SapientML: Synthesizing Machine Learning Pipelines by Learning from Human-Written Solutions

Ripon K. Saha[†], Akira Ura[§], Sonal Mahajan[†], Chenguang Zhu[★¶], Linyi Li[‡¶], Yang Hu[★¶],
Hiroaki Yoshida[†], Sarfraz Khurshid[★], Mukul R. Prasad[†]
[†]Fujitsu Research of America, Inc., [§]Fujitsu Ltd., [★]The University of Texas at Austin,
[‡]University of Illinois at Urbana-Champaign
{rsaha,ura.akira,smahajan,hyoshida,mukul}@fujitsu.com
{cgzhu,huyang,khurshid}@utexas.edu,linyi2@illinois.edu

ABSTRACT

Automatic machine learning, or AutoML, holds the promise of truly democratizing the use of machine learning (ML), by substantially automating the work of data scientists. However, the huge combinatorial search space of candidate pipelines means that current AutoML techniques, generate sub-optimal pipelines, or none at all, especially on large, complex datasets. In this work we propose an AutoML technique SapientML, that can learn from a corpus of existing datasets and their human-written pipelines, and efficiently generate a high-quality pipeline for a predictive task on a new dataset. To combat the search space explosion of AutoML, SapientML employs a novel divide-and-conquer strategy realized as a three-stage program synthesis approach, that reasons on successively smaller search spaces. The first stage uses meta-learning to predict a set of plausible ML components to constitute a pipeline. In the second stage, this is then refined into a small pool of viable concrete pipelines using a pipeline dataflow model derived from the corpus. Dynamically evaluating these few pipelines, in the third stage, provides the best solution. We instantiate SapientML as part of a fully automated tool-chain that creates a cleaned, labeled learning corpus by mining Kaggle, learns from it, and uses the learned models to then synthesize pipelines for new predictive tasks. We have created a training corpus of 1,094 pipelines spanning 170 datasets, and evaluated SapientML on a set of 41 benchmark datasets, including 10 new, large, real-world datasets from Kaggle, and against 3 state-of-the-art AutoML tools and 4 baselines. Our evaluation shows that SapientML produces the best or comparable accuracy on 27 of the benchmarks while the second best tool fails to even produce a pipeline on 9 of the instances. This difference is amplified on the 10 most challenging benchmarks, where SapientML wins on 9 instances with the other tools failing to produce pipelines on 4 or more benchmarks.

CCS CONCEPTS

• **Software and its engineering → Automatic programming**.

KEYWORDS

Program Synthesis, Machine Learning, AutoML, Program Analysis

1 INTRODUCTION

The explosive growth in machine learning (ML) applications, over the past decade, has created a huge demand for data scientists (DS) and ML practitioners to develop real-world ML solutions. The 2018 LinkedIn Workforce Report showed a shortage of 151,717 DS, nationwide [26], that had grown to 250,000 by 2020 [32]. Automatic machine learning, or AutoML, holds the promise of addressing this shortfall [16, 19, 51]. AutoML can improve productivity of data science teams and cover gaps in expertise.

Given a dataset and a predictive task (e.g., classification or regression) AutoML aims to create an ML pipeline that trains an optimized ML model for the given task. Simply put, the pipeline is a sequence of ML operators that processes data to make it suitable for learning (feature engineering (FE)), fits a suitable ML model on it (model selection), and calculates the predictive performance of the model. One of the prominent instances of AutoML, the subject of much research recently, is creating supervised ML pipelines for tabular data [7, 10, 15, 30, 40, 49, 50]. This paper also focuses on this formulation of AutoML.

AutoML has been traditionally solved as a search and optimization problem – *selecting* the best pipeline from a space of candidates [15, 30, 43, 49, 50]. However, ML pipelines are also programs, in fact relatively small, highly structured domain-specific programs, that could be amenable to program synthesis. Further, public repositories like Kaggle [21] and GitHub contain hundreds of thousands of human-written ML pipelines that could serve as starting points for synthesizing new pipelines. Indeed, program synthesis by mining or learning from existing program corpora has been successfully deployed for other endpoints of synthesis [3, 25, 28, 29, 33]. Emerging research [7, 10] demonstrates the promise of this perspective for ML pipeline synthesis. Our work also follows this philosophy but offers a novel take on the core challenge of AutoML.

The central challenge of AutoML is the massive *combinatorial* search space of candidate ML pipelines it exposes – compositions of different potential FE operators, each applied on different columns of the subject dataset, composed with one of several potential models or their ensembles. Further, each pipeline component may have its own space of hyper-parameters. Previous AutoML

[¶] These authors contributed to this work as interns at Fujitsu Research of America.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510226

techniques adopt several approaches to combat this combinatorial explosion. Some try to search a restricted search space by excluding FE from consideration [15, 43, 49], searching specific pipeline topologies [30], or a pre-compiled explicit corpus of synthetic pipelines [15, 49, 50]. Others try to prune the search space by using learned language models coupled with aggressive dynamic evaluation of *partial pipelines* [10] or by warm-starting search using constraints mined from human-written pipelines [7]. However, navigating the huge combinatorial search space of AutoML remains an open problem. In fact, *none* of the above techniques apply FE transforms to specific dataset columns, as a human DS would. Instead, they are blindly applied to the complete dataset, ostensibly to avoid injecting another set of hyper-parameters into the search space.

Insight. Our key insight is that the root cause of the AutoML search-space explosion is because previous AutoML techniques reason on complete ML pipelines (*combinations* of various ML components) as a single entity, ostensibly to capture dependencies between ML components. However, we observe that in many practical instances, the decision on whether to include a particular component (say an *imputer*) in a pipeline can be made based primarily on properties of the dataset (whether or not it has missing values), *independent* of other components. Indeed, human DS often employ such *best practices* when manually constructing ML pipelines. Once the set of *plausible components* to use for a given dataset are known, they can be used to assemble a target pipeline or a small population of plausible target pipelines to choose from. This would substantially mitigate the combinatorial explosion coming from exploring arbitrary combinations of components. Further, we hypothesize that these DS best practices are represented in publicly available human-written ML pipelines (say on Kaggle). Thus, these pipelines can potentially be mined to learn and then replicate human DS decision making to create viable pipelines for new datasets.

Proposed approach. Pursuant to the above insight, we propose an AutoML technique SapientML [1], that can learn from a corpus of existing datasets and their pipelines, and generate a high-quality pipeline for a predictive task on a new dataset. To combat the search space explosion of AutoML, SapientML employs a novel divide-and-conquer strategy, realized as a three-stage program synthesis approach that reasons on successively smaller search spaces. The first stage uses meta-learning to train a meta-model (offline phase) which is then used to *independently* predict the *suitability* of each ML component with respect to the given dataset (in the online phase). Specifically, this meta-model captures the relationship between features of the dataset (e.g., the presence of missing data values) and desired components in the pipeline (e.g., a data interpolation component). This prediction yields a ranked list of *pipeline skeletons*. Each pipeline skeleton is an (unordered) set of plausible ML components, to constitute a pipeline, each component mapped to specific (or all) dataset columns on which it should be applied. In the second stage, the skeletons are then *instantiated* into a small pool of viable concrete pipelines using a pipeline *dataflow model* mined from the corpus, and a small library of standard implementations for each ML component. For each candidate skeleton the pipeline components are correctly ordered and incompatible components discarded using the dataflow model, and each component instantiated using code templates from the library. Dynamically evaluating these few pipelines (the most expensive operation), in the third stage, yields the best pipeline. The concept of a multi-stage approach employing more expensive analyses on successively smaller spaces has been successfully used in other domains, including automatic program repair [28] and code search [27], among others. Our specific design is customized for ML pipeline synthesis.

We instantiate SapientML as part of a fully automatic end-to-end tool-chain that mines datasets and corresponding pipelines from Kaggle, automatically cleans and labels each pipeline, learns from this corpus and then synthesizes ML pipelines for predictive tasks on new datasets. We evaluate SapientML on a set of 41 benchmark datasets, including 10 new, large, real-world datasets from Kaggle and against 3 state of the art AutoML tools (AL [10], auto-sklearn [15], TPOT [30]) and 4 baselines. Our evaluation shows that SapientML produces the best or comparable accuracy on 27 of the benchmarks while the second best tool (AL), fails to even produce a pipeline on 9 of the instances. Further, on the most challenging 10 benchmarks SapientML wins on 9 instances with the other AutoML tools failing on 4 or more benchmarks.

This paper makes the following main contributions:

- **Technique:** A learning-based AutoML technique SapientML, that can efficiently synthesize high-quality supervised ML pipelines, using a novel divide-and-conquer approach to circumvent the combinatorial state-space explosion of AutoML.
- **Tool:** An implementation of SapientML as part of an automated tool-chain that creates a cleaned, labeled learning corpus by mining Kaggle, learns from it, and uses the learned models to then synthesize pipelines for predictive tasks on new datasets.
- **Evaluation:** A substantial evaluation of SapientML on a benchmark of 41 datasets, including 10 new, large, real-world datasets from Kaggle, comparing it to 3 state of the art AutoML tools and 4 baseline techniques for creating ML pipelines.

2 PROBLEM DEFINITION

A *tabular dataset*, $D = (X \times Y) \in \mathcal{D}$ is sampled from a distribution over a domain $\mathcal{X} \times \mathcal{Y}$ where $\mathcal{X}$ and $\mathcal{Y}$ denote an input domain and an output domain respectively. X is comprised of n rows and d columns, called *features*, where each row represents an observation consisting of d values from $\mathcal{X}$. Similarly Y is comprised of n rows and t columns where each row is a t-tuple of values or labels from $\mathcal{Y}$. A *supervised predictive task* on D is to learn a prediction function $h : X \longrightarrow Y$ such that $y \approx h(x)$. A predictive task is called a *classification* task when the y is discrete and called a *regression* task when y is continuous. For multi-label classification and multivariate regression, $|t| > 1$. Applying supervised machine learning (ML) to a predictive task requires a training version of the dataset D_{train} to train an ML model, and a held out test dataset, D_{test} to evaluate its performance. A single dataset D can also be split into D_{train} and D_{test}.

Given a dataset D, an *ML pipeline* ($P \in \mathcal{P}$) is a sequence of FE components followed by a model component that realizes a given predictive task. Hence, $P = [c_f^1, c_f^2, .., c_f^k, c_m]$ represents a pipeline with k FE components and one model. A pipeline *component* $c \in C$ is comprised of one or more API calls, and associated glue code, that together performs an atomic data-specific pipeline task, e.g.,

[1] an AutoML approach harnessing the wisdom (*sapere*) of human (*sapien*) data scientists.

⟨FE:OrdinalEncoder(card4, . . .), 0.73⟩ C'_f
⟨FE:OneHotEncoder(card4, . . .), 0.70 ⟩
⟨FE:Imputer(card2, card3, . . .), 0.81⟩
⟨FE:LinearScaler(X), 0.69⟩
⟨FE:DataBalancer(X), 0.58 ⟩

⟨MODEL:CatBoostClassifier(X), 1⟩
⟨MODEL:ExtraTreesClassifier(X), 2⟩ C'_m
⟨MODEL:XGBClassifier(X), 3⟩

(a) Predicted FE and Model components by the skeleton predictor with their probability scores and rank respectively

C'_f ∪ ⟨MODEL:CatBoostClassifier(X)⟩

C'_f ∪ ⟨MODEL:ExtraTreesClassifier(X)⟩

C'_f ∪ ⟨MODEL:XGBClassifier(X)⟩

(b) Three skeletons generated by the pipeline seeding phase

⟨FE:Imputer(card2, card3, . . .)⟩
⟨FE:OrdinalEncoder(card4, . . .)⟩
⟨FE:LinearScaler(X)⟩
⟨FE:DataBalancer(X) ⟩
⟨MODEL:CatBoostClassifier(X)⟩

(c) First skeleton after ordering and redundancy removal

Figure 1: Artifacts of Pipeline Seeding and Pipeline Instantiation Phases for IEEE-CIS-Fraud-Detection Example

filling missing values or transforming a categorical column to a set of numeric columns. There are two kinds of components: i) the FE components (c_f) that transforms a feature (x) or a set of features ($X' \subset X$) including data wrangling tasks, and ii) the model components (c_m) that performs the actual learning and prediction.

Given dataset $D = D_{train} \cup D_{test}$, a predictive task on D, and an accuracy metric σ (e.g., F1 score for classification and R^2 for regression problems respectively), our aim is to create an executable machine learning pipeline P for this dataset and task that maximizes σ on D_{test}. We pose this *pipeline synthesis* problem as a program synthesis problem with quantitative objectives, akin to [17].

3 MOTIVATING EXAMPLE

In this section, we illustrate the use-case and mechanics of our technique using a real-world dataset *IEEE CIS Fraud Detection* [44], provided by the company Vesta and hosted on Kaggle. It contains 591K rows of data, each corresponding to an e-commerce transaction represented by a rich set of 394 features. The features are mainly numeric (e.g., transaction amount) and string categorical values (e.g., device type). Some features are missing in some transactions. The predictive task is to label a transaction as fraudulent or not, based on its features, i.e., a binary classification task.

Use Case. Creating a pipeline for a predictive task may take a long time for a DS. The DS needs to decide on the appropriate set of feature engineering (FE) (c_f) and model (c_m) components to use, the right dataset columns (features) to apply each of them on, and then instantiate them in the right order so the pipeline executes on the dataset (D) without errors. Given the huge space of possibilities for these decisions, data scientists typically rely on their understanding of D, past experience, and often brute-force trial and error, to complete this laborious task [24]. AutoML tools can accelerate this process substantially, especially for a novice DS, by providing her with a good-quality, executable pipeline for potential last-mile optimization.

Key Challenge. Real-world, large, complex datasets like IEEE CIS Fraud Detection present particularly challenging cases for current AutoML tools. In order to navigate the huge combinatorial search space of possible candidate pipelines tools such as TPOT [30] and auto-sklearn [15] restrict themselves to numeric data, which ensures smaller, simpler pipelines. Thus, they cannot even run on the given dataset since it has string categorical features. The state-of-the-art tool AL [10] uses a combination of learned language model and aggressive dynamic evaluation of partial pipelines to search for a viable solution. However, in this case, it evaluates 1,641 partial and 1,310 complete pipelines in 1.5 hours (on a 8 vCPU and 32GB memory machine) and finally crashes due to an internal timeout without producing any pipeline.

SapientML's three-stage program synthesis approach proves to be quite effective on this example. In the first stage (Section 4.3), SapientML uses a machine-learned model, trained on its *meta-learning corpus* of human-written pipelines, to generate a ranked-list of *pipeline skeletons*, to construct viable pipelines. For the present example, SapientML first predicts five potential FE components and the top three most appropriate models in Figure 1a to generate three pipeline skeletons in Figure 1b. The predicted components broadly agree with human intuition. For instance, OrdinalEncoder and OneHotEncoder are reasonable transforms to encode the String-based features in the dataset and the use of DataBalancer comports with the significant imbalance between the number of fraudulent and valid transactions in the dataset. Further, the choice of the CatBoost model is consistent with the previous research [42] showing that CatBoost performs well for classification on large, imbalanced data. SapientML further uses the decision rules learned by the skeleton predictor to infer the relevant features in the dataset where each FE transform will be applied. For example, Figure 1a shows that SapientML targets card2, . . . for SimpleImputer.

In the second stage (Section 4.4), SapientML concretizes the pipeline skeleton into a set of executable pipelines. To this end, it uses the confidence scores included in the skeleton as well as a *pipeline dataflow meta-model* mined (offline) from the learning corpus to discard redundant FE components, and order the components in a syntactically correct fashion, to produce ordered skeletons, as shown in Figure 1c. For instance, the analysis concludes that both OrdinalEncoder and OneHotEncoder are to be applied on the same dataset columns but cannot be simultaneously used. Thus, OneHotEncoder which has a lower confidence score, is discarded. As another example, Imputer is ordered before OrdinalEncoder, following the mined component order relation. Next the ordered skeletons are transformed into a set of concrete pipelines (three in this case).

In the final stage, SapientML evaluates these candidate pipelines on a held-out validation dataset (derived from only the training dataset, *not* the testing dataset) and returns the highest accuracy pipeline. Figure 2 shows an abridged version of this pipeline.The pipeline implements a rich set of five FE components each applied to its appropriate columns and paired with a CatBoost gradient-boosting classification model. SapientML takes only 8 mins to produce this pipeline and produces a respectable 0.82 F1 score.

```
import pandas as pd   # LOAD DATA
__train_dataset=pd.read_csv("training.csv", delimiter=",")
__test_dataset=pd.read_csv("test.csv", delimiter=",")

from sklearn.impute import SimpleImputer   # FE TRANSFORM 1
import numpy as np
_NUMERIC_COLS_WITH_MISSING_VALUES = ['card2', 'card3', .. 'V339']
for _col in _NUMERIC_COLS_WITH_MISSING_VALUES:
    __imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
    __train_dataset[_col] = __imputer.fit_transform(__train_dataset[_col].values.reshape(-1,1))[:,0]
    __test_dataset[_col] = __imputer.transform(__test_dataset[_col].astype(
                                          __train_dataset[_col].dtypes).values.reshape(-1,1))[:,0]

_STRING_COLS_WITH_MISSING_VALUES = ['card4', 'card6',..., 'M9']   # FE TRANSFORM 2
...  ## Apply SimpleImputer for string columns similar to the FE transform 1 shown above

from sklearn.preprocessing import OrdinalEncoder   # FE TRANSFORM 3
...  ## Apply OrdinalEncoder to categorical columns ['ProductCD', 'card4', .., 'M9']

__feature_train = __train_dataset.drop(['isFraud'], axis=1)   # DETACH TARGET
__target_train =__train_dataset['isFraud']
__feature_test,  __target_test = __test_dataset.drop(['isFraud'], axis=1), __test_dataset['isFraud']

from sklearn.preprocessing import StandardScaler   # FE TRANSFORM 4
...  ## Apply StandardScaler to __feature_train and __feature_test

from imblearn.over_sampling import SMOTE   # FE TRANSFORM 5
__feature_train, __target_train = SMOTE().fit_resample(__feature_train, __target_train)

from catboost import CatBoostClassifier   # MODEL
__model = CatBoostClassifier()
__model.fit(__feature_train, __target_train)
__y_pred = __model.predict(__feature_test)

from sklearn import metrics   # EVALUATION
print(metrics.f1_score(__target_test, __y_pred, average='macro'))
```

Figure 2: Abridged version of pipeline generated by SapientML for the IEEE-CIS-Fraud-Detection prediction task

4 APPROACH

4.1 Overview

Figure 3 presents a high-level overview of the SapientML system. It has an offline and an online phase. In the offline phase SapientML creates a corpus of human-written pipelines and their datasets, called the *meta-learning corpus*, by mining data-science repositories (Kaggle in our case) and automatically curating the data for learning, through denoising, augmentation, and labeling. The meta-learning corpus is then used to build two meta-models, namely the *skeleton predictor* meta-model and the *pipeline dataflow meta-model*. In the online phase, given a new dataset and a predictive task (classification or regression) defined on it, SapientML uses two meta-models to synthesize a supervised ML pipeline for the given dataset and task, which maximizes some accuracy metric (e.g., F1 or R2).

SapientML navigates the huge combinatorial search space of AutoML through a novel three-stage program synthesis approach that reasons on successively smaller search spaces. The first stage, called *pipeline seeding*, uses the skeleton predictor derived through meta-learning on the meta-corpus, to independently predict the *suitability* of each ML component to appear in an ML pipeline for the given dataset, based on the meta-features of the dataset. This prediction yields a *pipeline skeleton*, an unordered set of plausible components, to constitute a solution pipeline. In the second, *pipeline instantiation* stage, this skeleton is concretized into a small pool of viable *candidate pipelines*, using the dataflow meta-model mined from the corpus, to correctly order, minimize, and instantiate the pipeline components. The final, *pipeline validation* stage selects the highest accuracy ML pipeline among the candidate pipelines by dynamically evaluating them. The following sub-sections describe the meta-corpus creation and the three pipeline synthesis stages.

4.2 Creation of the Meta-Learning Corpus

This step automatically mines and curates a high-quality corpus that includes human-written ML pipelines and their datasets, to power the meta-learning of SapientML. These pipelines naturally capture the expertise and domain knowledge of human DS as opposed to creating a relatively small, homogeneous, synthetic ML pipeline corpora used by some other AutoML techniques [15, 50] that incurs significant computational cost. To build the corpus, we first mine the datasets and their pipelines from Kaggle [23] – a popular data-science repository. Specifically, we collected top 350 datasets based on user votes, and up to 100 top-voted pipelines per dataset, giving us around 2,500 initial pipelines. These raw pipelines and datasets are further denoised, augmented, and labelled to make them suitable for learning by SapientML. Our final corpus is comprised of 1,094 pipelines across 170 datasets.

4.2.1 Denoising pipelines. Human-written notebooks on Kaggle often contain *noise* in the form of exploratory data analysis, visualization, and debugging code that while useful for human comprehension, are irrelevant to ML model execution. Further, some pipelines may no longer be executable due to various issues such as deprecated APIs and differences in the run time environment. To

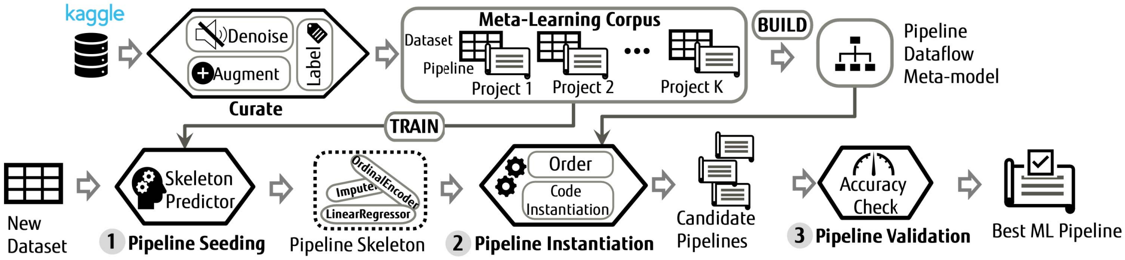

Figure 3: Overview of SapientML system.

construct a clean meta-learning corpus, we first discard any pipeline that fails to run successfully on our environment. Then to remove the noise in each executable pipeline, P, we first heuristically identify a criteria line (l_{cr}), which performs the final prediction task. In pipelines using the popular python ML libraries such as Scikit-learn[34] and XGBoost [48] this is typically a call to the `predict` API function. Then, we compute a forward slice P_{forw} and a backward slice P_{back} from l_{cr}, by applying standard dynamic program slicing [1] and concatenate P_{forw}, l_{cr}, and P_{back} to yield the clean pipeline P_{clean}. We compare the accuracy scores of P and P_{clean} and discard P_{clean} if it obtains a lower score than P.

4.2.2 Augmenting pipelines. This step is motivated by the observation that human-written pipelines may not contain the best representative ML model choice for each dataset. This could happen in pipelines written by novice data scientists or because of the availability of newer, better models after the pipeline was written. Presence of sub-optimal ML models in our meta-learning corpus in turn degrades the quality of the pipelines synthesized by SapientML. To alleviate this problem we employ a *data augmentation* technique to systematically replace sub-optimal models in meta-corpus pipelines with better, i.e., higher-accuracy, models. Data augmentation [36] is commonly employed in machine learning flows to improve the predictive quality of training data.

Generation of candidates. To improve the performance score of a denoised pipeline P_{clean} with model c_m, our data augmentation technique systematically replaces the model c_m in P_{clean} by each viable model in the corpus, $C_m = \{c_m^1, \ldots, c_m^b\}$ *one at a time*, to create a set of candidate pipelines $\mathcal{P}_{mutated} = \{P_{clean}^1 \ldots P_{clean}^b\}$. To identify the code for c_m, we start with l_{cr} (defined in Section 4.2.2) and compute a backward slice up to the model declaration. Next we identify the variable names of the model object, X_{train}, Y_{train}, X_{test}, and Y_{test} through static analysis. Finally we replace the declaration of the old model c_m with a new model c_m^i to generate P_{clean}^i.

Selection. Each mutated pipeline in $\mathcal{P}_{mutated}$ is run on the corresponding dataset, and the best mutated pipeline, P_{clean}^{best}, showing the highest improvement in the performance score replaces the original pipeline in the corpus.

4.2.3 Creation of abstract pipelines for meta-learning. The objective of creating a meta-learning corpus is to provide SapientML necessary training data to learn the relationship between various dataset properties and ML components. However, since human-written ML pipelines are diverse in terms of implementation, it is challenging to learn the relationships between various dataset properties and raw code snippets. To keep the meta-learning tractable, we represent a pipeline at an abstract level as a sequence of ML components, $P = [c_f^1, c_f^2, .., c_f^k, c_m]$ where c_f^i and c_m represent the labels of FE and model components respectively.

To this end, first we automatically annotate each component with a label that has two pieces of information: ⟨*component_type*:*API_Name*⟩. We primarily distinguish two types of components: feature engineering (`FE`) and `MODEL`. The automatic labeling process involves two steps: i) extracting the API name, and ii) identifying whether a particular API is an `FE` or a `MODEL` component. We perform an AST analysis to extract API names from each statement, ignoring any APIs that are part of boilerplate code. For example, almost every FE engineering APIs in Scikit-learn library are accompanied by a template code that contains `fit`, `transform`, or `fit_transform` APIs. Once we have all the API names, we first annotate the model component (already identified in Section 4.2.2 for each pipeline in our corpus). All components appearing before the MODEL component are labeled as FE components.

At this point, the abstract pipeline P is presented at the API level. However, there are many labels in the corpus that are functionally similar across pipelines. For example, a data scientist can use either the `fillna` API from `pandas` or `SimpleImputer` from `sklearn` to fill out the missing values in a dataset. To learn meaningful patterns of meta-features with respect to these labels, we have to group the components that are semantically similar. Applying domain-knowledge in ML is a standard practice and meta-learning is no exception. Therefore, we investigated the labels in our project corpus and group them based on their functionality by studying the API documentation. Then we assigned each group a functional label. For example, we grouped `FE:fillna`, `FE:interpolate`, `FE:SimpleImputer` and `FE:KNNImputer` together and mapped to a unified label `FE:Imputer` since they all are used to impute missing values in a dataframe.

4.3 Stage 1: Pipeline Seeding

Given a dataset (D) and predictive task, the objective of the pipeline seeding stage is to produce a ranked-list of *pipeline skeletons*, $\mathcal{S} = [S_1, \ldots S_k]$. This is used to constitute concrete candidate pipelines in the subsequent pipeline instantiation stage (Section 4.4).

A pipeline skeleton (S) is a (unordered) set of plausible ML components that includes zero or more FE components and one model component (Definition 4.3). To predict the ML components in S, SapientML uses a meta-learning model, called the *skeleton predictor*, trained during the offline meta-training phase. The skeleton predictor is architected as a set of sub-models, each of which learns a mapping between properties (*meta-features*) of a dataset D and

Table 1: Summary of Meta-Features

High-Level Property	Meta-features
Shape of dataset (3)	Number of rows, features, and targets
Missing entries (1)	Presence of missing values
Feature types (10)	Presence and #features, whose data type is numeric, number category, string category, text, and date.
Measure of symmetry (4)	Skewness and Kurtosis (normal, uniform, and tailed)
Measure of Distribution (6)	Normal, Uniform, and Poisson distribution for features and target
Tendency and Dispersion (3)	Normalized mean, standard deviation, variation across columns
Correlated features (3)	Pearson correlation (min, max, number of correlated features)
Outliers (2)	#features that contains few or many outliers.
Value frequency (3)	Number of features whose values are sparse, imbalanced, dominant
Target property (3)	Imbalanced, continuous or categorical.

Table 2: Meta-Targets(C: Classification, R: Regression)

Feature Engg.	Model	C	R	Model	C	R
Imputer	RandomForest	x	x	SVM	x	x
OrdinalEncoder	ExtraTrees	x	x	LinearSVM	x	x
OneHotEncoder	LightGBM	x	x	LogisticRegression	x	x
TextVectorizer	XGBoost	x	x	Lasso	-	x
TextPreprocessor	CatBoost	x	x	SGD	x	x
DateFeaturization	GradientBoosting	x	x	MLP	x	x
LinearScaler	AdaBoost	x	x	MultinomialNB	x	-
LogScaler	DecisionTree	x	x	GaussianNB	x	-
DataBalancer	-	-	-	-	-	-

the likelihood of a specific ML component (*meta-target*) appearing in a pipeline for D.

Definition 4.1 (Meta-features). A set of meta-features, $\Phi = \{\alpha_1, \alpha_2, \ldots, \alpha_l\}$, quantify the characteristics of a dataset where each meta-feature is computed by a function $\alpha_i : D \to \mathbb{R}$ that takes the dataset as input and outputs a real number.

4.3.1 *Design of Meta-Features.*

A good set of meta-features should have three properties: i) they are expressive enough to characterize the dataset, ii) they are succinct enough so that there exist some meaningful patterns that skeleton predictor can learn with respect to the ML components, and iii) they are efficient to compute. For example, the choice of `FE` and `MODEL` component often depends on the meta-features such as number of records and features in D and feature types. Based on the existing literature [10, 15] and our experience, we compute 38 meta-features to characterize the datasets in our meta-training corpus. Table 1 presents the list of meta-features used in SapientML.

Definition 4.2 (Meta-Targets). A set of ML components $C = C_f \cup C_m$ that define the prediction space of skeleton predictor where C_f and C_m represent FE and model components respectively.

4.3.2 *Meta-targets.*

Each ML component in the abstract pipelines created in Section 4.2.3 is a valid meta-target for SapientML. However, to learn any meaningful pattern between meta-features (Φ) and a particular ML component c, we need sufficient occurrences of c in the meta-corpus. Therefore, we excluded any ML components that appeared less than five times in our corpus. This filtering criteria provided us 9 FE components and 29 model components (15 classification models and 14 regression models) as meta-targets. Table 2 summarizes the ML components that SapientML's meta-model predicts.

4.3.3 *Design of Skeleton Predictor (Meta-Models).*

Given a set of meta-features, Φ computed from D, the objective of skeleton predictor is to predict a set of plausible FE components and model components to generate pipeline skeletons defined in Definition 4.3.

Definition 4.3 (Skeleton). A skeleton, $S = \{\langle c_f^1(X_1), \rho_1\rangle, \ldots, \langle c_f^q(X_q), \rho_q\rangle, c_m(X)\}$ is a set of tuples comprised of q FE components and one model component where $\langle c_f^i(X_i), \rho_i\rangle$ represents that the FE component c_f^i will be used in the pipeline with a probability ρ_i and applied on $X_i \subset X$ features in D.

We use the following insights to design our skeleton predictor. First, a pipeline may require several FE components and in many cases the decision of using a particular FE component can be made based on a few meta-features without depending on other FE components. Although occasionally there can be some dependencies between the ML components, our experimental results show that this design decision leads to faster generation of pipelines without sacrificing accuracy. To this end, we design the FE component predictor as a set of binary classifiers $\{\lambda_1, \ldots \lambda_9\}$ that predicts whether a particular FE component $c_f^i \in C_f$ should be used in the generated pipeline for the target dataset, D. On the other hand, since by design SapientML allows only one model $c_m \in C_m$ in a skeleton, we cast the model selection problem as a ranking problem and design a learning-to-rank model to rank all the model components for D.

Definition 4.4 (Skeleton Predictor). The skeleton predictor is comprised of a set of sub-models, $\Lambda = \{\lambda_1, \ldots \lambda_9, \lambda_m\}$ where each sub-model approximates a function, $\lambda_i : \mathbb{R}^l \to y_i'$ where $\mathbb{R}^l$ is a set of meta-feature values and y_i' is a probability score of an ML component (*meta-target*) appearing in a pipeline for D.

Sub-models to predict the FE components. FE components are generally applied on a sub-set of features of D. For example, an `<FE:Imputer>` is generally applied on the features with missing values. Therefore, SapientML aims to predict not only an FE component (c_f^i) for D but also infers the subset of features $X' \subset X$ in D on which c_f^i would be instantiated on. To facilitate the determination of X' for c_f^i, a `DecisionTreeclassifier` is a natural fit since the classifier would learn a set of precise conditions w.r.t. the meta-features to select c_f^i for D and later we can analyze those conditions to infer X' for c_f^i. However, `DecisionTree` models tend to overfit with a large number of features [35]. To minimize the effect of irrelevant features, we first perform a *point biserial correlation* analysis between the meta-features (Φ) and each FE component (c_f^i) and use only the meta-features ($\Phi' \subset \Phi$) that exceeds a certain correlation threshold to train λ_i. As a result, we get a set of binary classifiers $\{\lambda_1, \ldots \lambda_9\}$ as the FE components predictor where λ_i is used to predict c_f^i.

Sub-model to rank MODEL components. Unlike the prediction of FE components, which often depends on a few meta-features, it is challenging to determine a few meta-features that can predict the performance of a particular model on D [49]. Therefore, instead of predicting one model based on a few meta-features, we design a learning-to-rank sub-model that considers all the meta-features Φ to rank all the model components in our corpus. Considering the size of our meta-training dataset, which is not very large and the fact that the ensemble models are better than a single model for complex learning task [5], we designed an ensemble of `LogisticRegression` and `SupportVectorMachine` to rank the model components. More specifically, these meta-models first compute a probability score for each model component and use the average score to sort the target model components.

4.3.4 Training the Skeleton Predictor (Offline). We trained all the sub-models in the skeleton predictor using the meta-training corpus. Since the proportion of pipelines having and not having a ML component is not equal in the corpus, we used balanced weighting strategy to solve the *class imbalance* problem. Further, we tuned the hyper-parameters through 5-fold cross validation and grid-search.

4.3.5 Generation of Pipeline Skeletons (Online). During pipeline generation, SapientML first computes the set of meta-features, Φ from D and passes it to the skeleton predictor (Λ), which returns a set of plausible FE components $\{c_f^1 \dots c_f^q\}$ with probability scores and a ranked-list of the model components $C'_m = [c_m^1, \dots c_m^r]$.

Inferring relevant features. For each c_f^i in the predicted set, SapientML infers the relevant features, $X'_i \subset X$ in D on which c_f^i can be successfully applied and create the semi-instantiated set of FE components, $C'_f = \{\langle c_f^1(X_1), \rho_1 \rangle, \dots, \langle c_f^q(X_q), \rho_q \rangle\}$. Such inference is important to help avoid pipeline failures caused by infeasible transforms, such as `StringVectorizer` applied to a numeric column. Further, it helps precisely identify the columns most suitable for the FE transform, e.g., applying `SimpleImputer` to only those columns that have missing values.

To infer relevant features for c_f^i, first SapientML access the the sub-model λ_i, which is a decision tree classifier that predicted c_f^i for D. Then SapientML extracts the decision path that led to the prediction. A decision path is a list of conditions in form of $[\alpha_1\ op\ v_1, \dots \alpha_u\ op\ v_u]$ where α, op, and v correspond to a meta-feature, >= or <, and a real number respectively. Then SapientML iterates over each feature $x_i \in X$ and selects x_i only if it satisfies at least one of the conditions in the decision path. For example, SapientML correctly applies the `OrdinalEncoder` on `card4`, which is a string categorical feature whereas it marks the `TransactionAmt` feature as irrelevant, which is indeed a numeric feature.

Generation. Finally SapientML selects Top-k models from C'_m and adds one-by-one to the selected FE components to generate k number of pipeline skeletons, $S = [\{C'_f \cup c_m^1\}, \dots \{C'_f \cup c_m^k\}]$.

4.4 Stage 2: Pipeline Instantiation

This stage synthesizes a set of concrete pipelines for the given user dataset (D) and its predictive task. Given a ranked-list $\mathcal{S}$ of pipeline skeletons produced by pipeline seeding (Section 4.3), this stage instantiates each skeleton S into a concrete pipeline P by first creating an ordered skeleton S_O representing a syntactically viable data flow, and then instantiating the components in S_O into a pipeline template, along with necessary glue code. This yields a set of k candidate pipelines, $\mathcal{P}_{cand} = \{P_1, P_2, \dots, P_k\}$.

4.4.1 Create Ordered Skeleton. The goal of this step is to order the components of S, and discard *incompatible components*, if any, to produce an *ordered skeleton* S_O, such as the one shown in Figure 1c. This operation uses a pipeline *dataflow meta-model* extracted by SapientML, from the meta-learning corpus, during the offline phase. We develop the description using the following terminology.

Definition 4.5 (Dataflow dependence). There exists a dataflow dependence between components c_i and c_j of a pipeline P for a dataset D iff there exists feature x_i of D on which both c_i and c_j are applied in P.

There exists a dataflow from c_i to c_j in P, denoted $c_i \xrightarrow{P} c_j$, iff there is a dataflow dependence between c_i and c_j and c_i precedes c_j in P. Dataflow dependence, as defined above, can be inferred through a simple static analysis. The details are elided for brevity. Although neither sound nor complete, this definition provides a simple, efficient way to capture dataflow in all but the most complicated pipelines.

The dataflow meta-model is a partial-order relation Δ capturing the dataflow between pipeline components, as observed in the corpus pipelines. Specifically,

$$\Delta = \{(c_i, c_j) \in C \times C \mid \exists P \in \mathcal{P}_L, c_i \xrightarrow{P} c_j \text{ and } \nexists P' \in \mathcal{P}_L, c_j \xrightarrow{P'} c_i\}$$

The dataflow meta-model is represented as a directed acyclic graph (DAG), $\mathcal{G}_\Delta$, whose nodes are the components C and directed edges $(c_i, c_j) \in \Delta$.

A skeleton S produced by pipeline seeding is transformed into an ordered skeleton S_O using the following steps. First, all dataflow dependencies are captured between potential skeleton components, using Definition 4.5. If there is any component pair c_i, c_j that has a dataflow dependence but no edge between c_i, c_j in $\mathcal{G}_\Delta$, this indicates an incompatible component. Hence the component with the lower predicted probability is discarded. In our motivating example (Figure 1), components `OneHotEncoder` and `OrdinalEncoder`, which happen to be semantic substitutes of each other (e.g., convert categorical columns to numeric) present such an instance. Hence, the lower probability component `OneHotEncoder` is discarded. Discarding all such components yields a reduced skeleton S'.

Next, a sub-graph of the dataflow meta-model $\mathcal{G}_\Delta$ with only the nodes in the reduced skeleton S' is extracted. Finally, a topological sort on this sub-graph provides a component order for the reduced skeleton S' consistent with $\mathcal{G}_\Delta$. This order is to create the ordered skeleton S_O. For our motivating example, `Imputer` precedes `OrdinalEncoder` in S_O. Reversing the order for a column with missing values would result in a pipeline crash.

4.4.2 Generate concrete pipeline. Each ordered skeleton S_O is converted into a concrete pipeline P by instantiating each component $c \in S_O$ in order, into a pipeline template of the kind shown in Figure 2. Specifically, each $c \in C_f \cup c_m$ is instantiated using a parameterized snippet drawn from a small pre-compiled library of standard component implementations, by appropriately filling the parameter holes. For example, `OrdinalEncoder` is instantiated by filling the columns hole with relevant columns ('ProductCD', 'card4', . . .). SapientML can also handle type-based instantiation of components. For example, `SimpleImputer` is instantiated differently for filling missing values in numeric vs. string columns.

4.5 Stage 3: Pipeline Validation

Each candidate pipeline $P \in \mathcal{P}_{cand}$ is dynamically evaluated to compute an accuracy score (F1/R^2), to find the best pipeline, P_{best}. SapientML internally splits the *user-provided* training data D_{train} into *internal* training and validation sets which are used for the training and validation of candidate pipelines within this stage. Therefore, the held out *test* dataset, D_{test} (shown as "test.csv" in Figure 2) is completely unseen to SapientML. Finally, P_{best} is used to train on D_{train} and evaluated on D_{test} for the accuracy score returned to the user.

5 EVALUATION

Our evaluation addresses the following research questions:

RQ1: How does SapientML perform compared to the existing state-of-the-art techniques?

RQ2: How robust SapientML is in producing good quality pipelines across trials?

RQ3: Does SapientML use its search space well to predict a diverse set of FE and model components?

RQ4: Does each of the novel technology components of SapientML contribute to its effectiveness?

5.1 Experimental Set-up

5.1.1 Implementation. SapientML is implemented in the Python programming language using approximately 5,000 lines of code. It includes a crawler to download ML projects, a set of tools for required static and dynamic analysis such as mining the order of ML components from a corpus, denoising pipelines, a meta-feature extractor, and machine learning models for the skeleton predictor. SapientML uses Kaggle Public APIs [22] for automatically downloading data, the Python-PL library [9] to instrument source code for dynamic program slicing, the scikit-learn [31, 34] library to implement meta-models for the skeleton predictor, and LibCST [20] for static analysis. SapientML uses Pandas, Numpy, and Scipy for computing meta-features and its own data analysis.

5.1.2 Benchmarks. We use a set of 41 benchmark datasets to evaluate SapientML. This includes the set of 31 datasets used in AL[10]. They include 12 datasets from the OpenML suite, 9 from PMLB, 4 from Mulan, and 6 from Kaggle. However, since most of these datasets are small and simple in nature, we have added 10 new datasets from Kaggle as representatives of large, real-world datasets which modern AutoML tools should handle. To systematically select the 10 new benchmark datasets we collected all the *Featured* and *Playground* Kaggle competitions completed since the year 2015. From these we selected ones operating only on tabular data and where the license permits academic research and use outside the competition. Finally, we selected 10 datasets based on size either in terms of large number of rows or columns, or have various types of columns. Table 3 presents the size, prediction task, and source repository for each benchmark dataset.

5.1.3 Experimental Methodology. SapientML is trained on our meta-learning corpus of 1,094 pipelines and corresponding cleaned datasets. Therefore, the 41 benchmark datasets are completely *unseen* to SapientML. Similar to AL [10] and auto-sklearn [15], we performed 10 trials of each experiment for each benchmark with a one hour time out. For each trial, we randomly split the *user-provided* dataset into *training* and *testing* data in a 75:25 split. Then SapientML generated a pipeline using *only* the *user-provided* training data and then reported its accuracy on the *user-provided* testing data. All the baselines and existing tools were run using the same train-test split of data in each trial to ensure a fair comparison. We use standard *macro* F1 scores and R^2 scores for classification and regression tasks respectively, and used the mean score of 10 runs to compare the results, following existing literature [10, 15]. We ran all tools on 4 vCPUs of Xeon E5-2697A v4 (2.60GHz) with 16GB memory for OpenML, PMLB, and Mulan datasets and on 8 vCPUs with 32GB memory for Kaggle datasets.

5.2 RQ1: SapientML versus state of the art

We compared the performance of SapientML to three state-of-the-art AutoML systems: auto-sklearn [15] (ver. 0.12.2), TPOT [30] (ver. 0.11.7), and AL [10], from its public distribution [8] using the same configurations as in [10]. auto-sklearn is an actively managed open-source project on Github with more than 6K stars. AL represents the most recent AutoML technique that also learns from human-written pipelines to generate supervised pipelines. In addition, we implemented two baseline tools Basic-ML and Default, representing basic ML techniques, following the methodology described in [10]. Specifically, Basic-ML applies `SimpleImputer` to fill numeric and string missing values with 0 and empty string respectively, `CountVectorizer` to transform all string columns to token counts, and then uses the `LogisticRegression` and `LinearRegression` models for classification and regression tasks respectively. Default always predicts the most frequent label for classification tasks or the mean value for regression tasks.

5.2.1 Quantitative Comparison. Table 3 presents the evaluation results in terms of average macro F1 and R^2 scores over 10 runs for classification and regression tasks respectively. Highest scores for each benchmark are marked as bold. We call them as *champion*. Furthermore, we performed a pair-wise Wilcoxon-signed-rank Test ($\alpha = 0.05$) to see whether the score difference between the champion and another tool for a benchmark is statistically significant across 10 trials. The underlined numbers represent the scores that are statistically similar to the champion. We call them as *winners*.

We start by comparing the two baseline tools: Basic-ML and Default to all other tools. As expected, Default's simplistic prediction performed the worst. Interestingly, Basic-ML is the champion on 4 datasets, since some of the datasets are simple and do not need any sophisticated pipelines. However, Basic-ML's overall performance is poor compared to any other AutoML tools, in terms of mean F1/R2 scores. Therefore, this result supports the *no free lunch* hypothesis [47] that no single pipeline is good for every dataset.

Comparing the performance of SapientML to other AutoML tools, Table 3 shows that SapientML outperforms the state-of-the-art AutoML tools in terms of successful pipeline generation, number of champions, and winners. SapientML generated a successful pipeline for each benchmark and trial, whereas AL, auto-sklearn, and TPOT failed on 9, 17, and 12 datasets respectively. There are several reasons for failures including not being able to handle various types of data, unexpected exceptions, applying FE components on inappropriate columns, or timeout.

Table 3: Effectiveness of SapientML compared to the state-of-the-art AutoML tools on the benchmark datasets. Bold numbers indicate the best scores; Underlined numbers are not statistically different from the best scores according to a Wilcoxon-signed-rank Test ($\alpha = 0.05$). ***Failed*** **means the tool has failed on all the 10 trials whereas** ***F[n]*** **means the tool has failed on** ***n*** **trials.**

Dataset	SapientML	AL	auto-sk.	TPOT	Basic ML	Default	Base. 1	Base. 2	Metric	Source	Rows	Cols
1049	0.78	0.73	**0.79**	0.78	0.63	0.47	0.78	0.64	F1	OpenML	1458	37
1120	0.87	0.83	**0.87**	0.86	0.75	0.39	F[5]	0.75	F1	OpenML	19020	10
1128	0.95	0.94	**0.97**	0.96	0.94	0.44	F[5]	0.94	F1	OpenML	1545	10935
179*	**0.79**	0.77	Failed	Failed	0.43	0.43	F[7]	0.79	F1	OpenML	48842	14
184	**0.77**	0.55	Failed	Failed	Failed	0.02	0.67	0.37	F1	OpenML	28056	6
293*	**0.96**	0.78	0.91	0.75	0.75	0.34	0.96	0.75	F1	OpenML	581012	54
38	**0.95**	0.94	Failed	Failed	0.57	0.48	0.87	0.83	F1	OpenML	3772	29
389	**0.83**	0.63	0.83	0.79	0.81	0.02	0.83	0.81	F1	OpenML	2463	2000
46	0.95	0.95	Failed	Failed	Failed	0.23	**0.96**	0.94	F1	OpenML	3190	60
554	0.98	0.91	**0.98**	0.83	0.92	0.02	0.98	0.92	F1	OpenML	70000	784
772	0.51	0.51	**0.51**	0.51	0.41	0.35	0.51	0.49	F1	OpenML	2178	3
917	0.90	0.90	0.90	**0.90**	0.65	0.35	0.90	0.65	F1	OpenML	1000	25
Hill_Valley_with_noise	0.95	0.73	**1.00**	0.99	0.95	0.33	0.95	0.95	F1	PMLB	1212	100
Hill_Valley_without_noise	0.99	0.69	**1.00**	1.00	1.00	0.32	1.00	1.00	F1	PMLB	1212	100
breast_cancer_wisconsin	**0.97**	0.95	0.97	0.96	0.93	0.38	0.97	0.93	F1	PMLB	569	30
car_evaluation	0.95	0.97	0.98	**0.99**	0.76	0.21	0.95	0.79	F1	PMLB	1728	21
glass	**0.74**	0.67	0.63	0.70	0.48	0.10	0.71	0.45	F1	PMLB	205	9
ionosphere	**0.94**	0.91	0.94	0.94	0.84	0.39	0.94	0.86	F1	PMLB	351	34
spambase	0.96	0.94	0.95	0.95	0.92	0.38	**0.96**	0.94	F1	PMLB	4601	57
wine_quality_red	**0.35**	0.33	0.33	0.34	0.22	0.10	0.34	0.29	F1	PMLB	1599	11
wine_quality_white	**0.44**	0.42	0.41	0.43	0.16	0.10	0.44	0.34	F1	PMLB	4898	11
detecting-insults-in-social-comm...	0.71	0.76	Failed	Failed	**0.77**	0.42	Failed	0.42	F1	Kaggle	3947	2
housing-prices	**0.89**	0.85	Failed	Failed	0.80	-0.00	Failed	F[6]	R2	Kaggle	1460	80
mercedes-benz	0.52	**0.53**	Failed	Failed	-2.0E+23	-0.00	-0.80	Failed	R2	Kaggle	4209	377
sentiment-analysis-on-movie-rev...*	**0.49**	0.39	Failed	Failed	F[7]	0.13	Failed	0.02	F1	Kaggle	156060	3
spooky-author-identification	0.78	0.80	Failed	Failed	**0.81**	0.19	0.78	0.19	F1	Kaggle	19579	2
titanic	0.79	0.71	Failed	Failed	**0.81**	0.38	Failed	0.79	F1	Kaggle	891	11
enb	0.98	0.98	**0.99**	Failed	0.89	-0.01	0.98	0.96	R2	Mulan	768	8
jura	0.60	**0.76**	0.48	Failed	0.52	-0.01	0.59	0.60	R2	Mulan	359	15
sf1	-0.09	F[4]	Failed	Failed	Failed	**-0.01**	-0.10	-0.05	R2	Mulan	323	10
sf2	**0.05**	F[3]	Failed	Failed	Failed	-0.00	-1.0E+23	-4.6E+22	R2	Mulan	1066	10
costa-rica*	0.91	0.87	Failed	Failed	0.21	0.19	**0.92**	0.55	F1	Kaggle	9557	142
Categorical-Feature-Enc...-Chal...-II*	**0.58**	0.54	Failed	Failed	Failed	0.45	Failed	Failed	F1	Kaggle	600000	24
Porto-Seguros-Safe-Driver-Pred...	0.49	F[2]	**0.52**	0.52	0.49	0.49	0.49	0.49	F1	Kaggle	595212	58
Kobe-Bryant-Shot-Selection*	**0.64**	Failed	Failed	Failed	0.36	0.36	Failed	0.62	F1	Kaggle	30697	24
whats-cooking	0.71	Failed	Failed	Failed	**0.71**	0.02	0.71	0.02	F1	Kaggle	39774	2
PUBG-Finish-Placement-Prediction*	**0.93**	Failed	-0.00	0.86	0.83	-0.00	0.93	0.83	R2	Kaggle	4446966	28
Santander-Value-Prediction-Chal...	0.12	Failed	0.26	**0.27**	-7.2E+17	-0.00	0.12	Failed	R2	Kaggle	4459	4993
IEEE-CIS-Fraud-Detection*	**0.82**	Failed	Failed	Failed	0.49	0.49	Failed	Failed	F1	Kaggle	590540	394
Quora-Insincere-Questions-Class...*	**0.75**	0.63	Failed	Failed	Failed	0.48	Failed	0.06	F1	Kaggle	1306122	3
DonorsChoose.org-App...-Screening*	**0.50**	F[5]	Failed	Failed	Failed	0.46	Failed	Failed	F1	Kaggle	182080	16
#champions	19	2	9	3	4	1	3	0				
#winners	27	9	17	12	5	1	14	3				
#failures	0	9	19	21	8	0	12	6				

In terms of performance score, SapientML is champion for 18 subjects, whereas the second best tool, AL, based on the number of successful pipelines, is champion for only 2 datasets. On the other hand, although auto-sklearn failed on highest number of datasets, it is champion for 9 datasets. These results indicate that auto-sklearn performs well in a limited scope. However, although AL has a broader scope, it has overall performed moderately. Interestingly, SapientML outperforms them both in terms of scope and performance. The same findings also hold in terms of number of winners. SapientML performed the best or comparable to the best for 27 datasets, which is the highest among all tools.

For the 10 more difficult datasets (marked with a * in Table 3) – the largest (*row* × *columns*) datasets requiring at least one FE component – SapientML performs even better relative to other tools. SapientML produces best or comparable performance on 9 of them, with AL failing to produce a pipeline on 4 of them and TPOT, auto-sklearn on most of them. These results illustrate the value of SapientML's divide-and-conquer synthesis to produce viable pipelines especially for large, complex datasets.

5.2.2 Qualitative Analysis. We analyze the results qualitatively using a few concrete examples. Benchmark `OpenML-293` presents an interesting case where every tool produced a pipeline since the dataset contains only numeric values. AL predicted and selected XGBoostClassifier through dynamic evaluation, which achieved a macro mean F1 score of 0.78. AL's prediction may suffer since it uses language model which depends on the previous two components for prediction. However, there is no need for FE components for this dataset. auto-sklearn selected a pipeline based on dataset similarity that performs standard scaling first and then uses `GradientBoostingClassifier`. It achieved an F1 score of 0.91, better than AL. However, SapientML predicted an even better model: `RandomForestClassifier`, which achieved the best F1 score: 0.96.

For the *sentiment-analysis-on-movie-reviews* dataset, auto-sklearn simply failed since it cannot handle textual data. In contrast, AL and SapientML both successfully generated pipelines by using a `TextVectorizer` component to convert text to numeric columns. However, AL selected the `LinearSVC` model that resulted in an F1 score of 0.39. On the other hand, SapientML selected an additional

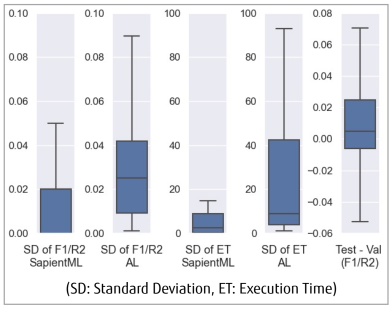

Figure 4: Robustness of SapientML

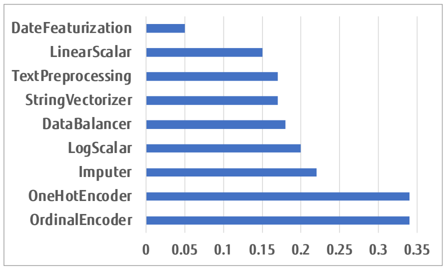

Figure 5: Distribution of FE predictions

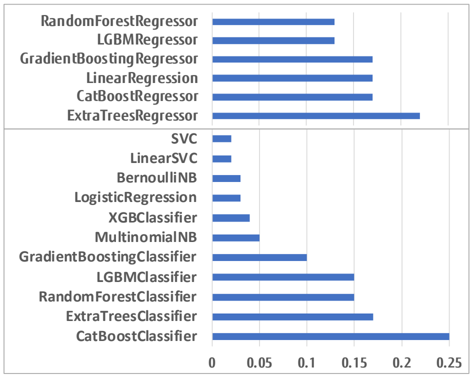

Figure 6: Distribution of model predictions

text preprocessing components that performs some basic cleaning and normalization of text. Further, it selected a better model `CatBoost` that provided an F1 score of 0.49.

Finally, the `jura` dataset presents a negative example for SapientML where AL achieved a significantly better score. On investigating the reason, we found that AL selected a model called `Ridge`, which is not used in our project corpus. Even for AL, this particular model was not highly ranked by its meta-model. However, since AL set a beam_size of 30 for this dataset, i.e., it evaluated 30 different models to select the best model, AL was successful in this case. AL could afford to perform such extensive evaluation for this dataset simply because the dataset is small and hence the dynamic evaluation is fast. However, for this extensive dynamic evaluation, AL failed to produce any pipeline for large datasets such as `IEEE` and `Donors` due to timeout. In contrast, SapientML uses only top 3 models based on its meta-model and performs overall the best.

5.3 RQ2: Robustness of SapientML

We analyze the robustness of SapientML in generating pipelines in terms of variation of performance scores and execution time across 10 trials. To investigate how much SapientML fluctuates across trials, we calculated the standard deviation of performance scores and execution time across 10 trials for each benchmark dataset. Figure 4 presents the distribution of the standard deviations across 10 trials for 41 benchmark datasets. The results show that SapientML is overall very stable across the 10 trials in terms of both accuracy and execution time. Both the 50th percentile (mean) and 75th percentile standard deviation for macro F1/R2 scores across all the benchmark are only 0.02, which is more stable than that of AL, which are 0.03 and 0.04 respectively. The same is true for the execution time. The 50th percentile and 75th percentile standard deviation of execution time are only 3 and 9 seconds respectively for SapientML, which are 9 and 41 seconds respectively for AL.

Finally, we investigate whether the generated pipelines are overfitted to the corresponding training data. To prevent overfitting, we already made sure that SapientML generates pipeline only using 75% data, and the generated pipeline is tested on 25% *held-out* test data. However, in this RQ, we investigate even further. Generally overfitting happens when a model performs very good on the validation data but performs poorly on the test data [39]. To this end, we compute the internal validation score based on which SapientML selected the best pipeline. Then we compare the validation accuracy with held-out test accuracy. As the fifth boxplot in Figure 4 shows, the 50th and 75th percentile difference between test and validation accuracy are 0.01 and 0.02 respectively. And interestingly, the differences are positive, which means that the final test scores are better than the validation scores for most of the subjects. We could not compare this result with any other tools since we do not have access to their validation scores.

5.4 RQ3: Diversity in Meta-Prediction

Figure 5 presents the distribution of predicted ML components in pipelines skeletons for all benchmarks and trials. The results show that the skeleton predictor was able to predict all the 9 FE components successfully. Among these components, accurate prediction of `Imputer`, `OrdinalEncoder` or `OneHotEncoder`, `TextVectorizer`, and `DateFeaturization` is important since any false negative predictions may lead to a crash during pipeline execution. Since SapientML is able to produce a successful pipeline for each trial in each benchmark dataset (Table 3), it is evident that the skeleton predictor accurately predicted these components. Similarly, as Figure 6 shows, the skeleton predictor is able to predict a wide range of model components. More specifically, it predicted 11 different classification and 6 regression models for 33 classification and 8 regression tasks respectively. As expected, some models such as `CatBoost` and `RandomForest` are dominant since they are fundamentally better. However, traditional models such as `LogisticRegression` or `SVC` are also predicted depending on the dataset. The overall results suggest that the predictions were effective for most of the datasets.

5.5 RQ4: Impact of novel components

This research question investigates the contribution of SapientML's two main components: i) pipeline seeding and ii) pipeline instantiation. To this end, we create two baselines:

Baseline1. This baseline uses the skeleton predicted by pipeline seeding but instantites each FE component on the entire dataset.

Baseline2. In this baseline, we further relax Baseline1 by replacing the pipeline seeding by a common skeleton to understand the combined effect of pipeline seeding and instantiation. To create the default skeletons, we take three most frequently used FE components in our corpus, one at a time, with the most frequent model. Thus, we try with three skeletons and take the best accuracy.

Table 3 shows the results of Baseline1 and Baseline2 (columns 8 and 9). From the results, it can be seen that the two baselines fail on 6 and 12 datasets, respectively. Further, their performance is poor due to the use of FE components on all columns in the dataset. Baseline1 achieves comparable performance to SapientML for datasets that are simple and do not require any FE components. However, the overall results show that both pipeline seeding and pipeline instantiation are important for SapientML to succeed.

6 LIMITATIONS & THREATS TO VALIDITY

External validity. Our framework has only been instantiated for ML pipelines in Python and evaluated on our 41 benchmark datasets. Thus our results may not hold outside this scope. We tried to mitigate this risk by using standard benchmarks from previous work [10, 15, 30] augmented with large, diverse, real-world datasets used on public data science competitions hosted on Kaggle.

Quality of data. Being a data-driven technique SapientML's performance is inherently limited by the quality of its training data. It is a well known problem that most notebooks available on Kaggle or GitHub cannot be locally re-executed [45, 46]. Thus, we could also mine only a fraction of the data (i.e., pipelines) potentially available on Kaggle. Further, the notebooks we did obtain vary significantly in quality and their use of specific libraries versus custom code. These differences manifest as noise in our analysis. We tried to mitigate these issues by developing simple but effective corpus augmentation (Section 4.2.2), pipeline denoising (Section 4.2.1) and by using semantic components classes (Section 4.3) to canonicalize pipelines. However, using a larger, cleaner data corpus could significantly strengthen our results.

Simple skeleton predictor model. Currently, our skeleton predictor uses a rather simple model that prioritizes features of the dataset and ignores correlations between (predicted) pipeline components. This approximation allows the model to perform well with limited data, as it did on our benchmarks. However, generating much more deeper or sophisticated pipelines might necessitate a more expressive model trained on substantially larger, cleaner data.

Manual definition of the pipeline space. Currently, we use a manual methodology to define the synthesis space of SapientML, including creating the clusters of APIs constituting the semantic FE classes (Section 4.3). We note that this is consistent with the practice of previous AutoML techniques [10, 15, 30]. However, we follow a transparent and systematic process (Section 4.3), so that SapientML can be easily generalized to other ML components once viable pipeline data demonstrating their use is available. However, this would still be limited to API-based ML components. The problem of mining and re-using arbitrary, custom ML transforms in pipeline synthesis remains a very interesting, open problem.

Hyper-parameter optimization (HPO). SapientML focuses on ML component selection and end-to-end pipeline instantiation. HPO is currently out of its scope. However, standard Bayesian optimization HPO [2] could be added as a post-processing step.

7 RELATED WORK

AutoML for tabular data. Previous AutoML techniques use different techniques to explore the huge combinatorial search space of potential candidate pipelines. TPOT [30] uses evolutionary search while ReinBo [40] uses Reinforcement Learning combined with Bayesian Optimization [18]. Auto-WEKA [43], and later Auto-Sklearn [13, 15], employ meta-learning on a corpus of synthetic optimized pipelines to *select* the most appropriate pipeline and then tune hyper-parameters using Bayesian Optimization. TensorOBOE [49, 50] builds on this approach using low rank tensor decomposition as a surrogate model for efficient pipeline search. AL [10] uses language models learned from human-written pipelines, in combination with aggressive dynamic evaluation of partial pipelines, to explore the pipeline space. AMS [7] mines constraints from corpora of human-written pipelines to help warm-start search-based AutoML like TPOT. SapientML shares AL and AMS's goal of learning from human-written pipelines. However, unlike all of the above approaches, which essentially reason on complete pipelines, SapientML combats AutoML combinatorial state space explosion through a novel divide-and-conquer approach of first reasoning on individual ML components and subsequently assembling a small pool of candidate pipelines for final analysis.

AutoML for DL models. This area is reviewed extensively in [16, 51]. This research focuses on synthesizing the neural network models themselves, through *neural architecture search (NAS)* [4, 19, 52], or on hyper-parameter optimization (HPO) [14, 19]. By contrast, SapientML addresses ML component selection and end-to-end pipeline instantiation, treating ML components as black-boxes.

Program synthesis for data wrangling. These techniques typically use input-output examples of data-frames as an input specification to synthesize programs implementing data wrangling operations (data pre-processing, cleaning, transformation) for the given dataset. They prune or navigate the synthesis program space by manually specified API constraints coupled with constraint-solving [12], automatically learning lemmas during synthesis [11], or using more general neural-network-backed program generators [6]. However, the PbE paradigm common to these techniques is not applicable to ML pipeline synthesis.

ML-based program synthesis. One class of approaches, such as [25, 33], use probabilistic models trained on programs extracted from large open repositories (e.g., Github and StackOverflow) to rank the space of candidate programs generated by the synthesizer. Another body of work [29, 37, 38, 41] leverages user-provided input-output examples, or natural language description, to create a search space for neural program synthesis, typically for simple domains such as string-manipulating programs. By contrast, our synthesis technique is specifically engineered to use a given dataset and its predictive task as the (only) specification for synthesis.

8 CONCLUSIONS

In this work we proposed a learning-based AutoML technique SapientML, to generate supervised ML pipelines for tabular data. SapientML combats the huge combinatorial search space of AutoML through a novel divide-and-conquer three-stage program synthesis approach that reasons on successively smaller search spaces. We have instantiated SapientML as part of a fully automated tool-chain that creates a cleaned, labeled learning corpus by mining Kaggle, learns from it, and uses the learned models to then synthesize pipelines for new predictive tasks. We evaluated SapientML on a set of 41 benchmark datasets and against 3 state-of-the-art AutoML tools and 4 baselines. Our evaluation showed that SapientML produced the best or comparable accuracy in 27 of the benchmarks while the second best tool failed to even produce a pipeline on 9 of the instances.

REFERENCES

[1] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. *ACM SIGPlan Notices* 25, 6 (1990), 246–256.

[2] AutoML.org. 2019. *SMAC v3 Project*. https://github.com/automl/SMAC3 Accessed in 2021.

[3] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360585

[4] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. 2017. Designing Neural Network Architectures using Reinforcement Learning. arXiv:1611.02167 [cs.LG]

[5] SV Barai and Yoram Reich. 1999. Ensemble modelling or selecting the best model: Many could be better than one. *AI EDAM* 13, 5 (1999), 377–386.

[6] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: Neural-Backed Generators for Program Synthesis. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 168 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360594

[7] José Cambronero, Jürgen Cito, and Martin Rinard. 2020. AMS: Generating AutoML search spaces from weak specifications. In *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Sacramento, California) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA.

[8] José P. Cambronero. 2019. *Public distribution of AL.* https://github.com/josepablocam/AL-public Accessed: August, 2020.

[9] José P. Cambronero. 2020. *python-pl*. https://github.com/josepablocam/python-pl Accessed: January 2021.

[10] José P. Cambronero and Martin C. Rinard. 2019. AL: Autogenerating Supervised Learning Programs. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 175 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360601

[11] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-Driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 420–435.

[12] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 422–436.

[13] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. 2020. Auto-Sklearn 2.0: The Next Generation. *arXiv:2007.04074v1 [cs.LG]* (2020).

[14] Matthias Feurer and Frank Hutter. 2018. *Hyperparameter Optimization.* https://www.ml4aad.org/wp-content/uploads/2018/09/chapter1-hpo.pdf

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

[16] Xin He, Kaiyong Zhao, and Xiaowen Chu. 2019. AutoML: A Survey of the State-of-the-Art. arXiv:1908.00709 [cs.LG]

[17] Qinheping Hu and Loris D'Antoni. 2018. Syntax-Guided Synthesis with Quantitative Syntactic Objectives. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 386–403.

[18] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization* (Rome, Italy) *(LION'05)*. Springer-Verlag, Berlin, Heidelberg, 507–523.

[19] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.). 2018. *Automated Machine Learning: Methods, Systems, Challenges.* Springer. In press, available at http://automl.org/book.

[20] Instagram. 2019. *LibCST:A Concrete Syntax Tree (CST) parser and serializer library for Python.* https://github.com/Instagram/LibCST Accessed in 2021.

[21] Kaggle. 2010. *Kaggle.* https://www.kaggle.com Accessed in 2021.

[22] Kaggle. 2010. *Kaggle Public API.* https://www.kaggle.com/docs/api Accessed: January, 2021.

[23] Kaggle. 2010. *Meta Kaggle.* https://www.kaggle.com/kaggle/meta-kaggle Accessed: January, 2021.

[24] Udayan Khurana, Horst Samulowitz, and Deepak Turaga. 2018. Feature engineering for predictive modeling using reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[25] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating Search-Based Program Synthesis Using Learned Probabilistic Models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 436–449.

[26] LinkedIn. 2018. *LinkedIn Workforce Report.* https://economicgraph.linkedin.com/resources/linkedin-workforce-report-august-2018

[27] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code Recommendation via Structural Code Search. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 152 (Oct. 2019), 28 pages.

[28] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair. 101–114.

[29] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. 2018. Neural Sketch Learning for Conditional Program Generation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=HkfXMz-Ab

[30] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. 2016. Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016* (Denver, Colorado, USA) *(GECCO '16)*. ACM, New York, NY, USA, 485–492. https://doi.org/10.1145/2908812.2908918

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[32] QuantHub. 2020. *The Data Scientist Shortage in 2020.* https://quanthub.com/data-scientist-shortage-2020/

[33] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 419–428.

[34] Scikit-Learn. 2007. *scikit-learn: Machine Learning in Python.* https://scikit-learn.org Accessed in 2021.

[35] scikit-learn 1.0.2. 2022. *Decision Trees.* https://scikit-learn.org/stable/modules/tree.html

[36] Connor Shorten and Taghi M Khoshgoftaar. 2019. A survey on image data augmentation for deep learning. *Journal of Big Data* 6, 1 (2019), 1–48.

[37] Chengxun Shu and Hongyu Zhang. 2017. Neural Programming by Example. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 1539–1545.

[38] Rishabh Singh and Pushmeet Kohli. 2017. AP: Artificial Programming. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)* (2nd summit on advances in programming languages (snapl 2017) ed.).

[39] Jyothi Subramanian and Richard Simon. 2013. Overfitting in prediction models–is it a problem only in high dimensions? *Contemporary clinical trials* 36, 2 (2013), 636–641.

[40] Xudong Sun, Jiali Lin, and Bernd Bischl. 2019. ReinBo: Machine Learning pipeline search and configuration with Bayesian Optimization embedded Reinforcement Learning. *arxiv preprint, https://arxiv.org/abs/1904.05381* 1904.05381 (2019).

[41] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A Grammar-Based Structural CNN Decoder for Code Generation. *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (07 2019), 7055–7062.

[42] Jafar Tanha, Yousef Abdi, Negin Samadi, Nazila Razzaghi, and Mohammad Asadpour. 2020. Boosting methods for multi-class imbalanced data classification: an experimental review. *Journal of Big Data* 7, 1 (2020), 1–47.

[43] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Chicago, Illinois, USA) *(KDD '13)*. Association for Computing Machinery, New York, NY, USA, 847–855.

[44] Vesta. [n.d.]. *IEEE-CIS Fraud Detection.* https://www.kaggle.com/c/ieee-fraud-detection Accessed: February, 2021.

[45] Jiawei Wang, Li Li, Kui Liu, and Haipeng Cai. 2020. Exploring How Deprecated Python Library APIs Are (Not) Handled. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 233–244.

[46] Jiawei Wang, Tzu yang Kuo, Li Li, and Andreas Zeller. 2020. Assessing and Restoring Reproducibility of Jupyter Notebooks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '20)*.

[47] David H Wolpert and William G Macready. 1997. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation* 1, 1 (1997), 67–82.

[48] XGBoost. 2016. *XGBoost: eXtreme Gradient Boosting.* https://github.com/dmlc/xgboost Accessed in 2021.

[49] Chengrun Yang, Yuji Akimoto, Dae Won Kim, and Madeleine Udell. 2019. OBOE: Collaborative Filtering for AutoML Model Selection. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, 1173–1183.

[50] Chengrun Yang, Jicong Fan, Ziyang Wu, and Madeleine Udell. 2020. AutoML Pipeline Selection: Efficiently Navigating the Combinatorial Space. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '20)*. Association for Computing Machinery, 1446–1456.

[51] Quanming Yao, Mengshuo Wang, Yuqiang Chen, Wenyuan Dai, Yu-Feng Li, Wei-Wei Tu, Qiang Yang, and Yang Yu. 2018. Taking Human out of Learning Applications: A Survey on Automated Machine Learning. arXiv:1810.13306 [cs.AI]

[52] Barret Zoph and Quoc V. Le. 2017. Neural Architecture Search with Reinforcement Learning. arXiv:1611.01578 [cs.LG]