Machine learning as a black box solution

Seminar - Computational Aspects of Machine Learning

Tao Xiang
Fakultät für Informatik
Technische Universitat München
Email: tao.xiang@tum.de

Abstract

Artificial intelligence is now used in various fields, but many non-experts have not learned AI-related knowledge such as machine learning. Therefore, a direction about automated machine learning came into being. This paper introduces the basic definition of automated machine learning and its three fundamental steps: data preprocessing, algorithm selection and hyperparameter optimization. This paper focuses on the CASH problem, which combines algorithm selection and hyperparameter optimization by treating the algorithms themselves as one hyperparameter, and introduces two types of algorithms to deal with CASH problems: grid search and random search in parallel search, SMAC and TPE in sequential optimization. Finally, the paper introduces two very popular Auto-ML tools: Auto-WEKA and Auto-Sklearn and discusses the possible future work in Auto-ML.

Index Terms

Machine Learning, Auto-ML, data preprocessing, algorithm selection, hyperparameter optimization , Auto-Sklearn, Auto-WEKA

I. INTRODUCTION

NOWADAYS artificial intelligence or machine learning becomes more and more relevant in our existence, more and more fields have involved the applying of artificial intelligence. As a result, more and more people want to use machine learning algorithms to varied unrelated research fields. However, plenty of those don't understand any machine learning algorithms, or only understand a bit basic. As to permit such novices to quickly use artificial intelligence, for instance, to search out the best (machine learning) algorithm and suitable hyperparameters of this algorithm for a particular research problem, without having to know the (machine learning) algorithms themselves, a research direction called "Auto-machine learning" gradually attracted scientists' eyeballs and has developed rapidly within the last ten years.

Auto machine learning, as the name suggests, automatically selects the optimized algorithm and the corresponding hyper-parameters for a given problem and then gives out the output. Therefore it can be visually understood as a black box because people don't know what's within the black box, and they only need to give input and observe the output, just as those non-experts don't know the machine learning algorithm used and also the corresponding hyperparameters, they only have to give the dataset and care about the results.

As mentioned in [1], every machine learning service should solve 3 fundamental problems: 1) whether and the way to preprocess the features of the dataset 2) which ml-algorithm to use on the given dataset 3) the way to set all hyperparameters of this algorithm. So principally any automated machine learning service should be able to solve these 3 problems automatically. In other words, Auto-ML service should automatically produce the test set predictions for a brand new dataset within a fixed computational budget. One formal definition of Auto-ML problem defined in [1] is stated as follows:

Definition 1 (Auto-ML problem). For $i = 1, \ldots, n+m$, let $\boldsymbol{x}_i \in R^d$ denote a feature vector and $y_i \in Y$ the corresponding target value. Given a training dataset $D_{\text{train}} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ and the feature vectors $x_{n+1}, \ldots, x_{n+m}$ of a test dataset $D_{\text{test}} = \{(x_{n+1}, y_{n+1}), \ldots, (x_{n+m}, y_{n+m})\}$ drawn from the same underlying data distribution, as well as a resource budget b and a loss metric $\mathcal{L}(\cdot, \cdot)$, the Auto-ML problem is to (automatically) produce test set predictions $\hat{y}_{n+1}, \ldots, \hat{y}_{n+m}$. The loss of a solution $\hat{y}_{n+1}, \ldots, \hat{y}_{n+m}$ to the Auto-ML problem is given by $\frac{1}{m} \sum_{j=1}^{m} \mathcal{L}(\hat{y}_{n+j}, y_{n+j})$

There are totally n+m sample data, among them n data are for training, m data are for testing, each sample datum has d features. The Auto-ML service automatically produces test set predictions and calculates its corresponding loss using different ML-algorithms and the goal is to find the best algorithm and hyperparameters so that the loss function is minimized.

In this paper we will mainly focus on the principle and essence of automated machine learning from a mathematical or computational perspective.

II. PRELIMINARIES

The problem we are facing exactly is automated and simultaneous selection of machine learning algorithms and their corresponding hyperparameters for a given (raw) dataset, so that it has the optimal performance. During this section, we will take a glance at the definitions of the 3 significant steps of automated machine learning: 1) Data Preprocessing 2) Algorithm Selection 3) Hyperparameter Optimization.

A. Data Preprocessing

Data preprocessing is the fundamental step of (automated) machine learning because the data machine learning algorithms expect are usually quite different from data in the real world. For instance, most ML algorithms expect numerical data in form of a matrix. However, data in real-world are usually structured or unstructured data with some different feature types like number and text, and they have possible missing values or categorical values which need to be encoded into numerical values first. On the other hand, proper data preprocessing can usually improve the accuracy of the final model because after data preprocessing, much information can be extracted from the features which could be useful to predict the target value. In the following we will shortly describe the basic pipeline for data preprocessing and how we could do everything in automatic way.

1) categorical features encoding: There are many ways to encode categorical values, one popular approach is one-hot encoding: the data preprocessor ("DP" in the remaining section) will first count the total number of categories n and then build a vector of length n for each entry, if the entry belongs to the i_{th} category, then the i_{th} entry of the vector will be set to 1, others to 0. Null vectors are used to represent the missing-category value.

2) dealing with missing values: DP will first check if the features with missing values are useful by calculating the variance of all other observable entries of those features. If the variance is smaller then a predefined threshold v^*, like 0.05, then we will simply discard this feature since it makes no big difference between the data. Otherwise, DP will fulfill these missing values with the average of the remaining values.

3) feature elimination: Features with very close values will be eliminated due to low variance. After doing the previous 2 steps, the data are numerical. Now DP will calculate variance for each column and remove the column with variance smaller than the variance threshold v^*.

4) Feature Scaling: The final step is to scale the features because features with much larger values could have more weight into final model which might not be true. This is very important for algorithms based on continuous functions. Typically DP will scale the values into the range 0 to 1 by calculating the position of the value v between the maximum value v_{max} and minimum value v_{min} of this feature. Mathematically, it can be written as $\frac{v-v_{min}}{v_{max}-v_{min}}$.

B. Algorithm Selection

Assuming that we have already found the optimal hyperparameters combination for each algorithm $A_i \in \mathcal{A}$, and now we need to find the optimal algorithm, so that the loss could be minimized. To be more precise, we do k-fold cross validation [2] for training and validation: we split the training data into k equal-sized partitions $\mathcal{D}_{\text{valid}}^{(1)}, \ldots, \mathcal{D}_{\text{valid}}^{(k)}$ and set $\mathcal{D}_{\text{train}}^{(i)} = \mathcal{D} \backslash \mathcal{D}_{\text{valid}}^{(i)}$ for $i = 1, \ldots, k$. we train the data on $\mathcal{D}_{\text{train}}^{i}$,and evaluate data on $\mathcal{D}_{\text{valid}}^{i}$, for $i = 1, \ldots, k$, then we record the mean loss as the final loss for each algorithm and finally we choose the algorithm with the minimal loss. Mathematically, it can be written as follows:

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

where $\mathcal{L}\left(A, \mathcal{D}_{\text{train}}^{(i)}, \mathcal{D}_{\text{valid}}^{(i)}\right)$ is the loss when using the algorithm A with its optimal parameters, trained on $\mathcal{D}_{\text{train}}^{(i)}$, and evaluated on $\mathcal{D}_{\text{valid}}^{(i)}$.

C. Hyperparameter Optimization

Almost every learning algorithm has its own hyperparameters. These hyperparameters could have a significant effect on the way the learning algorithm works, and the performance of the algorithm could be influenced to a large extent. For instance, the number of neurons in a hidden layer, or just the number of hidden layers, which plays an important role in neural network.

Assuming that for a certain algorithm A we have n hyperparameters: $\lambda_1, ..., \lambda_n$, with domains $\Lambda_1, ..., \Lambda_n$ respectively, and the hyperparameter space Λ is a subset of the cross-product of the domains: $\Lambda \subset \Lambda_1 \times \cdots \times \Lambda_n$. Then we need to find hyperparameter $\lambda^* \in \Lambda$, so that the algorithm A with λ^* has the optimal performance. Like before, we use k-fold cross-validation so that our predictions could be more accurate. We record the mean loss as the final loss for each hyperparameters combination. Mathematically, it can be expressed as follows:

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

where $\mathcal{L}\left(A_{\boldsymbol{\lambda}}, \mathcal{D}_{\text{train}}^{(i)}, \mathcal{D}_{\text{valid}}^{(i)}\right)$ is the loss when using the algorithm A with the parameter λ, trained on $\mathcal{D}_{\text{train}}^{(i)}$, and evaluated on $\mathcal{D}_{\text{valid}}^{(i)}$.

Notice that such hyperparameter spaces are often in a hierarchical structure, such as tree-structured or directly acyclic graph (DAG), since some hyperparameters are only active if other certain hyperparameters are active. For instance, the hyperparameters of a support vector machine's polynomial kernel are only active if we choose the polynomial kernel as the kernel function. This is the so-called conditional hyperparameters [3].

D. Combined Algorithm Selection and Hyperparameter Optimization (CASH)

In the past, people always select algorithm and hyperparameters separately, which generally results in huge computational overhead. To tackle that, Thornton et al. [4] raised an idea to process algorithm selection and hyperparameter optimization simultaneously, i.e. the CASH problem: *combined algorithm selection and hyperparameter optimization*. It was also proved that the recent Bayesian optimization algorithms find combination of algorithms and hyperparameters that is usually superior to the existing baseline methods, especially on large datasets [4] . A formal definition of CASH problem [1] is given as follows:

Definition 2 (CASH). Let $\mathcal{A} = \left\{A^{(1)}, \ldots, A^{(R)}\right\}$ be a set of algorithms, and let the hyperparameters of each algorithm $A^{(j)}$ have domain $\Lambda^{(j)}$. Further, let $D_{\text{train}} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ be a training set which is split into K cross-validation folds $\left\{D_{\text{valid}}^{(1)}, \ldots, D_{\text{valid}}^{(K)}\right\}$ and $\left\{D_{\text{train}}^{(1)}, \ldots, D_{\text{train}}^{(K)}\right\}$ such that $D_{\text{train}}^{(i)} = D_{\text{train}} \backslash D_{\text{valid}}^{(i)}$for $i = 1, \ldots, K$. Finally, let $\mathcal{L}\left(A_{\boldsymbol{\lambda}}^{(j)}, D_{\text{train}}^{(i)}, D_{\text{valid}}^{(i)}\right)$ denote the loss that algorithm $A^{(j)}$ achieves on $D_{\text{valid}}^{(i)}$ when trained on $D_{\text{train}}^{(i)}$ with hyperparameters λ. Then, the Combined Algorithm Selection and hyperparameter optimization (CASH) problem is to find the joint algorithm and hyperparameter setting that minimizes this loss:

$$A^{\star}, \boldsymbol{\lambda}_{\star} \in \underset{A^{(j)} \in \mathcal{A}, \boldsymbol{\lambda} \in \Lambda^{(j)}}{\operatorname{argmin}} \frac{1}{K} \sum_{i=1}^{K} \mathcal{L}\left(A_{\boldsymbol{\lambda}}^{(j)}, D_{\text{train}}^{(i)}, D_{\text{valid}}^{(i)}\right).$$

Basically, the CASH problem can be regarded as a hyperparameter optimization problem of a single hierarchical structure, since the algorithms could be seen as a special hyperparameter as well. Formally, we extend the hyperparameter space $\boldsymbol{\Lambda} = \Lambda^{(1)} \cup \cdots \cup \Lambda^{(k)}$ to $\boldsymbol{\Lambda}' = \Lambda^{(1)} \cup \cdots \cup \Lambda^{(k)} \cup \{\lambda_r\}$, where $\Lambda^{(i)}$ is the hyperparameter space of algorithm A_i, and $\{\lambda_r\}$ is the set of all given algorithms. We then place this new hyperparameter at root level and make all other hyperparameter subspaces conditional on it. For instance, the hyperparameters subspace $\Lambda^{(i)}$ is only active when λ_r takes the value of A_i.

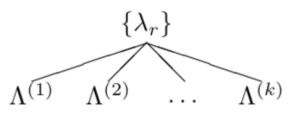

Fig. 1: hyperparameter space of CASH problem

Principally the CASH problem can be tackled in various ways. Here we are going to cover two common types of algorithms for hyperparameter tuning: grid search and random search of parallel search, SMAC and TPE of sequential optimization.

III. GRID SEARCH

Grid Search is an exhaustive searching algorithm for hyperparameter optimization: it traverses all possible hyperparameter combinations to find the hyperparameter combination that minimizes loss. The process is the same as the permutation and combination we learned in school: Assume we have n hyperparameters, and the i_{th} hyperparameter has m_i possible values, then there are totally $\sum_{i=1}^{n} m_i$ combinations.

A. Advantages and Disadvantages

Obviously one advantage of grid search is that it's very easy to implement. Besides, grid search considers all possible combinations, which at the end could output the real optimal hyperparameter combination theoretically. Otherwise, grid search could be implemented parallelly, as the computation for each combination is independent from others. Grid search is reliable in low dimensional hyperparameter space.

However, the computational overhead of grid search in high dimensional hyperparameter space is too much, and it treats the hyperparameters all equally, which is not that suitable here as CASH problem has many conditional hyperparameters. Since grid search has many drawbacks, Bengio at al [9] has raised random search algorithm in 2012 and has shown that random experiments are more efficient than grid experiments for hyperparameter optimization in the case of several learning algorithms.

IV. RANDOM SEARCH

Unlike grid search, random search works by randomly sampling hyperparameter combinations from hyperparameter space and compare them with the current best combination to decide whether to update the best or not. A pseudo-code is given as follows [10]:

Algorithm 1 RandomSearch

```
1: Best $\leftarrow$ some initial random candidate solution
2: while time budget for optimization has not been exhausted do
3:   $\mathcal{S} \leftarrow$ a random candidate solution
4:   if Loss $(\mathcal{S})$ < Loss (Best) then
5:     Best $\leftarrow \mathcal{S}$
6: end while
7: return Best
```

A. Advantages and Disadvantages

Like grid search, random search is also easy to understand, implement (no "hyperhyperparameter"), and trivially parallelizable. Since random search selects the candidate randomly, it's hard to cover every corner in the hyperparameter space. This is nevertheless good when the hyperparameters are not equally important (like the CASH problem), which is why under certain time limit, random search beats grid search in many cases. Besides, there are also many experiments [13] showing that for some problems random search can give not bad results compared with more advanced optimization approaches like Bayesian optimization.

One drawback of random search is unnecessarily high variance. Since it's entirely random and not intelligent at all, the luck plays a part. What's more, when the dimension of hyperparameter space is relatively high, the performance degrades.

V. BAYESIAN OPTIMIZATION

Grid and random search are relatively inefficient because they do not use the previous results to choose next promising hyperparameter. As a result, they often spend a significant amount of time evaluating "bad" hyperparameters. In contrast, Bayesian approaches keep track of past evaluation results to build a probabilistic model $\mathcal{M}_{\mathcal{L}}$ that captures the dependence of loss c on hyperparameter setting $\lambda \in \mathbf{\Lambda}$. This model is represented as follows:

$$p_{\mathcal{M}_L}(c \mid \lambda)$$

The model is used to choose the next promising hyperparameter (in so-called acquisition function, details in subsection B), and with more and more past results, it will be more and more accurate and choose more and more promising hyperparameters in return.

One detailed formalization of Bayesian optimization for hyperparameter optimization is the *Sequential Model-Based Optimization* [SMBO;5].

A. Sequential Model-Based Optimization

In a nutshell, Sequential Model-Based Optimization (within one iteration) searches the next promising hyperparameter configuration λ using the acquisition function, then stores the hyperparameter and its loss in a specific set $\mathcal{H}$, which we call *History Set*, and updates the model $\mathcal{M}_{\mathcal{L}}$ based on the new $\mathcal{H}$. This procedure is repeated until a certain time limit is reached. After that, the hyperparameter λ with minimal loss in $\mathcal{H}$ is returned. Following is the algorithm of SMBO:

Algorithm 2 SMBO

```
1: initialise model $\mathcal{M}_L; \mathcal{H} \leftarrow \emptyset$
2: while time budget for optimization has not been exhausted do
3:   $\lambda \leftarrow$ candidate configuration from $\mathcal{M}_L$
4:   Compute $c = \mathcal{L}\left(A_{\boldsymbol{\lambda}}, \mathcal{D}_{\text{train}}^{(i)}, \mathcal{D}_{\text{valid}}^{(i)}\right)$
5:   $\mathcal{H} \leftarrow \mathcal{H} \cup \{(\boldsymbol{\lambda}, c)\}$
6:   Update $\mathcal{M}_L$ given $\mathcal{H}$
7: end while
8: return $\lambda$ from $\mathcal{H}$ with minimal $c$
```

There are two "hyperhyperparameters" here: the acquisition function and the probabilistic model $\mathcal{M}_{\mathcal{L}}$.

B. Acquisition Function

The criterion for selecting the hyperparameter by acquisition function is that the hyperparameter should have maximal ability of improvement, so that the loss could be much smaller in the next iteration. Typically, acquisition function $f : \mathbf{\Lambda} \rightarrow \mathbf{R}$ reflects the ability of improvement of the given hyperparameter λ, so we need to find λ that maximizes f in each iteration.

There are many well-known acquisition functions and here we only introduce two simple examples for the demonstration.

1) probability of improvement: One example of acquisition functions is the *probability of improvement*. Assuming that c_{min} is the minimal loss currently known for us and $c(\boldsymbol{\lambda})$ the loss when applying the hyperparameters $\boldsymbol{\lambda}$, we define a "reward function"

$$u(\boldsymbol{\lambda}) = \begin{cases} o, & \text{if } c(\boldsymbol{\lambda}) > c_{min} \\ 1, & \text{if } c(\boldsymbol{\lambda}) \leq c_{min} \end{cases} ,$$

which is also very intuitive: When the loss of the potentially better hyperparameters λ is less then the current minimal loss c_{min}, then we give λ a reward of 1, otherwise 0. The probability of improvement acquisition function is defined as expected reward as a function of λ:

$$\mathrm{E}_{\mathcal{M}_{\mathcal{L}}}\left[u(\boldsymbol{\lambda})\right] := \int_{-\infty}^{c_{\min}} 1 \cdot p_{\mathcal{M}_L}(c \mid \lambda) dc$$

The problem of this acquisition function is that it may only return a λ with local maximum rather than global maximum. To tackle that, we introduce another well-known acquisition function: the *positive expected improvement (EI).*

2) positive expected improvement (EI): Similar as before, let c_{min} be the current minimal loss and $c(\boldsymbol{\lambda})$ be the loss when applying λ, the positive improvement function over c_{min} is defined as:

$$I_{c_{\min}}(\boldsymbol{\lambda}) := \max\left\{c_{\min} - c(\boldsymbol{\lambda}), 0\right\},$$

which is basically of the same essence as the reward function before: if the loss of the candidate is less than the current loss minimum $c_{\min}$ then we give $\boldsymbol{\lambda}$ a positive value $c_{\min} - c(\boldsymbol{\lambda})$, otherwise 0. The positive expected improvement acquisition function is defined as:

$$\mathrm{E}_{\mathcal{M}_{\mathcal{L}}}\left[I_{c_{\min}}(\boldsymbol{\lambda})\right] := \int_{-\infty}^{c_{\min}} \max\left\{c_{\min} - c, 0\right\} \cdot p_{\mathcal{M}_L}(c \mid \lambda) dc$$

EI(λ) is large for hyperparameter λ with low predicted cost and for those with high predicted uncertainty; thereby, it offers an automatic tradeoff between exploitation (focusing on known good parts of the space) and exploration (gathering more information in unknown parts of the space) [5].

C. Probabilistic Model

One main difference between SMBO algorithms is the model class $\mathcal{M}_L$ used. Common models are like Gaussian Process, Random Forest Regression and Tree Parzen Estimators (TPE). Here we will go deep into the latter two models, since they can handle hierarchical hyperparameters which are proper for the task of combined algorithm selection and hyperparameter optimization.

D. Summary of SMBO

Here I would generalize each iteration in SMBO into two main steps:

1) Step 1: selecting the next promising hyperparameter and updating $\mathcal{H}$: This step basically picks the hyperparameter λ that can maximize the acquisition function and adds (λ, cost of λ) to history H.

2) Step 2: updating model $\mathcal{M}_{\mathcal{L}}$ based on updated $\mathcal{H}$: The implementation of this step varies for different SMBO algorithms. But generally, the model is updated based on updated history set in step 1.

Notice that model $\mathcal{M}_{\mathcal{L}}$ is also a part of the acquisition function in step 1, that's to say step 1 and step 2 interacts each other: the model is used to choose next promising hyperparameter, and the hyperparameter in return could affect the model to select similar hyperparameters in next iterations. (SMBO keeps narrowing the search region)

In the following we will go deep into two instantiations of SMBO.

VI. SEQUENTIAL MODEL-BASED ALGORITHM CONFIGURATION

Sequential Model-Based Algorithm Configuration is an instantiation of SMBO, it supports many different probabilistic models which are able to reflect the dependence of the loss function c on hyperparameters λ, for instance, the approximate Gaussian processes and random forest. Here we would use the random forest models, since the regression trees in random forest are known to perform well for categorical input data, so that random forest can usually make more precise predictions [11].

A. Model: Random Forest (step 2)

Random forest consists of several individual regression trees that are more or less the same as decision trees but have real values (here: the target loss c). In the following we will see how the model is built based on history $\mathcal{H}$ in step two.

1) Train The Regression Trees: To build the model, we first have to train our regression trees (thus random forest). For that we have to determine how many regression tress we should use and the split ratio for each tree.

To keep the computational overhead small, people usually use $B = 10$ regression trees in the random forest and the split ratio is by default $\frac{5}{6}$, what's more, a node should have at least 10 data points to be split [5].

The training dataset is exactly $\mathcal{H}$, (set of hyperparameters and corresponding costs selected in past iterations), which is growing in each iteration. This means the model is gradually being precise. To make sure each tree accepts different training data, we simply make several copies of $\mathcal{H}$ and randomly sample n data for each tree, where n is the size of $\mathcal{H}$.

2) Build The Probabilistic Model: In each iteration after trees are trained, each individual tree has its own prediction of cost for the hyperparameter λ. We compute the random forest's predictive mean $\mu_{\boldsymbol{\lambda}}$ and variance $\sigma^2_{\boldsymbol{\lambda}}$ as the empirical mean and variance of these individual trees' predictions. Then we model/update $p_{\mathcal{ML}}(c \mid \boldsymbol{\lambda})$ as the normal distribution $\mathcal{N}(\mu_{\boldsymbol{\lambda}}, \sigma^2_{\boldsymbol{\lambda}})$. This model is again used to select the next promising hyperparameter in the next iteration (see subsection B). Like we said before, we have to find hyperparameter that maximizes the acquisition function (subsection B in section 5).

B. Acquisition Function: EI (step 1)

SMAC uses the positive expected improvement (EI) as the acquisition function:

$$E_{\mathcal{ML}}[I_{cmin}(\boldsymbol{\lambda})] = (c_{\min} - \mu_{\boldsymbol{\lambda}}) \cdot \Phi(\frac{c_{\min} - \mu_{\boldsymbol{\lambda}}}{\sigma_{\boldsymbol{\lambda}}}) + \sigma^2_{\boldsymbol{\lambda}} \cdot \varphi(\frac{c_{\min} - \mu_{\boldsymbol{\lambda}}}{\sigma_{\boldsymbol{\lambda}}})$$

(for detailed derivation process please see the last page), where f and F is the density function and cumulative distribution function of random variable c; φ and Φ is the density function and cumulative distribution function of standard normal distribution.

As we said in subsection B, section 5: we need to find hyperparameter that maximizes the acquisition function. To make the computational overhead small, SMAC performs a simple multi-start local search [5], which basically compute EI for all the past hyperparameters stored in $\mathcal{H}$ and pick ten hyperparameters with maximal values, then do a local search for each hyperparameter. The local search is also very intuitive: It first normalizes all the numerical hyperparameters to range 0 to 1, and generates four neighbours for each numerical hyperparameter with value v using a univariate Gaussian distribution with mean v and deviation 0.2, then compares EI value of this hyperparameter with its four neighbours and changes the focus to the neighbour with maximal EI and do the process again. This process is repeated until none of neighbours have larger EI, and then this focused hyperparameter is returned.

C. Advantages and Disadvantages

SMAC is good for CASH problem because the regression trees in random forest model are good for categorical and hierarchical input data. What's more, SMAC also optimizes previous SMBO algorithms in many details:

1) The way to find hyperparameter maximizing acquisition function: Previous SMBO simply applied random sampling of hyperparameters (in particular, they evaluate EI for 10000 random samples), which is not only costly but also not good for sparse hyperparameter space. In contrast, local search saves a lot of computation.

2) Dealing with overfitting: Because we need to return $\boldsymbol{\lambda}$ in $\mathcal{H}$ with minimal cost c, there is a variable that stores the incumbent λ with minimal cost so far, and in each iteration, SMBO will compare the incumbent with the candidate hyperparameter selected by acquisition function and update it if necessary.

Above is what normal SMBO algorithms do, which sometimes lead to overfitting. To tackle that, SMAC interleaves randomly-sampled hyperparameters [5]. More specifically, before comparing the incumbent and candidate in each iteration, SMAC will first sample a hyperparameter uniformly at random, then compare these three together. This has an impact that each hyperparameter is possible to be selected in each iteraion, which to some extent reduces overfitting.

However, SMAC still has improvement space, such as to perform better in context of high dimensional hyperparameter space and to better handle hyperparameters with large per-run captimes for each target algorithm run [5];

VII. TREE-STRUCTURED PARZEN ESTIMATOR APPROACH

Let's look at another modeling strategy and EI optimization scheme for the SMBO algorithm: *Tree-structured Parzen Estimator Approach (TPE).* TPE is still Bayesian optimization, and it has a deep relationship with SMAC. It is a non-standard Bayesian optimization algorithm based on the tree structure Parzen density estimation.

A. General Model (step 2)

The way TPE builds the model $\mathcal{M}_{\mathcal{L}}$ is different from that of SMAC. Whereas the SMAC approach models $p(c \mid \boldsymbol{\lambda})$ directly, TPE applies Bayesian rule and substitutes $p(c \mid \boldsymbol{\lambda})$ with $p(\boldsymbol{\lambda} \mid c)$ and $p(c)$:

$$p(c \mid \boldsymbol{\lambda}) = \frac{p(\boldsymbol{\lambda} \mid c) \cdot p(c)}{p(\boldsymbol{\lambda})} \tag{1}$$

Let's first have a look at $p(\boldsymbol{\lambda} \mid c)$: TPE models $p(\boldsymbol{\lambda} \mid c)$ as one of two density functions $\ell(\boldsymbol{\lambda})$ and $g(\boldsymbol{\lambda})$, depending on whether the loss c is greater or less than a predefined threshold value c^*, which is something related to *target performance*. Intuitively speaking, when the cost c of a certain hyperparameter λ is less than c^*, we consider λ as a good hyperparameter, otherwise a bad hyperparameter.

To do that, we first have to define the target performance. In [7] it's defined as γ quantile of search results:

$$\gamma = p(c < c^*) = \int_{-\infty}^{c^*} p(y)\, dy$$

Back to the implementation: In each iteration we split the history $\mathcal{H}$ into two subsets $\mathcal{H}_\ell$ and $\mathcal{H}_g$ according to the threshold c^*, and $\mathcal{H}_\ell$ is for building $\ell(\boldsymbol{\lambda})$, $\mathcal{H}_g$ for $g(\boldsymbol{\lambda})$. Generally, $p(\boldsymbol{\lambda} \mid c)$ can be written as follows [4]:

$$p(\boldsymbol{\lambda} \mid c) = \begin{cases} \ell(\boldsymbol{\lambda}), & \text{if } c < c^* \\ g(\boldsymbol{\lambda}), & \text{if } c \geq c^* \end{cases}$$

Where $l(\boldsymbol{\lambda})$ is a density function learned from all the hyperparameters in $\mathcal{H}_\ell$ and $g(\boldsymbol{\lambda})$ is a density function learned from all the hyperparameters in $\mathcal{H}_g$.

B. Tree Structured Parzen Estimators (step 2 cont.)

Now let's go deep into how TPE models $\ell(\boldsymbol{\lambda})$ or $g(\boldsymbol{\lambda})$. The two densities ℓ and g are modeled using Parzen estimators (also known as kernel density estimators), which are here tree-structured processes involving discrete and continuous hyperparameters.

1) Tree Structured: It means that the hyperparameter space is tree-like: the value chosen for one hyperparameter determines what hyperparameter will be chosen next and what values are available for it. For each hyperparameter (node) in $\mathcal{H}_\ell$ or $\mathcal{H}_g$, a 1-D Parzen estimator is created to model the density of this hyperparameter, which has different forms depending on whether the hyperparameter is continuous or discrete. (see next subsection). When a new hyparameter configuration $\boldsymbol{\lambda}$ is added to $\mathcal{H}_\ell$ or $\mathcal{H}_g$, only the active hyperparameters' 1-D estimators are updated.

To evaluate a candidate hyperparameter's probability estimate, TPE starts at the root of the tree and walks all the way to the leaves along a path that only uses active hyperparameters. At each node on the path, the 1-D estimator of this node will compute a probability and at the end the individual probabilities are combined on a pass back to the root of the tree.

2) 1-D Parzen Estimator: For each continuous hyperparameter with value v in $\mathcal{H}_\ell$ or $\mathcal{H}_g$, the 1-D estimator is constructed as a Gaussian distribution with mean v and the deviation is set to the greater of the distances to the left or right neighbour, but clipped to remain a reasonable range. For discrete hyperparameters: supposing one discrete hyperparameter is represented as a vector of n probabilities p_i, where $i \in [n]$, then the posterior vector elements were proportional to $np_i + C_i$, where C_i counts the occurrences of choice i in $\mathcal{H}_\ell$ or $\mathcal{H}_g$ [7]. This means that discrete hyperparameters are estimated with probabilities proportional to the number of times that a particular choice occurred in $\mathcal{H}_\ell$ or $\mathcal{H}_g$.

C. $p(\boldsymbol{\lambda})$ and $p(c)$ (step 2 cont.)

As for $p(\boldsymbol{\lambda})$, we could simply rewrite it using γ , $\ell(\boldsymbol{\lambda})$ and $g(\lambda)$:

$$p(\boldsymbol{\lambda}) = \int_{-\infty}^{\infty} p(\lambda \mid c) \cdot p(c) dc = \int_{-\infty}^{c^\star} p(\lambda \mid c) \cdot p(c) dy + \int_{c^\star}^{+\infty} p(\lambda \mid c) \cdot p(c) dy = \gamma \cdot l(\lambda) + (1-\gamma) \cdot g(\lambda).$$

We don't necessarily need a probabilistic model for $p(c)$ since it could be simplified into other terms in the later computation. Now we got clear with each part in (1), let's move to the details for computing the acquisition function.

D. Acquisition Function: EI (step 1)

Here we use the positive expected improvement as the acquisition function too. After a piece of derivations (details in last page), we know that it's proportional to a quantity which can be computed in close-form from γ, $g(\boldsymbol{\lambda})$ and $l(\boldsymbol{\lambda})$:

$$E\left[I_{c_{\min}}(\boldsymbol{\lambda})\right] \propto \left(\gamma + \frac{g(\boldsymbol{\lambda})}{\ell(\boldsymbol{\lambda})} \cdot (1-\gamma)\right)^{-1} \tag{2}$$

From (2) we know that to maximize the improvement we would like $\boldsymbol{\lambda}$s with high probability under $l(\boldsymbol{\lambda})$ and low probability under $g(\boldsymbol{\lambda})$. TPE maximizes $\left(\gamma + \frac{g(\boldsymbol{\lambda})}{\ell(\boldsymbol{\lambda})} \cdot (1-\gamma)\right)^{-1}$ by generating many hyperparameter configurations at random and evaluating them according to $\frac{g(\boldsymbol{\lambda})}{\ell(\boldsymbol{\lambda})}$, then it picks $\boldsymbol{\lambda}$ with the smallest value of $\frac{g(\boldsymbol{\lambda})}{\ell(\boldsymbol{\lambda})}$.

E. Advantages and Disadvantages

TPE has been shown to scale to higher dimensions with little overhead by Eggensperger et al.[14] and to parallelize easily [7].

One of the most significant disadvantage of TPE is that it does not model the interactions between the hyper-parameters, because the tree structure requires each hyperparameter is independent from others and two or more hyperparameters are not allowed to appear together in one single node. This results in lack of ability to deal with hyperparameters that strongly interact.

VIII. AUTO-WEKA & AUTO-SKLEARN

Auto-WEKA is an efficient and completely automated tool that can solve the combined algorithm selection and hyperparameter optimization (short: CASH) problem utilizing the full range of classification algorithms in WEKA. It's based on the recent Bayesian optimization techniques we discussed before. It was shown in [4] that on 21 prominent datasets Auto-WEKA often outperformed the standard algorithm selection/hyperparameter optimization methods, especially on large datasets.

Auto-Sklearn is another Auto-ML technique that made two improvements for Auto-WEKA [1]: First, it includes a meta-learning step before the Bayesian optimizer for finding potentially good instantiations of machine learning frameworks among the given ML frameworks. This is achieved by collecting both performance data and the meta-features, i.e. the characteristics of the dataset which can be computed efficiently and can help to determine which algorithm to use on a new dataset. As stated in [1], meta-learning contributes to a considerable boost in efficiency. Second, it includes an automated ensemble construction step after the Bayesian optimizer to tackle one shortage of the previous Auto-ML algorithms: All other models except the optimal model are discarded once the optimal model has been found. This shortage makes the previous optimization algorithms very wasteful and not robust. Basically, this approach store these models to construct an ensemble rather than discarding them, since ensembles often outperform individual models [12]. Auto-Sklearn is proved to be more efficient and robust than the previous hyperparameter optimization algorithms [1].

IX. CONCLUSION

In this paper we have introduced three fundamental steps of Auto-ML: Data Preprocessing, Algorithm Selection, Hyperparameter Optimization and focused on the CASH problem and four well-known algorithms for solving it. Whereas brutal force algorithms like Grid or Random Search spend much time on bad hyperparameters, SMBO algorithms search the next good hyperparameter based on previous results, keeping narrow the search region. That's why SMBO algorithms show a better performance than Grid or Random Search in general.

Generally each iteration of SMBO could be divided into two steps: picking the next hyperparameter and updating the model. That's also where the instantiations of SMBO differ. SMAC and TPE are more proper to solve CASH problems because they are both tree-structured so that they are good to tackle hierarchical hyperparameter space.

However, there is one common drawback for SMBO algorithms: The performance degrades (to a different extent) for high dimensional hyperparameter space. Experiments show that TPE has a better performance for high dimensional hyperparameter space aigainst other SMBO algorithms. So one challenging for SMBO algorithms is to deal with high dimensional hyperparameter space in the future.

Besides the traditional Auto-ML algorithms, another direction called "meta-learning" is getting more and more popular in the recent years. It also has another name: "learn to learn", which is motivated by humans being able to learn new things fast after having learned some similar things.

Anyway, the time of AI comes, where Auto-ML plays an import part. Meta-learning gives the machine ability to learn things quickly, which is close to real human beings. Personally I think robots could behave like a real human in the near future.

REFERENCES

[1] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, Frank Hutter, *Efficient and Robust Automated Machine Learning*

[2] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proc. of IJCAI-95*, pages 1137-1145,1995

[3] F. Hutter, H.H.Hoos, K. Leyton-Brown, and T. Stuetzle. ParamILS: an automatic algorith configuration framework. *JAIR*, 36(1):267-306, 2009

[4] Chris Thornton, Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, *Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms*

[5] F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. *Proc. of LION-5*, pages 507-523,2011

[6] D.R. Jones, M. Schonlau, and W.J. Welch. Efficient global optimization of expensive black box funcctions. *Journal of Global Optimization,* 13:455 -492, 1998

[7] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kegl. Algorithms for hyperparameter Optimization. In *Proc. of NIPS-11*, 2011

[8] Anonymous author. *Massive Parallel Hyperparameter Tuning*

[9] James Bergstra and Yoshua Bengio. Random Search for hyperparameter Optimization. *Journal of Machine Learning Research* 13 (2012) 281-305

[10] https://www.youtube.com/watch?v=7WWnWSPymdc

[11] L. Breiman. Random forest. *Machine Learning*, 45(1):5-32, 2001

[12] I. Guyon, A. Saffari, G. Dror, and G. Cawley. Model selection: Beyond the Bayesian/Frequentist divide. *JMLR*, 11:61-87, 2010

[13] http://www.argmin.net/2016/06/20/hypertuning/

[14] Eggensperger, K.; Feurer, M.; Hutter, F.; Bergstra, J.; Snoek, J.; Hoos, H. H.; and Leyton-Brown, K. 2013. Towards an empirical foundation for assessing Bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization*

Appendix

A. Derivations of acquisition function in SMAC

$$\begin{aligned}
E_{\mathcal{ML}}[I_{cmin}(\boldsymbol{\lambda})] &= E_{\mathcal{ML}}[\max(c_{\min}-c,0)] \\
&= \int_{-\infty}^{c_{min}} \max(c_{\min}-c,0)\cdot p(c)dc + \int_{c_{min}}^{\infty} \max(c_{\min}-c,0)\cdot p(c)dc \\
&= \int_{-\infty}^{c_{min}} (c_{\min}-c,0)\cdot p(c)dc \\
&= \int_{-\infty}^{c_{min}} c_{\min}\cdot p(c)dc - \int_{-\infty}^{c_{min}} c\cdot p(c)dc \\
&= c_{\min}\cdot F(c_{\min}) - \int_{-\infty}^{c_{min}} c\cdot p(c)dc \\
&\overset{(4)}{=} c_{\min}\cdot F(c_{\min}) + \sigma_{\boldsymbol{\lambda}}^2\cdot f(c_{\min}) - \mu_{\boldsymbol{\lambda}}\cdot F(c_{\min}) \\
&= (c_{\min}-\mu_{\boldsymbol{\lambda}})\cdot F(c_{\min}) + \sigma_{\boldsymbol{\lambda}}^2\cdot f(c_{\min}) \\
&= (c_{\min}-\mu_{\boldsymbol{\lambda}})\cdot \Phi(\frac{c_{\min}-\mu_{\boldsymbol{\lambda}}}{\sigma_{\boldsymbol{\lambda}}}) + \sigma_{\boldsymbol{\lambda}}^2\cdot \varphi(\frac{c_{\min}-\mu_{\boldsymbol{\lambda}}}{\sigma_{\boldsymbol{\lambda}}})
\end{aligned} \tag{3}$$

The computation of formula (4) is as following: (notice that the density function of normal distribution $\mathcal{N}(\mu,\sigma)$ is $\varphi(x;\mu,\sigma)=\frac{1}{\sqrt{2\pi}\cdot\sigma}\cdot\exp(-\frac{(x-\mu)^2}{2\sigma^2})$

$$\begin{aligned}
\int c\cdot p(c)dc &= \int (c-\mu_{\boldsymbol{\lambda}})\cdot p(c) + \mu_{\boldsymbol{\lambda}}\cdot p(c)dc \\
&= \int (c-\mu_{\boldsymbol{\lambda}})\cdot \frac{1}{\sqrt{2\pi}\cdot\sigma_{\boldsymbol{\lambda}}}\cdot\exp(-\frac{(c-\mu_{\boldsymbol{\lambda}})^2}{2\sigma_{\boldsymbol{\lambda}}^2})dc + \mu_{\boldsymbol{\lambda}}\cdot F(c) \\
&= \frac{1}{\sqrt{2\pi}\cdot\sigma_{\boldsymbol{\lambda}}}\cdot(-\sigma_{\boldsymbol{\lambda}}^2)\cdot\exp(-\frac{(c-\mu_{\boldsymbol{\lambda}})^2}{2\sigma_{\boldsymbol{\lambda}}^2}) + \mu_{\boldsymbol{\lambda}}\cdot F(c) \\
&= \mu_{\boldsymbol{\lambda}}\cdot F(c) - \sigma_{\boldsymbol{\lambda}}^2\cdot f(c)
\end{aligned} \tag{4}$$

For Integration interval $[-\infty, c_{\min}]$ it's $\mu_{\boldsymbol{\lambda}}\cdot F(c_{\min}) - \sigma_{\boldsymbol{\lambda}}^2\cdot f(c_{\min})$.

B. Derivations of acquisition function in TPE

$$\begin{aligned}
E\left[I_{c_{\min}}(\boldsymbol{\lambda})\right] &= E_{\mathcal{ML}}[\max(c_{\min}-c,0)] \\
&= \int_{-\infty}^{c_{min}} \max(c_{\min}-c,0)\cdot p(c\mid\boldsymbol{\lambda})dc + \int_{c_{min}}^{\infty} \max(c_{\min}-c,0)\cdot p(c\mid\boldsymbol{\lambda})dc \\
&= \int_{-\infty}^{c_{min}} (c_{\min}-c,0)\cdot p(c\mid\boldsymbol{\lambda})dc \\
&= \frac{\int_{-\infty}^{c^*}(c^*-c)\cdot p(\boldsymbol{\lambda}\mid c)\cdot p(c)dc}{p(\boldsymbol{\lambda})} \\
&= \frac{l(\boldsymbol{\lambda})\cdot\int_{-\infty}^{c^*}(c^*-c)\cdot p(c)dc}{\gamma\cdot l(\boldsymbol{\lambda})+(1-\gamma)\cdot g(\boldsymbol{\lambda})} \\
&= \frac{l(\boldsymbol{\lambda})\cdot\left(c^*\cdot\int_{-\infty}^{c^*}p(c)dc - \int_{-\infty}^{c^*}c\cdot p(c)dc\right)}{\gamma\cdot l(\boldsymbol{\lambda})+(1-\gamma)\cdot g(\boldsymbol{\lambda})} \\
&= \frac{l(\boldsymbol{\lambda})\cdot\left(c^*\cdot p\left(c<c^*\right) - \int_{-\infty}^{c^*}c\cdot p(c)dc\right)}{\gamma\cdot l(\lambda)+(1-\gamma)\cdot g(\boldsymbol{\lambda})} \\
&= \frac{l(\boldsymbol{\lambda})\cdot c^*\cdot\gamma - l(\boldsymbol{\lambda})\cdot\int_{-\infty}^{c^*}c\cdot p(c)dc}{\gamma\cdot l(\boldsymbol{\lambda})+(1-\gamma)\cdot g(\boldsymbol{\lambda})} \\
&= \frac{c^*\cdot\gamma - \int_{-\infty}^{c^*}c\cdot p(c)dc}{\gamma+(1-\gamma)\cdot\frac{g(\boldsymbol{\lambda})}{l(\boldsymbol{\lambda})}} \\
&\propto \left(\gamma+\frac{g(\boldsymbol{\lambda})}{\ell(\boldsymbol{\lambda})}\cdot(1-\gamma)\right)^{-1}
\end{aligned} \tag{5}$$