AN AUTOMATED FRAMEWORK TO GENERATE END-TO-END MACHINE LEARNING PIPELINES

A Thesis

by

ANURAG KAPALE

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2019

Major Subject: Computer Science

ABSTRACT

The recent developments in machine learning have shown its applicability in numerous real-world applications. However, building an optimal machine learning pipeline requires considerable knowledge and experience in data science. To address this problem, many automated machine learning (AutoML) frameworks have been proposed. However, most of the existing AutoML frameworks treat the pipeline generation as a black-box optimization problem. Thus, failing to incorporate basic heuristics and human intuition. Furthermore, most of these frameworks provide very basic or no feature engineering abilities. To tackle these challenges, in this thesis, we propose an automated framework to generate end-to-end machine learning pipelines. By survey of 100s of Kaggle kernels and extensive experimentation, we finalized a set of heuristics which enhances the pipeline optimization problem. We also implemented a system to automate feature engineering, which could generate 100s of features to produce better representation of the data. Additionally, the framework provides interpretations about why certain models and features were selected by the system. This would help the users to further improve the pipeline. Finally, our experimentation shows consistent performance across various datasets.

ACKNOWLEDGEMENTS

I truly appreciate my advisor, Dr. Xia 'Ben' Hu, for his support, guidance, and encouragement throughout the course of my research. I would also like to thank my committee members, Dr. Bobak Mortazavi and Dr. Yang Shen for their guidance and valuable comments on the research. Furthermore, I would like to thank all DATA lab members. It was a wonderful experience and a lot of fun working with such a great group of people. Finally, my deepest gratitude goes to my family for their support all through these years.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee consisting of Dr. Xia Hu and Dr. Bobak Mortazavi of the Department of Computer Science and Engineering and Dr. Yang Shen of the Department of Electrical and Computer Engineering. All work for the thesis was completed independently by the student.

Funding Sources

This work was, in part, supported by DARPA (#FA8750-17-2-0116) and NSF (#IIS-1750074). The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

NOMENCLATURE

ML	Machine Learning
AutoML	Automated Machine Learning
SMAC	Sequential Model-based Algorithm Configuration
TPE	Tree Parzen Estimator
SMOTE	Synthetic Minority Oversampling Technique
KNN	K-Nearest Neighbors
GBM	Gradient Boosting Machine
SVM	Support Vector Machine
ROC	Receiver Operating Characteristics
AUC	Area Under the (ROC) Curve

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

In recent years, Machine Learning has been successfully used in many real-world applications. From Finance, Marketing, Advertisement to Healthcare, there are numerous use-cases of these techniques. The widescale applications and promising results have led to the proliferation of the libraries, providing out of the box implementation of various machine learning algorithms and techniques. Libraries such as Scikit-learn [1], XGBoost [2], Light-GBM [3] and frameworks like TensorFlow [4], PyTorch [5] are extensively used to develop machine learning solutions. However, generating an optimal machine learning pipeline takes a lot more than using these algorithms. It requires a significant amount of skills and experience. A typical workflow involves repeated rounds of feature extraction, feature selection, model development, benchmarking, hyper-parameter tuning, etc. Moreover, for every change in data distribution, this process needs to be repeated. These challenges led to the rise of the automated machine learning (AutoML) systems, which try to address this repetitive nature of work and reduce the entry barriers to the people with minimal data science experience.

In this thesis, we propose a framework to automate the end-to-end process of generating machine learning pipelines. Given a tabular dataset and time budget, our framework aims to come up with the machine learning pipeline with the optimal performance for the given task.

We envision this framework to be useful for two types of user groups: basic users and machine learning experts. The basic users could make use of its intuitive interface to easily generate a well-performing model for their data and desired task. On the other hand, machine learning experts could use our system to automate and simplify parts of their workflow.

Imagine a scenario where a data scientist is developing a stock trade prediction model. It is unknown before-hand which methods would work the best. He would need to spend hours to develop the pipelines, benchmark and tune them for the optimal performance. Moreover, different stocks may have different data distributions as well as different features associated with them. This would require him to re-do most of his work for each one of them. Our system could help by providing a decent baseline to start the development. It could give insights about which models and features perform the best and consequently help him further tune the performance. More importantly, when developing models for the other stocks, based on his experience, he could specify a better hyper-parameter search space and configuration. Thus, overall our system could help him improve the predictive performance at the same time significantly reducing the workload.

1.1. Objectives

The major goal of our framework is to automate the problem of generating machine learning pipelines for tabular datasets for the tasks of classification and regression. Additionally, it should satisfy the following objectives:

1.1.1. User-friendly Design

One of the major goals of our framework is to provide a simple and intuitive interface to the user. The user should be able to use the library with few simple lines of code without needing to spend hours learning its APIs. Additionally, it should be flexible enough to accommodate various user needs. If a user comes with a preferable pre-existing workflow for the given problem, then he should be able to extend that to our framework without significant overhead.

1.1.2. Good Baseline Performance

In many machine learning applications especially in the domains such as finance or healthcare, the predictive performance of the model is very important. Even the small improvements in the performance could be worth thousands of dollars. Therefore, our framework should be consistently able to produce good baseline performance across various real-world problems.

1.1.3. Automated Feature Engineering

Feature engineering is the process of using domain knowledge of the data to create features that make machine learning algorithms work [6]. However, creating useful features from the data requires a good deal of experience and deep understanding of the data. Our framework aims to simplify the process by going through multiple steps of feature generation and feature selection to provide a set of useful features. These features can be used by the framework to improve its performance or by the user to include in his workflow.

1.1.4. Interpretability

In addition to generating an optimal machine learning pipeline, our framework should also provide the user with few key insights about which models/features perform well for the given task. This could help the user to further improve the performance by better tuning the search space and focusing on better set of features.

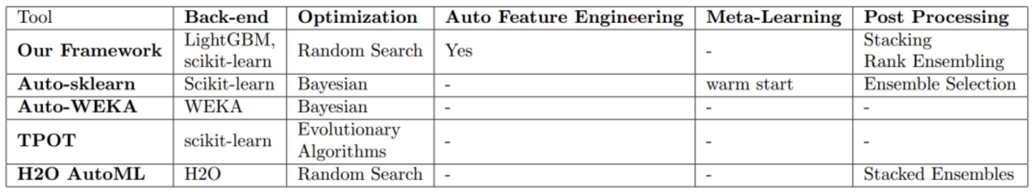
2. RELATED WORK

Over the years, various methods have been proposed to automate the complete machine learning pipeline or the parts of it. Most of the previous work has been focused on treating the automated machine learning as an optimization problem to find the best model and hyper-parameter settings. Moreover, in recent years, a considerable amount of attention has been given to topics such as Neural Architecture Search and Automated Feature Engineering. In the following sections, we would briefly explain the previous work in each of these aspects.

2.1. Model and Hyperparameter Optimization

The automated model and hyperparameter search methods intend to tune the hyper-parameters and the models to build the entire machine learning pipeline automatically. Various optimization methods such as Bayesian Optimization, Evolutionary Algorithms, Random Search, etc. are used for this purpose.

Auto-WEKA [20] is one of the first AutoML platforms. It is based on the WEKA [8] models and uses SMAC [10] (a variant of Bayesian Optimization) for optimizing the hyper-parameters. auto-sklearn [7] uses Scikit-learn models along with SMAC based optimization. It tunes the entire pipeline of preprocessors and models using conditional hyperparameter spaces. It also uses the meta-learning to warm start the search, based on the dataset properties. On the other hand, TPOT [12] uses evolutionary algorithms along with the Scikit-learn models. H2O AutoML [14] uses the random search-based optimization followed by stacked ensembles to enhance the performance. A brief comparison of each of these platforms along with our framework is given in Table 1 [19].



Tool	**Back-end**	**Optimization**	**Auto Feature Engineering**	**Meta-Learning**	**Post Processing**
Our Framework	LightGBM, scikit-learn	Random Search	Yes	-	Stacking Rank Ensembling
Auto-sklearn	Scikit-learn	Bayesian	-	warm start	Ensemble Selection
Auto-WEKA	WEKA	Bayesian	-	-	-
TPOT	scikit-learn	Evolutionary Algorithms	-	-	-
H2O AutoML	H2O	Random Search	-	-	Stacked Ensembles

Table 1 Reprinted from [19]: Comparison of AutoML tools

2.2. Automated Feature Engineering

Automated Feature Engineering aims to generate informative and discriminative features from the given data, followed by selecting the most useful ones for the model. Feature engineering requires human insight and good understanding of the data. Therefore, it is very difficult to automate. There are only a handful number of frameworks which support automated feature engineering. FeatureTools [9] is an open-source framework based on Python. It automatically generates features by applying feature transformation operations, such as mean or sum. AutoCross [15] is an automatic feature crossing framework by 4Paradigm. It uses cross-product of categorical features to automatically capture interactions between them, which is followed by beam search to find the optimal feature set.

In our experience, FeatureTools is more relevant for the relational databases and cannot generate many meaningful features for flat tabular datasets. While AutoCross works on tabular datasets, it mainly focuses on generating high order cross features and therefore is computationally expensive.

2.3. Neural Architecture Search

Due to the recent widespread success of deep learning, there has been a growing amount of interest in automating the search and design of the neural network structures. NAS [16], NASNet [17], ENAS [13], DARTS [11], AutoKeras [18] are few of the examples of using AutoML for deep learning. Due to the high computational cost and lower interpretability of the deep neural networks, we limit our focus on the traditional machine learning models. Therefore, further discussion of these techniques is out of the scope for this work.

3. PROPOSED FRAMEWORK

In this section, we propose an automated machine learning framework for tabular datasets in order to satisfy the objectives we discussed earlier: (1) intuitive user interface (2) good baseline performance (3) automated feature engineering and (4) interpretations of the feature/model selection.

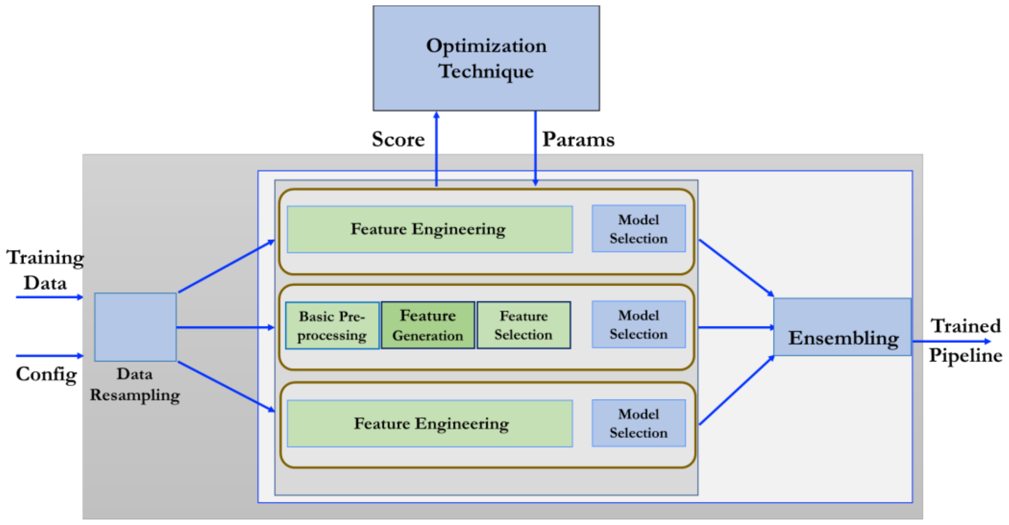


Figure 1 The architecture of the proposed framework

The general architecture of the framework is shown in Figure 1. Similar to the most machine learning algorithms, the framework operates in three phases. First, during the initialization phase, it takes-in the configuration parameters from the user, validates them and defines the components and their search spaces accordingly. Next, during the training phase, it takes the training data and target labels as input, runs the optimization to find the best set of components for the pipeline. Finally, during the test phase, it takes in the test data and make predictions using the previously trained pipeline. For each of these phases, our framework provides an intuitive single line interface. Thus, a user can use the framework with just 3 lines of code without spending a lot of time in learning the APIs.

Our framework consists of 5 major components: Data resampling, Feature engineering, Model selection, Optimization technique, and Ensembling. In the following sections, we would explain each of these components in further details.

3.1. Data Resampling

Oftentimes for the classification problems, the training data contains an imbalanced distribution of the classes. Most of the machine learning models have a bias towards the majority class, which makes them overlook the instances of the minority class. Thus, the model fails to correctly classify the minority class samples from the test data. To address this problem, we use the following resampling techniques to equally represent the minority classes:

1. Up-sampling: This technique randomly samples the minority classes with replacement and adds duplicate instances to the dataset. This process is repeated until the desired class ratio is achieved. This technique is effective only when the imbalance ratio is low. For the higher levels of imbalance, adding duplicate instances will cause the model to overfit to the repeated samples. Therefore, we use this technique only when the ratio of the number of majority class samples to the number of minority class samples is less than 2.

2. Data Augmentation: When the imbalance ratio is higher, instead of duplicating the minority samples, it is preferred to add artificial samples that are representative of the minority class. To do so we use the Synthetic Minority Oversampling Technique (SMOTE) algorithm [24]. The simplified pseudocode of the algorithm is shown in Figure

2. For every sample in the minority class, its K nearest neighbors are found. Then, one of the samples and its nearest neighbor are selected randomly. Next, the vector difference between the sample and its neighbor are multiplied by a random number between 0 and 1. A new sample is generated by adding this scaled difference to the selected sample point. This process is repeated until the desired class ratio is achieved.

```
Algorithm 1: SMOTE algorithm
   Input: Minority class samples: $T$, Target number of samples: $N$
   Output: Optimal hyper-parameter value: $\lambda^*$
1  for $x$ in $T$ do
2      $NN[x] \leftarrow$ FindNearestNeighbors($x$)
3  while $N > 0$ do
4      $x \leftarrow$ SelectRandom($T$)
5      $x_k \leftarrow$ SelectRandom($NN(x)$)
6      $x' \leftarrow x +$ rand(0, 1) $* \mid x - x_k \mid$
7      $T \leftarrow \mathrm{T} \cup x'$
8      $N \leftarrow N - 1$
```

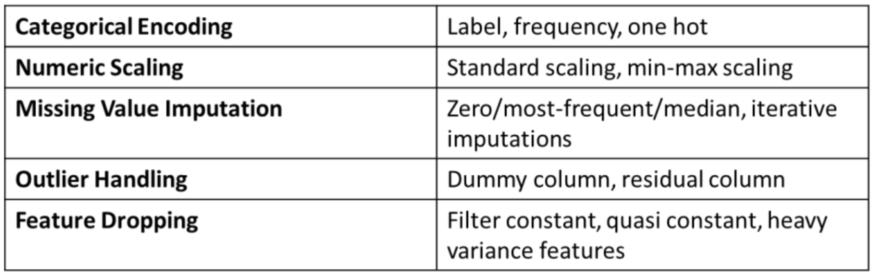
Figure 2 SMOTE algorithm

3.2. Feature Engineering

Good features can significantly improve the predictive performance. Therefore, feature engineering is one of the most important parts of a machine learning pipeline. It roughly consists of basic preprocessing, feature generation and feature selection.

3.2.1. Basic Preprocessing

Most machine learning models require clean numeric data. However, in real-world scenarios, the data is often messy with categorical variables, missing values, outliers, etc. There are a variety of techniques available to deal with each of these aspects. The right set of preprocessing techniques depends on many factors including the data distribution, model type, task type, evaluation metric, etc. It is challenging to select an optimal subset from such a wide range of options. Therefore, based on the heuristics, we designed a search space of various preprocessing operators for categorical encoding, missing value imputations and outlier handling. The list of some of these preprocessing operators is given in Table 2. We then use the optimization methods to select from this space, the optimal set of preprocessing operations applicable to the given scenario.



Categorical Encoding	Label, frequency, one hot
Numeric Scaling	Standard scaling, min-max scaling
Missing Value Imputation	Zero/most-frequent/median, iterative imputations
Outlier Handling	Dummy column, residual column
Feature Dropping	Filter constant, quasi constant, heavy variance features

Table 2 List of basic preprocessing operations

3.2.2. Feature Generation

Feature generation is the process of transforming and combining the data to create useful features. Based on the inspiration from the Deep Feature Synthesis [9] and AutoCross [15], we designed a system to generate features in an automated manner. The process relies on the observation that the most common features can be generated by sequentially applying a list of simple data-agnostic operations. These operations could be considered as the building blocks of the feature engineering. Therefore, we implemented our feature engineering technique using many such basic operations called primitives. Primitives are the simple computations that can be applied on a single column (first order) or multiple columns (second order).

3.2.2.1. First-Order Primitives

These primitives are applied directly to the single column (or feature) of the data and output the transformed version of it. For example, a label encoder takes as an input a feature column and maps each of the feature value to a numeric value. In similar manner, the first order primitives map one feature to one or many features. Most of the pre-processing, feature generation, and feature selection methods in our framework are implemented using the first order primitives.

3.2.2.2. Second Order Primitives

Very often the target depends not only on a single feature but also on interactions between two or more of them. Second order features try to find the simple interactions between two (or more) columns by combining them based on the defined aggregation strategy. The examples of some of these aggregation strategies are mean, common count or max. Table 3 provides an example of calculating a second order primitive. In this example, ‘Genre’ and ‘Country’ are categorical feature whereas the ‘Duration’ is a numerical feature. To generate categorical-categorical (cat-cat) interaction (‘Genre-Country’), we use ‘common count’ as the aggregation strategy. Thus, we count the number of common occurrences for the different values of the input features. The set (‘Sci-Fi’, ‘Japan’) occurs twice in the dataset, so its corresponding value is set to 2 whereas the rest of the combinations are mapped to 1. Similarly, to find categorical-numerical (cat-num) interaction, we use the ‘mean’ as the aggregation strategy. So, for ‘Genre-Duration’ feature, we take the mean of all the values of ‘Duration’ for the given value of ‘Genre’.

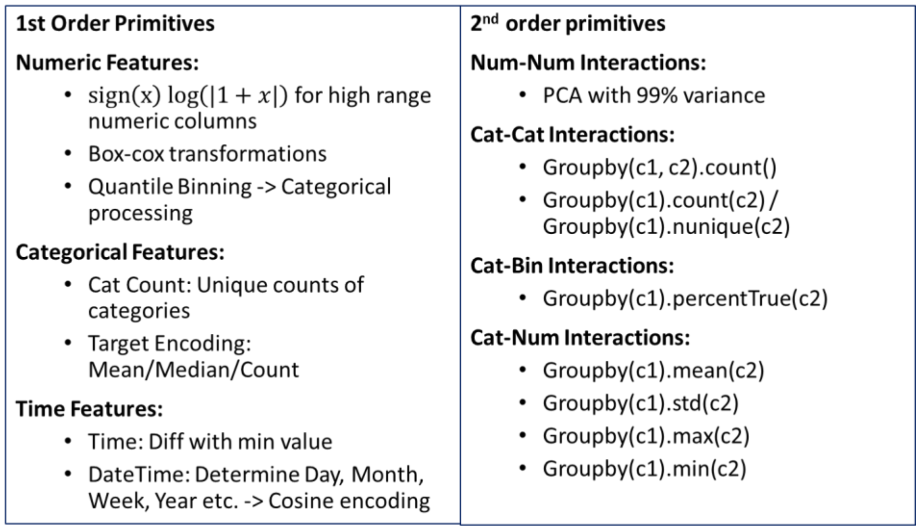
Thus, the value for the records corresponding to ‘Sci-Fi’ are mapped to 190 (i.e. mean of 180, 190, 200) and ‘Action’ are mapped to 110.

Second order features are useful as they may capture better correlation with the target. For example, in our case, the target is ‘Y’ whenever ‘Genre-Country’ is high and vice versa. In this way, cross features often help to improve its prediction power of the model by capturing the complex relationship between features.

Name	Genre	Country	Duration	Genre-Country	Genre-Duration	User Review
A	Sci-Fi	Japan	180	2	190	Y
B	Action	USA	100	1	110	N
C	Sci-Fi	Japan	200	2	190	Y
D	Sci-Fi	USA	190	1	190	N
E	Action	Japan	120	1	110	N

Table 3 Example of the second order primitives

Based on the data types and the properties of the feature columns, we apply a list of these first and second order primitives. This process generates 100-1000s of features, many of which are filtered using the feature selection methods. The list of some of the first order (single column) and second order (cross column) primitives is shown in Table 4.



1st Order Primitives	2nd order primitives
Numeric Features:	**Num-Num Interactions:**
• $\text{sign}(x)\ \log(\|1 + x\|)$ for high range numeric columns	• PCA with 99% variance
• Box-cox transformations	**Cat-Cat Interactions:**
• Quantile Binning -> Categorical processing	• Groupby(c1, c2).count()
Categorical Features:	• Groupby(c1).count(c2) / Groupby(c1).nunique(c2)
• Cat Count: Unique counts of categories	**Cat-Bin Interactions:**
• Target Encoding: Mean/Median/Count	• Groupby(c1).percentTrue(c2)
Time Features:	**Cat-Num Interactions:**
• Time: Diff with min value	• Groupby(c1).mean(c2)
• DateTime: Determine Day, Month, Week, Year etc. -> Cosine encoding	• Groupby(c1).std(c2)
	• Groupby(c1).max(c2)
	• Groupby(c1).min(c2)

Table 4 List of feature primitives

3.2.3. Feature Selection

Feature selection is the process of finding a feature subset, which will be most informative to the model. Any machine learning model is as good as the data input to it. Irrelevant or mutually correlated features can negatively affect the performance of the model, as the model can overfit to these irrelevant features. Additionally, reducing the number of features can save the computational and memory requirements of the model. Especially for our framework, as we use automated feature engineering to generate a lot of features, a good feature selection technique is essential to select the most useful ones and filter out the rest. However, finding a perfect subset of features is an NP hard problem, so instead we approximate using greedy selection methods.

We propose a two-step feature selection algorithm which uses univariate feature selection method to filter out the basic uninformative features followed by a feature importance method to remove the features with low feature important score.

Step 1: Univariate Selection: First we remove the features with very low variance (constant value) or extremely high variance (almost all different values). Next, we calculate the Pearson correlation score of each of the features with the target. Finally, based on the threshold, we remove the features with low correlation score.

Step 2: Feature Importance: We train a Light-GBM model on all of the features remaining after the first step. We use the trained model's feature importance scores, to find which features are more useful. We normalize these scores by dividing them by their mean. The features with the importance score below the threshold are filtered out.

The complete procedure is explained in Figure 3:

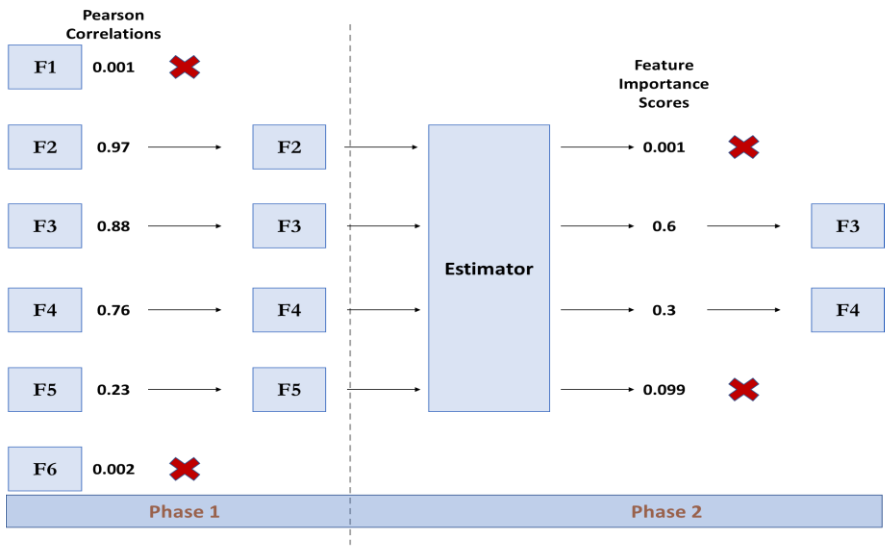
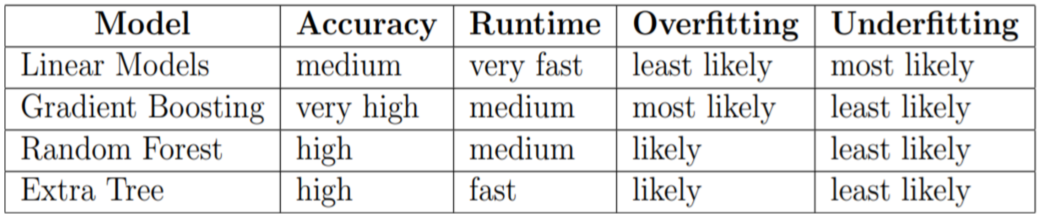


Figure 3 Two-step feature selection method

3.3. Model Selection

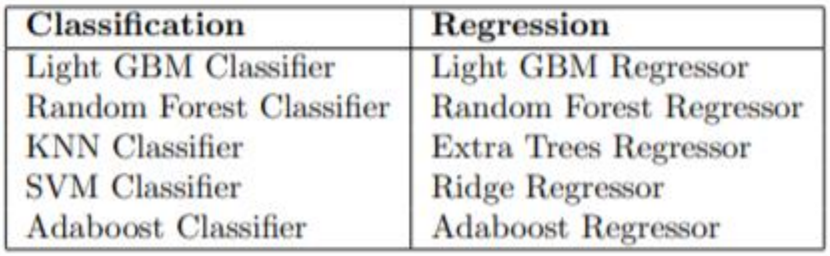
A single machine learning model cannot perform well for every scenario. Every model has its strengths and weaknesses. For example, linear models are very good at differentiating between linearly separable classes. Random Forests on the other hand can learn complex decision boundaries but often overfit the linearly separable cases. Table 5 [26] shows a very general comparison of strengths and weaknesses of various model types.



Model	Accuracy	Runtime	Overfitting	Underfitting
Linear Models	medium	very fast	least likely	most likely
Gradient Boosting	very high	medium	most likely	least likely
Random Forest	high	medium	likely	least likely
Extra Tree	high	fast	likely	least likely

Table 5 Reprinted from [26]: Comparison of model types

We envision our platform to perform consistently across various practical problems, therefor we selected number of different models in our optimization search space. Moreover, as different types of models do good in different scenarios, we selected diverse set of estimators from various model families. The list of supported models is shown in Table 6.



Classification	Regression
Light GBM Classifier	Light GBM Regressor
Random Forest Classifier	Random Forest Regressor
KNN Classifier	Extra Trees Regressor
SVM Classifier	Ridge Regressor
Adaboost Classifier	Adaboost Regressor

Table 6 List of supported models

3.4. Optimization Techniques

At its core, every AutoML system can be considered as an optimization problem to find the best settings for the pipeline viz. model, hyper-parameters and feature engineering components. In this section we would review the optimization techniques used by our framework. There are various optimization strategies such as Grid Search, Random Search, Bayesian Optimization, Genetic Algorithms, Reinforcement Learning etc. Each of them has many variations. We compared many of these methods based on their predictive performance with respect to the time budget. Based on our experimentation, we shortlisted following two options to be used for our framework:

3.4.1. Random Search

The random search samples the hyper-parameters randomly without replacement and evaluates them on the cross-validation data. In spite of being an extremely simple strategy, it tends to provide fairly good performance even in the case of small time-budgets. For n independent random samples, the probability of finding the top k percentile solution is greater than or equal to $1 - (1-k)^n$. Therefore, with only 100 trials, there is 0.9945 probability to find the top 5% of the hyper-parameter settings. The probability increases exponentially with the increase in the number of trials. Moreover, one does not need to maintain the history of trials. It is enough to store only the best setting found. Thus, the strategy requires $O(1)$ memory and is easy to restart without maintaining a state.

3.4.2. Bayesian Optimization

Bayesian optimization is a well-known technique to find optimal values for non-convex black box functions. The advantage of Bayesian technique is that it makes the smarter choices for the hyper-parameter search based on the history. It builds a probabilistic surrogate model which tries to approximate the objective function based on the hyper-parameters. It then uses an acquisition function to sample a set of hyper-parameter values. The true objective function is evaluated using these hyper-parameters which in turn are used to update the surrogate model. This process is repeated until the maximum number of iterations are reached. Finally, the optimal hyper-parameters are selected based on the surrogate model.

For our framework we use a variant of Bayesian optimization, which is based on Tree Parzon Estimator (TPE) proposed by Bergstra et al [27]. One disadvantage of Bayesian optimization is that it generally takes longer to converge. Therefore, when the time budget it low we use random search and in case of higher time budget we warm-start the Bayesian optimization with the random search, which can lead to faster convergence.

3.4.3. Multi-Step Hyper-Parameter Search

For optimizing a machine learning pipeline, one needs to optimize each of its component, along with their individual hyper-parameters. This makes the search space is exponentially large. Optimizing over this huge hierarchical search space requires significant time and computational budget. Therefore, to reduce the computational complexity, we take a greedy approach and do the optimization in multiple steps.

Step 1 Feature Optimization: In this step, we focus on optimizing the feature engineering parameters. Thus, we limit our search space to the all of the feature engineering parameters along with only the small number of base models from each model family. We then optimize over this smaller search space to select the best K feature engineering parameters.

Step 2 Combined Optimization: In this step, we combine the top K feature engineering parameters found in step 1, along with the complete model hyper-parameter space. We again optimize over this combined search space to pick the best pipeline/s.

This greedy approach allows us to partly decouple the optimization of feature engineering parameters and the model hyper-parameters. This significantly reduces the computational complexity of the optimization.

3.5. Ensembling

Ensembling can be defined as the meta learning algorithm which combines several machine learning models into one predictive model. Almost in every Kaggle competition the better use ensembling is the differentiating factor between the top performing solutions and the others. In machine learning, ensembling techniques have been extensively used to reduce bias, reduce variance or to improve the predictive performance of the model. With proper implementation, ensembling can also provide a form of regularization and prevent the models from overfitting. For ensembling to be effective, the base models should not be correlated with each other and there should be no data leakage between the base models and the ensembling algorithm.

Therefore, to implement ensembling we need to make two major choices: (1) How should we select the base estimators? and (2) Which ensembling technique should be used? We will explain each of them in the following sections:

3.5.1. Strategies to Select Base Estimators

In spite of the performance gains, training and predictions of the ensembling is computationally expensive. Therefore, we need to select only a few base models. We explored following strategies for selecting the K base models for the ensembling:

3.5.1.1. Top-K Estimators

This strategy sorts the search trials based on their performance and selects the top K estimators. While it picks the best performing estimators, it fails to provide diversity because very often the top search trials come from the same type of estimator.

3.5.1.2. Random Selection

It uses random selection with replacement to pick the models for ensembling. While this achieves diversity, when the K is low, it often ends up with many weak estimators. Thus, this strategy is most useful for constructing a large ensemble.

3.5.1.3. Diverse Selection

This strategy sorts the search trials and groups them based on the estimator family. It then picks equal number of best estimators from each estimator family. This ensures diversity as well as strong base estimators for the ensemble. In our experience, this strategy leads to the better performance even for a small or medium sized ensemble.

Therefore, by default we use the ‘diverse selection’ strategy to select up to 50 models for ensembling. For the number of base models higher than 200, we switch to ‘random selection’ strategy.

3.5.2. Ensembling Techniques

Once we finalize the base models, we combine the predictions of the base models by stacking them together in an array of size: *number of samples x number of base estimators*. Each entry in the array is a prediction made by a given estimator on a given data sample. There are many techniques to train the ensembling algorithm such as bagging, boosting, ensemble selection [21] etc. In our case, the base models are heterogeneous, therefore following two strategies work the best:

3.5.2.1. Rank Ensembling

Rank ensembling makes predictions by combining the results of the base model by a defined aggregation operation such as majority voting or averaging. It is one the most basic and yet useful techniques for ensembling. It is very convenient as it does not require any training for the ensembling algorithm. During prediction phase it simply applies the aggregation operation across multiple base models.

3.5.2.2. Stacking

Stacking considers the base pool of predictions as features and learns a meta-estimator to combine these predictions. It was proposed by Wolpert [25] and has been widely popularized after the Netflix Prize competition, where it was used as the important winning strategy by the top teams. For our framework, we use Light GBM models as meta-estimator.

To avoid any sort of data leakage, we divide the training data in two parts: train data and holdout data. The base estimators are trained only on the train data and the stacking meta estimator is trained on the predictions made by the base estimators on the holdout data. This practice is also known as blending. To further optimize the results, we also do hyper-parameter optimization for the meta-estimator (Light-GBMs).

4. EXPERIMENTS AND RESULTS

In this section we perform experiments on real world datasets, to show the efficacy of our proposed framework. We would like to answer the following questions in this section: (1) How to effectively evaluate the performance of an AutoML framework? (2) How well does our framework compare with other AutoML frameworks across various types of datasets? (3) How efficient is our framework with respect to the allocated time budget? (4) Which components of the framework contribute the most towards the performance? (5) Can the framework provide interpretations about which features, and models should be selected?

To answer these questions, we design an elaborate experimental setup along with the choices of datasets and evaluation metrics. Then, we compare the overall predictive performance of our framework with two well-known AutoML frameworks: auto-sklearn and H2O AutoML. Next, we compare the performance of these 3 frameworks across various time budgets to see their efficiency. Then, we run an ablation study to compare the performance benefits of using feature engineering and ensembling for our framework. Finally, we show the methods provided by our framework to interpret the features, models and hyper-parameters of the pipeline.

4.1. Experimental Setup

Benchmarking the performance of an automated machine learning system has many challenges. Many of these challenges are also applicable to the general machine learning systems, however due to the data driven nature of AutoML systems, they are even more prominent to them.

First, there is a significant amount of variance in the performance of the same AutoML system in different trial runs. Most of the components of the machine learning pipeline such as the optimization techniques, models, cross-validation, sampling etc. have some amount of randomness involved. Thus, the overall performance of the system could vary across runs. To minimize this problem, we fixed the random seed for Python, NumPy and all of the other modules used. Moreover, to avoid overfitting to a single random seed, we tested the performance across 5 different seeds.

Second, the time and computational budget plays a very important role. Some AutoML systems converge faster whereas others converge slower but give better performance. Therefore, it is difficult to estimate the appropriate budget to make a fair comparison. Comparing to the evaluation framework used by other works [7, 20] and considering the practical constraint we fixed out budget to be 30 mins for each run.

Finally, we need to ensure that each experiment is run on the same dataset and similar train/test splits. Therefore, we used the OpenML [23] APIs to define a uniform interface, which can fetch the datasets, extract the feature type information and provide the train/test splits. This guarantees the consistency in terms of using the same dataset across various runs.

4.1.1. Dataset Selection

A good automated machine learning system should perform well on diverse practical scenarios. The datasets chosen for the experimentation should be representative of various real-world problems. Therefore, to find such representative datasets, similar to [19], we used the following approach:

First, we should avoid very easy datasets. Some of the datasets are so easy that even an untuned Random Forest could give a comparable performance. These kinds of datasets make it unclear if the difference in performance is due to the choice of technique or the natural variance between the methods. Therefore, we tried to limit the number of such datasets.

Next, there should be diversity in terms of the task type and data type distribution. We considered different scenarios with the varying number of features/samples, number of categorical/numerical columns, number of missing values etc.

Finally, the datasets should be representative of the real-world scenarios. Therefore, except for a few famous artificial datasets such as ‘kr-vs-kp’ we mostly picked the real-world datasets.

Based on the above considerations, we selected 21 datasets for classification and 11 datasets for regression from various sources such as Kaggle, UCI [28], KDD contests and OpenML [23], etc. We plan to add more datasets with further diversity to ensure consistent results across datasets.

Table 7 and Table 8 show the list of the datasets selected for classification and regression respectively along with their attributes:

Name	#Samples	Task_Type	#Numeric	#Categorical	#MissingValues
iris	150	Multiclass (3)	4	1	0
ecoli	336	Multiclass (8)	7	1	0
blood-transfusion-service-center	748	Binary	4	1	0
fri_c1_1000_25	1000	Binary	25	1	0
credit-g	1000	Binary	7	14	0
pc4	1458	Binary	37	1	0
OVA_Breast	1545	Binary	10935	1	0
quake	2178	Binary	3	1	0
Titanic	2201	Binary	3	1	0
fbis.wc	2463	Multiclass (17)	2000	1	0
splice	3190	Multiclass (3)	0	61	0
sick	3772	Binary	7	23	6064
wilt	4839	Binary	5	1	0
mushroom	8124	Binary	0	23	2480
eeg-eye-state	14980	Binary	14	1	0
MagicTelescope	19020	Binary	10	1	0
kropt	28056	Multiclass (18)	0	7	0
adult	48842	Binary	2	13	6465
KDDCup09_appetency	50000	Binary	192	39	8024152
mnist_784	70000	Multiclass (10)	784	1	0
covertype	581012	Binary	54	1	0

Table 7 List of classification datasets

Name	#Samples	#Numeric	#Categorical	#MissingValues
analcatdata_negotiation	92	5	1	26
fri_c3_100_25	100	26	0	0
analcatdata_gsssexsurvey	159	5	5	6
cholesterol	303	7	7	6
cleveland	303	7	7	6
plasma_retinol	315	11	3	0
liver-disorders	345	6	1	0
chscase_census2	400	8	0	0
pbc	418	14	6	1033
meta	528	20	2	504
kin8nm	8192	9	0	0

Table 8 List of regression datasets

4.1.2. Evaluation Metrics

The choice of evaluation metric affects how one interprets the performance of different AutoML systems. For example, in case of highly imbalanced classes, if only the accuracy was to be evaluated, then the system which exclusively predicts the majority class could perform better. In real-life scenario, the evaluation metrics are chosen based on the use-case at hand. However, it is almost impossible to guess it beforehand. For the benchmarking purposes, we need to use an evaluation metric which can act as a good proxy for the practical scenario. Based on the survey of the similar works [7, 20], we finalized the following evaluation metrics for our experiments.

Classification: Balanced Accuracy, AUC ROC.

Regression: Mean Absolute Error (MAE), R2 score.

4.2. Results

4.2.1. Overall Performance

To evaluate the overall performance of our framework, we tested it on the list of datasets, for the time budget of 30 mins. To account for the variance in the results, we repeated the experiments 5 times, with different seeds. The final score was calculated by using the median of the performances across these 5 runs. We compare the performance of our framework with two other AutoML frameworks viz. auto-sklearn and H2O AutoML.

For classification, we used balanced accuracy as an evaluation metric. Table 9 shows the performance scores of each framework for classification. The best performing AutoML framework is underlined. As it can be seen, our framework performs the best on 14 out of 21 datasets for classification. Even when it does not perform the best, the score is within 2-3% range of the best performance.

Name	Our framework	AutoSklearn	H2O AutoML
adult	0.778455	0.780152	**0.788948**
mushroom	**1.000000**	**1.000000**	**1.000000**
kropt	**0.774079**	0.427179	0.683019
credit-g	**0.754761**	0.721429	0.700024
BNG(kr-vs-kp)	**0.882037**	0.500000	0.865097
ecoli	0.925000	**0.975000**	0.974576
splice	**0.959884**	0.949495	0.934023
Titanic	**0.786666**	0.746644	0.771465
quake	0.533336	**0.571995**	0.564187
wilt	**0.946280**	0.891502	0.880351
iris	**0.973333**	0.966667	0.971751
pc4	0.714263	0.734375	**0.753590**
fri_c1_1000_25	0.907004	0.903784	**0.923333**
fbis.wc	**0.783667**	0.772888	0.743992
covertype	**0.833977**	0.816623	0.786543
sick	**0.934782**	0.913043	0.928041
eeg-eye-state	0.926657	0.939524	**0.951197**
KDDCup09_appetency	**0.924734**	0.913277	0.678534
MagicTelescope	0.842984	0.859643	**0.868566**
blood-transfusion-service-center	**0.613818**	0.599415	0.612548
OVA_Breast	**0.970142**	0.966456	0.942331

Table 9 Overall performance for classification (balanced accuracy)

Name	Our Framework	AutoSklearn	H2O AutoML
analcatdata_gsssexsurvey	**0.463762**	0.475978	0.481276
plasma_retinol	**165.595675**	167.957828	168.170444
kin8nm	**0.126088**	0.187507	0.167034
cleveland	**0.719198**	0.747834	0.738289
fri_c3_100_25	0.994159	**0.991869**	0.992632
pbc	**0.450224**	0.476568	0.467787
liver-disorders	**2.369421**	2.4096	2.396207
analcatdata_negotiation	**0.607866**	0.636591	0.617757
meta	**274.324583**	299.177176	294.570989
cholesterol	33.765439	**32.927914**	33.355266
chscase_census2	**0.306153**	0.381204	0.382781

Table 10 Overall performance for regression (MAE)

The performance results for regression are shown in Table 10. In case of regression, our framework performs the best in 9 out of 11 datasets. Even in the rest 2 datasets, its performance is very close to the best. Thus, the experiment demonstrates how our framework shows consistent performance in both classification as well as regression.

4.2.2. Efficiency with Respect to Time-Budget

For an AutoML system, in addition to the performance, it is also important to consider the time taken for the optimization. In this experiment, we compare the performance of the 3 AutoML systems with respect to the different time budgets. We run each of the framework from the beginning for the time budgets of 1, 2, 4, 8, 16, 30, 60 and 120 min for 2 different datasets: ‘sick’ and ‘kropt’. The comparison is plotted in Figure 4 and Figure 5.

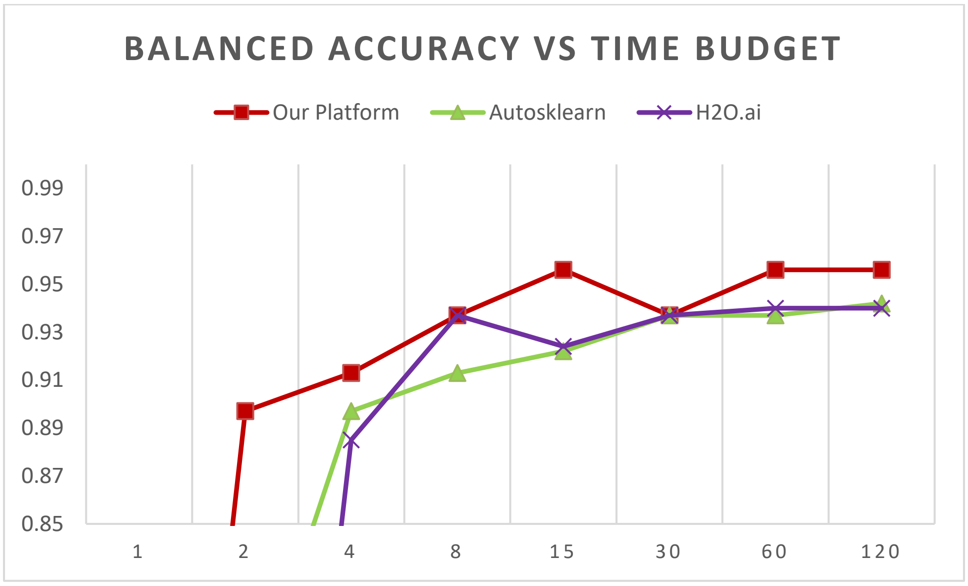


Figure 4 Performance w. r. t. time budget ('sick' dataset)

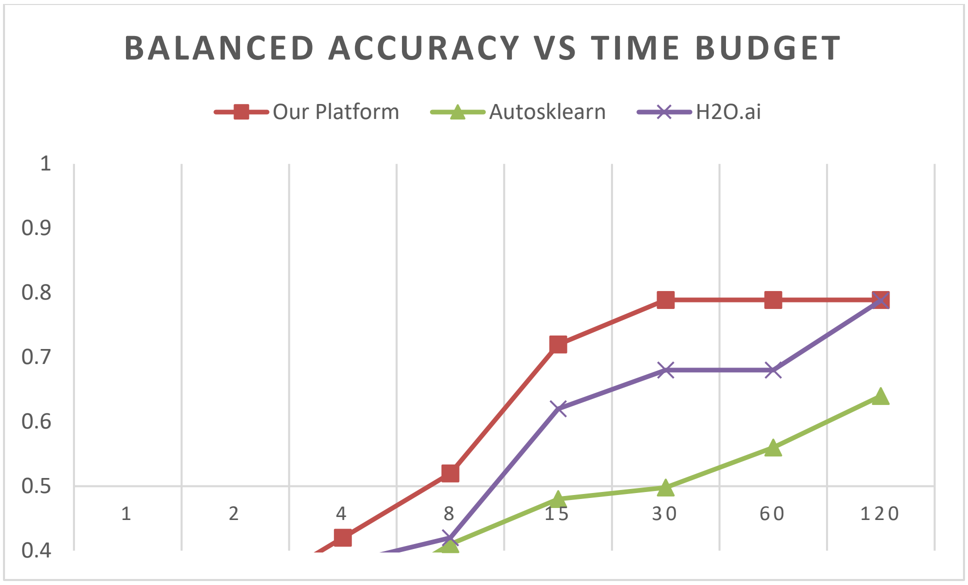


Figure 5 Performance w. r. t. time budget ('kropt' dataset)

As it can be seen, our framework achieves a decent performance in lesser time as compared to the other two. In case of ‘kropt’ dataset, where it performs as good as H2O AutoML, our framework achieves it in 30 min mark, as compared to 120 min. In our opinion, this can be partly attributed to the multi-step hyper-parameter optimization technique, which greedily optimizes the feature engineering and model selection in multiple steps.

4.2.3. Evaluation of the Pipeline Components

To evaluate the importance of feature engineering and ensembling for our framework, we performed an ablation study on 5 different datasets. First, we evaluated the complete framework. Next, we evaluated it again, removing the ensembling. Finally, we removed both ensembling and feature engineering only keeping the basic preprocessing to encode the categorical values and impute the missing values. The comparison can be seen in Table 11.

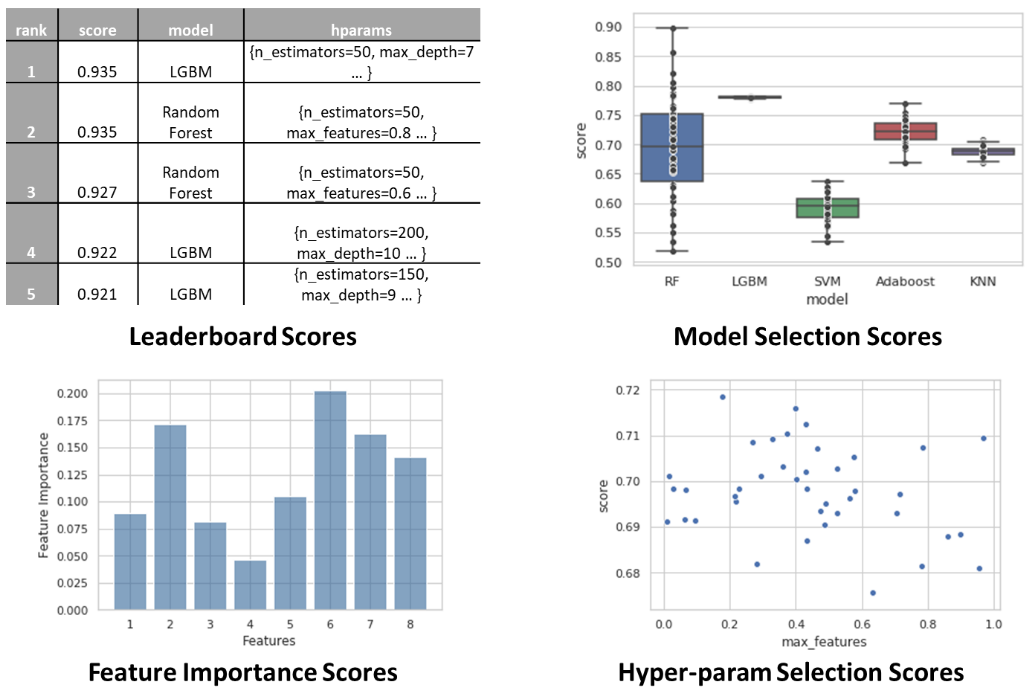
Dataset Id	Complete Pipeline	Remove Ensembling	Remove Feature Engineering
38	0.935	0.927	0.895
46	0.959	0.913	0.910
179	0.778	0.782	0.764
184	0.774	0.743	0.696
293	0.834	0.814	0.787

Table 11 Comparative evaluation of the pipeline components

As we can see, except for Dataset 179, both ensembling and feature engineering resulted in performance gains. Even in case of the dataset 179, the performance drop due to ensembling is negligible (0.004). Overall, on an average, the ensembling leads to about 2% increase in the balanced accuracy whereas the feature engineering leads to over 3% increase. The similar trend is observed with the other datasets as well.

4.2.4. Interpretability of AutoML System

In this section, we will show how our framework can provide the interpretations of various models, features and hyper-parameters selected in the pipeline. This is partly inspired by ATMSeer [29], which is an interactive tool to visualize and refine AutoML search space. Interpretability of AutoML framework is very helpful to the expert users, who could use these interpretations to further tune the search space and design an even better pipeline. Our framework provides interpretations for four different aspects: pipeline, model, feature and hyper-parameters. Figure 6 shows an example of interpretations for each of these aspects.



rank	score	model	hparams
1	0.935	LGBM	{n_estimators=50, max_depth=7 … }
2	0.935	Random Forest	{n_estimators=50, max_features=0.8 … }
3	0.927	Random Forest	{n_estimators=50, max_features=0.6 … }
4	0.922	LGBM	{n_estimators=200, max_depth=10 … }
5	0.921	LGBM	{n_estimators=150, max_depth=9 … }

Figure 6 Example of the interpretations provided by the framework

First, on the pipeline level, the framework provides the leaderboard for every pipeline tried during the optimization. It includes the cross-validation score of the pipeline along with the model and list of hyper-parameters. This gives a high-level overview of the performance across various instances of pipelines.

Second, on the model level, the framework can show the distribution of scores for each model type. It provides an insight about the mean and variance in the performance of each model types. For the example in Figure 6, Light-GBM consistently provides the best performance. On the other hand, the performance of KNNs varies significantly.

Third, on the feature level, the framework can provide feature importance scores for every feature generated. These scores are same as the ones used by the feature selection method. They provide some intuition on which features are more useful to the model. An expert user can use this understanding to further tune the feature engineering and create more useful features.

Finally, on the hyper-parameter level, the framework can provide the spread of the cross-validation scores for the range of values of the given hyper-parameter. This is useful to roughly find the optimal range of hyper-parameter values, for the given problem.

5. CONCLUSIONS AND FUTURE WORK

In this thesis, we proposed an automated framework to generate end-to-end machine learning pipelines. The framework is intuitive and provides a simple to use interface. For every component of the AutoML systems, we surveyed and benchmarked various techniques to find the best set of heuristics. These heuristics combined with the optimization methods allowed the framework to achieve comparable performance across various datasets. We also implemented a method to automate feature engineering, which could generate 100s of features to provide better representation of the data. Finally, our framework provides interpretation scores for the AutoML process, which provides the user some explanation on which models and features perform the best for the given problem.

This work would be extended by assembling it into an open source package, AutoKaggle. This would make these techniques easily accessible for the use of wider audience. Another natural extension would be to support more tasks such as time series analysis or anomaly detection and more types of datasets such as multi-entity datasets. Both of these lines of work are really important as most of the data in finance and medical field contains time series data and are in the form of relational tables. Finally, more work can be done to improve the interpretability of the AutoML system. Currently, our framework provides importance scores for various model and features combinations. In addition to this, it would be helpful to provide a graphical interface to guide user at each step of the process.

REFERENCES

[1] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, Édouard Duchesnay. “Scikit-learn: Machine Learning in Python”, The Journal of Machine Learning Research, 12, p.2825-2830, 2/1/2011.

[2] Chen, Tianqi, and Carlos Guestrin. “XGBoost: A Scalable Tree Boosting System”, Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD’16, 2016, 785-94. Accessed May 7, 2017.

[3] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. “Lightgbm: A highly efficient gradient boosting decision tree”, Advances in Neural Information Processing Systems, pages 3149–3157, 2017.

[4] M. Abadi, A. Agarwal et al., “Tensorflow: Large-scale machine learning on heterogeneous distributed systems”, 2016.

[5] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. “Automatic differentiation in PyTorch”, NIPS 2017 Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques, Long Beach, CA, US, December 9, 2017, 2017.

[6] Wikipedia contributors. (2019, September 29). Feature engineering. In Wikipedia, The Free Encyclopedia. Retrieved 15:29, October 25, 2019, from https://en.wikipedia.org/w/index.php?title=Feature_engineering&oldid=918620875.

[7] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. “Efficient and robust automated machine learning”, Neural Information Processing Systems (NIPS), 2015.

[8] Hall, M. et al. “The WEKA data mining software: an update”, SIGKDD Explor. 11, 10–18 (2009).

[9] J. M. Kanter and K. Veeramachaneni. “Deep feature synthesis: Towards automating data science endeavors”, IEEE International Conference on Data Science and Advanced Analytics, DSAA 2015, Paris, France, October 19-21, 2015, pages 1–10. IEEE, 2015.

[10] F. Hutter, H. H. Hoos, and K. Leyton-Brown. “Sequential model-based optimization for general algorithm configuration”, International Conference on Learning and Intelligent Optimization, pages 507–523. Springer, 2011.

[11] H. Liu, K. Simonyan, and Y. Yang. “DARTS: Differentiable architecture search”, International Conference on Learning Representations, 2019.

[12] R. S. Olson, N. Bartley, R. J. Urbanowicz, and J. H. Moore. “Evaluation of a tree-based pipeline optimization tool for automating data science”, Proceedings of the Genetic and Evolutionary Computation Conference 2016, GECCO ’16, pages 485–492, New York, NY, USA, 2016. ACM.

[13] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. “Efficient neural architecture search via parameters sharing”, Proceedings of the 35th International Conference on Machine Learning, volume 80 of Proceedings of Machine Learning Research, pages 4095–4104, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[14] H2O.ai. H2O AutoML, August 2017. URL http://docs.h2o.ai/h2o/latest-stable/

h2o-docs/automl.html. H2O version 3.14.

[15] L. Yuanfei, W. Mengshuo, Z. Hao, Y. Quanming, T. WeiWei, C. Yuqiang, Y. Qiang, and D. Wenyuan. “Autocross: Automatic feature crossing for tabular data in real-world applications”, arXiv preprint arXiv:1904.12857, 2019.

[16] B. Zoph and Q. V. Le. “Neural architecture search with reinforcement learning”, arXiv preprint arXiv:1611.01578, 2016.

[17] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. “Learning transferable architectures for scalable image recognition”, The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2018.

[18] H. Jin, Q. Song, and X. Hu. “Auto-keras: Efficient neural architecture search with network morphism”, arXiv:1806.10282, 2018.

[19] P. Gijsbers, E. Ledell, J. Thomas, S. Poirier, B. Bischl, and J. Vanschoren, “An Open Source AutoML Benchmark,” 6th ICML Workshop on Automated Machine Learning, pp. 1–8, 2019.

[20] Thornton, C., Hutter, F., Hoos, H. H. & Leyton-Brown, K. “Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms”, Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ’13, 847– 855 (ACM, New York, NY, USA, 2013).

[21] Caruana R, Niculescu-Mizil A, Crew G, Ksikes. “Ensemble selection from libraries of models”, twenty-first international conference on Machine learning, July 04–08, 2004, Banff, Alberta, Canada.

[22] D. Opitz, R. Maclin. “Popular ensemble methods: An empirical study”, J. Artificial Intelligence Res., 11 (1999), pp. 169-198.

[23] J. Vanschoren, J. van Rijn, B. Bischl, and L. Torgo. “OpenML: Networked science in machine learning”, SIGKDD Explorations, 15(2):49–60, 2013.

[24] N. V. Chawla, K.W. Bowyer, L. O. Hall, W. P. Kegelmeyer. “SMOTE: Synthetic Minority Over-Sampling Technique”, Journal of Artificial Intelligence Research, vol. 16, 321-357, 2002.

[25] Wolpert, D.H. (1992). “Stacked Generalization. Neural Networks”, Vol. 5, pp. 241-259, Pergamon Press.

[26] Ngo, K.T. “Stacking Ensemble for auto_ml”, Virginia Tech. Available online: https://vtechworks.lib.vt.edu/handle/10919/83547 (accessed on 12 September 2018).

[27] James S. Bergstra, Remi Bardenet, Yoshua Bengio, and Bálázs Kégl. “Algorithms for hyper- parameter optimization”, Advances in Neural Information Processing Systems 25. 2011.

[28] Dua, D. and Graff, C. “UCI Machine Learning Repository” [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[29] Qianwen Wang, Yao Ming, Zhihua Jin, Qiaomu Shen, Dongyu Liu, Micah J Smith, Kalyan Veeramachaneni, and Huamin Qu. “Atmseer: Increasing transparency and controllability in automated machine learning”, Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, page 681. ACM, 2019.