each iteration, the matrix is passed to the recommender. The recommender's proposal is tried on the new row in an experiment, and the matrix is updated.

new user would rate the movie. Similarly, for machine learning problems, we can use information about how machine learning pipelines have performed on previous datasets in order to predict which ones would be good to try on a new dataset.

Figure 1-4 illustrates how one can solve a recommendation problem. The first figure shows the scores for each possible pipeline (on x-axis) for each dataset (different legends). The second figure contains a new dataset and scores for a couple of the pipelines. We want to predict the scores for the other solutions, which we can do by leveraging the information in the first graph. We can see that the two known scores in the second graph closely match those of dataset 2 in the first figure, so we can extrapolate that the scores for pipelines 2 and 3 in the second graph would also closely match those of dataset 2. Because pipeline 3 was the best for dataset 3, we could recommend that solution for our new dataset.

Figure 1-5 shows how a typical recommender system is put to use in the context of AutoML. An established matrix contains information about the performance of different pipelines on datasets. We then have a new dataset for which we would like to

receive pipeline recommendations (added as a new row). This information is passed to the recommender. We propose a specific pipeline to try, run an experiment evaluating the proposed pipeline on the dataset, and receive a score. We update the matrix with this new score and continue the recommendation process.

1.5 BTB

In this thesis, we describe Bayesian Tuning and Bandits, BTB, a new software library that brings together these three disparate search approaches. BTB is publicly available on GitHub[1].

Data scientists and statisticians have an immense amount of knowledge when it comes to designing meta-modeling approaches, creating newer versions of selection algorithms, and designing effective recommender systems *via* matrix factorization. Oftentimes these approaches are designed and tested outside the context of AutoML. As the AutoML field rapidly expands [21], we imagine that state-of-the-art solutions for each of the above search procedures will evolve, be debated, and constantly change.

The researchers developing these algorithms and models are also often separate from the software engineers trying to search for and implement a machine learning pipeline for their problem. The goal of our open source is to create a centralized place where:

- researchers/experts can contribute better approaches for these search techniques without worrying about building out an elaborate testing and evaluation framework, and
- users can benefit from the performance gains of a state-of-the art-technique without having to understand the complex statistics required to build it and without integrating with multiple libraries.

Prior to this thesis, the library contained abstractions to solve the tuning and selection problems. We focused on expanding and perfecting the abstractions in this

[1] https://github.com/HDI-Project/BTB/

library so that experts could readily contribute to the library and users could integrate it into their projects. We focused heavily on user experience with the library, in order to ensure that the search process integrates seamlessly into existing workflows. While BTB can be used to solve the aforementioned search problems in any application, for the purpose of this thesis we focus on its application to AutoML.

1.6 Thesis organization

The rest of this thesis is as organized as follows:

Chapter 2 explains the detailed abstractions and `apis` for each of the main types in BTB. Chapter 3 explains in detail the recommender system and the implemented types of recommenders. Chapter 4 demonstrates how easy BTB is to integrate into existing systems through a series of example systems that use the library.

Chapter 5 discusses some of the steps that were necessary to prepare the library for being open-sourced and accepting expert contributions. Chapter 6 covers the contribution process and how different tuners will be evaluated against each other.

Chapter 7 summarizes the results and contributions made throughout the thesis.

Chapter 2

Tuners and Selectors

To reiterate our goals: in order for the BTB library to be broadly adopted, its abstractions must fit intuitively into a work flow. In order for experts to contribute to the library, the abstractions must also provide flexibility to contribute to different parts of the library. Previous abstractions for the tuner and selector systems can be found in [29]. In this chapter, we present the changes we made to the abstractions over the course of this thesis, and the rationale behind them.

2.1 Tuner

The tuner proposes a new hyperparameter set that could potentially improve the score. To do this, it first uses a meta-modeling technique to model the relationship between the hyperparameters and the score. It then finds the space that is predicted to maximize the score, and proposes a candidate drawn from that space.

2.1.1 API Changes

We modified the tuner API from [29] to make it simpler for the user. During integration attempts, we noticed that the previous API required the user to store and maintain all the hyperparameter set and their scores tried so far. In each iteration, the user was required to call `fit` with all of the this data to learn a meta model, even though

only a few new hyperparameter sets were evaluated each time (in our case, only one hyperparameter set). The API worked like this in order to remain consistent with scikit-learn's `fit` and `predict` API [6].

We found a few issues with this API:

– *Too focused on meta modeling-based optimization*: The API assumes that the tuner will use a meta modeling-based approach for optimization, but this may not always be the case.

– *Creates confusion when optimizing ML pipelines*: The most common use case for this library is tuning machine learning pipelines. A user is likely to use a `fit` and `predict`-like API for this fitting a proposed pipeline to the data. We noticed that using a similar API for hyperparameter tuning can create a lot of confusion.

– *Not intuitive*: Generally, in a traditional scikit-learn model, the user does not add more data to the training set over time. Instead, he or she starts out with a full dataset, calls `fit` once on all of the data, and then calls `predict` on any new pieces of data. For tuners, however, the user almost always adds data incrementally, as new hyperparameter combinations are tried sequentially. This makes the scikit-learn API less than ideal.

2.1.2 User API

We modified the API to use an `add` and `propose` method. Table A.3 in Appendix A shows the user-facing methods and attributes of a tuner. Below, we discuss a use case for each of these methods, and describe how they serve the user.

- `add`: In each iteration, the user passes two lists to the tuner. The first is a list of dictionaries representing the hyperparameter combinations that they have tried. The second is a list of the corresponding scores for each of these hyperparameter combinations. As new hyperparameters are tested, this continues in an iterative fashion, where the user specifies only new hyperparameters and

their scores, as opposed to everything each time. The tuner updates the available hyperparameter/score data and refits the meta-model on the resulting dataset. This method was not part of the original API– instead, the user called `fit` directly. While both `apis` are clean, the new API allows for a much better user experience. It removes the overhead of requiring the user to manage the storage of hyperparameter combinations and their scores, and it fits the end user's work flow naturally, because s/he updates it only with the results of the latest trial.

- `propose`: The user calls `propose` to receive a new hyperparameter set to try. Behind the scenes, the tuner first creates a list of candidate hyperparameter combinations. It uses the meta-model to predict scores for each of these combinations. Then, it uses the specified acquisition function to choose which candidate to propose to the user. We modified the existing API so that the user receives the hyperparameter combination in the form of a dictionary. A dictionary is an especially useful way to represent hyperparameters, as users can use it to update the parameters of a scikit-learn model directly.

These two methods are for the end user only. The logic involving adding data to the model (or using the trained model to propose a new candidate) is fixed, and these methods will not be overridden by contributors writing their own tuners. Figure 2-1 shows a code snippet demonstrating how a user might use a tuner.

2.1.3 Contributor API

With the exception of the constructor, the methods that a contributor or developer might override are completely separate from those that a user uses. These methods are described in Table A.4 in Appendix A. We abstracted the data transformation into the `add` and `propose` steps.

For this library, our goal is to get expert contributions. Commonly overridden functions are described as follows:

- `fit`: The `fit` method gets an array of hyperparameter sets that have already been converted into numerical values and their scores. Most modeling methods

```python
tunables = [
    ('n_estimators', HyperParameter(ParamTypes.INT, [10, 500])),
    ('max_depth', HyperParameter(ParamTypes.INT, [3, 20]))
]
tuner = GP(tunables) #instantiate a Gaussian process based tuner
for i in range(10):
    params = tuner.propose()
    # create random forest using proposed hyperparams from tuner
    model = RandomForestClassifier(
        n_estimators=params['n_estimators'],
        max_depth=params['max_depth'],
        n_jobs=-1,,
    )
    model.fit(X, y)
    predicted = model.predict(X_test)
    score = accuracy_score(predicted, y_test)
    # record hyper-param combination and score for tuning
    tuner.add(params, score)
print("Final-Score:", tuner._best_score)
```

Figure 2-1: A Python code snippet illustrating how to use a tuner's user API. The code illustrates how to use a tuner to tune a `number of estimators` hyperparameter between 10 and 500 and `maximum depth` of a tree between 3 and 20 for a random forest. The code snippet is given training data X, y and testing data X_{test} and Y_{test}. The tuner does not require the user to store any scoring data on their end. The user only has to call two methods after initialization, `add` to add new data to the tuner and `propose` to receive a new recommendation from the tuner.

devised by experts require numerical variables. An expert contributor would override the existing fit method to contribute a new meta modeling technique. An example would be fitting a Gaussian process to the hyperparameter score data.

- `predict:` An expert can override the `predict` function in order to use the meta-model to predict the mean score and standard deviation that would be given to a candidate hyperparameter combination. An example would be using the trained Gaussian process to predict the score of a given hyperparameter combination.

- `acquire:` An expert can override the `acquire` method in order to select which candidate to propose given a list of predictions (mean and standard deviation). Two examples would be returning the score with the highest predicted mean, or the one with the highest predicted expected improvement [28].

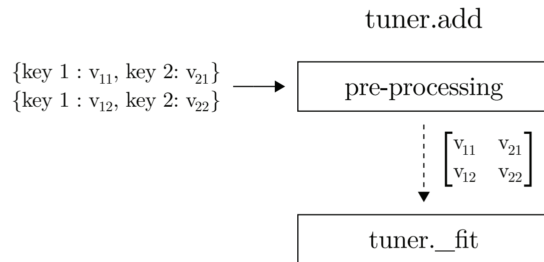

Figure 2-2: Interaction between the user API and developer-overridden methods for the tuner class `add` method.

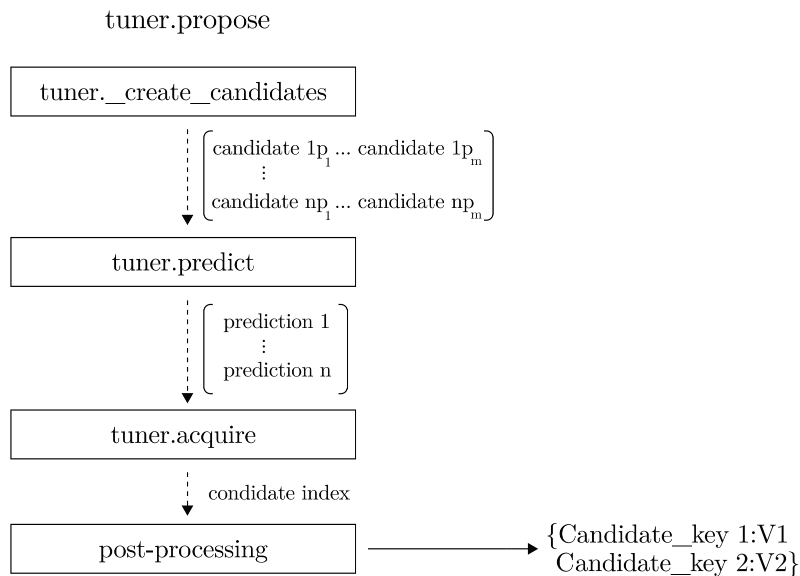

Figure 2-3: Interaction between the user API and developer-overridden methods for the tuner class `propose` method

Figures 2-2 and 2-3 show how the contributor-written methods integrate into the user-facing API for tuners.

Our design enables expert contributions in the following ways:

Flexibility in tuner design: Experts are allowed to contribute new meta-modeling technique. This is very important, because new techniques are constantly being developed and improved upon as the field of AutoML expands. Contributors are also given control of the acquisition function, which is used to propose a solution from a list of predictions. This is also very important, as there is little agreement in the AutoML expert community as to whether one function consistently outperforms another. Figure 2-4 shows how pairing a series of different modeling techniques and acquisition functions can create a series of different tuners. The choice of the Gaussian process (GP) meta-modeling technique versus the Gaussian copula process (GCP) breaks the tuners into two categories. (These ideas are implemented by modifying the tuner's `fit` and `predict` method.)

The tuners are next split based on their acquisition function. The basic GP and GCP-based tuners use the maximum predicted score to determine which candidate to propose. The other tuners use maximizing expected improvement as the acquisition function. This is implemented in the tuner classes by overriding the `acquire` method. Finally, the tuners differ as to whether they include a concept of "velocity" along with the model, only favoring the trained model's predictions over a uniform tuner if the velocity of the scores is large enough. The `GPEiVelocity/GCPEiVelocity` tuners include this concept, and `GPEi` and `GCPEi` do not. This concept is integrated in the tuner by calculating the velocity in `fit`. In `predict`, if the calculated velocity is not above a certain threshold, the model defaults to using a uniform tuner.

Handling data transformations: In this design, the expert contributors don't have to worry about the hyperparameter data types. Most methods that experts want to contribute to require numeric data. We do data transformations and reverse transformations outside the core fit/predict/acquire methods, so that experts that choose to contribute to the methods don't have to worry about how they are handled.

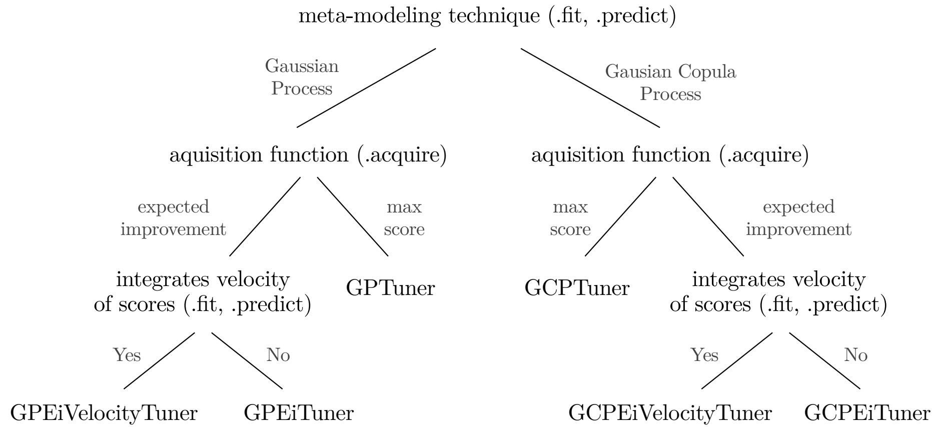

Figure 2-4: Relationship between six different tuners. Includes the developer API functions overridden to implement the tuner.

2.2 Selector

A selector chooses from a set of discrete options by evaluating each of their past performances. Most implemented selector types use a multi-armed bandit technique to determine which option to select next. The selector API was established in the ATM paper, and was not modified during this thesis [29].

2.2.1 User API

- `select`: The user passes a dictionary to the selector's `select` method. Each key in the dictionary corresponds to a choice available to the selector. Each value is a time-ordered list of scores achieved by this choice. The historical score data is first converted to rewards via the `compute_rewards` function. Next, `bandit` is called on the resulting historical rewards data in order to make a selection. The selected option for this time step is then returned to the user.

Table A.1 in Appendix A shows the methods a user would employ to create and use a selector object. The user only has to keep track of a dictionary mapping each of the choices to all of its corresponding scores. Figure 2-5 shows an example code snippet demonstrating how a user could use a selector. Because the choices are represented as keys of the dictionary, the user has a lot of flexibility as to which data type is used for

```python

# Instantiate tuners
rf_tuner = GP(('c', HyperParameter(ParamTypes.FLOAT_EXP, [0.01, 10.0])))
svm_tuner = GP(('n_estimators', HyperParameter(ParamTypes.INT, [10, 500])))

# Create a selector for these two pipeline options
choice_scores = {'RF': [], 'SVM': []}
selector = Selector(choice_scores.keys())

for i in range(SELECTOR_NUM_ITER):
    # Using score data, use selector to choose next pipeline to tune
    next_pipeline = selector.select(choice_scores)

    if next_pipeline == 'RF':
        # give tuning budget Random Forest pipeline
        for i in range(TUNER_NUM_ITER):
            params = rf_tuner.propose()
            model = RandomForestClassifier(n_estimators=params['n_estimators'])
            model.fit(X, y)
            rf_tuner.add(params, accuracy_score(model.predict(X_test, y_test))
        choice_scores['RF'] = rf_tuner.y

    elif next_pipeline == 'SVM':
        # give tuning budget to SVM pipeline
        for i in range(TUNER_NUM_ITER):
            params = svm_tuner.propose()
            model = SVC(C=params['c'])
            model.fit(X, y)
            svm_tuner.add(params, accuracy_score(model.predict(X_test, y_test))
        choice_scores['SVM'] = svm_tuner.y
```

Figure 2-5: Python code snippet illustrating how to use a selector alongside a tuner to choose whether to next tune a random forest or an support vector machine (SVM) classifier. The `n_estimators` is a hyperparameter for a random forest and has the range $[10, 500]$. The `c` is a hyperparameter for SVM and has the range $[0.01, 10.0]$. The code snippet is supplied with training data X, y and testing data X_{test} and Y_{test}. `TUNER_NUM_ITER` specifies the number of iterations of tuner when a particular pipeline (in this case only a classifier) and `SELECTOR_NUM_ITER` specifies the number of iterations for the selector.

each of the choices. This allows for great flexibility on the user's behalf, as they can represent the choices in whatever way they find most convenient.

2.2.2 Contributor API

When writing a new selector, there are three main methods a contributor would override. These methods and their desired functionality are described in Table A.2 in

Appendix A. The contributor can modify the bandit's strategy to favor more recently chosen options, options with the highest velocity of reward, etc. The general selection methodology is as follows: First, the selector converts the scores into rewards, as determined by the `compute_rewards` function. Then, it calls `bandit` on the resulting choice reward combinations to propose a choice.

- `compute_rewards`: The `compute_rewards` method takes as its input a list of historical time-ordered score data and returns a list of rewards. The contributor can decide how the selection option scores correspond to the rewards. For example, if the distribution that the score comes from changes over time, it may not be optimal to consider old score data, as it is unlikely to accurately represent the current distribution. The contributor can then override the `compute_rewards` function to consider the rewards to be non-zero for only the most recent k points.
- `bandit`: In the `bandit` method, a dictionary is received that maps a discrete choice to a list of rewards (the output of `compute_rewards`). The bandit method proposes a choice based on the rewards associated with each of the choices.
- `select`: In the `select` method, the user passes a dictionary to the selector. Each of the keys in the dictionary corresponds to a choice the selector can make, while the values consist of a time-ordered list of the scores that results from each choice. The historical score data is first converted to rewards via the `compute_rewards` function. Next, `bandit` is called on the resulting historical rewards data to make a selection. Finally, the method returns the selected option for this time step to the user.

Figure 2-6 shows how the contributor API interacts with the user-facing API for the select method. Contributors can decide how to pre-process the score data, how to convert the score data into the concept of a reward for the bandit, and which bandit function should be used.

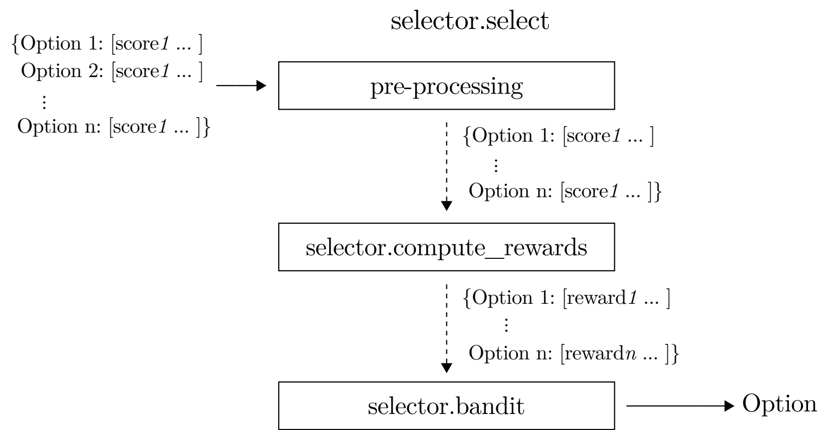

Figure 2-6: Interaction between user-facing API and developer overridden methods for selector class select method

Chapter 3

Recommender

Over the course of this thesis, we created a collaborative filtering-based search process from scratch. We call this a recommender, and describe it in detail in Section 3.3. We designed the API to closely match the tuner API in our library.

3.1 Overview of recommender problem

Suppose we have fitted a number of different machine learning pipelines on a number of datasets, and have recorded the resulting accuracy metric of each pipeline on each dataset. The knowledge gained about the performance of a particular pipeline on several datasets can, in some cases, be transferred to a new dataset.

For example, suppose there is a convolutional neural network that predicts whether a certain image contains an animal. Now, suppose we have the prediction problem of determining whether or not a certain image contains a *dog*. Because there is a similarity between the two problems, the architecture that performed best on the animal problem is likely to have high predictive accuracy for the dog problem. We can generalize this by leveraging the knowledge we have gained from running a series of machine learning pipelines in order to build a system which recommends a new candidate pipeline to try for a given unseen dataset.

3.1.1 Related Work

Multiple research groups have explored the topic of pipeline recommendation. Researchers at MIT have developed Delphi [11], a platform for machine learning that uses a distributed system for trying a series of pipelines and a recommender system to suggest a pipeline to the user. The recommender for Delphi uses a different methodology to make predictions than the one described in this thesis, as it does not use matrix factorization to reduce the space and sparsity of the matrix. The Delphi system is also not extensible, and the recommender methodology can not be changed.

In [14], the authors use probabilistic matrix factorization to infer the data missing from the matrix (including the new dataset for which they wish to propose a pipeline). The recommender acquires the predictions based on expected improvement, and the resulting system is tested on a matrix composed of various pipeline scores on a series of OpenML datasets. This recommender system is used in a series of experiments, and the work released pertains only to the results derived in the paper itself. This is a methodological contribution one which can be added to our open source and be made available to users.

3.1.2 Notation

In this section we will establish notations and naming conventions in use throughout the system. In the recommender system, the goal is to predict which of a candidate set of pipelines, whose hyperparameters are all fixed to certain values, will yield the highest accuracy score. Two pipelines can have the same steps, but may differ only by the hyperparameter values. For example, there can be two different pipelines, each of which use principal component analysis (PCA) on the input followed by a random forest, as long as the two pipelines have different hyperparameter values. This is because different hyperparameter combinations can yield dramatically different results when trained on the same dataset.

At the heart of the recommender system is the matrix that holds the known accuracy scores of given pipelines on given datasets. This matrix will be referred to as

the dataset pipeline performance matrix (dpp). In this matrix, each row represents a unique dataset and each column represents a unique machine learning pipeline. In the matrix, the value at row i and column j is the accuracy score (typically calculated using cross-validation) of pipeline i tried on dataset j.[1] If the pipeline has not been tried on the dataset, the value is 0.

Now suppose we are given a new dataset for which we want a pipeline recommendation. We will refer to this as d_new. For d_new we create a new row in the dpp matrix filled with zeros, to indicate that no pipeline has been tried. Throughout the recommendation process, we are consistently trying new pipelines on d_new and updating its row with the accuracy scores achieved by those pipelines. Each recommender object has only one d_new for which it recommends pipelines. The vector representation of the scores of the pipelines on d_new will be referred to as dpp-vector. Pipeline performance information is stored only in dpp-vector; the dataset is not represented as a row in dpp-matrix. In Section 3.3 we will describe the API, and in Section 3.2 we will describe the methods used for recommending.

3.2 Methods

Sparseness possess a huge problem for the recommender. Due to computational costs associated with fitting a pipeline to a dataset and evaluating it *via* cross validation, it is not feasible to try all of the possible pipelines on all of the known datasets. In fact, each dataset will only have been tried on a small subset of the pipelines, leading to very few non-zero entries in the dpp matrix. For example, in the matrix used in this chapter to demonstrate the efficacy of the recommender, we had a total of 9,384 unique pipelines, and each dataset was only tried on 363 of them.

If the dpp matrix were to make recommendations directly, the results would likely not be optimal, as the algorithm would probably only choose the dataset with the greatest number of pipelines that overlap with the ones that are tried on d_new,

[1]The score can be any metric, but for the purpose of this thesis it refers to the mean f1-score calculated during cross validation.

regardless of the performance of these pipelines. We can solve this problem using matrix factorization (mf). Matrix factorization projects the matrix onto a lower-dimensional space, reducing the dimensionality with the aim of reducing the sparsity. We can then compare the rankings of the values in the lower-dimensional space. Below, we present the current recommender we built.

Matrix factorization-based recommender

1. Use matrix factorization to reduce the dimensionality/sparsity of the `dpp-matrix`. We use non-negative matrix factorization [13, 8] to decompose `dpp-matrix` into a `n-datasets`×`num-desired-components` matrix (`dpp-matrix-decomposed`) as shown:

$$\texttt{dpp-matrix} \approx \texttt{dpp-matrix-decomposed} \cdot H \tag{3.1}$$

where H is some `num-desired-component`s × `num-pipelines` matrix.

2. Gather data about pipeline performances on `d_new` in the form of `dpp-vector`. Similar to `dpp-matrix`, each column j in the 1D vector represents the performance of the pipeline corresponding to that index on `d_new`.

3. Use the trained mf model to project `dpp-vector` on the the reduced dimensionality space yielding `dpp-vector-decomposed`. This can found by solving:

$$\texttt{dpp-vector-decomposed} \approx \texttt{dpp-vector} \cdot H^{-1} \tag{3.2}$$

Where H is the matrix calculated in Equation 3.1, found during the `fit` process of the matrix factorization.

4. In the reduced space, find the matching dataset in `dpp-matrix-decomposed` that is closest to the `dpp-vector-decomposed` as determined by Kendall-Tau distance (ie greatest Kendall Tau Agreement). In case of a tie, choose at random from the tied options.

In order to find the closest matching dataset, we chose to compare the rankings of the pipelines between two datasets, as opposed to the closeness of performance scores. Because we used matrix factorization to reduce the sparsity of `dpp-matrix` and `dpp-vector`, we find the closest matching dataset and compare ranking in the reduced

space. NMF is known to have clustering-like effects, where each cluster is a column in the reduced space and the cluster membership is determined by whichever column has the largest value for the row [22]. By calculating the Kendall Tau agreement in the reduced space, we are essentially comparing the relative-rankings of the reduced pipeline (column) clusters for each dataset.

We compare rankings because we care about predicting the highest-performing pipeline rather than the actual score of this pipeline. The matching dataset may only compare relatively in score performance, as one of the datasets may be much harder to accurately classify than the other. We define the matching dataset as the dataset that has the highest Kendall Tau agreement [18]:

$$\text{Kendall Tau agreement} = \frac{(\text{number agreeing pairs} - \text{number non-agreeing pairs})}{n(n-1)/2} \tag{3.3}$$

The higher the agreement is, the more that the two datasets agree on the comparative ranking of the pipelines. This is important because the actual projected values of the pipeline scores may have widely different ranges. Suppose we have 5 pipelines, numbered 1 through 5, along with 2 datasets, B and C. We are trying to find the dataset that most closely matches dataset A (which has been tried on pipeline 1 and 2). Suppose that dataset A is very similar to dataset C.

Table 3.1: Potential scores for three datasets A, B, C on 5 pipelines 1,2,3,4,5

Dataset	Pipeline 1	Pipeline 2	Pipeline 3	Pipeline 4	Pipeline 5
A	0.5	0.6	0.7	0.8	0
B	0.9	0.7	0.8	0.6	0.5
C	0.5	0.6	0.7	0.8	0.9

Table 3.1 represents a matrix of potential scores for this scenario and Table 3.2 shows the corresponding rankings. For simplicity, we will not use matrix factorization, and will instead compare the rankings in the space of all five pipelines.

If we were to look only at the number of times the raw rank value agrees, dataset

Table 3.2: Score rankings corresponding to table 3.1

Dataset	Pipeline 1	Pipeline 2	Pipeline 3	Pipeline 4	Pipeline 5
A	2	3	4	5	1
B	5	4	3	2	1
C	1	2	3	4	5

B would match dataset A, as they agree 1 time and dataset C and A never agree. Under the Kendall-Tau agreement, however, we look at pairwise agreements. Dataset C and A agree on the following pairs:

- Pipeline 1 $<$ Pipeline 2
- Pipeline 1 $<$ Pipeline 3
- Pipeline 2 $<$ Pipeline 3
- Pipeline 1 $<$ Pipeline 4
- Pipeline 3 $<$ Pipeline 4
- Pipeline 2 $<$ Pipeline 4

giving a total of 6 agreements. Dataset B and A agree on the following pairs:

- Pipeline 1 $>$ Pipeline 4
- Pipeline 3 $>$ Pipeline 5
- Pipeline 2 $>$ Pipeline 5
- Pipeline 4 $>$ Pipeline 5

giving a total of 4 agreements. By the Kendall-Tau agreement, Dataset C is closest to Dataset A. Given the dataset pipeline results, Dataset C is clearly the better option, as the scores match perfectly for every pipeline other than Pipeline 5.

5. Compose a list of candidate pipelines for the new dataset made up of all untried pipelines on the dataset.

6. When making predictions, rank the pipelines based on their rankings for the matching dataset.

7. Acquire the predictions by choosing and proposing the pipeline with the highest ranking. In the case of a tie, chose at random from the options that tie.

3.3 Design

The recommender system attempts to find the best pipeline for a dataset, given a set of candidate pipelines. We have the following requirements:

`dpp-matrix` At the core of the recommender system is the `dpp-matrix`. This must be made available to the system. Each column of the matrix is a different pipeline and each row is a dataset. The value at i, j is the accuracy score for pipeline j on dataset i. It is up to the user to maintain some sort of data structure mapping the column index to the actual pipeline that it represents. This is a valid design decision, as the user must already possess knowledge about the pipeline that the column represents in order to create and fill in the recommender matrix or try a specified pipeline on `d_new`.

Same accuracy score: In order for the recommender system to work, the evaluation metric used for the score must be the same across all datasets. This is because the recommender system works by comparing the scores between different pipelines. If the evaluation metrics differ, the scores have different maximum and minimum values, making score-based ranking of pipelines impossible.

3.4 User API

The API for recommenders matches that of tuners as closely as possible, in order to keep the APIs consistent and because they are used similarly. Users will likely try each pipeline individually. Using the `add` method, users can incrementally add the score for each pipeline, rather than keeping track of all the data (pipelines and scores) and calling `fit` on everything. Table A.5 in Appendix A shows the recommender class methods available to users.

- `add`: The user passes a dictionary to the recommender. The keys in the dictionary are integers representing pipeline indices that the user has tried for their new dataset. The value is the accuracy score for that pipeline on their dataset. This method is very similar to the `add` method for a tuner. The recommender updates

```
1 #m = Matrix of dataset-pipeline-performances
2 #num_iterations = Number of iterations
3 recommender = Recommender(m)
4 for i in range(num_iterations):
5       pipeline_index = recommender.propose()
6       score = result of trying pipeline index on dataset
7       recommender.add({pipeline_index, score})
8 best_pipeline_index = recommender._best_pipeline_index
```

Figure 3-1: Python pseudo-code snippet illustrating how to use a recommender's user API

its internal state with the new pipeline-score data and refits the meta-model on the new updated data. No external data storage is required on the user side.

- `propose`: The user calls `propose` to receive a new pipeline index proposed by the model that may improve the score. As with the tuner, the recommender first creates a list of candidate pipelines that haven't yet been tried on the given dataset. It uses the meta-model to predict scores for each pipeline. Next, it uses the specified acquisition function to select which of these candidates to propose to the user. While this method involves a lot of computations, the API is kept easy-to-use for the user. It naturally fits into an iteration structure where the user can receive a pipeline index from the recommender, try it and score it, and then update the recommender with its score.

Figure 3-1 shows an example pseudo-code describing how the user would use the recommender system. As we can see, the API is very clean and easy for users to use. The user also has easy access to the best pipeline and its score, which are stored as class attributes. Minimal code is required on the user's part in order to integrate a recommender system.

3.5 Contributor API

Table A.6 in Appendix A shows the recommender methods that contributors who wish to write their own recommender can override. As with the user API, the contributor API closely matches that of the tuner, so that experts who have contributed to one

type of object have a minimal learning curve when contributing to another. This also helps keep the library consistent. The following is a list of recommender methods that the contributor can override.

- `fit`: The contributor overrides the `fit` process to fit the meta-model to both the `dpp-matrix` data composed of known pipeline and score combinations along with a vector of pipeline scores on `d_new`. We abstracted this functionality into its own method to match the general fit/predict API that is commonly used by machine learning libraries. This way, contributors can simply implement their meta-modeling technique as a model and call `model.fit` in the recommender API. One example would be fitting a matrix factorization model to the `dpp-matrix` and using that to reduce the dimensionality of both the `dpp-matrix` and the pipeline-performance vector of `d_new`.

- `predict`: The contributor overwrites the `predict` function to use the meta-model to rank a series of candidate pipelines. As with `.fit`, we abstracted `predict` to its own function to match the scikit-learn practice of calling `model.fit` and `model.predict`. The contributor uses their meta-model to make predictions. An example would be using the relative rankings of the dataset whose pipeline scores most closely match the `d_new`. We chose to have the `predict` function rank the candidates rather than assigning them a raw score, because while similar problems may be useful for predicting relative performance, they are less likely to accurately predict the true numerical performance of a pipeline on a dataset.

- `acquire`: The contributor overwrites the `acquire` method to select which candidate to propose given a list of predictions. An example would be returning the pipeline that ranks the highest according to its prediction.

Similar to the tuner, the public methods `add` and `propose` are fixed and not meant to be overridden by contributors. This is because they include data processing and other nuances abstracted away from the contributor. The API for the contributor

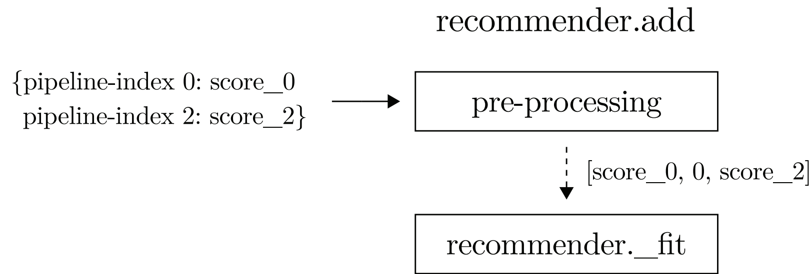

Figure 3-2: Interaction between user API and developer-overridden methods for recommender class add method

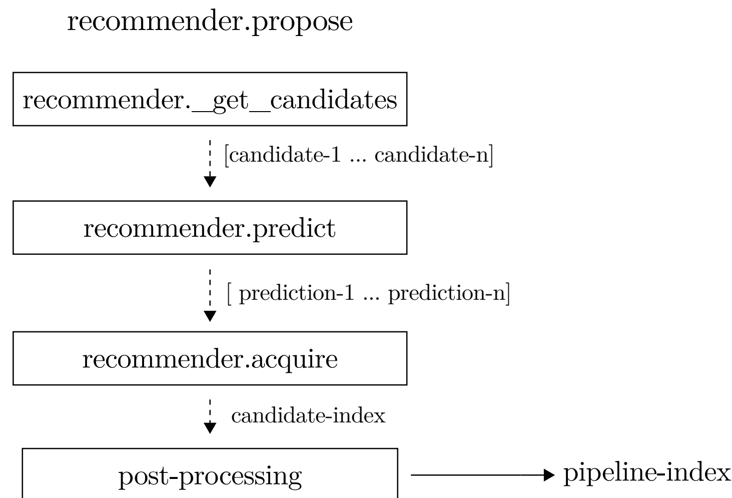

Figure 3-3: Interaction between user API and developer-overridden methods for recommender class propose method

is designed to be as simple as possible. The contributor deals only with an array of scores for fitting. Figures 3-2 and 3-3 show how those methods that a contributor can override interact with the methods that the user calls for a recommender.

The contributor does not have to override all of the methods. The default `get_candidates` and `acquire` functions are likely sufficient for most recommenders. The default `get_candidates` returns all pipelines that haven't yet been tried on the new dataset. The default acquisition function returns the index corresponding to the maximum ranking. At a minimum, contributor should override `fit` to specify how the model should be fit to the `dpp-matrix`, and `predict` to specify how it should use the fitted model to rank different pipelines.

3.6 Evaluation

We evaluate the mf-based recommender against a uniform sampling method that proposes a pipeline chosen at random from the candidate list of pipelines for each iteration.

3.6.1 Data

Creation of `dpp-matrix`: In order to generate the dataset-pipeline performance matrix, we used data gathered by a series of ATM runs using grid search to find the best pipeline for many different datasets[2]. 420 datasets were used in this experiment. Each dataset was run on roughly 600 different pipelines [3].

The data from these runs was stored in four csv files. The schema of these csv files is described in the documentation here: https://github.com/HDI-Project/ATM/blob/master/docs/source/database.rst.

In the raw ATM results file, the pipeline information was stored as in a string with the model type (eg SVM, random forest) and all of the hyperparameter values for the model. We first aggregated all of the pipelines by model type and hyperparameter configuration. This helped us identify 19,389 unique pipelines. We assigned each unique pipeline an index and then grouped the pipelines by dataset ID. There were 286,577 dataset-pipeline runs to begin with.

We further reduced the pipelines to a subset consisting of ones that ATM currently supports, so that when a specification of the pipeline is given, we are able to learn a model and predict using scikit-learn. We ended up with 9,384 pipelines.

After this selection, we found that each of the 420 datasets had been tried on an average of 363 different pipelines. We mapped each dataset ID to a list of pipelines and scores. From there, we constructed the `dpp-matrix` by filling in dataset and pipeline performance pairs of the `dpp-matrix`. Where the dataset-pipeline performance is unknown, the matrix is filled in with zeros.

[2]The raw data from the runs is available at https://s3.us-east-2.amazonaws.com/atm-data-store
[3]We are calling these pipelines, but they only consisted of a classifier

Ultimately, the matrix we have constructed is about 96% sparse, as each dataset has been tried on approximately 4% of the pipelines. This matrix is publicly available at https://s3.amazonaws.com/btb-data-store/recommender-evaluation/dpp-matrix/

Evaluation datasets: We subselected a set of 30 datasets. Table 3.3 gives the list of these datasets. For each of these datasets, we will make recommendations and evaluate the performance of the mf-based recommender system.

Table 3.3: List of datasets used in evaluation of matrix factorization-based recommender.

Dataset Name	Dataset Name
mf_analcatdata_asbestos_1	analcatdata_bankruptcy_1
AP_Endometrium_Prostate_1	auto_price_1
breast-tissue_1	chscase_health_1
diggle_table_a1_1	fri_c0_100_10_1
fri_c1_100_25_1	fri_c1_100_50_1
fri_c2_100_25_1	fri_c2_100_50_1
fri_c4_100_25_1	fri_c4_100_50_1
fri_c4_100_100_1	fri_c4_250_100_1
humandevel_1	machine_cpu_1
meta_batchincremental.arff_1	meta_ensembles.arff_1
meta_instanceincremental.arff_1	pasture_1
pasture_2	pyrim_1
rabe_166_1	sonar_1
visualizing_ethanol_1	visualizing_slope_1
fri_c0_250_50_1	witmer_census_1980_1

Experiment Setup: With the `dpp-matrix` and evaluation datasets at hand, we evaluate a mf-based recommender as follows:

1. We choose 30 random datasets (named in Table 3.3) as our candidate datasets, for which we want to find the best possible pipeline[4].

2. For each dataset, we:

[4]The train-test splits for these datasets are available at https://s3.us-east-2.amazonaws.com/atm-data-store/grid-search/

(a) Remove the row corresponding to this dataset from the dataset-pipeline performance matrix. We pretend that this is a new dataset, `d_new`.

(b) Use the remaining matrix to fit a recommender model.

(c) Use the recommender to propose a pipeline index for this dataset.

(d) Based on the pipeline index, get the string representation of the pipeline. Parse this to create and train a machine learning model on the dataset using ATM's `Model` class.

(e) Score the model based on the mean f1-score from 5-fold cross validation.

(f) Add the score to the matrix at the i, j location, where i corresponds to the dataset index and j is the pipeline index.

3. Continue repeating steps (a) through (e) until we have proposed and evaluated 50 pipelines. These correspond to 50 iterations.

4. Store all of the recommendation/score data for later evaluation.

Together, steps 2, 3, and 4 make up one `trial`. In order to assure confidence in the results and assess statistical significance, we run 20 trials on each dataset. Each experiment is made up of 20 trials run on a particular dataset.

3.6.2 Storing the results

We record the results of the experiments for each dataset-recommender pair in a separate csv file. The first row in the csv file is the first pipeline proposed by the recommender for an experiment. The second row is the resulting average f1-score of this pipeline on the dataset after 5-fold cross validation. The third row is the standard deviation of the f1-score for the pipeline calculated from the cross-validation. The columns continue in this manner for each of the trials. Each row in the csv represents an iteration along with the results for all of the trials at that iteration. Table 3.4 shows how this csv format translates into a table of results. Storing the results in this format means that we can easily use the Pandas Python library [20] to turn the CSV

file into a DataFrame for computations. As we are storing the mean and standard deviation of the f1-score for each pipeline on the dataset, we have a lot of flexibility around how we analyze and use the results.

Table 3.4: Storage of Results of recommender Evaluation

trial_0_ pipeline_ index	trial_0_ mean_f1	trial_0_ std_f1	trial_20_ pipeline_ index	trial_20_ mean_f1	trial_20_ std_f1
trial_0_ pipeline_ index_ iteration_0	trial_0_ mean_f1_ iteration_0	trial_0_ std_f1_ iteration_0	trial_20_ pipeline_ index_ iteration_0	trial_20_ mean_f1_ iteration_0	trial_20_ std_f1_ iteration_0
trial_0_ pipeline_ index_ iteration_ 50	trial_0_ mean_f1_ iteration_ 50	trial_0_ std_f1_ iteration_ 50	trial_20_ pipeline_ index_ iteration_ 50	trial_20_ mean_f1_ iteration_ 50	trial_20_ std_f1_ iteration_ 50

We use a standard naming scheme of `recommender-typ_dataset-name_results.csv` to make it easy to find the results for a specific experiment. The results are stored in a public S3 bucket [5]. This open-sources our results, and allows others to use the data that we have generated for their own experiments.

3.6.3 Evaluation of Results

We compare our matrix factorization recommender against a uniform recommender that proposes each pipeline with equal probability.

For an experiment on a dataset d that has a recommender r, we define the `best-so-far` metric $\texttt{pbest}_\texttt{i}{}^d$, or the score of the best-performing pipeline by iteration i as:

$$\texttt{pbest}_\texttt{i}{}^\texttt{d} = \sum_t \frac{max_{0 \leq j \leq i} f(p_i^t, d)}{t} \tag{3.4}$$

where $f(.)$ is the cross-validated accuracy score when pipeline p_i^t is evaluated on

[5]https://s3.amazonaws.com/btb-data-store/recommender-evaluation

dataset d. The accuracy metric used in our experiments is f1-score. p_i^t is the pipeline proposed by the recommender at iteration i in trial t for this dataset. Thus, $\texttt{pbest}_{\texttt{i}}{}^d$ gives the best score achieved by the recommender at iteration i, averaged across all the trials.

Evaluation Metrics

In order to compare two recommender methods, we compare their $\texttt{pbest}_{\texttt{i}}{}^d$ scores using a series of metrics.

- **Average $\texttt{pbest}_{\texttt{i}}{}^d$ at iteration 5, 10, 25, 50:** We take the mean of all the experiments (across different datasets) for each recommender type. Because the same accuracy metric (in our case, the f1-score) is used across datasets, this average effectively compares the overall performances of the two recommenders.

- **Wins for average $\texttt{pbest}_{\texttt{i}}{}^d$ at iteration 5, 10, 25, 50:** We also count the number of time that each method "won." To do this, we take each dataset at a specified iteration, check which $\texttt{pbest}_{\texttt{i}}{}^d$ is higher, and simply tally the "wins" for each method.

- **Average percentage difference in $\texttt{pbest}_{\texttt{i}}{}^d$ at iteration 5, 10, 25, 50:** In addition to comparing the mean $\texttt{pbest}_{\texttt{i}}{}^d$, we also calculate the mean percentage difference in $\texttt{pbest}_{\texttt{i}}{}^d$, in order to quantitatively measure any performance increase. This metric enumerates any potential performance gain that would result from using a matrix factorization recommender instead of a uniform recommender. We calculate the percentage difference in $\texttt{pbest}_{\texttt{i}}{}^d$ for a dataset between uniform and matrix factorization recommender ($1 - \frac{\texttt{mf-pbest}_{\texttt{i}}{}^d}{\texttt{uniform-pbest}_{\texttt{i}}{}^d}$), and then average over all datasets.

- **90% Confidence interval around average $\texttt{pbest}_{\texttt{i}}{}^d$ for iteration 50:** By leveraging the scores from all of the trials, we can calculate a 90% confidence interval around the calculated $\texttt{pbest}_{\texttt{i}}{}^d$ at iteration 50 for a given dataset. We then average this across all 30 datasets. By comparing the size of the confidence

interval, we can gauge how consistent the system is when predicting and how reliable the results are for the final iteration.

Statistical Significance: As we are only comparing two recommender, we chose to use a Wilcoxon signed-rank test to determine the significance of our results [16]. The result of this test will allow us to accept or reject the null hypothesis that the results for the matrix factorization and uniform recommender come from the same distribution. For the $\texttt{pbest}_{\texttt{i}}{}^{d}$ and 90% Confidence Interval $\texttt{pbest}_{\texttt{i}}{}^{d}$, we apply a Wilcoxon significance test to the results of these metrics on each dataset for the two recommenders.

3.7 Results

Table 3.5: Maximum `p_best`, Number of Wins and C.I. Lower Bound Results from Recommender Evaluation on 30 Datasets; Each Experiment Ran 20 Times

Metric	Max `p_best`				# Wins				Lower Bound of 90% C.I.
iteration	5	10	25	50	5	10	25	50	50
Uniform	0.66	0.70	0.74	0.77	18	1	0	0	0.75
Matrix Factorization	0.66	0.73	0.77	0.79	12	29	30	30	0.78

Table 3.6: Performance Increase Percentage Results from Recommender Evaluation on 30 Datasets; Each Experiment Ran 20 Times

Metric	% Performance Increase			
iteration	5	10	25	50
Uniform	N/A	N/A	N/A	N/A
Matrix Factorization	+2.4%	+6.8%	7.6%	5.1%

Table 3.5 shows the results of comparing the uniform and matrix factorization recommenders, which were evaluated on 30 datasets. The matrix factorization recommender always outperforms uniform. The matrix factorization, on average, leads to

Table 3.7: Statistical Analysis of Results from Recommender Evaluation on 30 Datasets; Each Experiment Ran 20 Times

Metric	Max `p_best`				Lower Bound of 90% C.I.
iteration	5	10	25	50	50
Wilcoxon p-value	0.81	3.9×10^{-6}	1.7×10^{-6}	1.7×10^{-6}	1.9×10^{-6}
Wilcoxon statistic	221	8.0	0	0	1.0
Significant	No	Yes	Yes	Yes	Yes

5.4% performance increase in the maximum f1-score with its proposed pipelines. After running a Wilcoxon signed-rank test, we get a significantly small p-value <0.05; thus, we can conclude with high probability that the results are statistically significant.

When comparing results across iterations, we can see that 10 iterations is approximately the point at which using a matrix factorization recommender pays off. At 5 iterations, because there is not enough pipeline performance data to aid in predicting good pipelines, the matrix factorization recommender is configured to propose random candidates. Between 10 and 25 iterations, the results are marginally better, and by the 50th iteration the results are almost 10% better. We ran a signed Wilcoxon significance test on the $\texttt{pbest}_\texttt{i}{}^d$ results for each dataset between each pair of these iterations and found that the differences in results were all significant (from iteration 5 to 10: $p-value$ =1.73×10^{-6} , statistic=0.0; from iteration 10 to 25: $p-value$ =1.73×10^{-6} , statistic=0.0; from iteration 25 to 50: $p-value$ =1.73×10^{-6} , statistic=0.0).

We compare the lower bound of a 90% confidence interval of the maximum f1-score of a pipeline proposed during an iteration of the experiment. Similar to the average maximum f1-score proposed by the recommender, the matrix factorization recommender performs about 4% better than uniform. As we can see, when we compare the lower bond of the confidence interval to the average (0.77 to 0.79), the results are very consistent and we are very confident that the average maximum f1-score will be around 0.79.

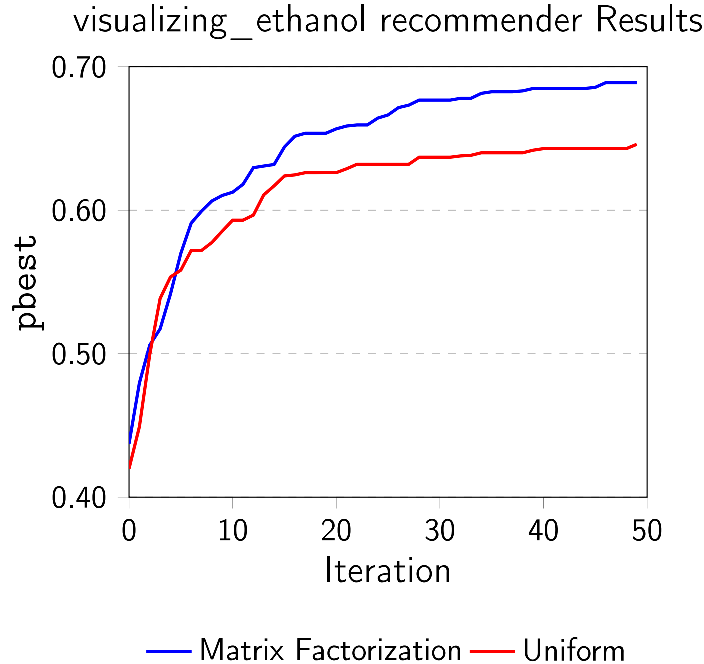

Figure 3-4: `pbest` at different iterations for matrix factorization and uniform recommenders on *visualizing_ethanol_1* dataset

3.7.1 Performance Graphs

Figure 3-4 shows the `pbest` (across trials) for specific iterations during an experiment for the *visualizing_ethanol* dataset. On this dataset, the matrix factorization recommender significantly outperforms the uniform recommender. For the first 5 iterations, the `pbest` between the matrix factorization recommender and the uniform recommender are very comparable. The `pbest` for both recommenders grows rapidly in the first few iterations as each chooses randomly. After a certain number of iterations, it becomes likely that the matrix factorization-based recommender will propose a pipeline that will perform well on the dataset. By iteration 50, there is a difference of about 0.05. This also explains why the uniform model performs quite well in general: It can achieve reasonably high accuracy without needing to learn anything about the matrix of pipeline performances.

3.7.2 Improvement Upon the Results

According to our evaluation metrics, the matrix factorization recommender significantly outperformed the uniform-based recommender. While the results were good, the dataset pipeline performance matrix is still very sparse (only around 3.8% complete) and we believe that a denser matrix would help improve the performance of the matrix factorization recommender.

During our experiments, we saved the f1-scores of each pipeline that was tried on a dataset, along with the pipeline index and the dataset. We can use this data to fill in more of the `dpp-matrix`, to further improve upon the results that the recommender system gives. We added the data from our initial experiment results, a uniform recommender tried on about 50 datasets and the matrix factorization recommender tried on about 30 datasets.

When we used these results to fill in the `dpp-matrix`, the average number of pipelines that each dataset has been tried on grew from approximately 363 to 515, which increased the density of the matrix by about 50%. After enough iterations of the evaluation have passed, the matrix will be much denser, even though there will likely be overlap between the proposed pipelines.

We chose not to integrate dataset-pipeline-performance results during the experiment, in order to keep the results consistent over multiple trials of the same experiment. This would make the trials non-independent, as later trials would use information from earlier trials.

Chapter 4

Current usage of BTB

In order to demonstrate the BTB library's usability and flexibility, we cite its use in three different systems. The first system, DeepMining, uses a tuner to tune a given machine learning pipeline for a dataset and parallel processing to evaluate pipelines faster. The second system, MIT TA2,[1] uses a tuner to generate the best machine learning pipeline possible for a specific dataset within a time limit. To make sure a pipeline is always present, the system uses a tuner to generate progressively better pipelines until a set time. The third system, ATM, uses both tuners and selectors to pick the best classifier for a dataset. The system runs in parallel, with multiple workers using tuners at the same time. In this chapter we will describe each of these systems, briefly highlighting how they use BTB.

[1]MIT TA2 is a system developed by MIT and Feature Labs for a DARPA D3M program [27]

Figure 4-1: An illustration of how data flows through a system that uses a tuner, such as DeepMining or TA2, which shows how score and hyperparameter information is shared within the system.

4.1 DeepMining

DeepMining is an open-source, end-to-end system that allows users to easily compose and tune machine learning pipelines for a specified dataset.[1]. The system is publicly available at GitHub.[2] Users specify a dataset, composed of labeled data, and a pipeline, composed of an ordered set of operations that they wish to tune. These operations can be standard library functions (for example, fitting a $sklearn.ensemble.RandomForestClassifier$), or custom functions, such as a user-written Gaussian blur function. For each step in the pipeline, the user specifies the hyperparameters, as well as their respective ranges. Users can also use any of a series of pipelines pre-written by the library maintainers, such as a traditional image pipeline. Once they specify their dataset and pipeline, DeepMining tunes the specified hyperparameters in order to maximize the performance of the resulting pipeline. The library also uses a variety of scoring methods, including Bag of Little Bootstraps, that use sampling methods to estimate the score of the pipeline.

Integrating with BTB: The original DeepMining library already contained logic for searching over the hyperparameter space in order to tune the pipeline [2]. However, this logic was buried inside the search class and not modularized. Switching the tuning algorithm (e.g. from a Gaussian process to a Gaussian copula process) was not easy, and required working around arguments only relevant to the tuning process throughout multiple parts of the system. Likewise, adding a new tuning methodology was complex and required rewriting core DeepMining functions.

DeepMining was refactored in the following ways:

– Logic was added to create, update, and fit pipelines, and to predict using a fitted pipeline to a separate library called `MLBlocks`.

– Tuning logic was removed and replaced with BTB integration. Inside the searching logic, one needs to call `tuner.propose()` to get a candidate set of hyperparameters, and `tuner.add()` to add the hyperparameters sets and their score.

[2]https://github.com/HDI-Project/DeepMining

```
1 def score_hyperparams(pipeline, hyperparams):
2      pipeline.set_hyperparameters(hyperparams)
3      score = pipeline.score()
4      return hyperparams, score
5
6 tuner = Tuner()
7 pipeline = pipeline object
8 num_parallel = number of parallel pipelines to score
9 for i in num_iterations:
10     candidates =  tuner.propose(num_parallele)
11     candidates, scores = map candidates to score_hyperarams( ↩
     pipeline, candidate)
12     tuner.add(candidates, scores)
13 tuner._best_score # score
14 tuner._best_hyperparams #hyperparameters
```

Figure 4-2: Python pseudo-code snippet illustrating how DeepMining uses BTB to score hyperparameter combinations in parallel.

– DeepMining allows users to score pipelines in parallel. BTB is flexible enough to accomodate this– it can be specified to propose multiple candidate sets at a time, which can then be tried in parallel. The resulting data can then be added back to the tuner, after which the search continues.

BTB helps to greatly simplify the DeepMining process's searching logic, and using BTB makes it easy to switch tuning algorithms. If new tuners are added to BTB, users can implement them in DeepMining without altering any of the DeepMining code.

Optionally, the tuner can be used with gridding in order to ensure that each combination of hyperparameters is only tried once. This prevents the tuner from getting stuck and proposing the same combination for multiple iterations. It also works in parallel: the candidates are created as a group, and the tuner ensures that none of them have been previously tried by checking whether they share the same grid spot with any of the pre-tried hyperparameter combinations. Each of the hyperparameter combinations is then tried in parallel. The hyperparameter/score combination is returned, and the results are concatenated together in a simple map/reduce. `tuner.add` is called with the list of candidates and scores, which adds them to the training data and refits the tuner. In this way, no concurrency issues arise. Figure 4-2 demonstrates how DeepMining uses BTB.

4.2 MIT TA2 System

The MIT TA2 system, which was built collaboratively by MIT and FeatureLabs,[3] is MIT's submission to DARPA's Data-Driven Discovery of Models Program, or D3M. D3M aims to allow non-experts to quickly and easily create machine learning models [27]. Systems will be given an unknown data set and a problem and must build the best possible pipeline solution on the fly. This competition breaks the problem into three tiers. TA1-level efforts create state-of-the-art primitives, including classifiers, regressors, imputers, scalers, and featurizers. TA2 is an end-to-end system which, when given a dataset of an unknown type, will construct the optimal pipeline using TA1 primitives, and produce a pipeline that can make predictions on new data points. TA3 teams focus on building an intuitive and useful UI on top of the TA2 system. The BTB library was integrated with MIT's TA2 system.

Importance of Tuning: In the D3M competition, the MIT TA2 system will be given only a dataset and a problem type (i.e. classification, regression, etc.). The system must determine the data type and construct the best possible pipeline for that combination on the fly. As the system is only allowed a certain amount of computational resources and running time, a tuner is very useful in this situation, because it can continue to tune the hyperparameters of the chosen pipeline until time runs out. As we do not know anything about the dataset ahead of time, and datasets can vary widely, it is impossible to pick a good hyperparameter combination for each pipeline that would work on all possible datasets. For example, for a random forest classifier run on a single table of features, a certain number of trees might be ideal for one dataset, but cause the model to overfit for another. In situations like this, the use of a tuner is critical to ensure that the pipeline performs as well as possible.

Differences from DeepMining: The TA2 system has slightly different goals than DeepMining. DeepMining gives the user complete control over the pipeline, the scoring methodology, the parallelization, and the tuning method. TA2, on the other hand, has to determine all of this by itself, with a constrained set of computational resources

[3] https://www.featurelabs.com/

```python
pipeline = Pipeline()
max_score = 0
tuner = Tuner(pipeline.get_hyperparameter_ranges())
for i in num_iterations:
    tuner.add(scores, hyperparameters)
    hyperparameters = tuner.propose()
    pipeline.set_hyperparameters(hyperparameters)
    pipeline.fit(x, y)
    score = pipeline.cv_score()
    if score > max_score:
        max_score = score
        yield pipeline
```

Figure 4-3: Python pseudo-code snippet illustrating how MIT TA2 uses BTB to yield the best possible pipeline over a series of iterations.

and in a limited amount of time. Because of this, MIT TA2 keeps each pipeline that outperforms the previous best pipeline, as the system may time out before the next tuning iteration completes. This is critical especially for deep learning-based pipelines, as training and cross-validation take a long time. In these situations, the tuner may not get through very many iterations before the timeout is reached.

Integration with BTB: Our current version of MIT-TA2 is similar to DeepMining in that it has a pipeline class that exposes hyperparameters and their ranges. Here, all pipelines are hard-coded with the logic that creates and tunes them. The evaluation metric is chosen based on the specifications given, and the pipeline is trained and evaluated accordingly.

LSTM Pipeline Example: A pipeline is composed of multiple steps, and how a single tuner can be used to tune the hyperparameters of all the steps simultaneously. In order to demonstrate this, we now describe an LSTM[15] pipeline for classification/regression on a text dataset used by the MIT TA2 system.

- **Steps and architecture**: We used a Python library called Keras [7] to implement the neural network based on an existing architecture for text classification/regression [5]. Figure 4-4 shows the overview of the architecture.

  - The first step is padding each of the texts so that they are the same length.
  - Next, we embed the texts into a vector by using the top N words. (For example, the most common word is a 1, second most common is 2, etc.)

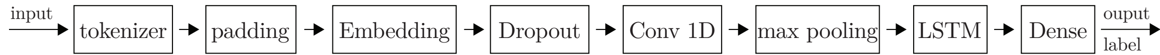

Figure 4-4: LSTM Pipeline Data Flow Diagram

  - We transform the text into a vector by letting the ith word be the number that word corresponds to (if it is in the top N words), or else 0.
  - Now that we have featurized the text, we can use the resulting feature vector to train a neural network. The neural net is composed of 5 layers. The first layer is a dropout layer, to prevent overfitting. The second layer is a one-dimensional convolutional layer, followed by a max pooling layer mapping the region of text to general trends. The result is inputted into an LSTM block. Finally, the LSTM block's output is fed into a densely-connected layer which outputs the prediction for the input.

- **Hyperparameters**: As shown in Table 4.1, there are hyperparameters for different steps in the pipeline. BTB is completely abstracted away from the role of particular tunables in the pipeline, and only focuses on finding the value of each that will maximize the score. Thus, we can use BTB to tune multi-step pipelines. In this example, `num_top_words` belongs to the first step in pipeline, the tokenizer. The `embedding_size` is used in the third step of the pipeline, the word embedding. `dropout_percent` is used in the first layer of the neural net, to specify what percentage of the neural network units should be dropped out. The `conv_kernel_dim` specifies the size of the convolutional kernel for the one-dimensional convolutional layer in the neural net. `pool_size` is used for the max pooling layer of the neural network.

This pipeline was evaluated on the 30_*personne* dataset, which is a binary classification problem. The resulting model had a 0.619 f1-score. When pipeline performances were compared, it was tied for second place out of 11 teams.

Table 4.1: Table of the LSTM Text pipeline's tunable hyperparameters, ranges, and description.

Hyperparameter	Type Range	Description
num_top_words	Int [1000, 40000]	The number of top words to use for the tokenizer preprocessing.
embedding_size	Int [100, 500]	The size of the feature vector after embedding.
dropout_percent	Float [0.1,0.75]	The percentage of the training data that should be dropped in the dropout layer.
conv_kernel_dim	Int [3,10]	The size of the kernel for the 1D convolutional layer.
pool_size	Int [2, 10]	The number of units to do a max pooling over.

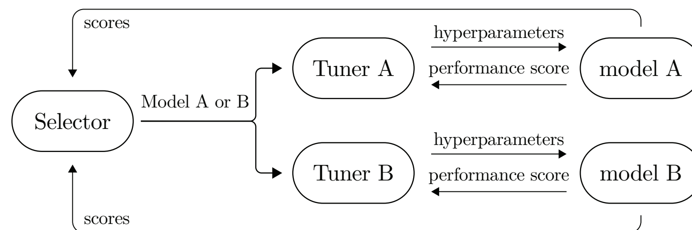

Figure 4-5: Diagram showing how information including scores, hyperparamters, and choices flows through ATM.

4.3 ATM

ATM implements a distributed, collaborative, and scalable system for automated machine learning [29]. The library [4] is open source and allows users to select among and tune machine learning pipelines in parallel. We showcase it here to describe how selectors, along with tuners, can be used in practice.

Overview of System

The goal of the ATM system is to select and tune a classifier from a series of potential classifiers in order to maximize the model score. Each potential classifier has a set machine learning model type and a set of fixed hyperparameters, along with a set of tunable hyperparameters, each of which has a specific range.

[4] https://github.com/HDI-Project/ATM

ATM uses a selector to determine the allocation of resources. Each of the classifiers is tuned by a different tuner, and each of the tuners is tuned by a different worker instance. Thus, multiple different classifiers are being tuned in parallel with different workers. The results are reported back to ATM and stored in a database. This example shows the versatility of the BTB library, as it can be used in parallel by multiple workers. Figure 4-5 shows a higher-level view of how ATM works.

Chapter 5

Open Source Preparation

BTB was released publicly along with ATM [19]. Before fully open-sourcing BTB and allowing for contributions from outside developers, we fleshed out the system in order to ensure maximum functionality.

5.1 Increased Functionality

The goal is for BTB to be adopted widely and eventually regarded as the top library for hyperparameter tuning and model selection. In order to achieve this goal, BTB must be able to tune a wide variety of hyperparameter types.

5.1.1 Handling Categorical hyperparameters

In its previous state, BTB supported both numerical and categorical hyperparameters. However, it could only tune the numerical values, and would propose the same categorical value over and over.

Difficulties: Adding support to tuning categorical hyperparameters poses non-trivial challenges. While these parameters come from a discrete space by definition, they must be mapped onto a continuous space in order for most tuners to work properly.

A naïve solution would to be to create a list of the possible categorical hyperpa-

rameters and map their value to the list's index, turning each categorical value into a numerical value. However, a categorical hyperparameter's values are definitionally independent of each other, and when modelling this space using a Gaussian process tuner, we assume dependence between them. For example, the assumption that a good performance by the categorical hyperparameter at index 4 would indicate a likelihood of a good performance at index 3 or 5 would be invalid, as the categorical hyperparameters can be in an arbitrary order. Thus, this approach would yield less-than-optimal results.

Transforming Categorical Values in Numerical Space: We represent each of the values of a categorical hyperparameter using the average of the scores given for the specific hyperparameter value. This means we have a valid reason to assume that values located near each other in this new numerical space will perform similarly. Table 5.1 gives an example of how to represent the categorical values. It also shows how the tuner-proposed values are mapped back to their categorical hyperparameter.

Table 5.1: Example showing how the values of a categorical hyperparameter can be mapped to a numerical representation and back

Categorical Value	Model Score
True	0.5
False	0.4
True	0.7
False	0.3

Categorical Value	Numerical Representation
True	0.6
False	0.35

Tuner Numerical Choice	Categorical hyperparameter Value
0.7	True
0.1	False
0.5	True

In this representation, if each of the categorical values is mapped to the average

of the score for that value, each value is closest numerically to the categorical value that averaged the most similar score. Therefore, if each categorical value is close to similar-scoring values, we have a valid ordering on the hyperparameters, and we can assume values near each other in the numerical space will perform similarly. While the variance of the other parameters also influences the score, the assumptions that we can make about predicting the score for a given hyperparameter value are much more valid than the previously stated naïve approaches.

5.1.2 Abstraction of Logic for Hyperparameter Transformation

In order to keep the tuner class as clean as possible and reduce bloat, we abstracted out the logic for hyperparameter transformation. Now each hyperparameter type has its own class, within which it specifies how it should be transformed and inverse-transformed. Certain types of hyperparameters must be transformed before a meta-modeling technique can be fit to the data. For exponential hyperparameters, for example, we search not for the hyperparameter value, but rather for the correct exponent. We map each hyperparameter to its logarithm, and search for the right value over that space. For categorical variables, we use the technique described above to map the hyperparameters to numerical values to search over.

API: Table A.7 in Appendix A shows the API for the `Hyperparameter` class. The API uses the same type of `fit_transform` and `inverse_transform` functions that scikit-learn uses. This makes it easy for new users to learn, as they have seen the design pattern before. Each `Hyperparameter` type is responsible for how the type should be mapped to a numerical space to search over or be modeled during the tuner's fit and predict process. It is also responsible for determining how to take a numerical value and map it back to a the hyperparameter it represents. Lastly, tuners can be used with gridding in order to avoid trying the same or extremely similar sets of hyperparameter combinations multiple times. Each hyperparameter

is responsible for determining how to map its search space to a specific number of points.

Extensible: If contributors want to add support for new hyperparameter types, or add a new methodology for transforming a hyperparameter to the numerical space, the `Hyperparameter` class is easily extensible – the contributor only has to define the constructor (based off a template), `fit_transform`, and `inverse_transform`. This allows contributors to easily add their own way of dealing with categorical hyperparameters, for example.

These hyperparameter objects are used by the tuners. The overall flow of tuners was changed so that `add` and `propose` don't need to be modified by contributors creating their own tuners. The `add` and `propose` methods deal with taking in/returning the hyperparameter in a human-readable format (dictionary mapping names to values). These two methods deal with transforming and inverse-transforming the hyperparameters, and passing the transformed values to fit and predict. They are completely abstracted and can be treated interchangeably – internally, a contributor who wishes to write their own tuner only has to deal with the hyperparameters as vectors of numbers. As tuner algorithms are generally just complicated statistical models, this is very useful, as the contributor doesn't have to worry about what to do with strings, Booleans, or other data types, and can focus on applying the best statistical formulae.

5.2 Testing

In order to prepare for outside contributions, we built out a rigorous testing environment. As we cannot merge a contributor's code without first ensuring it doesn't break anything in the library, there must be a series of unit tests that assess the functionality of the entire library. At each pull request, the test suite can be run in order to ensure that the proposed change or addition does not break any existing functionality. We built out unit tests for each of the classes and subclasses, to ensure that there was full code coverage over the entire library. We integrated Travis Continuous Integration

[1] with the library, which provides feedback about whether or not each pull request breaks any of the tests. It is run on every commit, and provides insurance that the master branch is always stable.

5.3 Examples

In order for users to find implementation simple, they must understand how to use the library and why it is helpful. In addition to documentation explaining this, we added three different examples on which users can base their own implementations.

The first example uses a tuner to show how we can tune the parameters of a Rosenbrock function[2] to find its minimum value. This example shows how tuners can be useful outside of the machine learning community. Additionally, since this problem requires searching for a minimum and the tuner always tries to maximize the output, we show how the tuner can still be used in this case. By negating all of the scores, we turn the minimization problem into a maximization problem.

The second example involves tuning the hyperparameters of a random forest trained on the MNIST dataset [10]. MNIST is a popular machine learning training dataset, composed of handwritten digits labeled with their values. This example illustrates how tuning can be used on to boost accuracy on any machine learning pipeline, even without a full end-to-end system.

Our final example combines a tuner and a selector. This example uses a selector to determine whether to tune a random forest or a SVM pipeline next. Once again, this model is trained on the MNIST dataset. This example shows how combining a selector with a tuner can make the end result even more powerful. Additionally, these examples show how integrating a tuner/selector into an existing system, or building one from scratch, is not complicated and does not require a lot of code.

[1] https://travis-ci.com/

[2] A Rosenbrock function has a hard-to-find global minima residing in an easy-to-find valley [23].

Chapter 6

Getting contributions from experts

While the existing BTB package includes a set of powerful pre-written methods, in order for the library to remain state-of-the-art, it must allow for contributions from AutoML and ML experts. Making the library amenable to such contributions is more complicated than most traditional Open Source projects, and has inspired the following additional goals:

Goal 1: Ensure that it is easy to write new methods We make sure that it is easy for contributors and experts to write new methods by giving them the following three things:

- **Clear well scafolded abstractions for contributions:** Using a well-defined API, we expose the key methods we imagine experts are most likely to want to change. For example, experts in AutoML often disagree about the meta-modeling technique and the acquisition functions. For each of the search techniques, we have broken the process down into methods, which we imagine they want to vary, with clear documentation of their inputs and outputs. They can also base their new methods off of a standard template.

- **Preprocessing:** So that experts can better focus on developing and creating new versions of key methods, we have abstracted away all the preprocessing of tunable hyperparameters. The methods they want to override only require simple numeric data and data structures they are familiar in using. We provide

all the additional routines, such as handling, searching for the scale over an exponential range, or transforming categorical variables into numeric values using a specific transformation.[1]

- **Easy to use contribution integration framework:** We make use of existing GitHub [2] infrastructure to enable contributions, and provide clear instructions on how to contribute. Contributors can integrate new methods in the form of pull requests. GitHub comments can be used if any additional follow-up is needed from the contributor. This process is transparent to maintainers, users, and contributors alike.

Goal 2: Maintain and ensure package stability To maintain package stability as new contributions come along, we have implemented the following additional frameworks:

- **Using a continuous integration testing framework:** Before a contributor's pull request containing a new tuner is integrated, the entire suite of tests for the package is run on the pull request. This will prevent users from writing a tuner that somehow breaks a different section of the package.

- **Naming Conventions:** Contributors must follow specific naming conventions when updating methods. For example, if they wish to contribute a tuner, they name their new tuner method as MyTuner.py and put it in the the tuning folder.

- **Unit Testing:** Contributors are required to write a basic series of unit tests based off of a template in order to test the specific behavior of the newly contributed method. They are also required to put them in MyTunerTest.py and in the tuning subdirectory of the test folder.

- **API Compliance Testing:** We envision a general-purpose API testing functionality that would test the functionality of all methods and their compliance.

[1] We imagine some experts can contribute new transformers as well.
[2] https://github.com/

Goal 3: Ensure code quality for new tuners: This is best achieved by having a set of standards and guidelines for code quality and style for any and all contributions. This can be achieved with two techniques: enforcing a single style guide over the entire packages and any contributions to it and using an automated code review tool for any and all contributions to ensure code quality.

For the BTB package, we chose to enforce the PEP8 style guide for Python [26]. This style guide ensures that code is readable, as it claims in general that code is read more than it is written.

Goal 4: Be able to evaluate the performance of contributions: A comprehensive evaluation mechanism is vital for attracting expert contributions. Unlike regular software engineering additions, method-related contributions require answering an additional question: "How good is this new method at achieving the end goal?" (In our case, this goal is optimizing machine learning pipelines.) Typically, experts create a new method, evaluate it extensively across several datasets, and summarize them in a paper [12] [4]. We imagined that this could be provided as a service, and that the workflow would look as follows:

- Experts create a new core method, test it locally on different datasets and do a pull request.
- After the pull request, we would test their contribution using CI, and for API compliance.
- We can then optionally review their code and provide feedback to ensure code quality.
- Once these three requirements are fulfilled, we will run a comprehensive evaluation across several datasets, and provide a comprehensive summary of how well their method performs against the existing ones in the library. This is completely automated and standardized.

We envision that enabling automatic evaluations across several datasets would motivate the experts – who, more often than not, are not software engineers – to go

through what may seem like an arduous process in order to contribute.

The first three goals are standard software engineering practice. For the rest of this chapter, we will focus on evaluations of a newly contributed methods and how they work, using the example of a newly contributed tuner.

6.1 Evaluating methodological contributions

In this section, we will use methodological contributions for tuners as an example. When an expert contributes a new tuner, we need to be able to compare it to existing tuners to determine whether or not it is better. The motivation for such a rigorous evaluation framework is twofold. One, proper evaluation would allow end users of BTB to determine which tuner is best to use. Two, knowing whether or not their tuner actually improved performance will motivate contributors, as well as provide good feedback.

Why is this difficult: It's easy to compare a pair of tuners for a specified dataset and machine learning pipeline – in fact, this is the sort of comparison contributors can do locally while evaluating their methods. However, choosing one particular dataset is not enough. At the same time, it is not enough to choose several datasets but *only* one machine learning pipeline. Difficulties in tuning usually come from the relationship between the dataset and the search space defined by the pipeline. For example, deep neural nets are known to be quite difficult to tune, as they require near-perfect hyperparameter values in order generate decent results. The classification accuracy of the resulting model can vary quite widely depending on the hyperparameters with which it was tuned. While this problem makes these model types ideal candidates for tuners, some of the hyperparameters for these models are also very sensitive. Learning rate, for example, often must be determined by choosing the right order of magnitude (ie 0.0001 or 0.1). The model is very sensitive to the value of the learning rate, though, and choosing a less than ideal learning rate often means the accuracy of the resulting model is capped quite low, even if all other hyperparameters are chosen perfectly.

6.1.1 Setting benchmarks

To enable a comprehensive evaluation, we create benchmarks that combine a set of datasets and a set of machine learning pipelines.

Dataset Choice: We have selected a set of 20 datasets shown in Table 6.1. We downloaded these from the OPENML platform and preprocessed them. They are all classification datasets. The train-test splits we use are available publicly on Amazon S3[3]. We have hundreds of datasets at the S3 location, and we can add more in the future. .

Table 6.1: List of datasets used in tuner evaluation

Dataset Name	Dataset Name
analcatdata_asbestos_1	analcatdata_bankruptcy_1
chscase_health_1	diabetes_numeric_1
diggle_table_a1_1	humandevel_1
rabe_166_1	visualizing_ethanol_1
visualizing_slope_1	witmer_census_1980_1

Pipelines: We chose two standard models from the scikit-learn library: MultiLayer Perceptron (MLP) and K-Nearest Neighbors (KNN). These models are relatively quick to train and evaluate compared to deep neural models. Each has some hyperparameters that are fixed and some that can be tuned. In order to make sure that the tuner cannot exhaustively try all combinations or randomly choose a near-perfect combination, there are at least 1000 possible hyperparameter combinations for each problem. As the number of choices grows exponentially with number of hyperparameters, we choose to have two free hyperparameters. Two hyperparameters would make predicting the score for a model more difficult to model than for one hyperparameter, but would allow a grid search over all combinations to finish in a reasonable amount of time. Table 6.2 lists the two models and the hyperparameters that we searched over during the evaluation, with their ranges.

Exhaustive Grid Search: First, we ran an exhaustive grid search using a uniform

[3]https://s3.us-east-2.amazonaws.com/atm-data-store/

Table 6.2: The model type and tunable hyperparameters for the search problems used in tuner evaluation

Multilayer Perceptron	
Hyperparameter	Value
hidden_size_layer1	[2,300]
alpha	[0.0001, 0.009]
KNN	
Hyperparameter	Value
n_neighbors	[1,20]
leaf_size	[1,50]

tuner with gridding to try all possible hyperparameter combinations in the search space. We scored each combination using the mean result of a 5-fold cross validation and used the f1-score as the evaluation metric. Scoring all possible hyperparameter combinations is useful for two reasons. First, it prevents the need to train and evaluate ML models during the tuner evaluation. Not having to train a full ML pipeline on each iteration of search during the evaluation will save a large amount of time, and will reduce the need for large computation servers during the evaluation step. While the tuner's meta-model may still take a non-trivial amount of time to `fit` and `predict`, overall evaluation time is substantially reduced. This also keeps the model performance for a certain candidate set of hyperparameters constant, so that tuners that propose exactly the same hyperparameters will score exactly the same.

Second, it allows us to rank the hyperparameter combinations proposed. Being able to compare a hyperparameter combination's performance to all possible combinations allows us to use a series of more interesting evaluation metrics, as opposed to just score. We can rank the hyperparameter combinations, which allows us to tell how the tuners' candidates performed against the absolute best possible hyperparameters. We choose to run the evaluation with 50 grid points on each axis, so that there are 1000 possible hyperparameter combinations for the KNN model and 2500 possible hyperparameter combinations for the MLP model.

6.1.2 Evaluation Metrics

Before we describe the various evaluation metrics, we describe the names we use to denote different aspects of the evaluation framework.

Search problem	A search problem is a combination of a dataset i and a pipeline p with a given set of hyperparameters $\alpha_1 ... \alpha_k$. We use s to denote a search problem.
Iteration	An iteration consists of one execution of a tuner over the search problem. In each iteration, a set of hyperparameters is proposed by the tuner and the pipeline is fitted and evaluated for a given metric using cross validation.
Trial	A trial is an end-to-end run of a tuner for a fixed number of iterations for a given search problem.
Experiment	An experiment is a set of repeated independent trials conducted for a search problem.

Basic measures: We calculate two basic measures per search problem, at different iteration numbers.

Best so far score, `mbest`: We define this score at the jth tuner iteration for a search problem s as:

$$\texttt{mbest}_j^s = \sum_t \frac{max_{0<z\leq j} f(d_i, \alpha_{1_z} ... \alpha_{k_z})}{t} \tag{6.1}$$

where $f(\dot{)}$ is the cross-validated value of the metric we are trying to optimize. Thus $\texttt{mbest}_j^s$ is the best possible score we achieved for this search problem at the jth iteration of the tuner, averaged across many trials. This is a fundamental unit of scoring of a tuner over which we can derive several statistics across several search problems (datasets+pipeline combinations). We call this `mbest` and use the notation $\texttt{mbest}(a, s, j)$ where j is index into the iteration number, a is the tuner and s is the search problem.

Rank, `rank`: We also evaluate the standing of the best-so-far achieved by the tuner relative to scores achieved for the search problem using grid search. The ranking of the `mbest` at j^{th} iteration for a search space is given by:

$$\texttt{rank}_{\texttt{j}}^{\texttt{s}} = \sum_t \frac{|i| - \sum_i U(\texttt{mbest}_{j,t}^s - \texttt{grid}_i)}{t} \tag{6.2}$$

where $\texttt{grid}_i$ is the score of the ith solution discovered by the grid search, U is the unit step function which is equal to 1 if $\texttt{mbest}_{j,t}^s \geq \texttt{grid}_i$ and $|i|$ is the total number of solutions generated during grid search. The metric above is averaged across several trials of the tuner for the search problem. This evaluation metric lets us determine exactly how close the tuner was to the best possible solution from the grid. This metric makes the differences between tuners on "easy" problems more pronounced, as all of the tuners might have had roughly 0.9 accuracy, but their ranks will differ by at least 1. Additionally, this metric standardizes scores across different search problems, as it compares the score to the maximum possible for the problem given the grid search, not the score's maximum possible value.

Statistics across multiple search problems: Given that we can calculate a best-so-far estimate and rank at any iteration for any search problem, we next derive statistics to compare pairs of tuners. Each of these statistics can be calculated at any iteration number, giving us a way to compare tuners working with different budgets.

Statistical Significance: In addition to knowing whether results differ between tuners, it is also important to know if the results are statistically significant. This will prevent random chance from influencing our choice of the best tuner. Given that we run each tuner for a number of search problems (in our case, 20 problems from 10 datasets, each tried on two pipelines), we can run a significance test for all of the tuners.

Based on the findings of [16] and [9], we know that comparing a new algorithm to a series of already-tried algorithms can be statistically difficult due to multiplicative comparison errors. We follow the suggestions laid out in these papers to determine statistical significance. We first compute a non-parametric Friedman test for each

evaluation metric over all of the tuners. This test shows us whether or not the differences in a given metric for any of the tuners is statistically significant. If this test fails, it is likely that the results are drawn from a similar distribution, meaning that none of the tuners truly outperform any of the others.

If the test succeeds, we still don't have any information about which tuner is performing significantly better. In this case, we follow up with a Bonferroni-Dunn post-hoc procedure that compares the results of the new tuner to each of the other tuners. This pairwise test will let us determine whether the difference between the two tuners' performance is significant. This test will likely give us more interesting results than the Friedman test, as we can then conclude (with high probability) that, for example, a tuner with an expected improvement acquisition function (GPEi) outperforms a tuner a with a simple maximum acquisition function (GP).

Given these two statistics, we compare two metrics:

- **Comparing rank**: At an iteration j, we can accumulate the ranks achieved by tuner A and B across multiple search problems and compare them.
- **Comparing best score**: At an iteration j, we can accumulate the best-so-far achieved by tuner A and B across multiple search problems and compare them.

6.1.3 Results of an experiment

Experiment Setup: Once we have grid search on each of the datasets for each of the ML pipelines, we will evaluate the tuners on each of these problems and evaluate the results based on their performance on each of the evaluation metrics. The experiment for evaluating a tuner is as follows for each dataset/ML pipeline combination:

1. Create a new tuner of the type we are testing.
2. Loop:
   (a) Use the tuner to propose a candidate set of hyperparameters.
   (b) Use the results from grid search to retrieve the calculated mean f1-score from 5-fold cross validation.

(c) Add the hyperparameter and score combination to the tuner.

3. Continue a-c until we have proposed 100 hyperparameter combinations.

4. Store all of the tuner-proposed hyperparameters and score data for each iteration for later evaluation.

In order to ensure confidence in the results and assess statistical significance, we run each of the outlined trials 20 times for each search problem. This will allow us to ensure that the results are consistent over multiple trials, similar to Section 3.6.3 for recommenders.

To test our evaluation procedure, we conducted an experiment with 10 different datasets, each tried on two pipelines. We compared the performance of seven tuners: a uniform tuner, three Gaussian process-based tuners, and three Gaussian copula process-based tuners (within each set of Gaussian tuners, there is one that acquires based on maximum prediction, one that acquires based on expected improvement, and one that acquires based on the velocity of the expected improvement). For each metric, we evaluate the performance that comes from a tuning budget of 25, 50, and 100 iterations. The results show that there is *not* a large performance increase between any of the tuners.

When the tuners are limited to few iterations, the results are not significant, meaning that none of them perform better than random (as benchmarked by the uniform tuner). Over more iterations, the GP tuner seems to do better than many of the other tuners, but in general, there does not appear to be a tuner that significantly and consistently outperforms the rest.

One interesting result to note is the number of times that the uniform tuner won. While we would expect this number to be extremely small or zero for larger-budget evaluations, a non-zero number of wins suggests that some of these problems were sufficiently difficult that modeling the hyperparameter space does not help and can actually hinder performance. An example distribution that could be extremely hard to model is shown in Figure 6-1. This invites further work regarding creating tuning methods and algorithms to allow the tuner to model this difficult distribution.

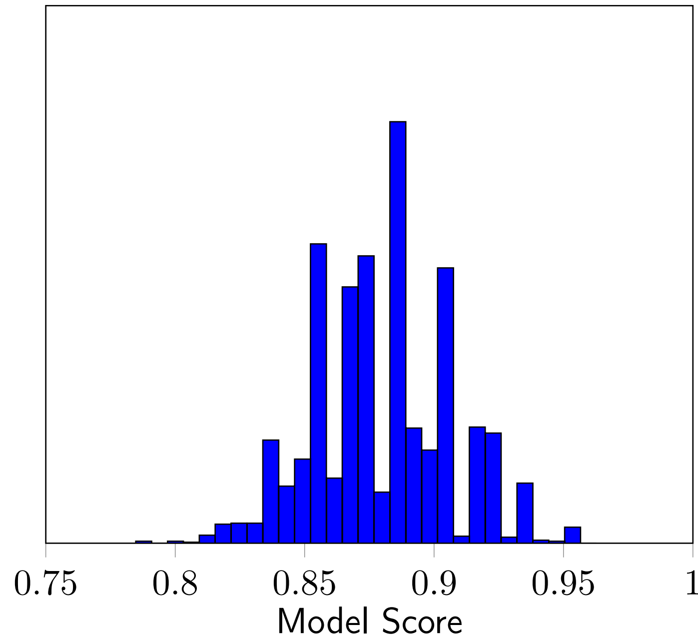

Figure 6-1: Histogram of a search problem with a difficult distribution. While it is easy to perform well on this search space, it is hard to model the distribution to consistently outperform a uniform tuner. This search problem is searching the KNN hyperparameters described in Table 6.2 on *housing_1* dataset.

Figures 6-3 and 6-2 and show the performance of different tuners on a “difficult” search problem (where a small number of hyperparameter combinations yield high f1-scores) for the f1-score of the best proposed hyperparameter combination by a certain iteration, and rank the best hyperparameter combination proposed (0 being the highest scoring combination, and N being the lowest scoring where there are N possible combinations). As we can see, the Gaussian process-based tuner performs the best for this problem.

6.2 Automation

We would like to automate the process of reviewing and merging contributions as much as possible. The code for a new tuner can require complicated statistics and algorithms, making it challenging and time consuming for a maintainer to methodically check the code quality of each contribution. If we rely too heavily on maintainers

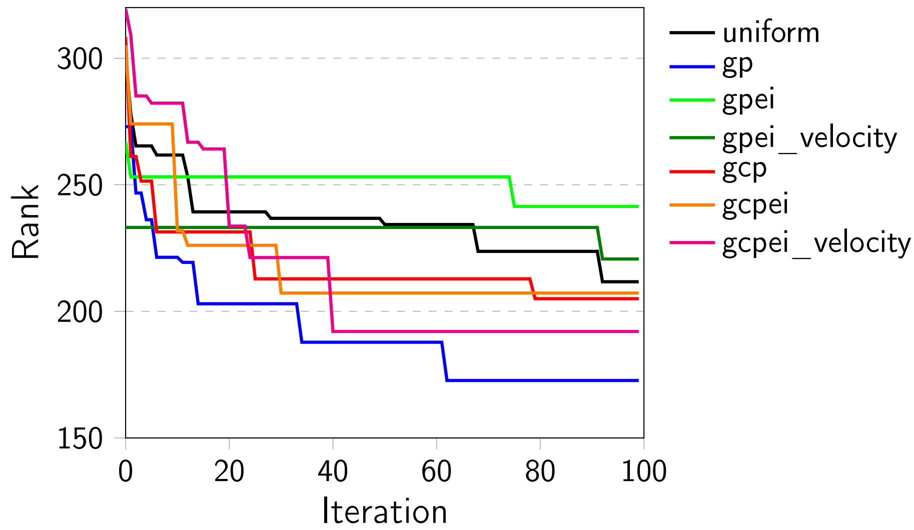

Figure 6-2: Comparison of the best ranking hyperparameter combination given a MLP pipeline on `chscase_health_1` dataset. The grid for this search problem had 1000 different hyperparameter sets.

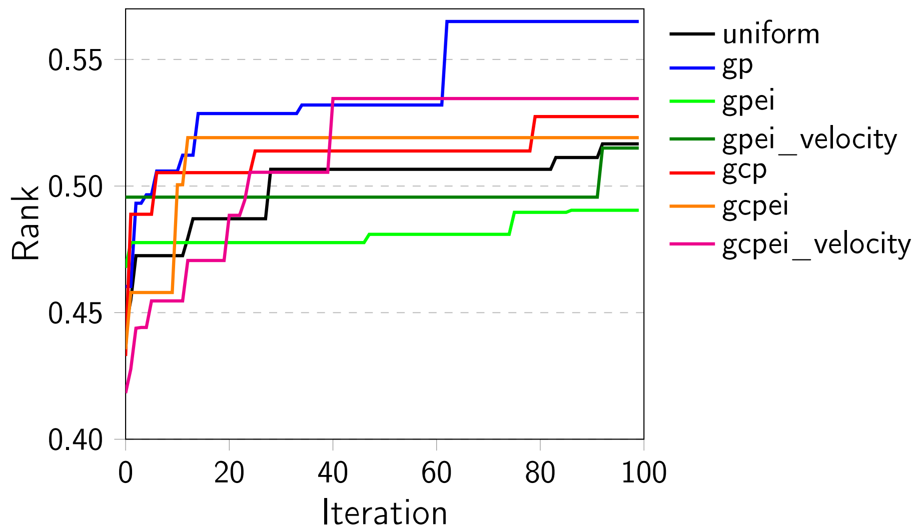

Figure 6-3: Comparison of the best mean f1 score yielded by a pipeline tuned for a given the KNN pipeline and the `chscase_health_1` dataset.

to review new contributions manually, it may take a very long time for them to be integrated into the library. However, it is important to maintain a high standard for code quality in the library, as this prevents bugs and allows for the consideration of edge cases that weren't previously considered.

We will use Codacy, an automated code analysis tool,[4] to help automate this process. We can configure Codacy with our chosen PEP8 style guide, and set standards for the contributions. Codacy will then automatically comment on contributors' pull requests, noting any style guide violations, unnecessary complexity, or other components that indicate low-quality code. The contributor can then fix the violations and update the pull request. Because these comments point out the exact line and violation, adhering to the style guide becomes very low effort for the contributor. It also makes things easier for the maintainer, as they can see the public comments that Codacy generated, and ensure that the final contribution adheres to the style guide. Additionally, Codacy will provide maintainers with tools to monitor the code quality of the entire library, allowing them to easily see which sections need work, or if a specific contributor continues to submit low quality tuners.

6.3 Workflow for creating and merging a new method

1. A contributor forks a BTB repository and develops a new tuner. While developing it, he has a subset of datasets and a evaluation script he can use to evaluate his tuner against others.

2. The contributor creates a pull request to the Master branch of BTB repository with their tuner code and includes a high-level description of the tuner in the pull request.

3. Travis CI and Codacy comment on the pull request with any detected issues.

4. The contributor revises the code and updates the pull request until all issues are solved.

[4] https://www.codacy.com/

5. The maintainer briefly reads through code to detect any tuner violations and comments with any issues if found.
6. The contributor addresses the comments and updates the pull request.
7. Once accepted, the maintainer downloads the pull request.
8. The maintainer runs a tuner evaluation in BTB-evaluate on the new tuner.
9. The maintainer merges the pull request into BTB and updates the relevant docs with the new tuner's evaluated performance.

This workflow ensures that the code standards are met while requiring minimal effort from the maintainer. The first part of the code review involves only Codacy, Travis CI and the author of the pull request. By the time it gets to the maintainer, basic functionality and code style and quality have been ensured. The maintainer only has to look at the code at a high level, determining that the user workflow with the new tuner will match that of all the other tuners. For example, if a contributor created a tuner that required the user to call a pre-process step on their data before `add`, the maintainer would not merge it, in order to ensure that the API is consistent over all tuners. As long as there are no API violations, the contributor can evaluate the tuner using BTB-Evaluate and merge the pull request along with the evaluated performance.

Chapter 7

Conclusion

7.1 Key Findings and Results

Through this thesis, we found that with the proper abstractions, we could devise a library that was both intuitive for data scientists to use and easy for AutoML experts to contribute to. We demonstrated this ease of use by integrating this library into multiple systems. We devised a recommendation algorithm that, when implemented in a recommender system and trained for a sufficient number of iterations, leads to on average a 5.1% performance increase over a uniform proposal method. We created an evaluation methodology to evaluate the quality of tuners. We found that the many tuners achieved a similar performance, and that further work could be done to create tuners that are better able to meta-model the hyperparameter space.

7.2 Contributions

In this thesis, we

1. Revised and finalized the apis for BTB objects.
2. Extended the existing BTB library to add support for recommendation problems.
3. Evaluated our recommender against a baseline uniform recommender.

4. Cleaned up the code repository and extended functionally to prepare the library for open-sourcing.

5. Designed a system for the automatic testing and evaluation of contributed code quality.

6. Designed a system for the automatic evaluation of BTB tuners in order to compare tuner performance.

Appendix A

Tables

Table A.1: User API for selectors

Methods	Parameters	Returns	Description
`__init__`	`choices`: a list of discrete choices from which the selector must choose	-	Initializes the selector object with the choices the selector must select between.
`select`	`choice_scores`: map of `{choice -> [scores]}` for each possible choice.	`choice`	Uses multi-armed bandit to select the next choice to maximize the total reward.

Table A.2: Developer API for selectors

Methods	Inputs	Returns	Description
`__init__`	`choices`: a list of discrete choices from which the selector must choose	-	Initializes the selector object with the choices the selector must select between.
`select`	`choice_scores`: map of `{choice -> [scores]}` for each possible choice.	`choice`	Uses multi-armed bandit to select the next choice to maximize the total reward.
`compute_rewards`	`scores`	`list of rewards`	Convert a list of scores associated with one choice into a list of rewards.
`bandit`	`choice_rewards`: mapping of choices to rewards that indicate their historical performance.	`choice`	Uses multi-armed bandit to return the choice that we should make next in order to maximize expected reward in the long term.

Table A.3: User API for tuners

Methods	Parameters	Returns	Description
`__init__`	`tunables`: ordered list of hyperparameter names and metadata objects. `gridding`: number of points on each axis on the grid.	N/A	Instantiates the tuner object.
`add`	`x`: dictionary of hyperparameter combinations. `y`: list of scores of the hyperparameter combinations.	None	Adds all hyperparameter/score data to the model and then refits the model on all of the data.
`propose`	`n`: number of combinations to propose.	List of dictionaries of proposed.	Uses tuner to propose a set of hyperparameters to try in order to maximize the score.
Attributes	-	Value	-
`_best_hyperparams`	-	Dictionary of tunables and their values.	-
`_best_score`	-	Score	-

Table A.4: Developer API for tuners

Methods	Parameters	Returns	Description
`__init__`	`tunables`: ordered list of hyperparameter names and metadata objects. `gridding`: number of points on each axis on the grid.	N/A	
`fit`	`x`, `y` matrix of hyperparameters and their scores.	None	Fits the model to the numpy representations of hyperparameters.
`predict`	`x`: matrix of hyperparameters	`y`: scores	Uses the model to predict scores for new hyperparameter candidate combinations
`_acquire`	`predictions`: array of predicted scores.	Index of acquired prediction.	Given a list of predictions, acquires list to one prediction.
`_create_candidates`	`n`: number of candidates to create.	`n` candidate hyperparameter combinations,	Creates a list of candidates hyperparameters to chose from.

Table A.5: User API for recommenders

Methods	Parameters	Returns	Description
`__init__`	`dpp_matrix`: matrix of dataset pipeline performance matrix.	-	Constructs the recommender object.
`add`	`data`: dictionary mapping pipeline indices to the accuracy value when tried on the dataset.	None	Adds all data points from data to the model and then refits the model on all of the data.
`propose`	None	Index of pipeline	Uses recommender system to propose the next pipeline (index) to try in order to maximize the score.
Attributes	-	Value	Description
`_best_pipeline`	-	Index of pipeline	Class argument of index of best scoring-pipeline tried on dataset.
`_best_score`	-	Score	Class argument of best score of pipelines tried on dataset.

Table A.6: Developer API for recommenders

Methods	Parameters	Returns	Description
`fit`	`X`	None	Fits the recommender model given the data vector X where index i in X is the score of pipeline i on the new dataset, or 0 if not tried.
`predict`	`indices`: a list of pipeline indices	A ranking of indices	Ranks indices based on predicted performance of pipeline index on the new dataset.
`acquire`	`scores`: a list of pipeline scores	Index	Based on predicted scores, returns the index into scores that maximizes the acquisition function.
`get_candidates`	`X`	Indices	Returns the candidate pipeline indices to try on X.

Table A.7: API of Hyperparameter BTB Class

Method	Parameters	Returns	Description
`fit_transform`	`x`,`y`	x	Fits (if necessary) the hyperparameter transformation model to `x` and then returns transformed `x`.
`inverse_transform`	`x`	x	Inverse transforms `x`.
`get_grid_axis`	`grid_size`	list	Returns grid points for hyperparameter based on range and grid size.

Table A.8: Results from tuner evaluation on 20 20 search spaces; Each experiment was run 20 times

Metric	Max Score				Std Max Score				Min Rank				Std Rank				# Wins			
iteration	10	25	50	100	10	25	50	100	10	25	50	100	10	25	50	100	10	25	50	100
Uniform	0.79	0.83	0.85	0.87	0.12	0.12	0.11	0.11	69.53	31.19	17.70	9.54	60.32	26.37	15.32	8.52	2	2	2	3
GP	0.80	0.84	0.86	0.88	0.12	0.11	0.11	0.10	66.70	28.41	15.71	7.79	65.68	26.61	15.74	7.82	4	6	10	9
GPEi	0.79	0.82	0.85	0.87	0.13	0.12	0.11	0.11	76.69	32.53	18.68	9.58	68.69	23.25	12.98	7.34	3	1	3	4
GPEiVelocity	0.79	0.82	0.85	0.87	0.13	0.12	0.11	0.11	72.19	35.18	18.13	10.50	58.88	27.50	19.50	9.92	3	1	3	2
GCP	0.80	0.83	0.86	0.88	0.12	0.12	0.11	0.11	69.96	35.57	18.57	8.00	66.25	34.60	17.53	8.27	6	5	3	6
GCPEi	0.79	0.83	0.85	0.87	0.13	0.12	0.11	0.11	76.09	34.17	17.91	9.59	60.86	24.51	14.90	8.41	2	2	4	4
GCPEiVelocity	0.79	0.83	0.85	0.87	0.13	0.12	0.11	0.11	72.22	33.26	18.75	9.38	59.88	29.75	17.94	8.09	2	3	3	4

Table A.9: Statistical analysis of results from tuner evaluation on 20 search spaces; Each experiment was run 20 times

Metric	Max Score				Min Rank			
iteration	10	25	50	100	10	25	50	100
Friedman p-value	0.09	0.14	0.04	0.01	0.42	0.04	0.02	0.00
Friedman Statistic	10.97	9.56	13.08	13.08	6.03	13.14	14.88	18.68
Significant	No	No	Yes	Yes	No	Yes	Yes	Yes

Bibliography

[1] Alec W Anderson. "Deep Mining: Scaling Bayesian Auto-tuning of Data Science Pipelines". MA thesis. Massachusetts Institute of Technology, 2017.

[2] Alec Anderson et al. "Sample, estimate, tune: Scaling bayesian auto-tuning of data science pipelines". In: *Data Science and Advanced Analytics (DSAA), 2017 IEEE International Conference on*. IEEE. 2017, pp. 361–372.

[3] James S Bergstra et al. "Algorithms for hyper-parameter optimization". In: *Advances in neural information processing systems*. 2011, pp. 2546–2554.

[4] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (Oct. 2001), pp. 5–32. ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324.

[5] Jason Brownlee. *Sequence Classification with LSTM Recurrent Neural Networks in Python with Keras*. July 2016. URL: https://machinelearningmastery.com/sequence-classification-lstm-recurrent-neural-networks-python-keras/.

[6] Lars Buitinck et al. "API design for machine learning software: experiences from the scikit-learn project". In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122.

[7] François Chollet et al. *Keras*. 2015. URL: https://keras.io.

[8] Andrzej Cichocki and Anh-Huy Phan. "Fast local algorithms for large scale nonnegative matrix and tensor factorizations". In: *IEICE transactions on fundamentals of electronics, communications and computer sciences* 92.3 (2009), pp. 708–721.

[9] Janez Demšar. "Statistical comparisons of classifiers over multiple data sets". In: *Journal of Machine learning research* 7.Jan (2006), pp. 1–30.

[10] Li Deng. "The MNIST database of handwritten digit images for machine learning research [best of the web]". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[11] Will D Drevo, Kalyan K Veeramachaneni, and Una-May O'Reilly. *Distributed, multi-model, self-learning platform for machine learning*. US Patent App. 14/598,628. May 2016.

[12] Manuel Fernández-Delgado et al. "Do we need hundreds of classifiers to solve real world classification problems". In: *J. Mach. Learn. Res* 15.1 (2014), pp. 3133–3181.

[13] Cédric Févotte and Jérôme Idier. "Algorithms for nonnegative matrix factorization with the β-divergence". In: *Neural computation* 23.9 (2011), pp. 2421–2456.

[14] Nicolo Fusi and Huseyn Melih Elibol. "Probabilistic matrix factorization for automated machine learning". In: *arXiv preprint arXiv:1705.05355* (2017).

[15] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. "Learning to forget: Continual prediction with LSTM". In: (1999).

[16] Magdalena Graczyk et al. "Nonparametric statistical analysis of machine learning algorithms for regression problems". In: *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*. Springer. 2010, pp. 111–120.

[17] F Hutter et al. "Automatic Machine Learning (AutoML)". In: *ICML 2015 Workshop on Resource-Efficient Machine Learning, 32nd International Conference on Machine Learning*. 2015.

[18] M. G. Kendall. "A New Measure Of Rank Correlation". In: *Biometrika* 30.1-2 (1938), pp. 81–93. URL: http://dx.doi.org/10.1093/biomet/30.1-2.81.

[19] MIT Laboratory. *Auto-tuning data science: New research streamlines machine learning*. Dec. 2017. URL: http://news.mit.edu/2017/auto-tuning-data-science-new-research-streamlines-machine-learning-1219.

[20] Wes McKinney et al. "Data structures for statistical computing in python". In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX. 2010, pp. 51–56.

[21] Janakiram MSV. *Why AutoML Is Set To Become The Future Of Artificial Intelligence*. Apr. 2018. URL: https://www.forbes.com/sites/janakirammsv/2018/04/15/why-automl-is-set-to-become-the-future-of-artificial-intelligence/.

[22] Cassio P de Campos et al. "Discovering Subgroups of Patients from DNA Copy Number Data Using NMF on Compacted Matrices". In: 8 (Nov. 2013), e79720.

[23] Victor Picheny, Tobias Wagner, and David Ginsbourger. "A benchmark of kriging-based infill criteria for noisy optimization". In: *Structural and Multidisciplinary Optimization* 48.3 (2013), pp. 607–626.

[24] Dmitry Plotnikov et al. "Automatic tuning of compiler optimizations and analysis of their impact". In: *Procedia Computer Science* 18 (2013), pp. 1312–1321.

[25] Paul Resnick and Hal R Varian. "Recommender systems". In: *Communications of the ACM* 40.3 (1997), pp. 56–58.

[26] Guido van Rossum, Barry Warsaw, and Nick Coghlan. *PEP 8 – Style Guide for Python Code*. 2013. URL: https://www.python.org/dev/peps/pep-0008/.

[27] Wade Shen. *Data-Driven Discovery of Models*. 2017. URL: https://www.darpa.mil/program/data-driven-discovery-of-models.

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

[29] Thomas Swearingen et al. "ATM: A distributed, collaborative, scalable system for automated machine learning". In: *IEEE International Conference on Big Data*. 2017.

[30] Stewart W Wilson et al. “Explore/exploit strategies in autonomy”. In: *Proc. of the Fourth International Conference on Simulation of Adaptive Behavior: From Animals to Animats*. Vol. 4. 1996, pp. 325–332.