Auto-FP: An Experimental Study of Automated Feature Preprocessing for Tabular Data

Danrui Qi[†*], Jinglin Peng[†*], Yongjun He[◇*], Jiannan Wang[†]

Simon Fraser University[†]
{dqi, jinglin_peng, jnwang}@sfu.ca

ETH Zürich[◇]
yongjun.he@inf.ethz.ch

ABSTRACT

Classical machine learning models, such as linear models and tree-based models, are widely used in industry. These models are sensitive to data distribution, thus feature preprocessing, which transforms features from one distribution to another, is a crucial step to ensure good model quality. Manually constructing a feature preprocessing pipeline is challenging because data scientists need to make difficult decisions about which preprocessors to select and in which order to compose them. In this paper, we study how to automate feature preprocessing (Auto-FP) for tabular data. Due to the large search space, a brute-force solution is prohibitively expensive. To address this challenge, we interestingly observe that Auto-FP can be modelled as either a hyperparameter optimization (HPO) or a neural architecture search (NAS) problem. This observation enables us to extend a variety of HPO and NAS algorithms to solve the Auto-FP problem. We conduct a comprehensive evaluation and analysis of 15 algorithms on 45 public ML datasets. Overall, evolution-based algorithms show the leading average ranking. Surprisingly, the random search turns out to be a strong baseline. Many surrogate-model-based and bandit-based search algorithms, which achieve good performance for HPO and NAS, do not outperform random search for Auto-FP. We analyze the reasons for our findings and conduct a bottleneck analysis to identify the opportunities to improve these algorithms. Furthermore, we explore how to extend Auto-FP to support parameter search and compare two ways to achieve this goal. In the end, we evaluate Auto-FP in an AutoML context and discuss the limitations of popular AutoML tools. To the best of our knowledge, this is the first study on automated feature preprocessing. We hope our work can inspire researchers to develop new algorithms tailored for Auto-FP. We release our datasets, code, and comprehensive experimental results at https://github.com/AutoFP/Auto-FP.

1 INTRODUCTION

Despite the recent advancement of deep learning for image and text data, most machine learning (ML) use cases in industries are still about applying classical ML algorithms to tabular data. For example, a recent survey of over 25,000 data scientists and ML engineers [5] shows that the most commonly used models are linear models (linear regression or logistic regression); the most popular ML framework is scikit-learn [4] (mainly designed for traditional machine learning).

Building a classical ML model on tabular data involves several tasks, such as feature preprocessing, feature selection, and hyperparameter tuning. An important question faced by many real-world data scientists is how to automatically complete each task. Fortunately, many research efforts have been devoted to answering this question. They conduct comprehensive surveys or experiments on feature selection [16, 22, 38], hyperparameter tuning [20, 68], data cleaning [40], feature type inference [54], etc. However, *feature preprocessing*, an essential task for classical ML, has not been well explored in the literature. This paper aims to fill this research gap.

There are many commonly used feature preprocessors in scikit-learn, such as *Binarizer*, *MaxAbsScaler*, *MinMaxScaler*, *Normalizer*, *PowerTransformer*, *QuantileTransformer* and *StandardScaler*. Intuitively, a feature preprocessor transforms features from one distribution to another. For example, *MinMaxScaler* transforms features by scaling each feature to a given range. *PowerTransformer* applies an exponential transformation to each feature to make its distribution more normal-like. Given a training dataset and a set of feature preprocessors [8], the goal of feature preprocessing is to construct a pipeline (i.e., a sequence of selected feature preprocessors) to scale and transform the features in the training set.

To construct a good pipeline, we need to answer two questions: i) which preprocessors to select; ii) in which order to compose them. Different answers will lead to different pipelines. Consider the following two pipelines:

$$P1 : \mathit{MinMaxScaler} \rightarrow \mathit{PowerTransformer}$$
$$P2 : \mathit{PowerTransformer} \rightarrow \mathit{MinMaxScaler} \rightarrow \mathit{Normalizer}$$

$P1$ selects two preprocessors and applies *MinMaxScaler* first and then *PowerTransformer*; $P2$ selects three preprocessors and applies *PowerTransformer* first and then *MinMaxScaler* and *Normalizer*.

It is not easy to use domain knowledge to determine which pipeline ($P1$ or $P2$) is better since it not only depends on the upstream training set but also on the downstream learning algorithm. To *manually* construct a pipeline, data scientists have to get into a trial-and-error mode, which will be both tedious and time-consuming. If a bad feature preprocessing pipeline is selected, it could even hurt the downstream model accuracy. To this end, this paper studies how to *automatically* search for the best feature preprocessing pipeline.

The brute-force solution that enumerates all possible pipelines is prohibitively expensive. Given a set of n preprocessors, a pipeline could contain 1, 2, $\cdots$, or n preprocessors, thus there will be a total of $\sum_{i=1}^{n} i^i \approx n^n$ pipelines to enumerate. Apparently, it is too expensive for the brute-force solution to enumerate all pipelines. We interestingly find the insight that the Auto-FP problem can be modelled as either a Hyperparameter Optimization (HPO) problem or a Neural Architecture Search (NAS) problem. This interesting insight enables us to extend many existing search algorithms from HPO and NAS to Auto-FP. With these search algorithms, we need to answer the following essential questions:

* The first three authors contributed equally to this research.

(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)
Num	Num	Num	Num	Num	Num	Num	Num
-1.5	-1.87	-0.3	0	-1	-1.72	0	0
1	-0.61	0.2	0.38	1	-0.71	0.17	1
1.5	-0.36	0.3	0.46	1	-0.46	0.33	1
2.5	0.15	0.5	0.61	1	0.07	0.5	1
3	0.40	0.6	0.69	1	0.35	0.67	1
4	0.90	0.8	0.85	1	0.93	0.83	1
5	1.41	1	1	1	1.53	1	1

Figure 1: Illustration of different feature preprocessors. (a) No preprocessor; (b) StandardScaler; (c) MaxAbsScaler; (d) MinMaxScaler; (e) Normalizer; (f) PowerTransformer; (g) QuantileTransformer; (h) Binarizer.

Q1. Which search algorithm performs better? There are many HPO and NAS search algorithms available, but it is unclear which algorithms will perform well for Auto-FP. Unlike HPO and NAS, Auto-FP focuses on optimizing a preprocessing pipeline, i.e., aiming to find the best combination as well as the best order of preprocessors, thus it has a unique search space for these search algorithms to explore. An algorithm performing well for HPO and NAS does not mean that it will also perform well for Auto-FP.

Q2. How to extend Auto-FP to support parameter search? To further enhance the performance of Auto-FP, we consider the scenario where a user wants to search for not only the best pipeline but also the best parameters associated with each preprocessor. The characteristics such as the cardinality of parameter search spaces can be varied and it is valuable to explore how to extend Auto-FP to support different search spaces.

Q3. What is the relationship between Auto-FP and AutoML? To further explore future opportunities, we put Auto-FP in an AutoML context to figure out the relationship between Auto-FP and AutoML. AutoML tools are typically equipped with a feature preprocessing module. One natural question is whether Auto-FP can outperform the FP part of a general-purpose AutoML. Another question is whether Auto-FP is important in the AutoML context. If yes, they can learn from our paper to improve their feature preprocessing module.

Our Contributions. For answering the above essential questions, our work makes the following contributions:

(1) ***Indicate the importance of FP.*** Feature preprocessing is a crucial step in classical ML. To the best of our knowledge, this is the first comprehensive study on this important topic.

(2) ***Formally define Auto-FP.*** We identify a number of widely used preprocessors in scikit-learn and formally define the Auto-FP problem. We find that this problem can be modelled as either a HPO or a NAS problem, and extend 15 search algorithms from HPO and NAS to Auto-FP.

(3) ***Categorize Auto-FP search algorithms.*** We conclude these search algorithms into a unified framework and categorize them into 5 categories: traditional algorithms, surrogate-model-based algorithms, evolution-based algorithms, RL-based algorithms and bandit-based algorithms.

(4) ***To answer Q1, Q2 and Q3***, we first conduct comprehensive experiments comparing 15 Auto-FP algorithms on 45 public ML datasets and provide recommendations for different scenarios. Then we explore two parameter search spaces with varying cardinalities and propose two extensions for Auto-FP, namely One-step (combining parameter and pipeline search) and Two-step (separating the searches). Thirdly, by situating Auto-FP within an AutoML context, we identify the limitations of current AutoML tools, examine if Auto-FP outperforms FP in AutoML tools, and highlight the significance of Auto-FP by comparing it to hyperparameter tuning in AutoML.

Table 1: 3-CV scores of decision tree models with different tree depths for downstream ML models (LR, XGB, or MLP).

LR		XGB		MLP	
Tree Depth	**3-CV Score**	**Tree Depth**	**3-CV Score**	**Tree Depth**	**3-CV Score**
1	0.65	1	0.70	1	0.54
2	0.65	2	0.67	2	0.65
3	0.54	3	0.67	3	0.67
No Limit	0.50	No Limit	0.67	No Limit	0.67

Experimental Findings. Our empirical results reveal a number of interesting findings and we summarize our main experimental findings as follows: 1) Evolution-based search algorithms typically achieve the highest overall average ranking. 2) Random search is a strong baseline. 3) Many sophisticated algorithms, such as RL-based, bandit-based and most surrogate-model-based algorithms, perform even worse than random search. 4) Different algorithms have different bottlenecks and model evaluation is the bottleneck in most cases. 5) *One-step* fits for extended low-cardinality search space and *Two-step* is preferred for extended high-cardinality search space. 6) Auto-FP outperforms the FP part in AutoML and it is important in the AutoML context.

Research Opportunities. According to our findings, we identify four future research directions: 1) warm-start search algorithms 2) allocate pipeline and parameter search time budget reasonably 3) benchmark Auto-FP on various data types and deep models. The code, datasets and experimental raw data are published on GitHub as a reference for the community. We hope our work can attract more research interest not only for Auto-FP but also for automating other individual tasks (such as data cleaning [35], feature generation and feature selection [28, 33, 45, 47, 56]) in AutoML context and building more powerful AutoML.

2 BACKGROUND AND MOTIVATION

In practice, users enhance model accuracy by incorporating feature preprocessing techniques to serve the purposes such as scale variance reduction, data distribution transformation and dimensionality reduction. Scaling with *StandardScaler* addresses varying scales, while *PowerTransformer* adjusts data for specific distributions. Techniques like *Binarizer* reduce dimensionality and improve efficiency. Proper selection of preprocessing techniques is crucial for accurate results. In this section, we first present the background of feature preprocessors, then validate whether feature preprocessing really matters, and finally explore the relationship between data characteristics and feature preprocessing.

2.1 Feature Preprocessors

Scikit-learn [4] is popular for developing classical ML on tabular data. Its feature preprocessing module offers various preprocessors[1] and Sklearn-doc [7] explains their practical usage and effectiveness. For example, scaling feature preprocessors is commonly used for improving model performance [15, 36]. Compared to the scaling feature preprocessors, discretization/binarization is less frequent. We discuss seven feature preprocessors in the following contents and provide a rigorous justification for choosing them.

1. *StandardScaler:* ML models may perform badly if individual features do not follow the standard normal distribution. *StandardScaler* standardizes a feature by removing its mean value and scales it by dividing the standard deviation [4]. Let μ and σ denote the mean and the standard deviation of a feature, respectively. For each value x in the feature, the scaled result is

[1] https://scikit-learn.org/stable/modules/preprocessing.html

calculated by $\frac{x-\mu}{\sigma}$. Figure 1(b) shows the result of applying *StandardScaler*. The μ and σ are 2.21 and 1.98. Thus, the transformed result of value -1.5 is $\frac{-1.5-2.21}{1.98} = -1.87$.

2. *MaxAbsScaler:* When the standard deviations of some features are very small like 10^{-8}, applying *StandardScaler* can cause unreasonable transformed features. In this situation, it is better to scale the values of each feature into a specific range. *MaxAbsScaler* scales each feature according to its maximum absolute value [7]. Let x be the maximum absolute value of the feature. For each value $v_1, v_2, \cdots$ of the feature, the scaled values will be $\frac{v_1}{x}, \frac{v_2}{x}, \cdots$. Figure 1(c) shows the result of applying *MaxAbsScaler* to the original feature. The maximum absolute value of the original feature is 5. Thus, the value -1.5 is transformed to $\frac{-1.5}{5} = -0.3$.

3. *MinMaxScaler:* Similar to *MaxAbsScaler*, *MinMaxScaler* transforms features by scaling each feature to a given range [7**?**] ([0, 1] by default). Let max and min denote the maximum and minimum values of the feature, respectively. The scaled result of x is calculated as $\frac{x-min}{max-min}$. Figure 1(d) shows the result of applying *MinMaxScaler* to the original feature. The max (min) value of the original feature is 5 (-1.5). Thus, the value 1 is transformed to $\frac{1-(-1.5)}{5-(-1.5)} = 0.38$.

4. *Normalizer:* ML models can be impacted severely by different scales of the features. *Normalizer* normalizes samples (*rows* of data) individually into unit norm [7]. Given a row vector, $x = [x_1, x_2, \cdots, x_n]$, each value x_i is scaled to $\frac{x_i}{||x||_2}$. Suppose the data only has a single column as shown in Figure 1(a). Figure 1(e) shows the result of applying *Normalizer* to the data. For the first row $x = [-1.5]$, the Euclidean length is $||x||_2 = \sqrt{(-1.5)^2} = 1.5$, thus the value -1.5 is transformed to $\frac{-1.5}{1.5} = -1$.

5. *PowerTransformer:* In some modelling scenarios, the normality of the features is desirable. When some features exhibit a large skewness, *PowerTransformer* can perform an exponential and monotonic transformation to each feature to make its distribution more normal-like. It can help ML models with normal-like input assumptions and help to handle highly skewed data [7]. The default PowerTransformer of Scikit-Learn, called Yeo-Johnson transformation, is defined as:

$$Yeo-Johnson(x) = \begin{cases} \frac{(x+1)^{\lambda}-1}{\lambda} & x >= 0, \lambda \neq 0 \\ log(x+1) & x >= 0, \lambda = 0 \\ \frac{1-(1-x)^{2-\lambda}}{2-\lambda} & x < 0, \lambda \neq 2 \\ -log(1-x) & x < 0, \lambda = 2 \end{cases} \quad (1)$$

where λ is automatically calculated. It aims to fit the transformed feature into a zero-mean, unit-variance normal distribution as much as possible. Figure 1(f) shows the result of applying *PowerTransformer* to the original feature. At first, the *PowerTransformer* finds the optimal λ automatically, which is 1.22. Then, the transformed results are calculated by Equation 1. For example, the transformed result of value -1.5 is $\frac{1-(1-(-1.5))^{2-1.22}}{2-1.22} = -1.34$.

6. *QuantileTransformer:* Similar to the applied scenarios of *PowerTransformer*, *QuantileTransformer* also performs a nonlinear transformation. *QuantileTransformer* transforms features independently into a uniform or a normal distribution [7]. Since the above *PowerTransformer* can be used for normal distribution, we utilize *QuantileTransformer* for uniform distribution. Intuitively, each transformed value represents its quantile position in the original feature column. For example, Figure 1(a) shows the original feature column: [-1.5, 1, 1.5, 2.5, 3, 4, 5]. As shown in Figure 1(g), it is transformed to $[\frac{0}{6}, \frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}, \frac{5}{6}, \frac{6}{6}] = [0, 0.17, 0.33, 0.5, 0.67, 0.83, 1]$.

7.*Binarizer:* Certain datasets can benefit from binarization, especially when binarization results present a high correlation with the target label [25]. *Binarizer* binarizes data into two values (0 or 1) according to a threshold. Values greater than the threshold are mapped to 1, otherwise to 0 [7]. The default threshold is 0, which means that negative values are mapped to 0, and non-negative values are mapped to 1. E.g., Figure 1(a) is the original feature without any preprocessing. Figure 1(h) shows the result of applying *Binarizer* to the feature. With the default threshold 0, the number -1.5 is transformed to 0, while the other values are transformed to 1.

How Do We Choose the Feature Preprocessors? We choose seven widely used feature preprocessors based on their effectiveness and interpretability. They have proven effective on popular models like LR and XGB and are easily interpretable when combined with specific datasets and models. We exclude less interpretable preprocessors like x-bit encoding. Our selection surpasses what popular AutoML tools offer, ensuring applicability across various scenarios. To accommodate additional preprocessors, our benchmark can be extended for further insights.

2.2 Motivation

Does FP Really Matter? To motivate Auto-FP, we conduct an exploratory experiment to examine whether feature preprocessing really matters by considering pipelines of lengths up to 4 (2800 pipelines in total). We apply each pipeline to a training set and train an LR model on the preprocessed data. Figure 2 presents the distribution of model accuracy for four datasets. The x-axis represents accuracy values, while the y-axis denotes the number of pipelines achieving each accuracy. The red line represents ML accuracy without preprocessing. Results demonstrate that different pipelines yield significantly varied model accuracy. For instance, on the Heart dataset, accuracy ranges from 0.49 to 0.88. Furthermore, a good pipeline can substantially improve accuracy, while a poor one can even degrade it. For example, on the *Pd* dataset, the best (worst) pipeline achieves 0.94 (0.24) accuracy, while no preprocessing (red line) achieves 0.81 accuracy. These findings emphasize the importance of feature preprocessing. To further motivate Auto-FP, we compare the accuracy of the 2800 pipelines with a given combination of FP represented by pipelines generated by the popular AutoML tool TPOT[48]. Table 2 shows the accuracy comparison between TPOT's FP pipeline and the best FP pipeline among the 2800 considered. For all four datasets, the best FP pipeline outperforms TPOT's pipeline, indicating the potential for improving performance by extending pipeline length and highlighting the need for Auto-FP.

Are there explicit data characteristic rules that can be used to infer the effectiveness of FP? To further motivate Auto-FP, we analyze the relationship between data characteristics and FP effectiveness. We consider 40 characteristics used in Auto-Sklearn [21], encompassing basic (e.g., *NumberOfClasses*), statistical (e.g., *SkewnessMean*), landmarking (e.g., *Landmark1NN*), and information-theoretic (e.g., *ClassEntropy*) measures. The detailed list of these data characteristics is shown in the technical report [6]. The datasets, downstream ML models, and the experimental environment we use are the same as in Section 5.

We investigate whether there are *data characteristic rules* that can be used to infer the effectiveness of feature preprocessing. For example, one possible rule is "*if features are highly skewed, then FP will be very useful for improving the downstream model accuracy*". Using 45 commonly used ML datasets with diverse characteristics, we construct a training data with 40 features

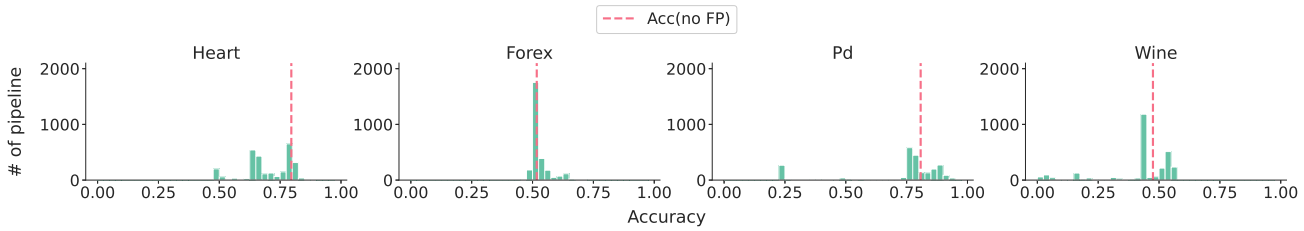

Figure 2: Distribution of LR accuracy with different feature preprocessing pipelines.

Table 2: Accuracy comparison between the TPOT FP pipeline and the best pipeline in Figure 2.

Dataset	TPOP FP Pipeline / Accuracy	Best FP Pipeline in Figure 2 / Accuracy
Heart	Binarizer / 0.8333	Normalizer ->StandardScaler ->Binarizer / 0.8367
Forex	Binarizer ->StandardScaler / 0.7001	MaxAbsScaler ->Normalizer ->Normalizer ->StandardScaler / 0.7042
Pd	MinMaxScaler / 0.9250	StandardScaler ->Normalizer ->MinMaxScaler / 0.9421
Wine	Binarizer ->Normalizer ->MaxAbsScaler / 0.5591	PowerTransformer ->Normalizer ->MaxAbsScaler ->QuantileTransformer / 0.5693

representing data characteristics and a binary label indicating whether FP significantly improves downstream model accuracy. To determine the label, we calculate the accuracy score A by directly inputting a dataset into a model without FP. Then we apply 200 randomly selected pipelines on the same dataset and model and obtain an accuracy score B. We compute B-A and assign label 1 if B-A > 1.5%, and label 0 if B-A < -1.5%. We train a decision tree on the training data and try to derive characteristic rules. Table 1 presents the 3-fold cross-validation (3-CV) scores for different downstream models at varying tree depths. Unfortunately, the 3-CV scores are consistently low, suggesting the absence of reliable data characteristic rules to predict FP effectiveness. This further emphasizes the need for Auto-FP.

3 AUTOMATED FEATURE PREPROCESSING

In this section, we first formally define the Auto-FP problem and then view Auto-FP problem as HPO and NAS.

3.1 Problem Formulation

Definition 3.1 (Feature Preprocessor). Given a dataset D, let r_i be its i-th row and c_j be its j-th column. A preprocessor $\mathcal{P}$ is a mapping function that maps dataset D to D', where each row r_i is mapped to r'_i and each column c_j is mapped to c'_j.

Example 3.2. This paper considers the seven feature preprocessors from Section 2.1. Given a dataset D and a feature preprocessor $StandardScaler(\cdot)$, we have $D' = StandardScaler(D)$.

Definition 3.3 (Feature Preprocessing Pipeline). Given a set of preprocessors, a pipeline $\mathcal{L}$ of size n is a composite function that contains a sequence of n preprocessors, denoted by $\mathcal{P}_1 \rightarrow \mathcal{P}_2 \rightarrow \cdots \rightarrow \mathcal{P}_n$. For a dataset $\mathcal{D}$, $\mathcal{L}$ maps it to dataset $\mathcal{D}'$, where $\mathcal{D}' = \mathcal{P}_n \circ (...\mathcal{P}_2 \circ (\mathcal{P}_1(\mathcal{D}))$.

Example 3.4. Given $\mathcal{D}$ and a feature preprocessing pipeline $PowerTransformer \rightarrow MinmaxScaler \rightarrow Normalizer$, we have $\mathcal{D}' = Normalizer \circ (MinmaxScaler \circ (PowerTransformer(\mathcal{D})))$.

Pipeline Error. Auto-FP problem is the problem of searching for the best pipeline. Thus, it is vital to figure out how to measure the quality of a given pipeline. We formally define *Pipeline Error* as follows, thus searching for the best pipeline means searching FP pipeline with minimal pipeline error:

Consider the model training process that gives a classifier C, a training dataset $\mathcal{D}_{train}$ with its labels, and a validation dataset $\mathcal{D}_{valid}$ with its labels. For a pipeline $\mathcal{L}$, it will be used to transform the training data $\mathcal{D}_{train}$ and get a new dataset $\mathcal{L}(\mathcal{D}_{train})$. Then, the classifier C will be trained on the new dataset. We denote the trained classifier as $C_{\mathcal{L}(\mathcal{D}_{train})}$, or $C_{\mathcal{L}}$ for simplicity when the context is clear. Clearly, a good pipeline should be the one that minimizes the error of the trained classifier $C_{\mathcal{L}}$. Since the test data is not available in the training stage, we measure the validation error of $C_{\mathcal{L}}$ on $\mathcal{D}_{valid}$, denoted by validation_error$(C_{\mathcal{L}}, \mathcal{D}_{valid})$. Then, we define the error of a pipeline $\mathcal{L}$ as:

$$\text{error}(\mathcal{L}) := \text{validation\_error}(C_{\mathcal{L}(\mathcal{D}_{train})}, \mathcal{D}_{valid}) \quad (2)$$

Now we have defined the error of a pipeline. Next, we formally define the pipeline search problem (i.e., automated feature preprocessing) in definition 3.5.

Definition 3.5 (Automated Feature Preprocessing). Given a set of preprocessor $\mathcal{S}_{prep}$, suppose that a feature preprocessing pipeline contains at most N preprocessors. Let $\mathcal{S}_{pipe}$ be the set of all pipelines constructed with $n = \{1, \cdots, N\}$ preprocessors chosen from $\mathcal{S}_{prep}$. The automated feature preprocessing (Auto-FP) problem aims to find the best pipeline with minimal error, i.e.,

$$\arg\min_{\mathcal{L} \in \mathcal{S}_{pipe}} \text{error}(\mathcal{L})$$

3.2 Auto-FP as HPO and NAS

Interestingly, Auto-FP can be viewed in two different ways.

Auto-FP as HPO. HPO aims to find the best combination of classifier and its related hyperparameters for a given dataset. Since different classifiers have different HP space (e.g., LR needs to tune "penalty" while Random Forest (RF) does not), its search process contains two steps and can be modelled as a search tree like Figure 3(a). The first step is to select a classifier to activate its corresponding HP space (e.g., RF). The second step is to determine the candidate option for each HP (e.g., n_estimators = 200). The search process of Auto-FP can also be divided into two steps as shown in Figure 3(b). The first step is to select the pipeline length (e.g., len = 7). The second step is to determine the specific preprocessor for each position (e.g., P_1 = Normalizer).

Auto-FP as NAS. NAS aims to find the best neural architecture for a given dataset. Auto-FP can be modelled as the chain-structure NAS problem, whose neural architecture has no skip connection among different layers and no multi-branch in each layer. That is, a chain-structure neural architecture is a sequence of operators (i.e., a pipeline) as shown in Figure 4(a). Its search process optimizes two factors holistically: (1) the max depth of a chain structure; (2) the operator put in each position. Similarly, for Auto-FP, the goal is also to find the best chain-structure neural architecture. The difference is that it puts a feature preprocessor (e.g., Normalizer) rather than a neural-architecture operator at each position.

Remark. HPO, NAS, and AutoFP all aim to find the best combination in large search spaces. HPO optimizes the combination

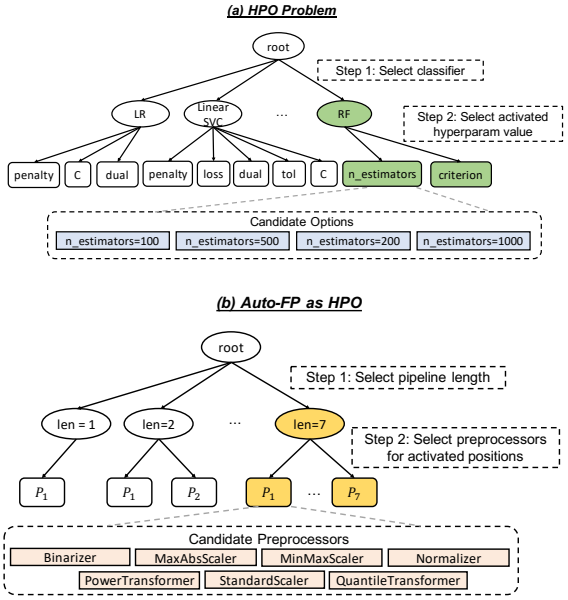

Figure 3: The analogy between HPO and Auto-FP.

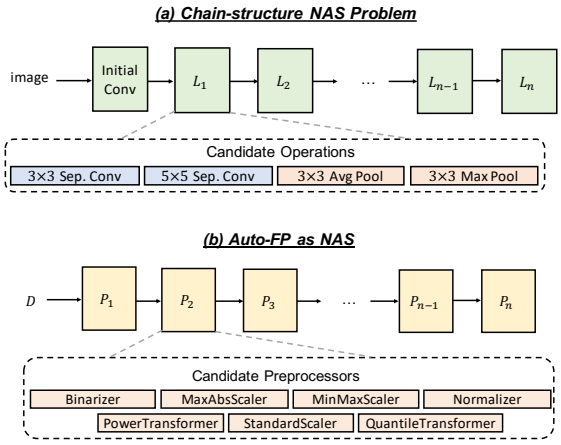

Figure 4: The anology between NAS and Auto-FP.

of classifiers and hyperparameters, NAS selects the combination of operators in neural architecture, and AutoFP searches for the optimal combination of feature preprocessors. Thus, HPO and NAS can cover AutoFP, which is the reason that we derive search algorithms for AutoFP from HPO and NAS area in Section 4.

4 AUTO-FP SEARCH ALGOITHMS

We identify 15 representative search algorithms from HPO and NAS for Auto-FP. To select the algorithms in our study, we referred to the popular Microsoft NNI tool [46] and two widely cited surveys [17, 27]. Our paper covers a significant proportion of HPO and NAS algorithms in NNI, i.e. 8 out of 12 HPO algorithms and 5 out of 10 NAS algorithms. Also, our search algorithms cover all categories of search algorithms in NNI. We initially divide them into 5 categories according to their optimizing strategies. Then we conclude them into a unified framework for analyzing their performance bottleneck in Section 5. Table 3 shows a summary of these algorithms.

4.1 Categories of Search Algorithms

Roughly, Auto-FP search algorithms can be divided into 5 categories including traditional, surrogate-model-based, evolution-based, RL-based and band-based algorithms. We introduce these algorithms in detail in the following content.

4.1.1 Traditional Algorithms. Traditional algorithms samples and evaluates one pipeline for each iteration without any initialization.

Random Search [13] randomly picks one FP pipeline from the search space and evaluates the pipeline with downstream ML model in each iteration.

Anneal [34] progressively approaches the best FP solution by comparing the current best pipeline to its neighbourhoods. In each iteration, it accepts the better neighbourhoods as the new best state and rejects the worse neighbourhoods.

4.1.2 Surrogate-model-based Algorithms. The existing surrogate-model-based algorithms utilize one surrogate model to model the relationship $p(e_x|x)$ between FP pipelines and the downstream model accuracy. The x represents FP pipelines and e_x represents the downstream model accuracy. In each iteration, they follow two steps: 1) fit the surrogate model with evaluated trials 2) generate the next promising pipeline to evaluate.

SMAC [29] uses the random forest as the surrogate model to handle high-dimensional and categorical input, which fits better for our Auto-FP scenario than Gaussian Process. At each iteration, *SMAC* fits random forest to represent $p(e_x|x)$, then generates the next promising FP pipeline for evaluation.

TPE [12] takes *Kernel Density Estimation* (KDE) as the surrogate model which gives the linear time complexity. It performs similar actions like *SMAC* at each iteration, i.e. refitting KDE and generating the next promising FP pipeline.

Progressive NAS [43] initially starts by considering single preprocessors as pipelines, evaluating them and training the surrogate model (*MLP* or *LSTM*). Then it expands the simple pipelines by adding more possible preprocessors. The surrogate model is used to select the next top-k pipelines for evaluation instead of one single pipeline generated by *SMAC* and *TPE* based on their score prediction. There are four variants of *Progressive NAS* according to the variants of the surrogate model, which are all listed in Table 3.

4.1.3 Evolution-based Algorithms. Evolution-based algorithms consider each individual FP pipeline as single DNA in a population. In each evolution step, some outstanding pipelines are selected, mutated and evaluated to update the population.

Tournament Evolution [51] randomly chooses S FP pipelines from the population in each step and the pipelines with the highest downstream model accuracy are used for mutation. There are two variants of *Tournament Evolution* because of two killing strategies, i.e. kill the oldest pipeline in the population (*TEVO_Y*) or kill pipeline with worst downstream model accuracy (*TEVO_H*).

PBT [31] updates the population gradually by replacing bad FP pipelines with the mutation of good pipelines. The "good pipelines" means the downstream model accuracies of these pipelines exceed the lowest bar. In each evolution step, *PBT* also injects more exploration by randomly generating FP pipelines with a fixed probability instead of just mutating.

4.1.4 RL-based Algorithms. The existing RL-based algorithms aims find an optimal policy π_θ which can maximize the expected cumulative reward. With the trial-and-error strategy, FP pipelines are sampled and evaluated to produce rewards.

REINFORCE [62] is a policy-gradient algorithm based on the Monte Carlo strategy, which tries to directly update θ of a policy. In each iteration, it samples one FP pipeline and updates θ according to the downstream model accuracy. The larger the

Table 3: Categories of Automated Feature Preprocessing Search Algorithms.

Category	Area	Search Alg	Surrogate Model	Initialization	# of samples / iter	# of evaluations / iter
Traditional	HPO	Random Search (RS) [13]	None	None	=1	=1
Traditional	HPO	Anneal [34]	None	None	=1	=1
Surrogate-Model-based	HPO	SMAC [29]	Random Forest	Random Search	>1	=1
Surrogate-Model-based	HPO	TPE [12]	KDE	Random Search	>1	=1
Surrogate-Model-based	NAS	Progressive NAS + MLP no ensemble (PMNE) [43]	MLP no ensemble	Single Preprocessors	>1	>1
Surrogate-Model-based	NAS	Progressive NAS + MLP ensemble (PME) [43]	MLP ensemble	Single Preprocessors	>1	>1
Surrogate-Model-based	NAS	Progressive NAS + LSTM no ensemble (PLNE) [43]	LSTM no ensemble	Single Preprocessors	>1	>1
Surrogate-Model-based	NAS	Progressive NAS + LSTM ensemble (PLE) [43]	LSTM ensemble	Single Preprocessors	>1	>1
Evolution-based	HPO	PBT [31]	None	Random Search	>1	>1
Evolution-based	NAS	Tournament Evolution-Higher (TEVO_H) [51]	None	Random Search	=1	=1
Evolution-based	NAS	Tournament Evolution-Younger (TEVO_Y) [51]	None	Random Search	=1	=1
RL-based	HPO	REINFORCE [62]	Parameter Matrix	None	=1	=1
RL-based	NAS	ENAS [49]	LSTM	None	=1	=1
Bandit-based	HPO	Hyperband [39]	None	None	>1	>1
Bandit-based	HPO	BOHB [19]	KDE	Random Search	>1	>1

downstream model accuracy of the sampled pipeline, the higher probability to choose feature preprocessors in this pipeline.

ENAS [49]considers all the FP pipeline architectures as DAGs and considers the whole search space as a large super-graph. The nodes in this super-graph represent the preprocessors, and the edges represent the edge flow. Each iteration utilizes a LSTM to predict the next edges that should be activated and the next preprocessors that should be used.

4.1.5 Bandit-based Algorithms. Bandit-based algorithms aim to create a trade-off between the number of evaluated FP pipelines and the evaluation time for each pipeline. It allocates more time for promising pipelines and successively discards others before the evaluation process is finished. Note that there are many types of bandit-base algorithms [9, 58]. However, the reason we choose Hyperband [39] and BOHB [19] here is that 1) they are two very popular HPO algorithms used specifically for HPO, 2) we take the analogy between Auto-FP and HPO.

Hyperband [39] considers *Succesive Halving* [32] as a subroutine and each running of SH is called a bracket. In each bracket, the outer loop controls the number of randomly sampled FP pipelines and the initial resource allocation, while the inner loop runs SH.

BOHB [19] indicates the shortcoming of pure *Hyperband*, that is, randomly generating sampled FP pipelines in each bracket wastes the limited budgets. Thus, it gives a mixture of randomly selected pipelines and pipelines generated by *TPE*, which helps to direct pipeline search without losing exploration.

4.2 Search Algorithm Framework

In fact, we notice that all these algorithms roughly follow the same search framework, as shown in Algorithm 1. It is an iterative framework mainly consisting of four steps: *Step 1*. Generate initial pipelines; *Step 2*. Update a surrogate model (optional); *Step 3*. Sample new pipelines; *Step 4*. Evaluate sampled pipelines and go back to Step 2 until the budget is exhausted. Finally, the pipeline with the lowest error is returned. In the following, we describe how each algorithm works at each step in detail.

Step 1: Generate initial pipelines (Line 2): As shown in the "Initialization" column of Table 3, most algorithms need an initialization step, i.e., generating initial pipelines. The evolution-based algorithms randomly generate initial pipelines to form an initial population. *RS*, *Anneal*, and *Hyperband* are the ones that do not need any initial pipelines. However, they still follow the framework in Algorithm 1 by setting $\mathcal{P}_{init} = \emptyset$. RL-based algorithms do not need any initial pipelines because *REINFORCE* uses a randomly generated parameter matrix as the initial policy and *ENAS* utilizes a randomly parameterized LSTM as the initial controller. Surrogate-model-based algorithms all need the initial pipelines and their evaluation results to build an initial surrogate model. For example, *TPE* leverages initial pipelines to generate an initial KDE, while *SMAC* uses them to generate an initial random forest model. *Progressive NAS* build initial *MLP* or *LSTM* with all single-preprocessor pipelines. Note that the initialization of all these algorithms except for *Progressive NAS* uses random search.

Algorithm 1 : A Unified Auto-FP Search Framework

1: **Input:** dataset $D = (D_{train}, D_{valid})$, time budget T, surrogate model $\mathcal{M}$ (optional), downstream ML model C
2: **Initialization**: Randomly sample and evaluate n_{init} pipelines with random search. // Step 1
3: elapsedTime = 0, $\mathcal{P}_{new} = \emptyset$, $\mathcal{P}_{eval} = \mathcal{P}_{init}$
4: **while** elapsedTime < T **do**
5: $\mathcal{M}$ = Update($\mathcal{M}, \mathcal{P}_{eval}$) // Step 2
6: $\mathcal{S}_{new}$ = get_sampled_pipelines() // Step 3
7: $\mathcal{P}_{new}$ = Eval($\mathcal{S}_{new}$) // Step 4
8: $\mathcal{P}_{eval} = \mathcal{P}_{eval} \cup \mathcal{P}_{new}$
9: **end while**
10: **return** *Pipeline with the lowest error from* $\mathcal{P}_{eval}$

Step 2: Update a surrogate model (optional) (Line 5): Surrogate-model-based algorithms leverage surrogate models to generate pipelines. As mentioned in Step 1, all of them need initial pipelines to initialize a surrogate model. After each iteration, there are one or more newly sampled pipelines evaluated. The surrogate model should be updated with the historical and newly generated pipelines. For instance, *SMAC* retrains its random forest, *TPE* refits its KDEs, and *Progressive NAS* refreshes its MLP or LSTM surrogate model. Other algorithms which include surrogate models also need to update their surrogate model: *BOHB* refits its KDE with pipelines trained with the highest iterations or estimators, *REINFORCE* updates its policy, and *ENAS* updates its LSTM model. Note that the algorithms without any surrogate model skip this step and directly go to Step 3.

Step 3: Sample new pipelines (Line 6): Different algorithms sample new pipelines with different strategies. In addition, they could sample one or multiple pipelines. *RS* and *Anneal* randomly sample a single pipeline at each iteration. *Hyperband* randomly

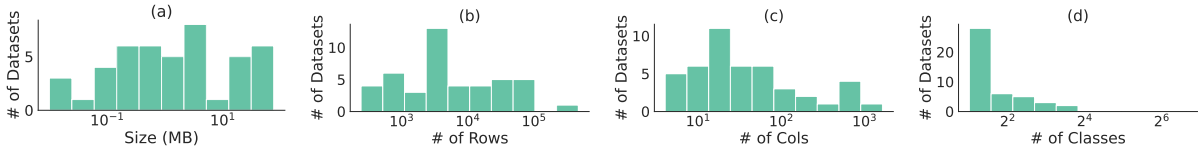

Figure 5: Statistics of 45 real-world ML datasets.

generates multiple pipelines in order to apply *Successive Halving* to early terminate poor-performing pipelines. *Tournament Evolution* tries to mutate the best one into another new child, while *PBT* generates multiple pipelines with its exploitation and exploration process.

The algorithms with surrogate models generate new sampled pipeline(s) with updated surrogate models. For example, *TPE*, *SMAC* sample a single pipeline with the best acquisition score based on updated KDE or random forest. *BOHB* follows the framework of *Hyperband*, thus at each iteration, it needs to generate multiple pipelines for Successive Halving. *REINFORCE* produces one sampled pipeline with the updated policy. *ENAS* generates one sampled pipeline for evaluation using its LSTM controller, and *Progressive NAS* generates its candidate pipelines with the prediction of LSTM or MLP.

Note that there is also one important difference in Step 3 among the surrogate-model-based algorithms, i.e., the number of sampled pipelines for evaluation. *Progressive NAS* generates top-k (k > 1) pipelines for evaluation. In the running process of *TPE* and *SMAC*, actually they produce several candidate pipelines when sampling. However, they pick up only one pipeline with the best acquisition score for the next-step evaluation.

Step 4: Evaluate sampled pipelines (Line 7): The last step is to evaluate newly sampled pipelines. The goal is to update the population (like *TEVO_H*, *TEVO_Y* and *PBT*) or collect new data to fit a better surrogate model. In this step, the preprocessed training data is used to train downstream ML models. Then, the preprocessed validation data is used to test the accuracy of trained ML models. The higher the accuracy, the better the preprocessing pipeline.

5 HOW DO DIFFERENT SEARCH ALGORITHMS PERFORM?

In practice, when users are looking for a feature preprocessing pipeline with good quality, they often constrain search time. In this section, we investigate how these search methods perform within specified time limits. At first, we compare these algorithms and rank them. Then, we analyze the performance bottleneck of these algorithms and identify the opportunities to further improve them.

5.1 Experimental Setup

Datasets. We search for a large collection of real-world datasets (in total 45 datasets) from the widely-known AutoML challenge website [1], an AutoML Benchmark [23], and Kaggle datasets [3]. Without losing generalization, for categorical and textual features, we need to first transform them into numerical features and then search FP pipelines for numerical features. That is why we focus on the 45 numerical datasets because the conclusion drawn from numerical datasets can be widely used in all scenarios.

As shown in Figure 5, these selected datasets have diverse characteristics in terms of file size, the number of rows/columns and binary/multi-classification. The file size of these datasets is from 0.01 MB to 75.2 MB. The number of rows of all datasets is from 242 to 464,809. The number of columns of all datasets is from 4 to 1,636. There are 28 binary classification datasets and 17 multi-classification datasets with up to 100 classes. These broad considerations promise the generalization of our experiments. The detailed information of datasets is shown in our technical report [6].

Search Algorithms. As shown in Table 3, we consider 15 search algorithms in total. Some of them have been included in famous `Python` libraries. E.g., `HyperOpt` [14] includes *Anneal* [34] and *TPE*, `SMAC3` [42] includes *SMAC* algorithm, and `HyperbandSter` [2] includes *Hyperband* and *BOHB*. We slightly change these libraries to make their algorithms support our scenario. For NAS algorithms like *Progressive NAS* and *ENAS*, they are originally implemented in `PyTorch`. However, they do not fit into our scenario, and thus we re-implement them based on their papers. We also implement *REINFORCE*, *Tournament Evolution*, and *PBT*. Note that all algorithms are implemented in `Python`, which is fair for comparison.

Downstream Classifiers. We evaluate search algorithms using three downstream classifiers: Logistic Regression (LR), XGBoost (XGB) and Multi-layer Perceptron (MLP). We choose the three ML models based on the recent survey [5]. *LR* is a linear model which takes the top popularity of all ML models, *XGB* is a tree-based model which takes the first popularity of complex ML models, and *MLP* is a neural network which gains more popularity recently. Choosing them means that our experimental results can provide insights on a wide range of scenarios. In terms of implementation, we use *LR* (set n_jobs = 1) and *MLP* in the Scikit-learn library with default parameters and use the *XGB* model (set n_jobs = 1) in the XGBoost library.

Training and Evaluation. For each dataset, we split it into training and validation with the proportion 80:20. The training set is used to generate trained downstream models after being preprocessed. And the preprocessed validation set is used to evaluate the trained models. Based on evaluations, search algorithms can derive information and choose the proper search direction for the next steps. After an iterative search of preset time limits, search algorithms stop their work and output the feature preprocessing pipeline with the highest validation accuracy.

Experimental Environment. We conduct experiments on a guest virtual machine with 110 vCPUs and 970GB main memory. The guest virtual machine runs on a Linux Kernel-based Virtual Machine (KVM) enabled server equipped with four Intel Xeon E7-4830 v4 CPUs clocked at 2.0GHz and 1TB main memory. Each CPU has 14 cores (28 hyperthreads) and 35MB Cache. All of the experiments are repeated five times and we report the average to avoid the influence of hardware and network.

5.2 Which Search Algorithm Performs Better?

We run the 15 algorithms on 45 datasets under 6 different time constraints: 60, 300, 600, 1200, 1800 and 3600 seconds. We choose these settings because it is more practical to mainly concern about the performance of FP with lower resources and leave some resources for tasks like feature generation and selection. Due to the space constraint, we report the general findings derived from the comprehensive experimental results on all datasets with all time constraints. Detailed experimental results are shown in the technical report [6].

Which search algorithm ranks better? To give a recommendation, we compute the average rankings of all 15 algorithms

Table 4: Overall Average Performance Ranking of All Search Algorithms.

Category	Traditional		Evolution-based			Surrogate-model-based						RL-based		Bandit-based	
Search Alg	RS	Anneal	PBT	TEVO_Y	TEVO_H	PMNE	PME	PLNE	PLE	SMAC	TPE	REINFORCE	ENAS	HYPERBAND	BOHB
LR Avg Ranking	6	12	**1**	2	3	4	5	11	14	7	8	10	15	9	13
XGB Avg Ranking	6	13	**1**	2	3	4	5	11	14	8	7	9	15	10	12
MLP Avg Ranking	7	12	3	4	5	**1**	2	6	10	8	9	14	15	11	13
Overall Avg Ranking	6	12	**1**	2	3	4	5	9	13	7	8	11	15	10	14

under all scenarios with at least 1.5% improvement of validation accuracy compared to no-FP (215 scenarios on *LR* + 90 scenarios on *XGB* + 196 scenarios on *MLP* = 501 scenarios). The reason we choose no-FP as a baseline instead of data processed by single preprocessors is that we want to compare the validation accuracy with/without FP instead of comparing the effectiveness between multiple and single preprocessors. Naturally, the scope of FP includes single preprocessors. The ranking value in each scenario is also according to validation accuracy. If there is a tie, we give the same ranking value. Table 4 shows the ranking results. We can see that *PBT* is ranked at the top, followed by the other two evolution-based algorithms. More specifically, when the downstream model is *LR* or *XGB*, evolution-based algorithms are highly recommended; when the downstream model is *MLP*, *PMNE* and *PME* are highly recommended. The overall average ranking of *RS* is 6, which is still a strong baseline. Our following analysis also takes *RS* as a baseline.

Why evolution-based algorithms outperform *RS*? One possible reason that evolution-based algorithms exceed *RS* is that they have more exploitation than *RS*, which can produce promising search directions for the next steps. For example, *TEVO_H* mutates the best pipeline of sampled pipelines from the population. Moreover, evolution-based algorithms require a small overhead to select the next pipeline since they just sample and then mutate. This allows them to evaluate many more pipelines under the same budget.

Why most surrogate-model-based algorithms do not outperform *RS*? We observe that most surrogate-model-based algorithms do not outperform *RS*, except *PMNE* and *PME*. The goal of utilizing surrogate models is to direct the search direction for the next steps through a model, thus precise directing is important for the performance of these surrogate-model-based algorithms. However, these surrogate-model-based algorithms that need initialization do not have enough data to promise the initial directions or only start with randomness. For example, the starting points of *PLNE* are just the 7 pipelines with only one preprocessor. Furthermore, the fitting process of these surrogate models is time-consuming, which causes fewer pipelines evaluated. For example, *SMAC* needs to train a random forest, *TPE* needs to fit many KDEs, *PLNE* and *PNE* need to fit a LSTM and multi number of LSTMs. This insight also inspires us that the search space of feature preprocessing pipelines is hard to learn by general surrogate models like the random forest, KDE and LSTM, and there is still space to improve these surrogate models for Auto-FP scenario especially. However, the special cases here are *PMNE* and *PME*. Even though they have non-precise starting points like *PLNE* and *PLE*, the overhead of the fitting process of MLP is very small (approximate to *RS* as shown in Figure 7), which leaves more time to train more precise MLP(s) with enough number of pipelines evaluated.

Why RL-based algorithms show poor performance? It is obvious that *REINFORCE* and *ENAS* do not perform well. There are two reasons. Firstly, the initialization of *REINFORCE* and *ENAS* is random, which is not so effective for finding a promising direction at the starting stage. Secondly, *REINFORCE* and *ENAS* employ the idea of stochastic gradient descent, which updates the policy after only one evaluation, which means the process of finding a good policy is slow and with lots of iterations.

Why bandit-based algorithms show low average ranking? The average rankings of *Hyperband* and *BOHB* are also behind *RS*. Their main early-stopping idea cannot grasp the correct pipeline ranking at the early stage for downstream ML models under our setting. Even though we try many possibilities of the two important parameters η and `min_budget` (See Figure 6. Due to the space limit, we only exhibit the result of *Jasmine* with the *LR* model), it is still hard to make the two algorithms exceed *RS*. Also, it is hard to determine which parameter is better with different downstream ML models and time limits. Clearly, how to improve *Hyperband* and *BOHB*, especially for the Auto-FP scenario, still needs further exploration.

Are there any frequent excellent feature preprocessor patterns? We tried to dig out if there are frequent patterns in the best pipelines of all 45 datasets searched by *PBT* (top 1 ranking search algorithm) with FP-growth [26] (a famous frequent pattern mining algorithm). However, the support of discovered patterns is very low, i.e. there are no obvious frequent patterns. This result further motivates our search idea and indicates that the Auto-FP problem is hard because of the large search space.

5.3 Performance Bottleneck Analysis

We investigate the performance bottleneck of different algorithms to identify opportunities to enhance their performance. We break down the performance into three parts. (1) "Pick": picking up FP pipelines to be evaluated, i.e. picking-up time, which includes `Step 2` and `Step 3` in Algorithm 1. (2) "Prep": preprocessing training and validation datasets with picked pipelines, i.e. preprocessing time, which is included in `Step 4` of Algorithm 1. (3) "Train": training ML models with preprocessed training dataset, i.e., training time, which is also included in `Step 4` of Algorithm 1. We conduct experiments on all datasets under different time limits. Due to the space constraint, we only show the results of 10 mins on 7 datasets in Figure 7. Note that *Hyperband* and *BOHB* are not shown because the two methods mix the picking-up and evaluation time and adopt partial training, making it impossible to record each part separately as other algorithms.

What is the most common bottleneck? Different search algorithms have different time distributions among the three parts, as shown in Figure 7. Obviously, "Train" is the bottleneck in most cases, followed by "Prep", then "Pick". "Train" and "Prep" are highly related to data size, i.e. the smaller data size, the shorter "Train" and "Prep" time. Therefore, reducing data size (e.g. by sampling) is meaningful for improving the performance.

Are there explicit data characteristic rules that can be used to figure out bottlenecks? To further indicate bottlenecks under different scenarios, we try to conclude the relationship between data characteristics and the type of bottleneck. According to the number of dataset dimensions, we split the 45 datasets into *high-dimensional datasets* (# of dimensions > 100) and *low-dimensional datasets* (# of dimensions <= 100). The *low-dimensional datasets* can also be split into three groups according to their size: small (size <= 1.6MB), medium (1.6MB < size <= 4MB) and large (size > 4MB). Combined with the complexity of the downstream ML model, we draw Table 5 to help researchers develop optimized

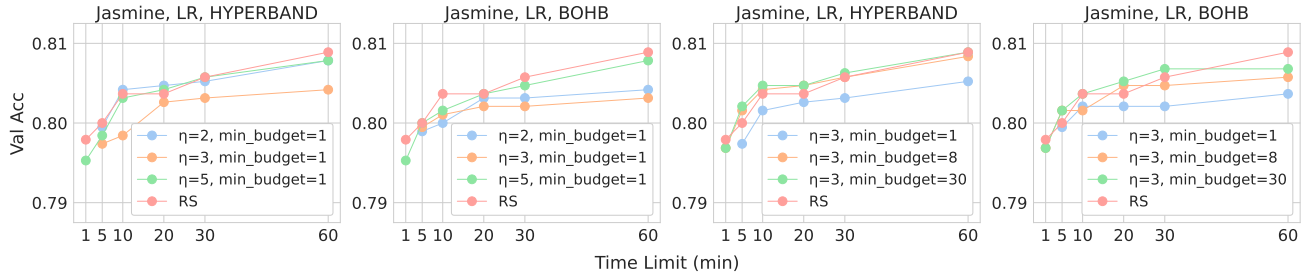

Figure 6: Parameter adjustment for *Hyperband* and *BOHB*. The upper two vary η. The lower two vary `min_budget`. Even with several parameter adjustments, *Hyperband* and *BOHB* still cannot outperform *RS*.

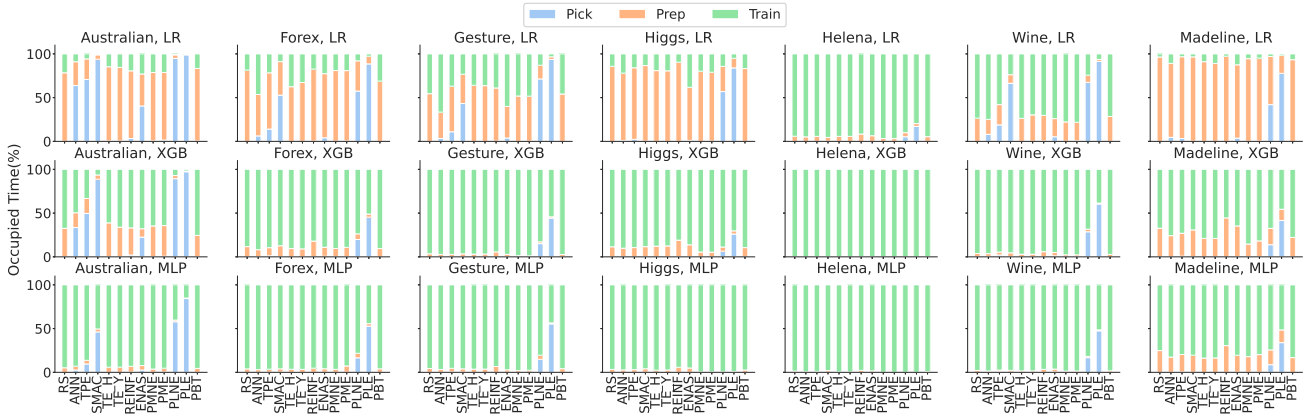

Figure 7: Overhead Percentage on 7 datasets with different downstream ML models. "Pick" means the overhead of picking up next pipelines. "Prep" means the overhead of preprocessing dataset with feature preprocessors. "Train" means the overhead of evaluating FP pipelines.

Table 5: Performance Bottleneck of Different Scenarios.

Dataset Dimensions	Dataset Size	ML Model	*RS*	*PBT*	*TEVO_H*	*TEVO_Y*
High	All	LR	Prep			
		XGB	Train			
		MLP				
Low	Small	LR	Prep/Train			
		XGB	Train			
		MLP				
	Medium	LR	Prep			
		XGB	Train			
		MLP				
	Large	LR	Prep/Train			
		XGB	Train			
		MLP				

searched algorithms tailored for different scenarios. For example, if some researchers tend to utilize *PBT* to search FP pipeline for high-dimensional dataset with *XGB* model, it is better for her to enhance the performance by solving the "Train" bottleneck.

5.4 Main Findings

Our main findings of this section are summarized as follows:

- Evolution-based algorithms, especially *PBT*, give the highest overall average ranking under all scenarios. They have more exploitation with a similar small overhead for picking up the next pipeline compared to *RS*.
- *RS* is still a strong baseline.
- Most surrogate-model-based algorithms except *PMNE* and *PME* do not outperform *RS* because of the imprecise initialization and time-consuming surrogate model fitting process.
- RL-based algorithms do not exceed *RS* because of the imprecise initialization and time-consuming policy learning process.
- Bandit-based algorithms do not exceed *RS* because the early-stopping cuts-off good pipelines in the early stage of training when the downstream ML models are *LR*, *XGB* and *MLP*.
- There is no obvious frequent feature preprocessor pattern always performing well.
- Different scenarios have different bottlenecks and "Train" is the bottleneck in most cases. Users can check Table 5 for indicating the most promising direction to enhance performance.

6 EXTENDING AUTO-FP TO SUPPORT PARAMETER SEARCH

In this section, we explore two extended search spaces and evaluate two approaches to extend Auto-FP to support parameter search. Furthermore, we also discuss in which situation one is superior to the other and explain the reasons.

6.1 Two Extended Search Spaces

We extend our search space by allowing preprocessors' parameters to have multiple possible values. For example, *Binarizer* has a parameter called "threshold". As mentioned in Section 2.1, its default value is 0. The extended search space extends the parameter space to a set of values such as {0, 0.2, 0.4, 0.6, 0.8, 1.0}. Based on the number of available values, i.e. the cardinality for each parameter, we present two types of extended search spaces: low-cardinality and high-cardinality search space.

Low-Cardinality Search Space. We first construct an extended search space as shown in Table 6, which is the low-cardinality search space. In low-cardinality search space, the value of max cardinality is small. For example, the max cardinality in Table 6 is the cardinality of `n_quantiles`, which is 8.

High-Cardinality Search Space. We construct another extended search space by significantly increasing the number of possible values for some parameters. We changed the `threshold` space of *Binarizer* into a list from 0 to 1 with a gap of 0.05, and the `n_quantiles` space of *QuantileTransformer* into a list from 10 to

Table 6: Extended Low-Cardinality Search Space. The max cardinality is the cardinality of `n_quantiles`, which is 8.

Preprocessor	Space Configures
Binarizer	threshold = [0,0.2,0.4,0.6,0.8,1.0]
MinMaxScaler	range_min=0 range_max = 1
MaxAbsScaler	No parameter
Normalizer	norm = ['l1', 'l2', 'max']
StandardScaler	with_mean = [True, False]
PowerTransformer	standardize = [True, False]
QuantileTransformer	n_quantiles = [10, 100, 200, 500, 1000, 1200, 1500, 2000] output_distribution = ['uniform', 'normal']

Table 7: Extended High-Cardinality Search Space. The max cardinality is the cardinality of `n_quantiles`, which is 1990.

Preprocessor	Space configures
Binarizer	'threshold' = from 0 to 1 with 0.05 step
MinMaxScaler	'range_min'=0 'range_max' = 1
MaxAbsScaler	No parameter
Normalizer	'norm' = ['l1', 'l2', 'max']
StandardScaler	'with_mean' = [True, False]
PowerTransformer	'standardize' = [True, False]
QuantileTransformer	'n_quantiles' = from 10 to 2000 with 1 step 'output_distribution' = ['uniform', 'normal']

2000 with a gap of 1. As shown in Table 7, in the high-cardinality search space, there are parameters with very high cardinality, i.e. the value of max cardinality is large. For example, the max cardinality in Table 7 is the cardinality of `n_quantiles`, which is 1990.

6.2 Two Approaches Supporting Parameter Search

We adopt two approaches to extend Auto-FP to support parameter search. The first one, called *One-step*, combines parameter search and pipeline search together. The second one, called *Two-step*, treats the parameter search and the pipeline search separately.

One-step. This approach considers each preprocessor with different selected parameters as different preprocessors. For example, *Binarizer(*`threshold=0`*)* and *Binarizer(*`threshold=1`*)* are considered two different preprocessors. In this way, for the low-cardinality search space, the number of preprocessors is even and will be increased from 7 to 6 + 1 + 1 + 3 + 2 + 2 + 16 = 31. After that, any search algorithm presented before can be directly applied to search for the best sequence of preprocessors along with associated parameter values.

Two-step. This approach consists of two steps. In the first step, it randomly selects the parameter values for each preprocessor. For example, `threshold=1` is selected for *Binarizer* and `with_mean=False` is selected for *StandardScaler*. In the second step, it runs a search algorithm with a short time limit (like 60s) to search for the best pipeline w.r.t. the selected parameter values. It repeats the two steps until the time budget is exhausted and finally returns the best overall pipeline. Note that other sample techniques in the first step besides random can be further explored, but it is outside the scope of this work.

6.3 One-step vs. Two-step.

We compare the two approaches on all datasets. We choose the *PBT* as the search algorithm because it shows the best overall average ranking.

For low-cardinality search space, One-step or Two-step? We first varied the time limit and used each approach to search for the best pipeline for the extended low-cardinality search space in Table 6. Due to the space constraint, Figure 8 shows the results on *Australian* and *Madeline*. The complete results are shown in our technical report [6]. The "Val Acc" refers to the average accuracy over five runs. We can see that for most cases, *One-step* outperforms *Two-step*. This is because *Two-step* only exploits at most one group of parameter values every minute. Even if we increase the time limit to 1 hour, it only goes through at most 60 groups of parameter values, which does not do enough exploration. In comparison, *One-step* can pick up more reasonable pipelines with more exploration.

For high-cardinality search space, One-step or Two-step? However, *One-step* has its own limitation. Compared to the low-cardinality search space in Table 6, the `n_quantiles` parameter of *QuantileTransformer* in the high-cardinality search space has about 2k possible values, which leads *One-step* to choose *QuantileTransformer* with a much higher opportunity. So does the `threshold` parameter of *Binarizer*. We run the same experiments as in Figure 8 w.r.t. the high-cardinality search space and get the results in Figure 9. We can see that in most cases, *One-step* performs worse than *Two-step* because *One-step* selects pipelines with many duplicate preprocessors. For example, it may return a pipeline like "QuanitleTransformer(n_quantiles=10) → QuantileTransformer(n_quantiles=50) → QuantileTransformer(n_quantiles=200)". It is natural because *QuantileTransformer* takes a large proportion of preprocessors in this space, which is 4000/4027 ≈ 99.3%, making the search algorithm less likely to select other preprocessors. *Two-step* can avoid this issue and thus perform better.

In summary, our experimental study shows that different types of extended search space fit different approaches. In low-cardinality search space, *One-step* is a preferred approach. However, in high-cardinality settings (e.g., one preprocessor dominates the search space), it may perform worse than *Two-step*. It is an open research problem to combine pipeline search and parameter search in a systematic way.

6.4 Main Findings

Our main findings of this section are summarized as follows:

- Different types of extended search space fit different approaches.
- For extended low-cardinality search space, *One-step* is preferred because it can do more exploration than *Two-step*.
- For extended high-cardinality search space, *Two-step* is preferred because it can avoid selecting many duplicated preprocessors in pipelines.
- Better combining pipeline and parameter search is still an important open problem which deserves further exploration.

7 PUTTING AUTO-FP IN AN AUTOML CONTEXT

AutoML aims to democratize machine learning by automating the search for the best ML pipeline. Recent decomposed Auto-ML works [41, 44, 60, 66] introduced space decomposition to accelerate the search process. However, existing approaches use a one-size-fits-all search strategy for all subspaces. A more reasonable choice is to design specific solutions for each subspace. By considering FP space as one individual search space in the AutoML context, we first figure out the answers to the two essential questions: *1) Does Auto-FP outperform the feature preprocessing module in AutoML? 2) Is Auto-FP Important in AutoML Context?* To answer these questions, we investigate three popular open-source AutoML systems and evaluate the effectiveness of Auto-FP in an AutoML context using our benchmark datasets. After getting

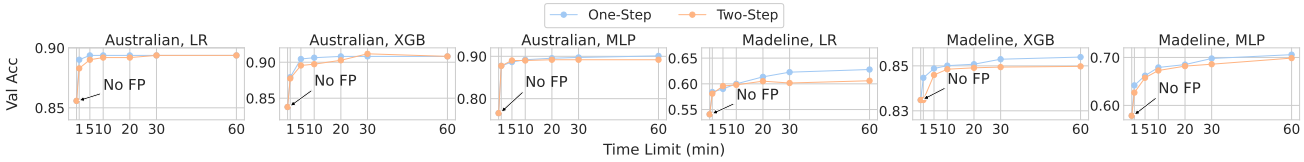

Figure 8: Comparison of One-step and Two-step in the extended low-cardinality search space in Table 6. One-step is preferred.

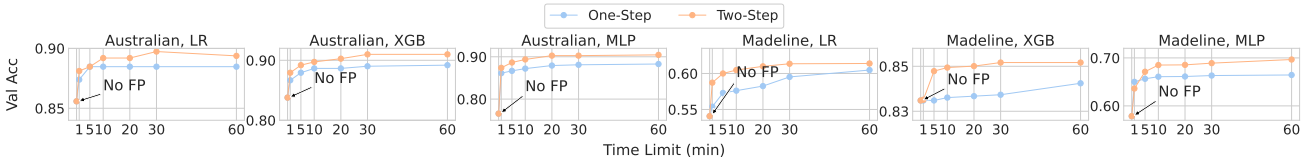

Figure 9: Comparison of One-step and Two-step in the extended high-cardinality search space in Table 7. Two-step is preferred.

the answers to the two essential questions, we provide the discussion on *What AutoML can learn from Auto-FP* to indicate the opportunities the Auto-FP experience can provide for building more powerful AutoML.

7.1 Does Auto-FP Outperform FP in AutoML?

Since FP is part of an ML pipeline, a general-purpose AutoML tool can be configured to solve an Auto-FP problem, by only enabling the FP component and disabling all other components. However, after investigating several popular AutoML systems, we find that their FP modules are quite limited. Table 8 shows the capability of the FP module of Auto-WEKA [59], Auto-Sklearn [21], and TPOT [48], respectively. We can see that Auto-WEKA does not provide any preprocessor, thus it cannot be applied to find an FP pipeline. In comparison, Auto-Sklearn provides five preprocessors, but each FP pipeline only contains a single preprocessor and one search algorithm. TPOT allows a pipeline to contain an arbitrary number of preprocessors, but compared to Auto-FP, it has fewer preprocessors and only considers one search algorithm.

We compare FP in AutoML with Auto-FP. We use TPOT which has five preprocessors and applies the genetic programming [10] algorithm for this comparison because it has a more sophisticated FP module compared to Auto-WEKA and Auto-Sklearn. We adopt the same experimental setting as Section 5.1 and set the time limit to 600 seconds. For TPOT, we disable all components except the FP module. For Auto-FP, the leading search algorithm *PBT* is employed. Figure 10 shows the results of six datasets and the complete results on all 45 datasets are shown in [6]. We can see that Auto-FP outperforms TPOT-FP in four, three and five datasets out of six datasets (24, 25 and 25 datasets out of 45 datasets) when downstream models are LR, MLP and XGB, respectively. Besides default search space, in Figure 11, Auto-FP outperforms TPOT-FP in four, three and five datasets out of six datasets (24, 29, 24 out of 45 datasets) when downstream models are LR, MLP and XGB. Obviously, the conclusion that Auto-FP outperforms FP in AutoML can be generalized into a wider search space. There are two reasons that Auto-FP outperforms TPOT-FP. Firstly, Auto-FP considers more feature preprocessors. Secondly, Auto-FP adopts a better search algorithm. This result validates the necessity of designing specific solutions for each subspace.

7.2 Is Auto-FP Important in AutoML Context?

To illustrate whether Auto-FP is important in the AutoML context, we explore whether Auto-FP is as important as other well-known important modules in AutoML. To answer the question, we compare Auto-FP against HPO, a well-known and highly effective module in AutoML. We also adopt the same experimental setting as Section 5.1 and set the time limit to 600 seconds. For TPOT, we disable all other parts except HPO. For Auto-FP, we still use *PBT* as the search algorithm. Figure 11 shows the results. We can see that Auto-FP outperforms HPO in all the datasets (40 and 37 outperform datasets out of 45 datasets) when the downstream models are LR and MLP. When the downstream model is XGB, out of six datasets, Auto-FP achieves better accuracy in three datasets (22 outperform and 2 competitive datasets out of 45 datasets). This result indicates that FP is as important as HPO and contributes greatly to downstream model performance improvement. Still in extended search space (Table 6), Auto-FP outperforms HPO in all datasets (40 and 36 datasets out of 45 datasets) when downstream models are LR and MLP. For XGB, Auto-FP is better in four datasets (29 outperform and 2 competitive datasets out of 45 datasets). Obviously, Auto-FP is still as important as HPO even in other search space.

Table 8: Feature preprocessing module in popular open-source AutoML systems.

AutoML System	Preprocessors#	Pipeline Len.	Search Algo.
Auto-WEKA [59]	0	0	SMAC [29]
Auto-Sklearn [21]	5	1	SMAC [29]
TPOT [48]	5	arbitrary	GP [10]

7.3 What can AutoML learn from Auto-FP?

Mainstream AutoML systems are monolithic and they aim to use a one-size-fits-all search algorithm to automate every step (feature preprocessing, feature selection, hyperparameter tuning, etc.) in an ML pipeline. Due to the use of a monolithic architecture, mainstream AutoML systems suffer from two limitations. Firstly, to control the overall search space, they tend to use a smaller search space for each step. For example, the search space of the FP module of Auto-Sklearn contains only five pipelines while that of Auto-FP contains about 1 million pipelines. Secondly, the one-size-fits-all search algorithm may not be suitable for every step of the pipeline. For example, the GP algorithm adopted by TPOT may not be the best search algorithm for hyperparameter tuning.

To overcome the two limitations, combining the previously mentioned space decomposition idea and designing specific solutions for each subspace is a promising direction. By employing space decomposition, AutoML can use a larger search space for each component without extending the whole search space exponentially. By designing specific solutions for each subspace, there is an opportunity for AutoML to improve its performance. To achieve this goal, the research community should conduct

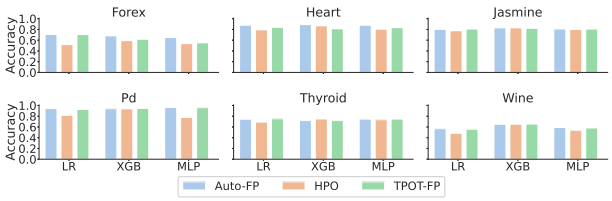

Figure 10: Evaluate Auto-FP in an AutoML context (default search space). In most cases, Auto-FP outperforms TPOT-FP and comparable to HPO module in default search space.

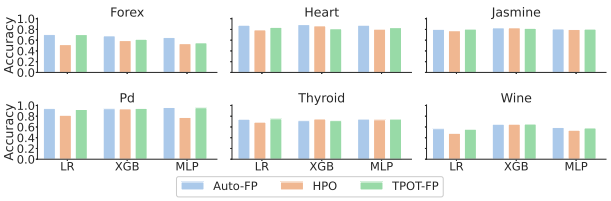

Figure 11: Evaluate Auto-FP in an AutoML context (extended search space in Table 6). In most cases, Auto-FP outperforms TPOT-FP and comparable to HPO module also in extended search space.

more benchmarks and develop the best solution tailored for each task, such as Automated Feature Generation/Selection.

7.4 Main Findings

Our main findings of this section are summarized as follows:

- Auto-FP outperforms FP in AutoML in most cases by considering larger search space and adopting better search algirithms.
- Auto-FP is important in the AutoML context. Specifically, it is as important as the well-known HPO module.
- Limited search space and a one-size-fits-all search algorithm are two limitations mainstream AutoML systems are suffering. Combining the popular space decomposition idea and deploying specific solutions such as Auto-FP for each subspace is a promising direction to refurbish current systems.

8 RESEARCH OPPORTUNITIES

Warm-start search algorithms. Evolution-based algorithms show a leading position in the Auto-FP scenario. Intuitively, a better initial population instead of random initialization is important for searching good pipelines faster. How to better warm start evolution-based algorithms for Auto-FP scenario is worthy to investigate. Meta-learning is an alternative way which is leveraged for warm-starting HPO [11, 24, 52, 53, 63–65, 70] and database tuning [71]. For Auto-FP, the initial population of newly-coming tasks can also be warm-started by historical tasks encoded by meta-features.

Allocate pipeline and parameter search time budget reasonably. For better supporting parameter search in the Auto-FP scenario, how to allocate the limited search time to different stages, i.e. pipeline search and parameter search is worthy of exploration [50, 61]. Allocating too much time for searching pipelines may reduce the opportunity to fine-tune good pipelines and get better performance. While allocating too much time for searching parameter may miss promising pipelines. There is still a trade-off between the time budget for searching pipelines and parameters.

Benchmark Auto-FP on various data types and deep models. In addition to our focus on tabular data, it is valuable to evaluate Auto-FP's performance on different data types like text and image data. Text data can benefit from feature preprocessors such as TF-IDF and word embeddings, while image data commonly utilizes random cropping and normalization. Combining these specific preprocessors with Auto-FP's existing ones could provide a more comprehensive understanding of its capabilities. Furthermore, while our paper explores popular ML models, deep models like DeepFM and DCN are dominant in specific domains like recommendation tasks. Auto-FP can be applied to deep models. However, deep models for specific tasks may require tailored search algorithms. Thus, benchmarking Auto-FP on deep models for specific tasks would offer practical insights for its application.

9 RELATED WORK

HPO and NAS. Auto-FP is similar in spirit to HPO and NAS. The goal of HPO is to find the best combination of a classifier and its related hyperparameters for a given dataset. There are many search algorithms proposed for HPO [12, 13, 19, 29, 30, 39]. NAS aims to automatically find the best DNN architecture. Zoph and Le [74] did the pioneering work in NAS. Later, many improved NAS algorithms are proposed [43, 49, 51, 72, 75]. In this paper, we model Auto-FP as HPO and NAS respectively in order to utilize these search algorithms. However, Auto-FP is fundamentally different because it has a totally difference search space. The search space of HPO includes classifiers and hyperparameters and the search space of NAS includes DNN operators, while the search space of Auto-FP includes feature preprocessors.

AutoML/Data Preparation Pipeline Search. Recently, AutoML attracts great attention from the database community [37, 41, 55, 66]. However, existing studies are mainly focused on optimizing the entire AutoML pipeline. We have discussed how AutoML systems can benefit from our study in Section 7. Our work is also related to *AutoPipeline* [67], which aims to generate multiple data preparation steps automatically. It would be interesting to explore whether our Auto-FP search algorithms can be applied to solve their problem.

AutoML Benchmark. OpenML [23] is a popular AutoML benchmark which utilizes 39 public datasets to evaluate classification performance with different time slots and different metrics. Zogaj et al. [73] conduct an extensive empirical study to investigate the impact of downsampling on AutoML results. Several benchmarks are published in the NAS area called NAS-Bench 101 [69], 201 [18] and 301 [57]. There are also benchmarks for some stages in the data science life cycle such as data cleaning [40] and feature type recognition [54]. Different from existing work, we are the first to benchmark automated feature processing.

10 CONCLUSION

In this paper, we studied an essential task in classical ML–feature preprocessing. We justified the importance of FP and pointed out that FP should be automated due to its large search space. We benchmarked Auto-FP with 15 search algorithms from HPO and NAS, 3 popular downstream ML models, and 7 widely-used preprocessors on 45 public datasets. We found that evolution-based algorithms take the top position. To further enhance Auto-FP, We also explored two kinds of extended parameter search space and compared two methods to support parameter search. We concluded that different search spaces fit different methods. In the end, we evaluated Auto-FP in an AutoML context and figured out that Auto-FP is an important part in the AutoML context which deserves a specific solution. How to decompose the AutoML search space reasonably and conduct effective solutions for other tasks in the AutoML context is a promising direction for the community to further explore.

REFERENCES

[1] 2021. AutoML Challenge Website. https://automl.chalearn.org/data.

[2] 2021. HpBandSter. https://automl.github.io/HpBandSter/build/html/index.html.

[3] 2021. Kaggle Datasets. https://www.kaggle.com/datasets.

[4] 2021. Scikit-learn: Machine Learning in Python. https://scikit-learn.org/stable/.

[5] 2021. State of Data Science and Machine Learning 2021. https://www.kaggle.com/kaggle-survey-2021.

[6] 2022. Auto-FP: An Experimental Study of Automated Feature Preprocessing for Tabular Data (Technical Report). (2022). https://github.com/AutoFP/Auto-FP/blob/main/Auto-FP(technical_report).pdf

[7] 2022. Scikit-learn Documentation. https://scikit-learn.org/stable/modules/preprocessing.html.

[8] 2022. Scikit-learn: Preprocessing Data. https://scikit-learn.org/stable/modules/preprocessing.html.

[9] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47 (2002), 235–256.

[10] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. 1998. *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications.* Morgan Kaufmann Publishers Inc.

[11] Rémi Bardenet, Mátyás Brendel, Balázs Kégl, and Michèle Sebag. 2013. Collaborative hyperparameter tuning. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013 (JMLR Workshop and Conference Proceedings)*, Vol. 28. JMLR.org, 199–207. http://proceedings.mlr.press/v28/bardenet13.html

[12] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*, John Shawe-Taylor, Richard S. Zemel, Peter L. Bartlett, Fernando C. N. Pereira, and Kilian Q. Weinberger (Eds.). 2546–2554. https://proceedings.neurips.cc/paper/2011/hash/86e8f7ab32cfd12577bc2619bc635690-Abstract.html

[13] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research* 13, 10 (2012), 281–305. http://jmlr.org/papers/v13/bergstra12a.html

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

[15] Jason Brownlee. 2020. *Data preparation for machine learning: data cleaning, feature selection, and data transforms in Python.*

[16] Girish Chandrashekar and Ferat Sahin. 2014. A survey on feature selection methods. *Computers & Electrical Engineering* 40, 1 (2014), 16–28.

[17] Krishna Teja Chitty-Venkata, Murali Emani, Venkatram Vishwanath, and Arun K. Somani. 2023. Neural Architecture Search Benchmarks: Insights and Survey. *IEEE Access* 11 (2023), 25217–25236. https://doi.org/10.1109/ACCESS.2023.3253818

[18] Xuanyi Dong and Yi Yang. 2020. NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=HJxyZkBKDr

[19] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research)*, Jennifer G. Dy and Andreas Krause (Eds.), Vol. 80. PMLR, 1436–1445. http://proceedings.mlr.press/v80/falkner18a.html

[20] Matthias Feurer and Frank Hutter. 2019. Hyperparameter optimization. In *Automated machine learning*. Springer, Cham, 3–33.

[21] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and Robust Automated Machine Learning. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'15)*. 2755–2763.

[22] George Forman et al. 2003. An extensive empirical study of feature selection metrics for text classification. *J. Mach. Learn. Res.* 3, Mar (2003), 1289–1305.

[23] P. Gijsbers, E. LeDell, S. Poirier, J. Thomas, B. Bischl, and J. Vanschoren. 2019. An Open Source AutoML Benchmark. *arXiv preprint arXiv:1907.00909 [cs.LG]* (2019). https://arxiv.org/abs/1907.00909 Accepted at AutoML Workshop at ICML 2019.

[24] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. 2017. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*. ACM, 1487–1495. https://doi.org/10.1145/3097983.3098043

[25] PC Hammer. 1962. Adaptive control processes: a guided tour (R. Bellman).

[26] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining Frequent Patterns without Candidate Generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein (Eds.). ACM, 1–12. https://doi.org/10.1145/342009.335372

[27] Xin He, Kaiyong Zhao, and Xiaowen Chu. 2021. AutoML: A survey of the state-of-the-art. *Knowl. Based Syst.* 212 (2021), 106622. https://doi.org/10.1016/j.knosys.2020.106622

[28] Franziska Horn, Robert Pack, and Michael Rieger. 2019. The autofeat Python Library for Automated Feature Engineering and Selection. In *Machine Learning and Knowledge Discovery in Databases - International Workshops of ECML PKDD 2019, Würzburg, Germany, September 16-20, 2019, Proceedings, Part I (Communications in Computer and Information Science)*, Peggy Cellier and Kurt Driessens (Eds.), Vol. 1167. Springer, 111–120. https://doi.org/10.1007/978-3-030-43823-4_10

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

[30] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Kevin Murphy. 2010. Time-bounded sequential parameter optimization. In *International Conference on Learning and Intelligent Optimization*. Springer, 281–298.

[31] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. 2017. Population Based Training of Neural Networks. *CoRR* abs/1711.09846 (2017). arXiv:1711.09846 http://arxiv.org/abs/1711.09846

[32] Kevin G. Jamieson and Ameet Talwalkar. 2016. Non-stochastic Best Arm Identification and Hyperparameter Optimization. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016 (JMLR Workshop and Conference Proceedings)*, Arthur Gretton and Christian C. Robert (Eds.), Vol. 51. JMLR.org, 240–248. http://proceedings.mlr.press/v51/jamieson16.html

[33] Gilad Katz, Eui Chul Richard Shin, and Dawn Song. 2016. ExploreKit: Automatic Feature Generation and Selection. In *IEEE 16th International Conference on Data Mining, ICDM 2016, December 12-15, 2016, Barcelona, Spain*, Francesco Bonchi, Josep Domingo-Ferrer, Ricardo Baeza-Yates, Zhi-Hua Zhou, and Xindong Wu (Eds.). IEEE Computer Society, 979–984. https://doi.org/10.1109/ICDM.2016.0123

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

[35] Sanjay Krishnan and Eugene Wu. 2019. AlphaClean: Automatic Generation of Data Cleaning Pipelines. *CoRR* abs/1904.11827 (2019). arXiv:1904.11827 http://arxiv.org/abs/1904.11827

[36] Max Kuhn and Kjell Johnson. 2021. *Feature engineering and selection: A practical approach for predictive models.* Chapman amp; Hall/CRC.

[37] Doris Jung Lin Lee, Stephen Macke, Doris Xin, Angela Lee, Silu Huang, and Aditya G Parameswaran. 2019. A Human-in-the-loop Perspective on AutoML: Milestones and the Road Ahead. *IEEE Data Eng. Bull.* 42, 2 (2019), 59–70.

[38] Jundong Li, Kewei Cheng, Suhang Wang, Fred Morstatter, Robert P Trevino, Jiliang Tang, and Huan Liu. 2017. Feature selection: A data perspective. *ACM computing surveys (CSUR)* 50, 6 (2017), 1–45.

[39] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *J. Mach. Learn. Res.* 18 (2017), 185:1–185:52. http://jmlr.org/papers/v18/16-558.html

[40] Peng Li, Xi Rao, Jennifer Blase, Yue Zhang, Xu Chu, and Ce Zhang. 2021. CleanML: A Study for Evaluating the Impact of Data Cleaning on ML Classification Tasks. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 13–24. https://doi.org/10.1109/ICDE51399.2021.00009

[41] Yang Li, Yu Shen, Wentao Zhang, Jiawei Jiang, Yaliang Li, Bolin Ding, Jingren Zhou, Zhi Yang, Wentao Wu, Ce Zhang, and Bin Cui. 2021. VolcanoML: Speeding up End-to-End AutoML via Scalable Search Space Decomposition. *Proc. VLDB Endow.* 14, 11 (2021), 2167–2176. http://www.vldb.org/pvldb/vol14/p2167-li.pdf

[42] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, René Sass, and Frank Hutter. 2021. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. arXiv:cs.LG/2109.09831

[43] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. 2018. Progressive neural architecture search. In *Proceedings of the European conference on computer vision (ECCV)*. 19–34.

[44] Sijia Liu, Parikshit Ram, Deepak Vijaykeerthy, Djallel Bouneffouf, Gregory Bramble, Horst Samulowitz, Dakuo Wang, Andrew Conn, and Alexander G. Gray. 2020. An ADMM Based Framework for AutoML Pipeline Configuration. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 4892–4899. https://ojs.aaai.org/index.php/AAAI/article/view/5926

[45] Shaul Markovitch and Dan Rosenstein. 2002. Feature Generation Using General Constructor Functions. *Mach. Learn.* 49, 1 (2002), 59–98. https://doi.org/10.1023/A:1014046307775

[46] Microsoft. 2021. *Neural Network Intelligence.* https://github.com/microsoft/nni

[47] Fatemeh Nargesian, Horst Samulowitz, Udayan Khurana, Elias B. Khalil, and Deepak S. Turaga. 2017. Learning Feature Engineering for Classification. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, Carles Sierra (Ed.). ijcai.org, 2529–2535. https://doi.org/10.24963/ijcai.2017/352

[48] Randal S Olson and Jason H Moore. 2016. TPOT: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on automatic machine learning*. PMLR, 66–74.

[49] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. 2018. Efficient Neural Architecture Search via Parameter Sharing. *CoRR* abs/1802.03268 (2018). arXiv:1802.03268 http://arxiv.org/abs/1802.03268

[50] Alexandre Quemy. 2020. Two-stage optimization for machine learning workflow. *Inf. Syst.* 92 (2020), 101483. https://doi.org/10.1016/j.is.2019.101483

[51] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. 2018. Regularized Evolution for Image Classifier Architecture Search. *CoRR* abs/1802.01548 (2018). arXiv:1802.01548 http://arxiv.org/abs/1802.01548

[52] Nicolas Schilling, Martin Wistuba, Lucas Drumond, and Lars Schmidt-Thieme. 2015. Hyperparameter Optimization with Factorized Multilayer Perceptrons. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part II (Lecture Notes in Computer Science)*, Annalisa Appice, Pedro Pereira Rodrigues, Vítor Santos Costa, João Gama, Alípio Jorge, and Carlos Soares (Eds.), Vol. 9285. Springer, 87–103. https://doi.org/10.1007/978-3-319-23525-7_6

[53] Nicolas Schilling, Martin Wistuba, and Lars Schmidt-Thieme. 2016. Scalable Hyperparameter Optimization with Products of Gaussian Process Experts. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2016, Riva del Garda, Italy, September 19-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science)*, Paolo Frasconi, Niels Landwehr, Giuseppe Manco, and Jilles Vreeken (Eds.), Vol. 9851. Springer, 33–48. https://doi.org/10.1007/978-3-319-46128-1_3

[54] Vraj Shah, Jonathan Lacanlale, Premanand Kumar, Kevin Yang, and Arun Kumar. 2021. Towards Benchmarking Feature Type Inference for AutoML Platforms. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1584–1596. https://doi.org/10.1145/3448016.3457274

[55] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. 2019. Democratizing Data Science through Interactive Curation of ML Pipelines. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1171–1188. https://doi.org/10.1145/3299869.3319863

[56] Qitao Shi, Ya-Lin Zhang, Longfei Li, Xinxing Yang, Meng Li, and Jun Zhou. 2020. SAFE: Scalable Automatic Feature Engineering Framework for Industrial Tasks. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1645–1656. https://doi.org/10.1109/ICDE48307.2020.00146

[57] Julien Siems, Lucas Zimmer, Arber Zela, Jovita Lukasik, Margret Keuper, and Frank Hutter. 2020. NAS-Bench-301 and the Case for Surrogate Benchmarks for Neural Architecture Search. *CoRR* abs/2008.09777 (2020). arXiv:2008.09777 https://arxiv.org/abs/2008.09777

[58] William R Thompson. 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika* 25, 3-4 (1933), 285–294.

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

[60] Chi Wang and Qingyun Wu. 2019. FLO: Fast and Lightweight Hyperparameter Optimization for AutoML. *CoRR* abs/1911.04706 (2019). arXiv:1911.04706 http://arxiv.org/abs/1911.04706

[61] Tianxiang Wang, Jie Xu, and Jian-Qiang Hu. 2021. A Study on Efficient Computing Budget Allocation for a Two-Stage Problem. *Asia Pac. J. Oper. Res.* 38, 2 (2021), 2050044:1–2050044:20. https://doi.org/10.1142/S021759592050044X

[62] Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8, 3 (1992), 229–256.

[63] Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. 2015. Sequential Model-Free Hyperparameter Tuning. In *2015 IEEE International Conference on Data Mining, ICDM 2015, Atlantic City, NJ, USA, November 14-17, 2015*, Charu C. Aggarwal, Zhi-Hua Zhou, Alexander Tuzhilin, Hui Xiong, and Xindong Wu (Eds.). IEEE Computer Society, 1033–1038. https://doi.org/10.1109/ICDM.2015.20

[64] Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. 2016. Two-Stage Transfer Surrogate Model for Automatic Hyperparameter Optimization. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2016, Riva del Garda, Italy, September 19-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science)*, Paolo Frasconi, Niels Landwehr, Giuseppe Manco, and Jilles Vreeken (Eds.), Vol. 9851. Springer, 199–214. https://doi.org/10.1007/978-3-319-46128-1_13

[65] Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. 2018. Scalable Gaussian process-based transfer surrogates for hyperparameter optimization. *Mach. Learn.* 107, 1 (2018), 43–78. https://doi.org/10.1007/s10994-017-5684-y

[66] Anatoly Yakovlev, Hesam Fathi Moghadam, Ali Moharrer, Jingxiao Cai, Nikan Chavoshi, Venkatanathan Varadarajan, Sandeep R. Agrawal, Tomas Karnagel, Sam Idicula, Sanjay Jinturkar, and Nipun Agarwal. 2020. Oracle AutoML: A Fast and Predictive AutoML Pipeline. *Proc. VLDB Endow.* 13, 12 (2020), 3166–3180. https://doi.org/10.14778/3415478.3415542

[67] Junwen Yang, Yeye He, and Surajit Chaudhuri. 2021. Auto-Pipeline: Synthesize Data Pipelines By-Target Using Reinforcement Learning and Search. *Proc. VLDB Endow.* 14, 11 (2021), 2563–2575. http://www.vldb.org/pvldb/vol14/p2563-he.pdf

[68] Li Yang and Abdallah Shami. 2020. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing* 415 (2020), 295–316.

[69] Chris Ying, Aaron Klein, Esteban Real, Eric Christiansen, Kevin Murphy, and Frank Hutter. 2019. NAS-Bench-101: Towards Reproducible Neural Architecture Search. *CoRR* abs/1902.09635 (2019). arXiv:1902.09635 http://arxiv.org/abs/1902.09635

[70] Wentao Zhang, Jiawei Jiang, Yingxia Shao, and Bin Cui. 2020. Snapshot boosting: a fast ensemble framework for deep neural networks. *Sci. China Inf. Sci.* 63, 1 (2020), 112102. https://doi.org/10.1007/s11432-018-9944-x

[71] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2102–2114. https://doi.org/10.1145/3448016.3457291

[72] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. 2018. Practical block-wise neural network architecture generation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2423–2432.

[73] Fatjon Zogaj, José Pablo Cambronero, Martin Rinard, and Jürgen Cito. 2021. Doing More with Less: Characterizing Dataset Downsampling for AutoML. *Proc. VLDB Endow.* 14, 11 (2021), 2059–2072. http://www.vldb.org/pvldb/vol14/p2059-zogaj.pdf

[74] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).

[75] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2017. Learning Transferable Architectures for Scalable Image Recognition. *CoRR* abs/1707.07012 (2017). arXiv:1707.07012 http://arxiv.org/abs/1707.07012