AMS: Generating AutoML Search Spaces from Weak Specifications

José P. Cambronero
Massachusetts Institute of Technology
Cambridge, U.S.A.
jcamsan@mit.edu

Jürgen Cito
TU Wien
Vienna, Austria
Massachusetts Institute of Technology
Cambridge, U.S.A
juergen.cito@tuwien.ac.at

Martin C. Rinard
Massachusetts Institute of Technology
Cambridge, U.S.A.
rinard@csail.mit.edu

ABSTRACT

We consider a usage model for automated machine learning (AutoML) in which users can influence the generated pipeline by providing a *weak pipeline specification*: an unordered set of API components from which the AutoML system draws the components it places into the generated pipeline. Such specifications allow users to express preferences over the components that appear in the pipeline, for example a desire for interpretable components to appear in the pipeline. We present AMS, an approach to automatically strengthen weak specifications to include unspecified complementary and functionally related API components, populate the space of hyperparameters and their values, and pair this configuration with a search procedure to produce a *strong pipeline specification*: a full description of the search space for candidate pipelines. AMS uses normalized pointwise mutual information on a code corpus to identify complementary components, BM25 as a lexical similarity score over the target API's documentation to identify functionally related components, and frequency distributions in the code corpus to extract key hyperparameters and values. We show that strengthened specifications can produce pipelines that outperform the pipelines generated from the initial weak specification and an expert-annotated variant, while producing pipelines that still reflect the user preferences captured in the original weak specification.

CCS CONCEPTS

• **Software and its engineering** → **Search-based software engineering**; **Automatic programming**; • **Theory of computation** → Program specifications.

KEYWORDS

automated machine learning, program mining, search-based software engineering

ACM Reference Format:
José P. Cambronero, Jürgen Cito, and Martin C. Rinard. 2020. AMS: Generating AutoML Search Spaces from Weak Specifications. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409700

1 INTRODUCTION

Automated machine learning (AutoML) [15, 23, 27, 37] promises to democratize the use of machine learning techniques by end users, allowing non-experts access to a tool that has become standard for tackling research and applications across domains as diverse as medicine, finance, and software engineering [17, 29, 34, 36]. AutoML tools typically take as input a tabular dataset along with a classification or regression task (i.e. predict a particular column) and generate an optimized composition of machine learning operators, drawn from a target API, to produce an executable pipeline.

At the core of AutoML tools lie search procedures that generate and evaluate possible pipeline candidates. Under the prevailing usage model [48], the end user treats the AutoML tool as a black box which produces a pipeline that outperforms other generated candidates based on some predictive performance metric (e.g. F1 score). In this setting the end user has no direct way of influencing the pipeline chosen by the system. However, a user may need to express preferences, beyond maximizing a predictive performance metric, to satisfy constraints such as pipeline interpretability, domain-specific best practices, and data scaling constraints, for example.

We propose the use of a *weak pipeline specification* as a way to provide partial user preferences to the AutoML tool. A weak pipeline specification consists of an unordered set of API components that the end user may want to appear in the resulting pipeline. This specification can be automatically extended to produce a *strong pipeline specification* that captures additional API components of interest, defines a set of hyperparameters and values to search over, and a search procedure to sample candidate pipelines. The strengthened pipeline specification can then influence the output pipeline produced by the AutoML tool by constraining the search space.

For example, the user might provide the Scikit-Learn component { LogisticRegression } as a weak specification. Strengthening this specification could add other linear models (e.g. linear SVM), would specify different types of regularization (e.g. L1/L2) and their weights, and would include the search procedure (e.g. genetic programming) used to sample pipelines. This proposed model of interaction allows the end user greater control over the eventual output pipeline, without negating the key advantage of AutoML: the user need not be an ML expert.

We introduce AMS (Figure 1), a system that automatically strengthens AutoML search space specifications. To carry out this strengthening, AMS exploits information in a code corpus and the target API's documentation. First, AMS automatically mines pairs of complementary API components from the selected code corpus, where two components are complementary if they co-occur frequently. To formalize this mining procedure, AMS uses normalized pointwise

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7043-1/20/11.
https://doi.org/10.1145/3368089.3409700

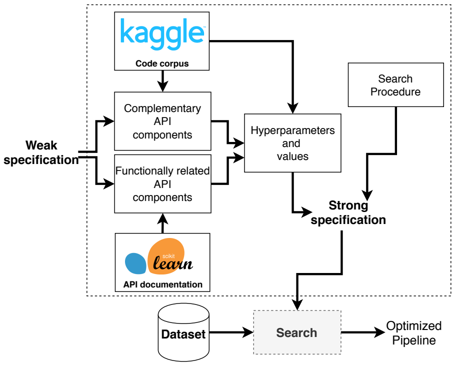


Figure 1: AMS system diagram. System boundaries are depicted as a dashed line. The user provides a weak specification, which is automatically extended by AMS to include complementary, functionally related API components, and key hyperparameters and a set of possible values.

mutual information [5] to rigorously characterize co-occurrence in probabilistic terms. These mined associations can then be used to extend the initial weak specification. Next, AMS identifies unspecified API components that may be functionally related to those in the original specification. To reason about component similarity, AMS applies BM25 [42], a popular and effective measure of lexical similarity, over the API's documentation. With this metric, AMS can identify components with the highest degree of relation to those in the original specification. Next, given that machine learning pipelines are known to exhibit different performance based on hyperparameter values [40], AMS uses frequency distributions, estimated from the selected code corpus, to define a hyperparameter search space for each component in the extended specification. Finally, AMS pairs this component configuration with a search procedure, which is used to sample candidate pipelines from the given space.

We empirically evaluate AMS's predictive performance, in terms of macro-averaged F1 score, over 9 datasets and 15 weak pipeline specifications. Our results show that, with two different search procedures, AMS produces pipelines that outperform the pipelines obtained using the initial weak specification and an expert-annotated version of the weak specification including hyperparameters and values. To quantify the extent of outperformance, we use the concept of a *win*. A pipeline wins when it obtains the highest score on a specification/dataset combination, and satisfies a minimum predictive score difference to rule out comparable scores.

When using genetic programming as a search procedure, AMS's specifications result in 38 wins compared to 12 under the weak specification extended with an expert hyperparameter space. When using random search as a search procedure, AMS's specifications result in 41 wins compared to 14 wins under the weak specification extended with an expert hyperparameter space. We also find that the pipelines produced using AMS's specification qualitatively reflect the influence of the weak specification.

To summarize, this paper makes the following contributions:

- We present a novel approach to automatically strengthen pipeline specifications for AutoML tools, allowing users to influence the final generated pipeline. Our approach relies on a probabilistic characterization of co-occurring API components, lexical similarity over alternative components' API documentation, and frequency distributions for hyperparameter spaces.
- We implement this approach (and share our evaluation dataset) in an open-source system called AMS[1].
- We evaluate AMS using 9 datasets from an existing AutoML paper [37], 15 weak pipeline specifications, and two different search procedures. Our results show that AMS's strengthened specifications produce higher performing pipelines. When using genetic programming, AMS specifications produce 38 wins compared to 12 from an expert-annotated variant of the weak specification, and 9 from the original weak specification. We see a similar number of wins when comparing approaches using random search.
- We qualitatively show that the distribution of components in the output pipelines produced using AMS specifications reflect the influence of the initial weak specification.

In the following sections we review the background on AutoML (Section 2), introduce the notion of pipeline specifications (Section 3), provide an illustrative scenario for the use of AMS (Section 4), detail the approach and design of AMS (Section 5), present experimental results (Section 6), provide context on related work (Section 7), outline possible threats (Section 8) and conclude (Section 9).

2 AUTOML BACKGROUND

We first formally introduce AutoML for classification [27]. Let $d \in \mathcal{D} : \mathbb{R}^{n \times m} \times \mathbb{N}^n$ be a dataset comprised of a matrix of n observations, each with m real covariates, and a vector of n natural number labels. Let $\mathcal{H} : \mathbb{R}^{n \times m} \rightarrow \mathbb{N}^n$ be the type of a pipeline program defined as a composition of preprocessing and learning algorithms – implemented as API components in a target library – along with their corresponding hyperparameter settings. A pipeline takes a dataset and predicts labels based on covariates. Let $\mathcal{S}$ be the search space of all possible pipeline programs. Let $e \in \mathcal{E} : \mathcal{H} \times \mathcal{D} \rightarrow \mathbb{R}$ be an evaluation function that scores the ability of a pipeline to successfully predict labels and generalize to unseen observations (e.g. cross-validated F1 score). Let $\text{cost} \in C : \mathcal{H} \times \mathcal{D} \rightarrow \mathbb{R}$ be a cost function that evaluates pipeline execution time on a dataset d, and $b \in \mathbb{R}$ be a search time budget. Then AutoML corresponds to the optimization problem

$$\underset{h \in \mathcal{S}}{\text{argmax}}\ e(h,d)\ s.t. \sum_{h \in \mathcal{S}} \text{cost}(h,d) \leq b$$

Given that the possible space of pipelines $\mathcal{S}$ is exponentially large, it is impractical to evaluate every pipeline in $\mathcal{S}$ within the given budget b. Effective AutoML systems will therefore typically have to perform a search over the space, evaluating only a subset of pipelines. The AutoML system iteratively searches and evaluates pipelines in $\mathcal{S}$, keeping track of its estimate of the best pipeline.

Existing AutoML systems employ a variety of search strategies to identify candidate pipelines. These strategies include genetic programming [37], Bayesian optimization [15], reinforcement learning [11], program-analysis-based search [7], and random search [18].

[1] https://github.com/josepablocam/ams

To improve the effectiveness of these strategies, systems may also incorporate prior knowledge about pipeline performance or impose additional structure on the candidate pipelines that can be generated. For example, an AutoML system may warm start the search by incorporating previously successful pipelines [15, 47], condition on textual dataset and algorithm descriptions [12], manually constrain the subset of algorithms available for pipeline definitions [37], or constrain the shape of possible pipelines [10, 11]. The latter two provide ways to restrict the type of pipelines produced, however, they require user involvement and expertise. In the following section, we present an approach that bridges this gap.

3 PIPELINE SPECIFICATIONS

The current usage model for AutoML typically emphasizes the lack of user involvement [48]. Under this model, the user presents the tool with their target dataset, for which they want to learn a classification pipeline, sets some computational budget, runs the tool, and accepts the pipeline produced by the AutoML tool. In this context, the AutoML tool receives no user feedback (beyond the input dataset), and the user is unable to influence the pipelines considered by the search procedure. Without any formal user feedback, the AutoML tool is unable to 1) exploit any user domain knowledge or 2) provide a pipeline that satisfies any desired user constraints (e.g. interpretability).

Most existing AutoML tools construct a pipeline by composing and configuring API components drawn from a higher-level ML library, such as Scikit-Learn [39].

Definition 3.1. API Component. An API component, in the context of our paper, refers to a public library function or class that can be composed with other components to create an ML pipeline. Most API components provide additional configuration options through the use of optional/default parameters.

We propose the use of *weak specifications* as a way for AutoML users to influence the pipelines produced by automatically subsetting the relevant set of API components, thus constraining the search space for candidates generated by the AutoML tool.

Definition 3.2. Weak Specification. A weak specification is an (unordered) set of API components, at least one of which is a regressor (if performing a regression task) or a classifier (if performing a classification task).

By providing a set of API components a user provides (partial) information regarding what they want: specifically a set of algorithms (e.g. classifiers, preprocessors) that should be considered for pipeline generation. We call this type of specification *weak* as it is incomplete along four key dimensions:

(1) it does not specify what hyperparameters are relevant
(2) it does not specify what values hyperparameters can take on
(3) other relevant API components may be missing
(4) it does not specify any order or compositional operators used to generate new pipelines from these components

Providing a weak specification allows a user to exert influence on the final pipeline produced, while at the same time not requiring deep API or machine learning expertise, as they do not have to manually detail the complete space. For example, a user can enforce a degree of interpretability on the optimized pipeline by writing a specification with a single linear model (e.g. logistic regression).

Definition 3.3. Strong Specification. Let h_c be a map from a subset of hyperparameters for component c to a collection of possible values. Let C be a map from component i to its respective h_i. Let P be a search procedure to generate candidate pipelines. A strong specification is a triple of the form $\langle C, (h_1, \ldots, h_n), P \rangle$.

A strong specification, in effect, defines a search space for an AutoML tool. We propose that this space can be derived from the weak specification, which expresses (partial) user preferences.

In the following sections, we detail our approach to automatically strengthening weak specifications to influence the AutoML search process. But first, we introduce an illustrative scenario to demonstrate a use case for AMS.

4 ILLUSTRATIVE SCENARIO

We follow the journey of a forensic scientist who is not a machine learning expert but wants to classify glass fragments [9, 13]. The forensic scientist has a high level understanding of different learning and preprocessing algorithms but is not aware of the various hyperparameters, possible values, or other suitable algorithms to consider. The scientist has heard of AutoML and thinks this might be a suitable tool to explore pipelines. However, they have clear constraints: no tree-based ensemble models, as the pipelines need to be easily interpretable. Unfortunately, AutoML tools are known to often produce tree-based ensemble models [14, 20], which are challenging to interpret [21].

They spent some time on the internet and found a related tutorial that detailed a Scikit-Learn [39] pipeline that may work for their use case (case 1 in Table 1).

The scientist will use this example pipeline (with no hyperparameters or values) as a weak specification. To evaluate their progress, they will use an existing classification dataset, "glass" [13], consisting of continuous measurements for 7 types of glass. The scientist performs a random 80/20 split for training/testing and evaluates pipelines using macro-averaged F1 score.

The scientist starts by naively running their specification directly as a pipeline with default hyperparameters, which results in an initial F1 score of *0.43*. Next the scientist uses the specification components, with default hyperparameters, as a configuration for the AutoML tool TPOT [37], which uses genetic programming to generate candidate pipelines. Applying TPOT to the weak specification (with no hyperparameters defined in the search space) results in a better score of *0.51*.

After consulting with a machine learning colleague, the scientist sets up a defined hyperparameter space (i.e. which hyperparameters to tune and set of possible values) for each component in the specification. The scientist then applies the same genetic programming search to the new configuration, resulting in pipeline number 3 in Table 1. Note that the shape of the optimized pipeline is the same as in the prior step, but now the regularization penalty and its weight varies. This step raised their score to 0.57.

The scientist now goes back to the original weak specification and uses AMS to automatically strengthen this weak specification (rather than manually specifying the full space). AMS extends the weak specification using a code corpus and the API's documentation. Applying the same search procedure to AMS's specification now results in the highest score of *0.75*. The final pipeline retains polynomial features, but replaces the variance threshold selector with a selector based on a specified false positive rate. The pipeline then stacks a SGDClassifier (with hinge loss) and uses logistic regression with an L1 penalty (to produce sparse coefficients). This embodies the spirit of the initially given specification, but substantially outperforms the rest of the approaches.

Table 1: Summary of scenario iterations based on the "glass" dataset showing the progression of score improvements. Note that component names are abbreviated for brevity.

#	Pipeline	Description	Score
1	PolyFeatures, MinMaxScaler, VarianceThreshold, LogisticRegression	Initial (naive) weak specification as a pipeline with default hyperparameters.	0.43
2	StackingEstimator(LogisticRegression), LogisticRegression	Applying AutoML tool TPOT (Genetic Programming) to the original specification without defining any hyperparameters.	0.51
3	StackEstimator(LogisticRegression ([Penalty: L1, Cost: 10]), LogisticRegression	Same as #2, but with expert-defined hyperparameter space for regularization (cost) and penalty.	0.57
4	PolyFeatures, SelectFPR, StackingEstimator(SGDClassifier [Loss: Hinge]), LogisticRegression [Penalty: L1, Cost: 100]	Applying genetic programming to the strong specification generated by our approach (AMS) given the weak specification.	0.75

5 AMS

We introduce AMS, a system that automatically strengthens weak pipeline specifications using an existing code corpus, an API's documentation, and a plug-in search procedure. Figure 1 shows a diagram of the system. AMS takes the user's weak specification as input. The system first extends the set of API components considered in the specification. To perform this extension, AMS relies on a code corpus, which exercises the target API, and on the API's natural language documentation. After the specification has been extended, AMS uses the code corpus to identify key hyperparameters for the API components in the specification and include sets of possible values they can take on. AMS then pairs this set of component configurations with a search procedure to produce a strong specification. The search procedure can then be used to iteratively sample and evaluate candidate pipelines, resulting in a final optimized pipeline.

We now present details on each step in the AMS system.

5.1 Unspecified (but Useful) API Components

AMS first extends the initial specification with additional components, which the user may not have included. Given a specification S and a new component c, c may be added to S if it satisfies one of the following two conditions: c is commonly used with a component already in S, or c could replace a component already in S.

The goal of the first condition is to identify *complementary components*. For example, if a classifier is often used with a particular preprocessing step, we say these components are complementary. The goal of the second condition is to identify *functionally related* components, which are alternatives to each other. For example, two different linear classifiers would be considered functionally related.

AMS relies on two different sources of information to identify components that satisfy each of these conditions. We first address complementary components.

5.1.1 Complementary Components. To identify complementary components, AMS exploits information from a crowd-sourced corpus of scripts, which exercise the target API. Each script in the corpus was written to target a single dataset, therefore two components used in the same script may be complementary. By using a code corpus to identify such components, AMS can automatically produce and update its inventory of complementary components to reflect current ML practices.

From the code corpus, the system extracts all scripts that contain a call to our target API library and records the set of API components used in each script. The intuition is that these sets can be used to measure the likelihood of components co-occurring, and that complementary components must (by definition) co-occur more frequently.

Formally, we compute the normalized pointwise mutual information (NPMI) [5] over the collection of all (unordered) pairs of co-occurring API calls in our code corpus to identify complementary components. Let X and Y be two random variables, representing possible components, defined over the the domain of our target API library. We define NPMI for two components $x \in X$ and $y \in Y$ as

$$\mathrm{NPMI}(x,y) = \frac{\log_2\left(\frac{p(x,y)}{p(x)p(y)}\right)}{-\log_2(p(x,y))} \tag{1}$$

where $p(x)$ is the fraction of pairs where either element is x divided by the number of all pairs, similarly for $p(y)$, and $p(x,y)$ is the fraction of pairs (x,y) or (y,x) divided by the number of all pairs.

NPMI ranges between -1 and 1, where -1 means the components never co-occur, 1 means the components always co-occur, and 0 means the components are independent. We compute the NPMI over the set of all pairs of co-occurring components (i.e. API components called in the same script). Eliminating pairs with an NPMI less than or equal to zero yields pairs of varying degree of complementarity.

When given a weak specification, we can identify all NPMI-positive pairs that share a component with the specification. For each such pair, the new potential component corresponds to the element in the pair that is not in the original specification. If more than one component in the original specification supports (i.e. co-occurs with) a new component, we compute an average NPMI. For each possible new component, we compute a weighted sum of the average NPMI and the fraction of original specification components that support it. The weighted sum balances average NPMI and support fraction based on a user-defined weight $\alpha \in [0,1]$. We then take the top K_{comp} new components and add them as complementary components to the original specification. Algorithm 1 describes this procedure.

5.1.2 Functionally Related Components. The goal of identifying functionally related components is to include algorithm alternatives in the specification. For example, the user's weak specification may indicate that they are interested in using linear models, but they may have not exhaustively listed all linear model alternatives. This task raises the challenge of reasoning about the semantics of API

Algorithm 1 Extracting Complementary Components

```
INPUT: A collection $P$ of pairs of API components co-occurring in a code
  corpus; a function NPMI that computes the normalized pointwise mutual
  information of two API components; a specification $S = \{c_1, \ldots, c_n\}$; a
  weight $\alpha \in (0, 1)$ to combine NPMI and support fraction; and an integer
  $K_{\text{comp}}$ for the maximum number of complementary components to take.
OUTPUT: A new specification $S'$ extended with at most $K_{\text{comp}}$ new components.
  procedure ComplementaryComponents
    ▷ Map from co-occurring pair to accumulator list of NPMI scores
    npmis ← {}
    for $c \in S, (p_1, p_2) \in P$ do
      if $(c \in (p_1, p_2)) \wedge (p_1 \notin S \vee p_2 \notin S)$ then
        new ← $p_2$ if $c_1 = p_1$ else $p_2$
        ▷ Accumulate the npmi score
        npmis[new] ← npmis[new] :: NPMI($p_1, p_2$)
    ▷ Compute average npmi and support fraction
    ▷ Combine using $\alpha$ to create score
    scores ← {}
    $n$ ← LEN($S$)
    for comp ∈ npmis do
      vals ← npmis[comp]
      scores[comp] ← $\text{Avg(vals)} * \alpha + \left(\frac{\text{Len(vals)}}{n}\right) * (1 - \alpha)$
    $S' \leftarrow S \cup \text{GetTopK}(\text{scores}, K_{\text{comp}})$
```

components. Rather than reason about component semantics, we rely on a simpler notion of similarity.

We would like to define a function $\text{SIM}(c_1, c_2)$ that computes a score for two API components, c_1 and c_2, such that a higher score corresponds to higher degree of semantic similarity. Given a component c_i, we can then sort all possible components in our target API in descending order based on their similarity score with respect to c_i.

AMS exploits the fact that the target library has natural language documentation for each component (as part of its developer documentation), which we assume details key aspects about their functional behavior. By mining the API's documentation, AMS can be used to automatically identify functionally related components in new target libraries or new versions of previously used libraries without the need for extensive expert annotation.

We define $\text{SIM}(c_1, c_2)$ to be computed over the documentation[2] for c_1 and c_2 and instantiate it to a classical relevance/similarity scoring technique: BM25 [42]. BM25, detailed below, produces a score for a document, given a query and a corpus of documents. A higher score indicates a higher degree of lexical correlation between the document and the query.

$$\text{BM25}(D,Q) = \sum_{i}^{n} \text{IDF}(C, q_i) \frac{f(q_i, D) * (k_1 + 1)}{f(q_i, D) + k_1 * \left(1 - b + b * \frac{\text{Len}(D)}{\text{AvgLen}(C)}\right)} \tag{2}$$

where D is a document, $Q = (q_1, \ldots, q_n)$ is a query comprised of q_i terms, C is a corpus of documents, and k_1 and b are score hyperparameters[3].

In our setting, the documentation for an existing component in the weak specification corresponds to the query, a particular API component's documentation corresponds to the document, and the entirety of the API's documentation corresponds to the document corpus.

AMS uses SIM to retrieve, and append, the top K_{rel} new components for each component in the original weak specification (i.e., we do not consider any complementary components added for purposes of this procedure). Algorithm 2 describes this procedure [4].

Algorithm 2 Extracting Functionally Related Components

```
INPUT: A collection $C$ of API components; a map $M$ from API component
  to documentation; a function SIM that computes the BM25 score between
  a query string and a document; a specification $S = \{c_1, \ldots, c_n\}$; and an
  integer $K_{\text{rel}}$ for the maximum number of functionally related components
  to take per component in $S$.
OUTPUT: A new specification $S'$ extended with at most $K_{\text{rel}}$ functionally
  related components per component in the original specification.
  procedure FunctionallyRelatedComponents
    ▷ Set of empty API components
    extension ← $\emptyset$
    for $c \in S$ do
      scored ← $\{(c', \text{SIM}(M[c], M[c'])) \text{ for } c' \in C \text{ if } c' \notin S\}$
      $c_K$ ← GetTopK(scored, $K_{\text{rel}}$)
      extension ← extension ∪ $c_k$
    $S' \leftarrow S \cup \text{extension}$
```

5.2 Identifying Hyperparameters and Values

Machine learning practitioners often spend a significant amount of time not just choosing pipeline components, but also tuning the hyperparameters associated with each component. Performance can significantly increase by identifying the appropriate hyperparameter values for a given dataset and pipeline [40].

AMS relies on the corpus of scripts that make calls to the target API to identify the set of relevant hyperparameters and possible values. This design choice hypothesizes that an AutoML system should focus on tuning the set of hyperparameters and hyperparameter values that human developers focus on tuning.

For each script in our code corpus that imports the target API, we parse the source code and identify calls to API class constructors. We extract the set of optional arguments in each constructor call and record each pair of (argument name, argument value) as a hyperparameter setting. The value recorded corresponds to a constant in the constructor call, or points to an unknown placeholder.

When given a specification, AMS takes each API component and identifies the set of top K_{params} hyperparameter names observed in the mined code for that component, along with the top K_{vals} values observed for each of the names. AMS adds the default value for each hyperparameter to the set of possible values (obtained by introspecting the class definition), and then emits this as the corresponding hyperparameter search space. Algorithm 3 describes this procedure.

Figure 2 shows a specification, originally just `sklearn.linear_model.LogisticRegression`, extended with a complementary component (Algorithm 1), a functionally related component (Algorithm 2), and hyperparameters and values (Algorithm 3).

[2] We perform standard preprocessing of the documentation strings such as tokenization, stemming, and extension with the path of the given component in the library's module structure.

[3] We use the gensim [41] BM25 implementation, where $k_1 = 1.5$ and $b = 0.75$ are implementation-defined constants.

[4] AMS also exposes functionality to label a weak specification component as "include" (by appending `:1`) or "exclude" (by appending `:0`), indicating that it must be included or excluded from the strengthened specification, respectively.

Algorithm 3 Adding API Component Hyperparameters and Values

```
INPUT: A map $P$ from API components to hyperparameter names and
    frequencies observed in calls; a map $V$ from hyperparameters to values
    and their frequencies observed in calls; a specification $S = \{c_1, \ldots, c_n\}$; an
    integer $K_{\text{params}}$ for the maximum number of hyperparameters to consider
    per component; and an integer $K_{\text{vals}}$ for the maximum number of values
    per hyperparameter to consider.
OUTPUT: A new specification $S'$ with at most $K_{\text{params}}$ hyperparameters
    per component and at most $K_{\text{vals}} + 1$ (including default value) per hyper-
    parameter.
  procedure HyperParamsAndValues
      ▷ Empty map from component to hyperparameter space
      $S' \leftarrow \{\}$
      for $c \in S$ do
          params ← GetTopK($P[c], K_{\text{params}}$)
          ▷ Empty configuration for component c
          $c_{\text{config}} \leftarrow \{\}$
          for $p \in$ params do
              values ← GetTopK($V[p], K_{\text{vals}}$)
              ▷ Append default value, if not included
              values ← values :: GetDefaultValue($p$)
              $c_{\text{config}}[p] \leftarrow$ values
          $S'[c] \leftarrow c_{\text{config}}$
```

```
{'sklearn.linear_model.LogisticRegression': {
  'C': [100000.0, 7, 1.0], 'penalty': ['l1', 'l2']},
'sklearn.feature_extraction.text.TfidfTransformer': {},
'sklearn.linear_model
    .SGDClassifier': {'loss': ['log', 'hinge'],
'penalty': ['l2', 'elasticnet']}}
```

Figure 2: A weak specification extended with one complementary component, one functionally related components, and two hyperparameters/values per component (plus a potential default value, if different).

5.3 Search Procedure

To fully satisfy the definition of a strong specification, AMS must add in a specific search procedure to the extended specification. AMS allows the use of different search procedures, which can be plugged into the system. In particular, the current implementation of AMS exposes a plug-in genetic programming search procedure, using TPOT [37] and a conceptually simple random search procedure implemented as part of AMS's codebase.

5.3.1 Genetic Programming. We use TPOT [37], a genetic programming based AutoML tool, as a search procedure. When using TPOT, we use the search space defined by AMS as the configuration available to the optimization process.

5.3.2 Random Search. AMS's implementation includes a hierarchical random search procedure to generate sequential (i.e. API components are chained in sequence) pipelines. Random search is known to perform better for algorithm configuration than equally simple alternatives such as grid search [4] and has also been successfully applied to related software engineering areas such as product line configuration [35]. To generate a pipeline, the search module samples a depth (up to a bound), then for each step in the pipeline it samples an API component from the configuration specified. For each hyperparameter in the chosen component's configuration, the search samples a value and sets it in that component's constructor. The search distinguishes between preprocessing and classifier components to generate valid candidate pipelines (i.e. the last step must always be a classifier). Candidate pipelines are cached to avoid re-training/evaluating pipelines, however, there is no effort to exhaustively search the space and if a pipeline is re-sampled a given number of times (100 in our implementation), the search procedure terminates.

Table 2: NPMI-based association rules mined from our code corpus to identify complementary API components categorized by algorithmic role. When both components in the association have the same role, we elide one for brevity.

Rule Type	# Rules	Mean Norm. PMI	SD Norm. PMI
classifier	78	0.18	0.11
(classifier, cluster)	1	0.37	-
(classifier, decomposition)	3	0.16	0.13
(classifier, feature extraction/selection)	29	0.19	0.15
(classifier, preprocessor)	31	0.20	0.13
(cluster, decomposition)	2	0.45	0.26
(cluster, preprocessor)	1	0.27	-
(cluster, regressor)	3	0.10	0.05
(decomposition, feature extraction/selection)	7	0.17	0.19
(decomposition, preprocessor)	4	0.24	0.26
(decomposition, regressor)	3	0.20	0.25
feature extraction/selection	3	0.35	0.25
(feature extraction/selection, preprocessor)	10	0.25	0.20
(feature extraction/selection, regressor)	4	0.17	0.12
preprocessor	3	0.32	0.26
(preprocessor, regressor)	6	0.20	0.21
regressor	97	0.19	0.07

6 EVALUATION

We now present our experimental results, which evaluate individual parts of our system (RQ1-RQ3, RQ5) and the overall performance of AMS (RQ4). First, we characterize the complementary API components extracted from our code corpus (RQ1). We evaluate AMS' ability to retrieve functionally related API components (RQ2). We then characterize the use of hyperparameters and their values in our code corpus, and evaluate the possibilities for improving classifier performance based on this information (RQ3). We evaluate AMS's ability to produce specifications that result in higher performance (RQ4). And finally, we explore the impact of the code corpus size on AMS's mined hyperparameters and complementary components (RQ5).

For our evaluation, we implemented AMS and its evaluation in approximately 5000 lines of Python. We use Scikit-Learn [39], a popular Python machine learning library, as the target API for pipelines. To mine complementary components and identify hyperparameters/values, we use the meta-Kaggle [28] dataset as our code corpus. The meta-Kaggle dataset contains over 3300 Python scripts.

6.1 RQ1: Complementary API Components

AMS mined 285 normalized PMI (NPMI) positive association pairs from our code corpus. These associations cover 69 different components (39.2% of all components in Scikit-Learn).

Table 2 details the distribution of associations based on the algorithmic role of each of the components, along with their mean and standard deviation NPMI.

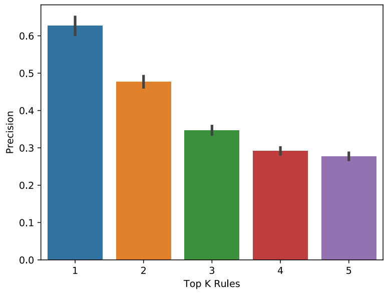


Figure 3: NPMI-based component extension can produce at least one complementary component for 82.68% of our test observations, with precision of approximately 60% when $K_{\mathbf{comp}} = 1$.

To evaluate the effectiveness of these NPMI-based component extensions, we conduct the following experiment. We take each script in our code corpus, and extract the set of Scikit-Learn components used. Using 10-fold cross validation (CV), we split this collection of components into a training fold and test fold. We use each training fold to compute NPMI, and we use the corresponding test fold to evaluate. For each set (ground truth) in the test fold, we take each component individually and use it as a query term to retrieve the top $K_{\text{comp}} \in [1,5]$ complementary components based on our approach (Algorithm 1, with $\alpha = 0.5$). We then compute precision as the fraction of retrieved components that are present in the full ground truth component set. Note that recall is not an appropriate measure of performance for evaluating complementary components, as recall implies our extensions need to be complete, but by definition we will only be able to cover components with strong co-occurrence patterns. Given this, we focus on precision.

We found that 82.68% of the sets in the test folds were covered (i.e. we were able to identify at least one complementary component). For $K_{\text{comp}} = 1$, we found that our NPMI-based approach yields a precision of 60%. This precision declines as expected when we increase K_{comp}, with a precision of approximately 28% when $K_{\text{comp}} = 5$. Based on these results, we configured AMS to use $K_{\text{comp}} \leq 3$. Figure 3 summarizes these results. These levels of complementary component retrieval sufficed for improved performance on our end-to-end benchmarks, but further improving complementary component precision could deliver additional gains.

6.2 RQ2: Functionally Related API Components

To evaluate AMS’s retrieval of functionally related components, we manually annotated our BM25-based ranking of API components for a given query component. To determine if two components were functionally related, we outlined a set of conditions that they should satisfy. Given a specification component Q (for query) and a possible extension component R (for related), we say they are functionally related if they satisfy the following:

- R could replace Q in a pipeline without raising an exception for the same dataset.
- Q and R belong to the same class of operators (e.g. classifier, regressor, value normalizer, decomposition algorithm, loss function).

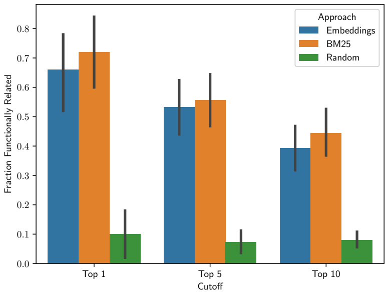


Figure 4: For 50 randomly sampled query API components, BM25 can retrieve close to 72%, 55%, and 44% functionally related components based on top 1, 5, and 10 cutoffs.

- If Q/R are classifiers/regressors, they must respect output shape constraints: a multi-task model can replace a single task model, but not vice-versa.
- If Q is (non-)linear, R must be always (non-)linear or must be (non-)linear based on a hyperparameter (e.g. SVM with a linear kernel)
- If Q is ensemble-based, R must be ensemble-based with one exception: R can be non-ensemble based if it is related (based on these rules) to the weak model class ensembled in Q.
- If Q is not ensemble-based, R may be ensemble-based if it uses a weak model class related to Q to create its ensemble.

To carry out our experiment, we randomly sampled 50 classes from Scikit-Learn and used these as queries. We chose to sample 50 classes as this covers approximately 28% of the components available in Scikit-Learn and balanced the need for detailed manual annotation. For each query, we retrieved the top 10 API components based on: 1) our BM25 metric, 2) cosine similarity using averaged pre-trained neural embeddings (which have been shown to be effective for the related task of code search [6]), and 3) a uniform random metric. We used (2) to compare the use of BM25 with another unsupervised approach to semantic similarity. We used BERT embeddings derived from a scientific text corpus [3]. We used (3) as a baseline to control for the extent to which our target API (Scikit-Learn) may have redundant components resulting in functionally related results through chance.

Figure 4 presents our results. The BM25-based ranking performed comparably (with no statistically significant difference) to the embeddings based approach. A random ranking results in approximately 10% functionally related results, across the top 1, 5, and 10 query results. In contrast, BM25 results in close to 72%, 55%, and 44% functionally related results across the same cutoffs, respectively. We opted to use BM25 in AMS, in contrast to the neural embeddings approach, given their comparable performance and the added advantage of avoiding the additional storage requirements imposed by per-token embeddings.

Note that while for purposes of this experiment, we allow functionally related components to include ensembled-variants of non-ensemble models, in our tool implementation users can exclude ensembles through a simple command line flag.

While we evaluated functionally-related API component retrieval using BM25 and cosine-similarity using BERT embeddings, AMS can use other information retrieval metrics. We also note that the task of specification strengthening, in the context of AutoML, is related but orthogonal to pure information retrieval. In particular, we generate a *new* search space configuration based on a weak specification, rather than searching over a stored (and pre-enumerated) set of configurations.

6.3 RQ3: Hyperparameters and Values

Figure 5 characterizes the hyperparameter tuning observed in our code corpus. In particular, we found that over 50% of the calls tune (i.e. explicitly set a value in the call) for under 20% of the hyperparameters available (5a); for about a third of API components the set of hyperparameters tuned is similar across calls (5b); and for over 70% of the hyperparameters observed, user calls choose few values (under 10 distinct values) (5c). This aligns with our intuition that human developers tend to tune a small set of hyperparameters, these are consistent across datasets/pipelines, and there are popular values that developers choose for each.

To demonstrate the possible impact of hyperparameter tuning, we performed the following experiment. We collected five datasets from the Penn Machine Learning Benchmarks (PMLB) [38]. The five datasets are healthcare-related classification tasks. We collected these 5 dataset to be independent from those used in RQ4. We then identified the top 5 most common classifiers[5] from our code corpus. For each classifier, we extracted the top 3 hyperparameters and top 3 values for each hyperparameter, along with the default values. We performed grid search over these values to evaluate all possible configurations. We then compared the best macro-averaged F1 score [16] from the grid search with the score obtained under the default configuration.

Figure 6 shows our results. In almost all cases, the hyperparameter space defined by the code examples in our corpus contained a setting which would have improved performance with respect to the default configuration. For the ensemble-based classifiers, ExtraTreesClassifier and RandomForestClassifier, this improvement could have been up to 10% on two of the datasets.

6.4 RQ4: Performance of Strong Specifications

Our performance experiments compare the following approaches:

- Weak Spec.: runs an ordered version of the original weak specification as a pipeline directly.
- Weak Spec. + Search: carries out a specified search procedure over the components defined in the weak specification (with default hyperparameters).
- Expert + Search: uses the set of hyperparameters/values defined in TPOT's default classifier configuration [1] for each component in the specification, and applies the specified search procedure. This choice of hyperparameter space corresponds to an expert AutoML developer identifying key hyperparameters and values. We also evaluated writing our own hyperparameter space and found that it performed comparably or worse, so we elide for brevity.
- AMS + Search: applies AMS to the weak specification to produce a full search space and then applies the specified search procedure.

Table 3: Components used to produce weak specification. *Expert + Search* uses TPOT's pre-defined hyperparameter search space [1] for each component.

Short name	Component
lr	Logistic Regression
rf	Random Forest
dt	Decision Tree
scale	Min-max value scaling
poly	Extract polynomial features
var	Variance-based feature selection
pca	PCA decomposition

For these experiments, we consider both search procedures available in AMS: genetic programming and random search.

Table 3 presents the individual components used to create the weak specifications for our experiments. We chose components that covered common machine learning operations: value scaling, feature derivation, feature selection, dataset decomposition, and varied forms of classification. For each such component, we also outline a subset of hyperparameters identified for tuning and their possible values, which are used in the *Expert + Search* approach.

We produced 15 weak specifications by combining the following 5 pre-processing weak specifications with each of the three classifiers (lr, rf, dt) - as outlined in Table 3: {} (no-preprocessing), {scale}, {poly, scale}, {poly, scale, var}, and {poly, scale, pca, var}.

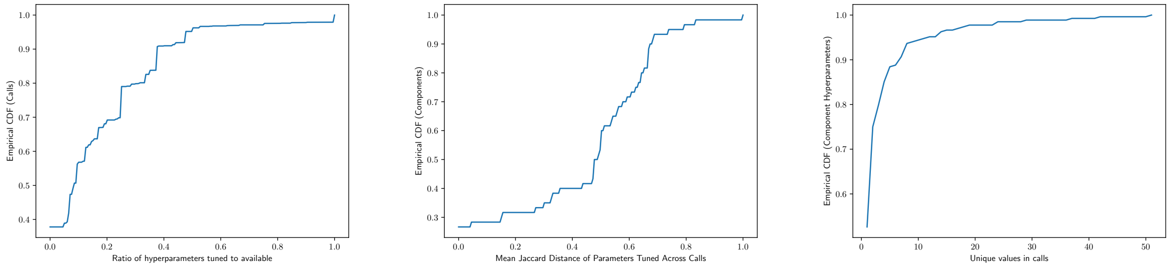
For our experiments we used all classification datasets from the original TPOT paper [37]; 9 in total. These datasets are: Hill-Valley-with-Noise, Hill-Valley-Without-Noise, breast-cancer-wisconsin, car-evaluation, glass, ionosphere, spambase, wine-quality-red, and wine-quality-white. All datasets are available through PMLB [38].

Our experiments used macro-averaged F1 score as a performance metric, where a higher score corresponds to better performance. Each search procedure uses this same score metric in their internal search loop. For each benchmark, dataset, and search procedure combination, we carried out 5-fold cross-validation (CV) with each of the approaches outlined previously. In each CV iteration, the training fold is used to find an optimized pipeline, and the test fold is used for evaluation. All approaches were provided a budget of 5 minutes per CV iteration (i.e. 25 minutes per dataset, for each specification and approach combination).

We evaluate AMS with the following configuration: a weak specification can be extended with at most 3 complementary components ($K_{\text{comp}} = 3$), where the npmi/support fraction weighing parameter is set to 0.5 ($\alpha = 0.5$), for each specification component we include up to 4 functionally related components ($K_{\text{rel}} = 4$), and we tune the top 3 hyperparameters per component ($K_{\text{params}} = 3$) by choosing from the 3 most common values per hyperparameter ($K_{\text{vals}} = 3$). We set the depth bound for the random search procedure to 4.

Figure 7 presents a count of the wins for each approach across both search procedures [8]. An approach *wins* when the average of the 5-fold CV test-fold performance metric is the highest across approaches for a given dataset and weak specification combination,

[5] excluding SVM, which did not terminate within a reasonable computing budget without additional data pre-processing for these datasets



(a) Fraction of hyperparameters tuned

(b) Distance between sets of hyperparameters

(c) Unique hyperparameter values used

Figure 5: Characterizing hyperparameter tuning in our code corpus.

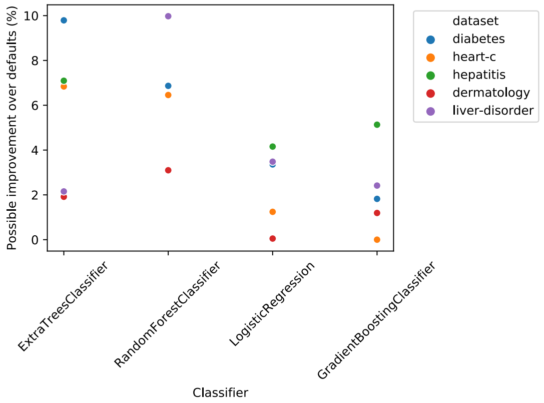


Figure 6: Possible improvements in macro-averaged F1 score by using hyperparameter settings in our code corpus, compared to the performance using defaults.

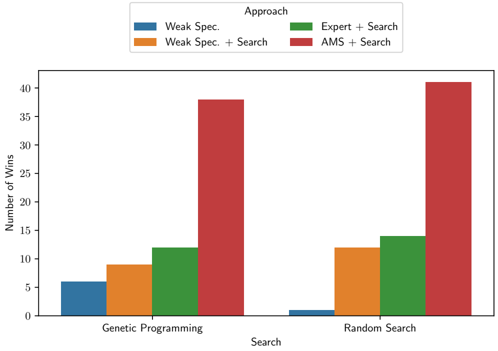


Figure 7: Wins for each approach across 270 experiments. Within a given search procedure, an approach *wins* when it obtains the highest average 5-fold CV test-fold performance for a dataset and weak specification combination, *and* this score is at least 1% higher (in absolute terms) than the next best score.

and the score is at least 1% (in absolute terms) higher than the next best score. We introduced a minimum performance difference threshold to eliminate cases where multiple approaches perform roughly equally on a specification/dataset combination. We varied the minimum difference threshold from 1% to 5% (absolute) and found that AMS obtained more wins than other approaches in all cases.

When using genetic programming as a search procedure, we see that *Weak Spec.* obtained 6 wins compared to 9 wins for *Weak Spec. + Search*. Under random search, *Weak Spec.* obtained 1 win and *Weak Spec. + Search* obtained 12 wins. *Expert + Search* obtained 12 wins when using genetic programming, and 14 wins when using random search. Under both search procedures, using AMS produced the majority of wins: 38 in the genetic programming experiments and 41 in the random search experiments.

Figure 8 presents the distribution of the top-10 Scikit-Learn operators as a fraction of the total count of operators in pipelines produced by genetic programming using AMS's strengthened specification for two different weak specifications. For comparison, running genetic programming over the full search space (as defined in TPOT's default classification configuration) produces pipelines where 60% of them have an ensemble-based model (one of GradientBoosting, ExtraTrees, XGBoost, or RandomForest), and at least one pipeline produced in 8 of the 9 datasets includes such a component. The skew towards ensemble-based models has been observed in other AutoML tools as well [14]. By using AMS a user can restrict the use of ensemble-based models, for example, if desired.

Comparison to other program-mining based AutoML tools. We also compared AMS to AL [7]. AL mines dynamic program traces to learn a probabilistic model for ML pipelines and uses this to generate sequential pipelines. A key advantage of AMS is that users can strengthen specifications without the need to collect a new corpus that reflects their initial specification. AMS can also mine information from otherwise un-executable programs and without access to the programs' target datasets, while AL requires program execution for its dynamic analysis.

To compare AL and AMS, we consider the weak specification of Scikit-Learn components [6]:

```
{LogisticRegression, LinearSVC, StandardScaler}
```

[6] names abbreviated for brevity

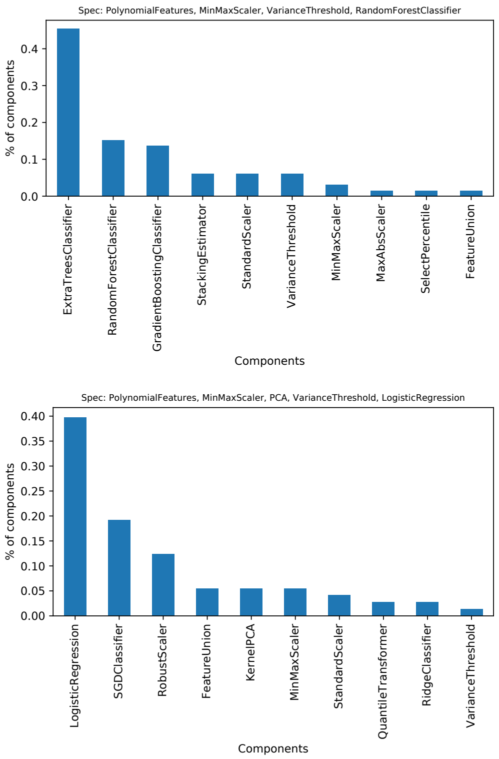


Figure 8: Example distributions of Scikit-Learn components in pipelines produced by genetic programming, based on AMS strengthening of the initial weak specification.

and run experiments on our 9 datasets. We use 5-fold CV, pair pipelines between CV folds in order to appropriately perform comparisons after removing pipelines that don't satisfy the weak specification, and then compute wins on the paired pipelines. If the pipeline for a system does not satisfy the specification, the other system's pipeline is assigned the win. AL is trained on the corpus presented in [7], which is restricted to programs it has already executed and from which it has extracted dynamic traces.

When AL is trained on the subset of programs that use at least one weak specification component, and AMS mines this same set of programs, we find that AL can produce pipelines that *still* deviate from the weak specification (as the full program traces may contain additional components). 21 of the 45 pipelines generated by AL did not satisfy the weak spec, while all of AMS do. After removing specification-violating pipelines, AMS obtains 29 wins and AL obtains 9. When AMS is trained on the full AMS corpus, AMS's wins increase to 35 (and all pipelines continue to satisfy the specification) and AL's decrease to 4. Finally, when AMS is trained on the AMS corpus and AL is trained on the full AL corpus (without any specification-related program pruning), 26 of the 45 AL pipelines do not satisfy the weak specification. After removing these pipelines, AMS obtains 42 wins and AL obtains 1 win.

6.5 RQ5: Impact of Corpus Size

AMS mines hyperparameters, their corresponding values, and complementary component association rules from a corpus of code examples. To evaluate the impact of varying corpus sizes on AMS, we sampled from 10% to 90% (in 10% increments) of the original 3,300 scripts. We repeated this sampling five times per sampling ratio. For each sampled corpus, we ran AMS's hyperparameter mining and complementary component mining.

Figure 9a shows the average fraction of hyperparameters missing for a given component, with respect to the hyperparameters found through the full corpus. For very small corpora, e.g. 10% (330 scripts), as expected the reduction in hyperparameters mined can be substantial. A moderate sized corpus, e.g. 50% (1650 scripts) covers most of the hyperparameters found in the full corpus.

Figure 9b shows the average reduction in possible hyperparameter values with respect to the full corpus. If we mine 5 possible values for a hyperparameter in the full corpus, and we mine 3 possible values in a downsampled corpus, we say that is a reduction of 2 possible values. We see that for a moderate sized corpus (e.g. 50%) the average reduction is approximately one possible value per hyperparameter.

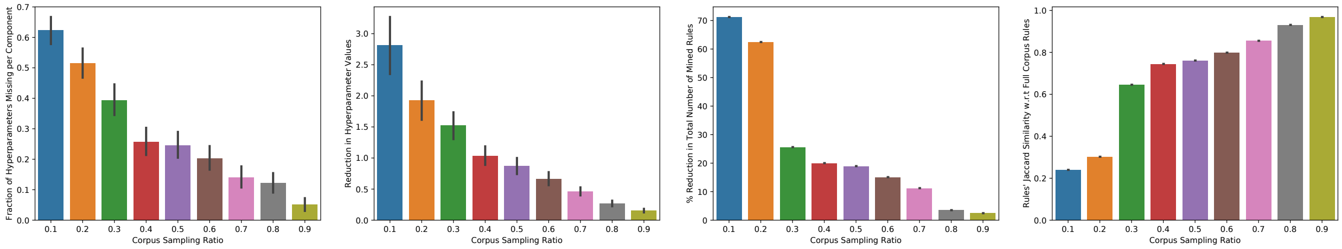
Figure 9c shows the decrease in number of complementary components mined, when compared to the full corpus. Small corpora (< 30% of the original size) display large decreases in the number of association rules found, but moderate sized corpora (e.g. 50%) mine approximately 80% as many rules as the full corpus. Figure 9d shows that for moderated sized corpora, the rules mined are relatively similar (approximately 0.8 jaccard similarity) to those mined from the full corpus.

7 RELATED WORK

Automated machine learning (AutoML) has received increased attention in the recent past. TPOT [37] uses genetic programming to automatically produce tree-structured classification and regression pipelines composed of Scikit-Learn [39] operators. Autosklearn [15] uses sequential model-based algorithm configuration (SMAC) [26] to generate Scikit-Learn pipelines and their hyperparameter settings. ReinBo [46] uses reinforcement learning to generate pipeline candidates and Bayesian optimization to tune their parameters. AL [7] learns a pipeline likelihood model from dynamic program traces and uses this to generate sequential Scikit-Learn pipelines, with default hyperparameter values. MLBazaar [45] builds up an AutoML system through a library of composable operators with a clean and unified interface.

In contrast to these systems, AMS focuses on providing AutoML users with a simple way of influencing the pipeline generation process: writing a weak specification, which can be automatically strengthened. This usage model empowers users to influence pipeline generation on a per-specification basis, rather than relying on distributional characteristics of a pipeline corpus (as in AL), on the developer pre-defined search spaces in the original AutoML tool (as in TPOT, ReinBo and Autosklearn), or manually specifying a new complete search space that reflects their preferences (as in TPOT's optional configurations or MLBazaar's top-down templates/configurations).

Search-based Software Engineering (SBSE) [22] provides a general framework through which to design and analyze AutoML systems,



(a) (b) (c) (d)

Figure 9: Impact of downsampling the full corpus. (a) shows the fraction of missing hyperparameters per component; (b) shows the number of missing possible values per hyperparameter; (c) shows the percent reduction in number of complementary component association rule; and (d) shows the jaccard similarity of the complementary component rules.

with the latter effectively being a instance of the former. SBSE has been successfully applied to problems such as automated testing of software with large test suites [32], synthesizing equivalent method call sequences [19], and optimizing product line configurations [35], among others. AMS allows users to approach AutoML in a grey-box setting, where their weak specification can influence the search process. Feedback of this form can enable an *“iterative process of refinement”* [22] to obtain solutions that satisfy user preferences.

Code corpora have enabled advances in various areas of software engineering. Code idioms mined from a corpus can be used to improve program synthesis and semantic parsing [44], as well as enabling context-sensitive developer queries [43]. At a smaller scale, automated example extraction [24] from specific scripts allows allows users to produce minimal working snippets. Large-scale corpora that exercise particular APIs can be used to mine preconditions for method calls [33], order-based specifications for chaining calls [2], code repair patterns [31], and drive semantic code search [6, 25, 30].

8 THREATS TO VALIDITY

We discuss potential limitations of this research based on design choices. In particular, we focus on threats to generalizability. First, our evaluation uses a particular ML framework (Scikit-Learn). We believe this threat is mitigated by the fact that Scikit-Learn is a widely-adopted ML library, used by over 92,000 GitHub repositories as of March 2020. Extending AMS to other popular libraries, such as Tensorflow, may be possible as long as these have high quality API documentation, with relevant keywords and explanations, and enough online examples for a code corpus[7].

Our code corpus (meta-Kaggle) represents a wide range of scripts written by different users targeting different datasets. Applying AMS to smaller code corpora may impact performance. In our experiments, we found our corpus of approximately 3,300 scripts delivered good performance, and our experiments with varying sizes of code corpus show that a moderate size (approximately 1650 scripts) can deliver reasonably high coverage of hyperparameters and complementary components when compared to our full sized corpus. Further increasing the size of the corpus can help mitigate this risk.

We evaluated two search procedures: genetic programming and random search. Other search procedures may potentially find pipelines with different characteristics and performance. However, both random and genetic search are commonly used methods in search-based software engineering and have shown good performance over a wide range of AutoML problems. The choice of evaluation datasets could also influence our results. We used the classification datasets from the original TPOT paper, which have also been used in the evaluation of existing AutoML research [8, 10]. The weak specifications in our evaluation are naturally a sample of possible specifications. However, we aimed to incorporate common operations and components in these specifications to reflect standard usage.

Finally, weak specifications must include at least one task-specific (i.e. regression/classification) component. We believe satisfying this requirement is facilitated by the wide availability of online resources (e.g. tutorials, blogs) describing basic library usage.

9 CONCLUSION

We introduced a new usage model for AutoML, where a user provides a set of API components as a weak specification and this specification can be automatically strengthened. Specifications enable users to exert control and express preferences over the resulting pipeline. We implement our strengthening approach – extending the specification with complementary components using normalized pointwise mutual information on an existing code corpus, functionally related components using a lexical similarity score over the target API’s documentation, frequency distributions on constructor calls in the code corpus to extract key hyperparameters and values, and a search procedure – in the AMS system. We evaluated AMS on 9 datasets and 15 weak specifications using two different search procedures. We show that the pipelines produced using AMS’s strengthened specifications outperform pipelines produced using the initial weak specifications and variants of the initial specifications annotated with expert-defined hyperparameter spaces.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their feedback, which improved the paper substantially, particularly on: an extended form of weak specifications, a discussion and comparison with AL, and an exploration of the impact of corpus size. This work was partially funded by DARPA HACCS-HR001118C0059.

[7] e.g. as of April 28th 2020 72,000 source code projects on GitHub used Tensorflow

REFERENCES

[1] [n.d.]. TPOT Default Classifier Configuration. https://github.com/EpistasisLab/tpot/blob/master/tpot/config/classifier.py. Accessed: 2020-03-05.

[2] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. 2007. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 25–34.

[3] Iz Beltagy, Arman Cohan, and Kyle Lo. 2019. Scibert: Pretrained contextualized embeddings for scientific text. *arXiv preprint arXiv:1903.10676* (2019).

[4] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of machine learning research* 13, Feb (2012), 281–305.

[5] Gerlof Bouma. 2009. Normalized (pointwise) mutual information in collocation extraction. *Proceedings of GSCL* (2009), 31–40.

[6] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 964–974.

[7] José P Cambronero and Martin C Rinard. 2019. AL: autogenerating supervised learning programs. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28.

[8] Boyuan Chen, Harvey Wu, Warren Mo, Ishanu Chattopadhyay, and Hod Lipson. 2018. Autostacker: A compositional evolutionary learning system. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 402–409.

[9] James Michael Curran, Tacha Natalie Hicks Champod, and John S Buckleton. 2000. *Forensic interpretation of glass evidence*. CRC Press.

[10] Alex GC de Sá, Walter José GS Pinto, Luiz Otavio VB Oliveira, and Gisele L Pappa. 2017. RECIPE: a grammar-based framework for automatically evolving classification pipelines. In *European Conference on Genetic Programming*. Springer, 246–261.

[11] Iddo Drori, Yamuna Krishnamurthy, Raoni Lourenco, Remi Rampin, Kyunghyun Cho, Claudio Silva, and Juliana Freire. 2019. Automatic Machine Learning by Pipeline Synthesis using Model-Based Reinforcement Learning and a Grammar. *arXiv preprint arXiv:1905.10345* (2019).

[12] Iddo Drori, Lu Liu, Yi Nian, Sharath C Koorathota, Jie S Li, Antonio Khalil Moretti, Juliana Freire, and Madeleine Udell. 2019. AutoML using Metadata Language Embeddings. *arXiv preprint arXiv:1910.03698* (2019).

[13] Ian W Evett and Ernest J Spiehler. 1989. Rule induction in forensic science. In *Knowledge Based Systems*. Halsted Press, 152–160.

[14] Fabio Fabris and Alex A Freitas. 2019. Analysing the Overfit of the Auto-sklearn Automated Machine Learning Tool. In *International Conference on Machine Learning, Optimization, and Data Science*. Springer, 508–520.

[15] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2019. Auto-sklearn: efficient and robust automated machine learning. In *Automated Machine Learning*. Springer, 113–134.

[16] Akinori Fujino, Hideki Isozaki, and Jun Suzuki. 2008. Multi-label text categorization with model combination based on f1-score maximization. In *Proceedings of the Third International Joint Conference on Natural Language Processing: Volume-II*.

[17] Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices* 51, 1 (2016), 499–512.

[18] Pieter Gijsbers, Erin LeDell, Janek Thomas, Sébastien Poirier, Bernd Bischl, and Joaquin Vanschoren. 2019. An open source AutoML benchmark. *arXiv preprint arXiv:1907.00909* (2019).

[19] Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, Mauro Pezzè, and Paolo Tonella. 2014. Search-based synthesis of equivalent method sequences. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 366–376.

[20] Isabelle Guyon, Imad Chaabane, Hugo Jair Escalante, Sergio Escalera, Damir Jajetic, James Robert Lloyd, Núria Macià, Bisakha Ray, Lukasz Romaszko, Michèle Sebag, et al. 2016. A brief review of the ChaLearn AutoML challenge: any-time any-dataset learning without human intervention. In *Workshop on Automatic Machine Learning*. 21–30.

[21] Satoshi Hara and Kohei Hayashi. 2018. Making Tree Ensembles Interpretable: A Bayesian Model Selection Approach. In *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 84)*, Amos Storkey and Fernando Perez-Cruz (Eds.). PMLR, Playa Blanca, Lanzarote, Canary Islands, 77–85.

[22] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 1–61.

[23] Xin He, Kaiyong Zhao, and Xiaowen Chu. 2019. AutoML: A Survey of the State-of-the-Art. *arXiv preprint arXiv:1908.00709* (2019).

[24] Andrew Head, Elena L Glassman, Björn Hartmann, and Marti A Hearst. 2018. Interactive extraction of examples from existing code. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.

[25] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv preprint arXiv:1909.09436* (2019).

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

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

[28] Kaggle. 2017. *Meta-Kaggle*. https://www.kaggle.com/kaggle/meta-kaggle/data

[29] Michael J Kane, Natalie Price, Matthew Scotch, and Peter Rabinowitz. 2014. Comparison of ARIMA and Random Forest time series models for prediction of avian influenza H5N1 outbreaks. *BMC bioinformatics* 15, 1 (2014), 276.

[30] Jason Liu, Seohyun Kim, Vijayaraghavan Murali, Swarat Chaudhuri, and Satish Chandra. 2019. Neural query expansion for code search. In *Proceedings of the 3rd acm sigplan international workshop on machine learning and programming languages*. 29–37.

[31] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 118–129.

[32] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 94–105.

[33] Hoan Anh Nguyen, Robert Dyer, Tien N Nguyen, and Hridesh Rajan. 2014. Mining preconditions of APIs in large-scale code corpus. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 166–177.

[34] João Nobre and Rui Ferreira Neves. 2019. Combining principal component analysis, discrete wavelet transform and XGBoost to trade in the financial markets. *Expert Systems with Applications* 125 (2019), 181–194.

[35] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 61–71.

[36] Pedro Paulo de Magalhães Oliveira Jr, Ricardo Nitrini, Geraldo Busatto, Carlos Buchpiguel, João Ricardo Sato, and Edson Amaro Jr. 2010. Use of SVM methods with surface-based cortical and volumetric subcortical measurements to detect Alzheimer's disease. *Journal of Alzheimer's Disease* 19, 4 (2010), 1263–1272.

[37] 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

[38] Randal S. Olson, William La Cava, Patryk Orzechowski, Ryan J. Urbanowicz, and Jason H. Moore. 2017. PMLB: a large benchmark suite for machine learning evaluation and comparison. *BioData Mining* 10, 1 (11 Dec 2017), 36. https://doi.org/10.1186/s13040-017-0154-4

[39] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research* 12, Oct (2011), 2825–2830.

[40] Philipp Probst, Bernd Bischl, and Anne-Laure Boulesteix. 2018. Tunability: Importance of hyperparameters of machine learning algorithms. *arXiv preprint arXiv:1802.09596* (2018).

[41] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, Valletta, Malta, 45–50.

[42] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[43] Naiyana Sahavechaphan and Kajal Claypool. 2006. XSnippet: mining for sample code. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 413–430.

[44] Eui Chul Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. 2019. Program synthesis and semantic parsing with learned code idioms. In *Advances in Neural Information Processing Systems*. 10824–10834.

[45] Micah J Smith, Carles Sala, James Max Kanter, and Kalyan Veeramachaneni. 2020. The machine learning bazaar: Harnessing the ML ecosystem for effective system development. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 785–800.

[46] Xudong Sun, Jiali Lin, and Bernd Bischl. 2019. Reinbo: Machine learning pipeline search and configuration with bayesian optimization embedded reinforcement learning. *arXiv preprint arXiv:1904.05381* (2019).

[47] Catherine Wong, Neil Houlsby, Yifeng Lu, and Andrea Gesmundo. 2018. Transfer learning with neural automl. In *Advances in Neural Information Processing Systems*. 8356–8365.

[48] Quanming Yao, Mengshuo Wang, Yuqiang Chen, Wenyuan Dai, Hu Yi-Qi, Li Yu-Feng, Tu Wei-Wei, Yang Qiang, and Yu Yang. 2018. Taking human out of learning applications: A survey on automated machine learning. *arXiv preprint arXiv:1810.13306* (2018).