Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2017-015 December 21, 2017

Generating Component-based Supervised Learning Programs From Crowdsourced Examples

Jose Cambronero and Martin Rinard

Generating Component-based Supervised Learning Programs From Crowdsourced Examples

José Cambronero
CSAIL
MIT
Cambridge, MA, USA
jcamsan@csail.mit.edu

Martin Rinard
CSAIL
MIT
Cambridge, MA, USA
rinard@csail.mit.edu

Abstract

We present CrowdLearn, a new system that processes an existing corpus of crowdsourced machine learning programs to learn how to generate effective pipelines for solving supervised machine learning problems. CrowdLearn uses a probabilistic model of program likelihood, conditioned on the current sequence of pipeline components and on the characteristics of the input data to the next component in the pipeline, to predict candidate pipelines. Our results highlight the effectiveness of this technique in leveraging existing crowdsourced programs to generate pipelines that work well on a range of supervised learning problems.

Keywords program synthesis, automated machine learning, code mining

1 Introduction

Supervised learning has now become mainstream computing practice [37]. It has been successfully applied to solve problems as varied as identifying email spam, mapping gene expressions to diseases, and predicting stock returns [8, 28, 33]. Indeed, it is now widely applied across many areas of modern data science, often by practioners whose primary interest and focus is on their domain.

Solutions to supervised learning problems now often take the form of a pipeline of components that 1) prepare the data for a core learning algorithm, 2) apply a core learning algorithm, and 3) evaluate the learned model on a held-out dataset [32]. These pipelines are typically built from off the shelf components developed (with significant effort) by machine learning experts. Prominent sources of general machine learning components include Python Scikit-Learn (*sklearn*) [9] and Java Weka [17]. Given the significant resources devoted to developing these components, and the large range of data transforms, learning algorithms, and evaluation metrics available, a modern practitioner must now navigate a complex space of potential pipelines to find a pipeline that provides a good match for the characteristics of the data set (and can therefore deliver an effective solution to the learning problem).

1.1 CrowdLearn

We present CrowdLearn, a system that learns to develop component-based supervised machine learning pipelines by using crowdsourced code and data examples. CrowdLearn starts with user data provided in a tabular form, with relevant features available as columns, and generates a set of pipelines consisting of API calls that process the data, fit a learning model, and evaluate it. Critically, CrowdLearn extracts relevant components from supervised learning pipelines written by humans and predicts a relevant sequence of components in the pipeline based on characteristics of the input data set. The goal is to provide users with effective, automatically generated supervised learning pipelines. Potential benefits include the automatic exploitation of machine learning expertise encoded in pipelines in supervised learning code repositories, the reduction/elimination of the need for users to become knowledgable about complex machine learning components and APIs, a reduction in the development time and expertise required to obtain effective learning pipelines, and enabling a broader range of users to productively apply supervised learning to problems in their domain.

1.2 Basic Approach

We collected a set of 500 human-written programs, each of which implements a supervised learning pipeline. Together, these programs solve 9 supervised learning problems. The programs and data sets are hosted on Kaggle [16], a data science website that hosts competitions, tutorials, and community forums for relevant machine learning topics. We instrument and execute the set of programs to extract the supervised learning pipeline. Specifically, our instrumentation collects dynamic program traces and includes a rich data abstraction for parameters to API calls. This abstraction captures key characteristics (e.g. data types, summary statistics, probability densities, column correlations, missing value frequency) of the input data to that component. We then use a slicing-based algorithm to extract a canonical representation of the supervised learning pipeline executed in each dynamic trace.

CrowdLearn uses a probabilistic model of program likelihood, conditioned on the current sequence of pipeline components and on the characteristics of the input data to the next component in the pipeline, to predict candidate pipelines. The extracted pipelines from the dynamic traces, along with the abstracted input data for each component in the extracted pipelines, form the training data for the probabilistic model of program likelihood. Given an input dataset, a pipeline depth bound, and a bound on the number of programs per depth, CrowdLearn uses the probabilistic model to incrementally construct candidate learning pipelines.

1.3 Results

We implemented CrowdLearn and trained it on a collection of 500 programs crowdsourced through the Kaggle web site [16]. We evaluate CrowdLearn by comparing it to *Autosklearn* [13], an automated machine learning tool that works with a fixed set of components. We evaluate our system on two collections of data sets. The first collection is sourced from Kaggle and Mulan [34]. These data sets include input data with a variety of different datatypes and multivariate outputs. CrowdLearn succesfully produces programs that execute out-of-the-box and perform better than a simple baseline. For these data sets, *Autosklearn* fails to produce a pipeline because its set of components does not include transforms necessary to successfully process the data sets (see Section 12.1). CrowdLearn succeeds because its training examples include transformations (missing from *Autosklearn*'s manually defined search space) that suitably process the input data.

The second collection of data sets is sourced from Scikit-Learn [9], the University of California, Irvine's Machine Learning Repository [23], and OpenML [35]. On these data sets, which have been curated to support machine learning research, CrowdLearn (working with a smaller pipeline search time) delivers comparable performance to *Autosklearn* (see Section 12.3).

Three of the data sets (`housing-prices`, `spooky-author-identification`, `titanic`) in our evaluation have open Kaggle leaderboards. As of this writing, CrowdLearn's top ranked program for these datasets outperforms 29%, 51%, and 91% of the submissions to these leaderboards, respectively.

1.4 Contributions

- **CrowdLearn**: It presents CrowdLearn, a new system that processes an existing corpus of crowdsourced machine learning programs to learn how to generate effective supervised learning pipelines.
  To the best of our knowledge, CrowdLearn is the first machine learning system that learns how to generate supervised pipelines from previous human programs.
- **Algorithm:** It presents the CrowdLearn algorithm, which uses a probabilistic model of program likelihood, conditioned on the current sequence of pipeline components and on the characteristics of the input data to the next component in the pipeline, to predict candidate pipelines.
- **Experimental Results:** It presents experimental results that characterize the performance of CrowdLearn on two collections of supervised learning problems.

```
 1 import xgboost
 2 import sklearn
 3 import sklearn.feature_extraction.text
 4 import sklearn.linear_model.logistic
 5 import sklearn.preprocessing.imputation
 6 import runtime_helpers
 7
 8 # transform inputs
 9 _t0 = runtime_helpers.ColumnLoop(sklearn.feature_extraction.text.
        CountVectorizer).fit(X_train)
10 X_train = _t0.transform(X_train)
11 _t1 = sklearn.preprocessing.imputation.Imputer().fit(X_train)
12 X_train = _t1.transform(X_train)
13
14 # fit machine learning model
15 _m2 = sklearn.linear_model.logistic.LogisticRegression().fit(
        X_train, y_train)
16
17 # evaluate on test data
18 X_val = _t0.transform(X_val)
19 X_val = _t1.transform(X_val)
20 _m2.score(X_val, y_val)
21
22 def predict(X_new):
23   X_new = _t0.transform(X_new)
24   X_new = _t1.transform(X_new)
25   return _m2.predict(X_new)
```

Figure 1. Top ranked Python program generated by CrowdLearn to perform classification on the `titanic` Kaggle dataset.

2 Example

We present an example that illustrates how to use CrowdLearn to solve a supervised learning problem, specifically the learning problem for the Titanic data set (`titanic`) currently hosted on Kaggle [19]. The user first acquires the training data from the Kaggle website. This data takes the form of a CSV file that contains a table of training data organized into rows and columns. Each row corresponds to an observation, with the features of the observation falling into the corresponding columns of the table.

The user provides the file name to CrowdLearn, as well as a column index indicating the target prediction column in the training data. CrowdLearn then splits the training file into training dataset and a held-out validation dataset, and then runs its learning algorithm to produce candidate pipelines (as described in Section 10.1).

2.1 CrowdLearn Pipeline

Figure 1 presents the highest ranked pipeline generated for the Titanic dataset. `X_train` and `y_train` correspond to the training data, and `X_val` and `y_val` correspond to the held-out validation dataset. The pipeline applies a sequence of transformations (lines 9 to 12), fits a logistic regression classifier (line 15), and evaluates the model's performance on the held-out validation data set (lines 18 to 20). The `predict` function (lines 22 to 25) produces new predictions for test data provided by the user.

The first component of the pipeline (lines 9-10) applies a transform that converts strings into tokens, counts the number of times each token appears in each string, and replaces the string in the data set with columns that count the number of occurrences of each token. This transformation is required because the learning algorithm (invoked at line 15) works only with numeric data (and fails if presented with a data set that contains text).

The second component of the pipeline (lines 11-12) applies a transform that imputes missing data (filling in missing data with the mean over entries present in the same column). Again, this transformation is required because the learning algorithm fails if there is any missing data. Because this imputation component works only with numeric data, it also requires the previous application of a transform (such as the string to token count conversion transform in the example pipeline) that converts strings to numeric values.

We note that this pipeline is tailored to the specific characteristics of the data on which it operates, the sequence of transforms which it applies, and the regression algorithm that processes the transformed data. The pipeline contains transforms specifically designed to work with data sets that contain text and missing entries, and these transforms must be applied in the specific pipeline order. Other data sets require different transforms (such as centering values to ensure that every column has zero mean, changes in dimensionality, and scaling the range of values) and different transform orders (see Section 12).

As of this writing, the Titanic dataset has an open leaderboard on the Kaggle web site. The CrowdLearn pipeline in Figure 1 outperforms 91% of the current submissions on this leaderboard.

2.2 CrowdLearn Algorithm

CrowdLearn generates this pipeline based on information it obtains by processing a training set of 500 existing supervised learning scripts crowdsourced through Kaggle. Working with these scripts, it extracts a canonical representation of the pipeline that each script implements. It uses this data, along with the characteristics of the data that appear at each pipeline stage, to obtain a probabilistic model of pipeline likelihood conditioned on the characteristics of the data and on the previous components in the pipeline. Given the user's input training data, CrowdLearn uses this probabilistic model to incrementally construct candidate pipelines. At each pipeline stage, it examines the incoming data and the previous pipeline components to estimate likely next components in the pipeline.

In our example, the string to token count transform is a likely first transform because this transform appears frequently in our set of training scripts that process data with text columns. The imputation component is a likely second component because this transform appears frequently in our set of training scripts that process data with missing values. The imputation component is much more likely in the second position of our pipeline because, in our training set of scripts, it is used only to process numeric data. CrowdLearn will also prune any candidate pipelines that attempt to place this component first because the pipeline will fail during evaluation when CrowdLearn runs the candidate pipeline on the training data.

CrowdLearn applies a logistic regression classifier to obtain discrete predictions because the training data contains discrete target vector values. It chooses this particular classifier because 1) it appears frequently in our training set of scripts, and 2) it performs well on the held out data set as measured by the scoring step (lines 18-20). Here CrowdLearn exploits the fact that the sklearn API for classifiers associates an appropriate scoring metric with each classifier (in the script the classifier is linked to the accuracy metric via the `_m2` logistic regression object).

3 Overview of Supervised Learning

Let $\mathcal{X}:\mathbb{R}^{n\times m}$ be the set of real-valued matrices with n rows and m columns. We refer to each row of $X\in\mathcal{X}$ as an *observation*. Let $\mathcal{Y}:\mathbb{R}^{n\times 1}$ be the set of real-valued column vectors with n entries. Let $\mathcal{H}:\mathcal{X}\rightarrow\mathcal{Y}$ be the set of functions mapping elements in $\mathcal{X}$ to elements in $\mathcal{Y}$, commonly known as a hypothesis space.

Supervised machine learning corresponds to choosing $h\in\mathcal{H}$ given some $X\in\mathcal{X}$ and $Y\in\mathcal{Y}$, such that $h(X)=Y$. It is often not possible to learn an h that satisfies this equality. It may also be undesirable to learn an exact mapping, as there may be unseen data for which h produces a poor result if it satisfies the equality. Often the choice of h is formulated as an optimization problem. Given some cost function $c:\mathcal{Y}\times\mathcal{Y}\rightarrow\mathbb{R}$ then $h^*=\text{argmin}_h c(h(X),Y)$. The process of choosing h^* is often referred to as *fitting* or *learning*. We denote the set of supervised machine learning algorithms by the set $\mathcal{L}:\mathcal{X}\times\mathcal{Y}\rightarrow\mathcal{H}$.

It is often the case that the domain of Y is not the reals, but rather a discrete set of labels. When the domain of Y is a discrete set of labels, the problem is called a *classification* problem. When the domain of Y is the reals, the problem is called a *regression* problem. For brevity, we formalize the regression case, but a similar formulation for classification is possible.

4 Canonical Supervised Learning Programs

Let a dataframe be a collection of (potentially named) column vectors with potentially different datatypes. Let $\mathcal{I}$ be the set of dataframes with n rows and m columns such that $\mathcal{X}\subset\mathcal{I}$. Let $I_{\text{train}},I_{\text{val}}\in\mathcal{I}$ and $Y_{\text{train}},Y_{\text{val}}\in\mathcal{Y}$.

Let $\mathcal{P}$ represent the set of component-based programs with structure implementing a supervised learning pipeline. A program in this set takes as inputs training data $train=(I_{\text{train}},Y_{\text{train}})$ and held-out validation data $val=(I_{\text{val}},Y_{\text{val}})$. The program learns $h\in\mathcal{H}$ using *train* and evaluates the performance of the learned hypothesis on *val*. The output of the program is h and the evaluation metric.

For most non-trivial applications, the input I_{train} to the program must be modified to satisfy constraints imposed by the learning algorithms available in $\mathcal{L}$. These may be hard constraints, such as type constraints, which if not satisfied

$$\llbracket((I_{\text{train}},Y_{\text{train}}),(I_{\text{val}},Y_{\text{val}}),[t_1,\ldots,t_n],l,e)\rrbracket =$$
$$\llbracket((t_1(I_{\text{train}}),Y_{\text{train}}),(t_1(I_{\text{val}}),Y_{\text{val}}),[t_2,\ldots,t_n],l,e)\rrbracket$$
$$\llbracket((I_{\text{train}},Y_{\text{train}}),(I_{\text{val}},Y_{\text{val}}),[],l,e)\rrbracket =$$
$$\llbracket(l(I_{\text{train}},Y_{\text{train}}),e,(I_{\text{val}},Y_{\text{val}}))\rrbracket$$
$$\llbracket(h,e,(I_{\text{val}},Y_{\text{val}}))\rrbracket = (h,e(h,I_{\text{val}},Y_{\text{val}}))$$

Figure 2. Semantics for programs in $\mathcal{P}$, the set of canonical supervised learning programs. These programs produce a fitted hypothesis and a hypothesis score on held-out validation data.

will crash the learning procedure, or soft constraints, such as normalized values, which if not satisfied may result in learning a suboptimal hypothesis. Programs in $\mathcal{P}$ may perform transformations of the input to satisfy these constraints.

Let $\mathcal{T} : \mathcal{I} \to \mathcal{I}$ be the set of functions that transform dataframes to a space of dataframes with potentially different dimensionality and datatypes. Let $\mathcal{E} : \mathcal{H} \times \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ be the set of functions that evaluate the performance of a hypothesis on held-out validation data and produce a model performance score. A program in $\mathcal{P}$ is defined as a five tuple $(train, val, T, l, e)$, where *train* is training data, *val* is held-out validation data, $T = (t_1,\ldots,t_k)$ is a sequence of k transformations where $t_i \in \mathcal{T}$, $l \in \mathcal{L}$ is a supervised learning algorithm, and $e \in \mathcal{E}$ is an evaluation metric.

Figure 2 shows the evaluation semantics of such a program. We use I^i_{train} to denote the output of transformation t_i on input I^{i-1}_{train}, with $I^0_{\text{train}} = I_{\text{train}}$. The output of this sequence of transformations is used to learn a hypothesis $h = l(I^k_{\text{train}}, Y_{\text{train}})$. To evaluate the hypothesis learned, we apply the same sequence of transformations T to I_{val} and use e to score the output of $h(I^k_{\text{val}})$ relative to Y_{val}. The output of the program is the hypothesis and its score on the held-out validation dataset.

5 Modeling Program Likelihood

We model the conditional probability of a canonical supervised learning program $(train, val, T, l, e) \in \mathcal{P}$ as the probability of writing the sequence of operations $t_1,\ldots,t_k,l$, given the training data. This probability is defined as:

$$\Pr(T,l|I_{\text{train}},Y_{\text{train}}) =$$
$$\Pr(l|T,I_{\text{train}},Y_{\text{train}})\prod_{i=1}^{k}\Pr(t_i|T_1^{i-1},I_{\text{train}},Y_{\text{train}})$$

where we use T_i^j to indicate the sequence $t_i \ldots t_j$, and T^0_* is the empty sequence.

We approximate this distribution by making a Markov assumption [6] of order j about the transformations and learning algorithm in the program. This assumptions means that the ith pipeline component is only a function of the j previous calls and its input data. By considering the input data for the ith call, rather than the initial data, we allow some additional information flow beyond the cutoff imposed by our Markov assumption. So our approximate conditional probability distribution is

$$\propto \Pr(l|T^k_{k-j}, I^k_{\text{train}}, Y_{\text{train}})\prod_{i=1}^{k}\Pr(t_i|T^{i-1}_{i-j}, I^{i-1}_{\text{train}}, Y_{\text{train}}) \qquad (1)$$

.

6 Abstracting Input for Learning

To estimate the conditional probability distribution defined in Equation (1), we need a data abstraction that is flexible across different inputs. This flexibility must account for varying dimensions, datatypes, and underlying distributions. We define such an abstraction operation $\alpha : \mathcal{I} \to \mathbb{R}^p$ which summarizes input data using a real-valued vector of dimension p. The vector is designed to capture key characteristics of the data. We often refer to the application of α as *summarizing* the input.

Using α, we cast the problem of learning the conditional probability distribution for supervised learning problems as a supervised learning problem itself, an approach known as meta-learning [15]. We consider each transformation t_i or learning algorithm l as a label, and train a discriminative classifier to predict the appropriate label given the previous j calls and the current call's input data.

The *feature map* that defines α presents a trade-off between the richness of the representation, sparsity in training data, and computational cost required to compute it. Producing very rich representations reduces the likelihood that we will observe many instances of the vector in our training data, and may be more expensive to compute during pipeline search. Very simple representations are unlikely to capture important features of the input data. Based on our own experience and existing literature [1, 30], we define α using the combination of the following features, which are averaged column-wise where relevant:

- **Type Features**: Collection type (e.g. list, array, set); distribution of types in collection elements (e.g. real, integer, string).
- **Features for Numeric Data**: Arithmetic mean; Geometric mean; Median; Minimum; Maximum; Maximum and minimum of z-score; Interquartie range; Skew; Kurtosis; Maximum and minimum of probability density function evaluation under normal, chi-squared, exponential, and gamma distributions; maximum and minimum of cumulative density function evaluation under normal, chi-squared, exponential, and gamma distributions; correlation; Count of missing values.
- **Features for Categorical Data**: Size of domain; Minimum, maximum, and mean frequency of elements in domain; Count of missing values.

7 Collecting Training Data

To empirically estimate the conditional probability distribution of supervised learning programs, we collected and analyzed a set of supervised learning programs crowdsourced through Kaggle.

7.1 Kaggle

Kaggle [16] is a data science website that hosts competitions, tutorials, and community forums for relevant machine learning topics. Users can write their own programs for solving data science problems and submit their predictions to the competitions. Kaggle offers users the ability to write and execute their programs as scripts in an extensive computational environment, providing popular data science languages such as R and Python, commonly used libraries, and access to Kaggle datasets.

7.2 Meta-Kaggle

Kaggle's *Meta-Kaggle* dataset [20] provides public access to a set of existing user scripts, competition information, and community information. We use a set of 500 programs from the *Meta-Kaggle* dataset to construct our corpus of canonical supervised learning programs. These programs solved one of 9 supervised learning problems. We downloaded the data for these problems, and used this to reproduce the execution of these programs.

7.3 Preprocessing the Corpus

We chose programs that were written in Python, a popular data science programming language. We removed all duplicate programs, based on comparing source code as a sequence of tokens after standardizing various formatting conventions. We also removed *pseudo-duplicates*, which we define as programs written by the same user or forked from the same parent script that have similarity ratio of 75% or greater. The similarity ratio used is implemented in Python's `difflib`, and is based on a fast subsequence pattern matching algorithm [29].

This pre-processing of the data is necessary as many of the Kaggle scripts are slight variations of previous scripts. For example we found scripts titled *Beating the Benchmark v1.0* and *Beating the Benchmark v1.1*, where the latter contained few semantic differences. This kind of iterative, ad-hoc version management is common in data science development [21], and often involves saving multiple versions of files, commenting out code, and changing call parameters.

We restricted our data collection to API calls to two popular Python machine learning libraries: Scikit-Learn (*sklearn*) [9] and XGBoost(*xgboost*) [10]. Scikit-Learn is an open source implementation of many popular machine learning algorithms and utilities, targeting medium sized datasets. XGBoost is a distributed machine learning library based on gradient boosting [14], with Python bindings. With this final restriction, we removed from our dataset any Kaggle user script that did not use either library.

7.4 Instrumenting Programs

We apply a simple set of static transformations to the Kaggle user scripts remaining after our preprocessing step. These transformations lift function calls and assign their output to a fresh local variable. For example, `f(g(x), h(x))` would become `v_1 = g(x); v_2 = h(x); v_3 = f(v_1, v_2)`. This allows us to easily add calls to a set of dynamic instrumentation functions. Recall that our goal is to obtain canonical supervised learning programs and estimate their conditional probability distribution, given input data. To do so, we need to collect information on the API calls made and the (abstracted) input to these calls. We use dynamic instrumentation to collect this data. Our dynamic instrumentation:

- unrolls loops once
- records any API calls to the target libraries
- records information on the object instance for method calls
- records call parameters abstracted using α (our data summarization function)
- records memory locations for data dependencies, if these data dependencies are the output of API calls to the target libraries
- updates memory locations associated with return values from API calls to the target libraries

We execute these instrumented Kaggle programs in a virtual machine and reproduce the original execution environment using Docker containers [25]. This environment includes the necessary libraries and the original input data for execution.

8 Extracting a Component Search Space

The instrumented programs produce a set of dynamic traces, along with information on the arguments and memory locations associated with each API component call. We need to label each call in these dynamic traces as an element in $\mathcal{T}$, the set of available transformations, $\mathcal{L}$, the set of available learning algorithms, or $\mathcal{E}$, the set of available evaluation functions. The calls with appropriate labels can then be collected to construct the respective sets.

Our approach allows us to avoid manually defining the set of relevant API components for constructing a supervised learning pipeline by relying on the experience of Kaggle users. In our evaluation, we show that learning the component search space allows our system to handle unexpected inputs. Learning from examples also gives CrowdLearn the flexibility to adapted to new settings, in which different transformations or learning algorithms may be favored by users, by observing existing examples.

8.1 Dynamic Trace Slicing For Canonical Program Extraction

Algorithm 1 describes our algorithm to 1) label each API call in a dynamic trace, and 2) extract a canonical supervised learning program. We exploit the package hierarchy in *sklearn* and *xgboost*'s bindings to label as an element of $\mathcal{T}$ any function or appropriate method call for a class defined in *sklearn*'s *decomposition*, *feature_extraction*, *feature_selection*, *pipeline*, *preprocessing*, and *random_projection* modules.

Similarly, we label as an element of $\mathcal{E}$ any function or appropriate method call for a class defined in *sklearn*'s: *calibration*, *metrics*, *model_selection* modules. We also use the fact that all learning algorithms in *sklearn* and *xgboost*'s Python bindings are implemented as classes with a `fit` method to apply the respective learning algorithm to data. We identify all calls to `fit` that are not already labeled and label these as elements in $\mathcal{L}$.

We say that a call that has been labeled as an element in $\mathcal{L}$ is a *seed* for a canonical supervised learning program. There may be multiple seeds in a single dynamic trace, yielding multiple canonical programs. For each seed in the trace, we slice the trace forward using the data dependencies on the seed call's return value to identify calls that depend on the output of that seed. These calls are labeled as elements of $\mathcal{E}$. We then slice the trace backwards using the data dependency information for the seed call's parameters to identify calls that are inputs to the learning algorithm. These calls are labeled as elements of $\mathcal{T}$. Concatenating the backward slice, the seed, and the forward slice produces a canonical program.

Call labeling in a canonical program is done on a per-seed basis. This means that a single method on the same class can be labeled differently across different canonical programs. This provides CrowdLearn with additional flexibility that is not possible with a pre-defined manual specification of the search space. Additionally, recall that our canonical supervised learning programs are an input to a probabilistic model, which is robust to our labeling algorithm making occasional mistakes.

Algorithm 1 From a dynamic program trace, extract canonical supervised learning programs where each call has been labeled as part of $\mathcal{T}$ (t), $\mathcal{L}$ (l), or $\mathcal{E}(e)$

INPUT: D, a dynamic trace collected through instrumentation of Kaggle user scripts; P, a pre-defined mapping from API calls to labels in e, t, *None*. Let SLICEFWD(d, s) and SLICEBWD(d, s) respectively, compute a forward and backward slice on dynamic trace d starting from call s using data dependencies.

OUTPUT: A set of labeled canonical supervised learning programs from a single trace.

```
1:  function CANONICAL-PROG($D$, $P$)
2:      progs ← $\emptyset$
3:      for call ← $D$ do
4:          call.label = $P$(call.method)
5:      end for
6:      seeds ← {call ∈ $D$|call.label = None ∧ call.method = .fit}
7:      for $s$ ← seeds do
8:          $s$.label ← $l$
9:          fwd ← SLICEFWD($D$,$s$)
10:         bwd ← SLICEBWD($D$,$s$)
11:         for call ← fwd do
12:             call.label ← e
13:         end for
14:         for call ← bwd do
15:             call.label ← t
16:         end for
17:         prog ← CONCATENATE(bwd,seed,fwd)
18:         progs ← progs∪prog
19:     end for
20:     return progs
21: end function
```

9 A Discriminative Classifier for Supervised Learning Programs

We use a discriminative classifier to model the conditional probability distribution of programs, given the input data. We use a common machine learning algorithm to do so: multi-label logistic regression with L_1 regularization.

9.1 Overview of Multi-label Logistic Regression

Given an input observation $x \in \mathbb{R}^m$, multi-label logistic regression will predict one of k labels, where k is the number of possible labels [7]. To predict the label, the model uses a linear combination of the input observation and a matrix of learned weights $W \in \mathbb{R}^{k \times m}$, where W_j is the row of weights associated with label j. For an input x, the classifier predicts

$$\Pr(j|x) = \frac{\exp(W_j^T x)}{\sum_{i=0}^{k}\exp(W_i^T x)}$$

$$\underset{j}{\operatorname{argmax}} \Pr(j|x)$$

To train such a classifier, we use a training dataset D of n labeled observations $(x^{(1)},y^{(1)}),\ldots,(x^{(n)},y^{(n)})$. where $y^{(i)}$ is the label for the ith input $x^{(i)}$. We can compute the likelihood of the training data as follows:

$$L(D) = \prod_{i}^{n} \Pr(y^{(i)}|x^{(i)})$$

We can then compute W by maximizing the likelihood, subject to a L_1 penalty on the magnitude of W. This can be formulated as a Lagrangian optimization problem

$$\underset{W}{\operatorname{argmax}} L(D) - \lambda|W|$$

where λ is a hyperparameter controlling the degree to which we penalize large weights (which can lead to overfitting). We used an off-the-shelf optimizer in *sklearn* [9] to compute W.

9.2 Applying Discriminative Classifiers to Supervised Learning Programs

Recall that in Equation (1) we defined the conditional probability of a canonical supervised learning program as proportional to

$$\Pr(l|T_{k-j}^{k},I_{\text{train}}^{k},Y_{\text{train}})\prod_{i=1}^{k}\Pr(t_i|T_{i-j}^{i-1},I_{\text{train}}^{i-1},Y_{\text{train}}) \qquad (2)$$

which has a Markov assumption of order j.

We instantiated j to 2 in our experiments, which means our model considers only the two previous calls, along with input, when computing the probability of the next component call. We also abstract out the input data using our α abstraction function. We trained two separate classifiers: one for transformations and one for learning algorithms. Let $\Pr_T$ be the conditional probability computed by the former classifier, and $\Pr_L$ be the conditional probability computed by the latter classifier.

We then redefine the conditional probability for a supervised learning program to

$$\Pr_L(l|t_{k-1},t_{k-2},\alpha(I_{\text{train}}^{k}),\alpha(Y_{\text{train}}))$$
$$\prod_{i=1}^{k}\Pr_T(t_i|t_{i-2},t_{i-1},\alpha(I_{\text{train}}^{i-1}),\alpha(Y_{\text{train}}))$$

We trained the two classifiers on the relevant calls in the canonical supervised learning programs extracted in Section 8.1. Algorithm 2 details how we constructed the training datasets for the two classifiers.

10 Generating Supervised Learning Programs

Now that we have a concrete way of quantifying the conditional probability of different supervised learning programs, we introduce our approach to generating new programs when given input data by the user.

10.1 Generation Approach

We define the depth of a supervised learning program as the number of transformations of the input data prior to the application of a learning algorithm. We generate programs using a component-based approach that enumerates programs up to a bounded depth. The enumeration is done by populating holes in a pre-defined sketch for supervised learning pipelines. We use the conditional probability of programs, as estimated by our transformation and learning algorithm classifiers, to prune programs during the search. Figure 3 shows the general structure of the sketches used for the programs we generate.

Algorithm 2 Train a classifier to predict transformations and a classifier to predict learning algorithm choice for supervised learning programs. Both classifiers use call n-grams and input data abstraction.

```
INPUT: CP, a set of a labeled canonical supervised learning
    programs; Let N-GRAM($c, p, n$) construct an n-gram of size
    $n$ components preceding call $c$ in program $p$.
OUTPUT: Two trained classifiers. $M_t : \mathcal{P} \to \mathcal{T}$ for transfor-
    mations, and $M_l : \mathcal{P} \to \mathcal{L}$ for learning algorithms.
 1: function TRAIN-CLASSIFIERS(CP)
 2:     for label $\in \{t,l\}$ do
 3:         $X \leftarrow ()$
 4:         $Y \leftarrow ()$
 5:         for $p \in$ CP do
 6:             obs $\leftarrow$ {call $\in p$ | call.label = label}
 7:             $x \leftarrow ((\alpha(o.\text{args}),$ N-GRAM$(o,p,2)) | o \in$ obs)
 8:             $y \leftarrow$ (o.method$|o \in$ obs)
 9:             $X \leftarrow$ APPEND$(X,x)$
10:             $Y \leftarrow$ APPEND$(Y,y)$
11:         end for
12:         $M_{\text{label}} \leftarrow$ TRAIN(LogisticRegression,$X$,$Y$)
13:     end for
14:     return $M_t, M_l$
15: end function
```

⟨*program*⟩ ::= ⟨*transform*⟩* ⟨*learn*⟩ ⟨*score*⟩

⟨*transform*⟩ ::= ⟨*transform_fit*⟩ ⟨*transform_apply*⟩

⟨*transform_fit*⟩ ::= t_i = #1.fit(I_{train})
  | t_i = ColumnLoop(#1).fit(I_{train})

⟨*transform_apply*⟩ ::= I_{train} = #1.transform(I_{train})

⟨*model*⟩ ::= m_i = #1.fit(I_{train}, Y_{train})

⟨*score*⟩ ::= m_i.score(I_{val}, Y_{val})

Figure 3. Structure of sketches for programs generated by CrowdLearn. `.fit`, `.transform`, and `.score` are part of the API for *sklearn* and *xgboost* components, which we use to instantiate our set of transforms ($\mathcal{T}$), learning algorithms ($\mathcal{L}$) and evaluation ($\mathcal{E}$). Elements of the form #*n* represent holes to be filled with API components.

CrowdLearn uses a greedy algorithm to both extend existing programs with new components and to produce the final set of programs for a given input. The algorithm takes as input a bound d on the depth of the programs to generate, a bound k on the number of programs to produce for a given depth, input training and held-out validation data. CrowdLearn builds programs incrementally, adding transformations (as single components and as part of a column-based bounded-loop) and partially executing the programs to eliminate candidate programs that fail. Algorithm 3 shows how the classifiers are used to predict the next call in a program.

Algorithm 4 shows how the extension of programs is used to greedily enumerate the space of possible programs of a given depth. At each depth, the algorithm takes the first k programs to succesfully execute after extension. These programs are the product of adding calls in descending order of conditional probability to candidate programs, which are themselves sorted in descending order of conditional probability. Each program is then extended with a learning algorithm fitting step, to construct complete programs of that depth. After the final set of programs has been produced, we sort the generated programs in descending order based on the evaluation metric on a held-out validation dataset. This search algorithm has time complexity $O(2 * d * k)$, where d is the depth bound, and k is the bound of programs per depth.

Algorithm 3 Greedily extending a sequence of programs with an additional component predicted by classifiers trained on canonical supervised learning program examples

INPUT: P, a sequence of programs sorted in descending order of conditional probability; m, a trained classifier that predicts calls in descending order of conditional probability; k, a bound on the number of programs to return. Let + mean extending a program with a new call. Let $\textsc{Executable}: \mathcal{P} \rightarrow \mathbb{B}$ be a predicate that evaluates to true if the program can be executed without errors.

OUTPUT: At most k programs, extended with one additional call.

```
1:  function PredictProgramExtension($P$, $m$, $k$)
2:      $P' \leftarrow ()$
3:      for $p \in P$ do
4:          $ops \leftarrow \textsc{Predict}(m, p)$
5:          $ps \leftarrow (p+op \mid o \in ops \wedge \textsc{Executable}(p+op))$
6:          $P' \leftarrow \textsc{Append}(P', ps)$
7:          if $|P'| \geq k$ then
8:              return $\textsc{First}(P', k)$
9:          end if
10:     end for
11:     return $\textsc{First}(P', k)$
12: end function
```

11 Evaluation Methodology

We compare CrowdLearn's generated programs to ensemble-based pipelines produced by *Autosklearn* [13], an automated machine learning system. *Autosklearn* and CrowdLearn both instantiate their program components with the *sklearn* API, use meta-learning to construct candidate programs, and handle regression and classification problems.

We used version 0.18.1 of the *sklearn* library for our experiments. This is the version of the library we used to execute the instrumented Kaggle example programs.

Algorithm 4 Greedy Enumeration of Supervised Learning Programs

INPUT: $I_{\text{train}}, Y_{\text{train}}$, input training data; $I_{\text{val}}, Y_{\text{val}}$, evaluation test data; d, a bound on the depth of the programs; k, a bound on the number of programs per depth; M_t, a classifier predicting the next transformation based on the existing program; M_l, a classifier predicting the learning algorithm based on the existing program. Let *base* be the empty program. Let $\textsc{SortPerf}$ sort programs in descending order based on the evaluation metric on the held-out validation dataset $I_{\text{val}}, Y_{\text{val}}$.

OUTPUT: A sequence of possible programs solving the supervised learning task presented.

```
1:  function Generate($I_{train}$, $Y_{train}$, $I_{val}$, $Y_{val}$, $d$, $k$, $M_t$, $M_l$)
2:      $P \leftarrow ()$
3:      $W \leftarrow (base)$
4:      for depth $\in 0 \ldots d$ do
5:          $P_{depth} \leftarrow \textsc{PredictProgramExtension}(W, M_l, k)$
6:          $P \leftarrow \textsc{Append}(P, P_{depth})$
7:          if depth $\neq d$ then
8:              $W \leftarrow \textsc{PredictProgramExtension}(W, M_t, k)$
9:          end if
10:     end for
11:     return $\textsc{SortPerf}(P, I_{val}, Y_{val})$
12: end function
```

We ran each benchmark ten times on AWS m4.xlarge machines with 16 GB of memory. Each iteration randomly split the dataset into training and test using a 75/25 split. Both *Autosklearn* and CrowdLearn were trained/tested on the same split of the data in each iteration. We present average results.

11.1 Benchmark Datasets

Table 1 shows the first collection of datasets used in our evaluation. The first column in the table shows the dataset name, the second column shows the type of supervised learning task, and the third column shows the source of the data. We used 6 data sets from Kaggle and 4 multivariate regression datasets from the Mulan project [34], a Java library for multi-target regression. These datasets were *not* used by the Kaggle programs that CrowdLearn uses to build its program likelihood model. We restricted our choice to datasets that were of medium size and medium dimensionality. We chose datasets that presented a combination of different datatypes across columns. We picked datasets that were associated with both closed and open Kaggle competition leaderboards. For open competitions, we did not use datasets that are part of *featured* competitions, as there are legal complications on publishing data. We used only *Playground* and *Getting Started* datasets.

Table 2 shows the second collection of datasets used in our evaluation. The first column in the table shows the dataset name, the second column shows the type of supervised learning task, and the third column shows the source of the data.

Dataset	Task	Source
detecting-insults-in-social-commentary	classification	Kaggle
sentiment-analysis-on-movie-reviews	classification	Kaggle
mercedes-benz	regression	Kaggle
spooky-author-identification	classification	Kaggle
housing-prices	regression	Kaggle
titanic	classification	Kaggle
sf1	multivariate regression	Mulan
sf2	multivariate regression	Mulan
enb	multivariate regression	Mulan
jura	multivariate regression	Mulan

Table 1. We use a first collection of datasets to evaluate CrowdLearn with data that have different datatypes and multiple target outputs.

We used 5 data sets available through *sklearn*'s `datasets` module, 3 from the UCI machine learning data repository, and 7 datasets from OpenML. We only chose datasets that did not require transformations outside of *Autosklearn*'s search space so that it works out-of-the-box with these inputs. We used these datasets to compare the predictive performance of the pipelines produced by CrowdLearn to the ensemble-based pipeline produced by *Autosklearn*.

Dataset	Task	Source
boston	regression	sklearn
iris	classification	sklearn
digits	classification	sklearn
diabetes	regression	sklearn
breast_cancer	classification	sklearn
website-phishing	classification	UCI
banknote-authentication	classification	UCI
airfoil-self-noise	regression	UCI
38	classification	OpenML
179	classification	OpenML
772	classification	OpenML
917	classification	OpenML
1049	classification	OpenML
1120	classification	OpenML
389	classification	OpenML

Table 2. We use a second collection of datasets to compare the predictive performance of pipelines generated by CrowdLearn and the ensemble-based pipelines produced by *Autosklearn*. We only chose datasets for which *Autosklearn* succesfully produced a pipeline out-of-the-box.

12 Results

Figure 4 summarizes the components used in the top 10 ranked pipelines generated by CrowdLearn across the benchmark datasets. Figure 4a shows learning algorithm components on the x axis and the fraction of pipelines that used this component as their learning algorithm on the y axis. A taller bar indicates a more popular learning algorithm. Figure 4b shows data transformation components on the x axis and the fraction of transformations using that component on the y axis. A taller bar indicates a more popular data transformation. Approximately 2% of the top 10 pipelines generated contained 4 API components, 33% contained 3 API components, 38% contained two components, and 26% contained a single component. The time to execute a pipeline is a function of the number of components, the complexity of the component implementations, and the size of the data that each component takes as input.

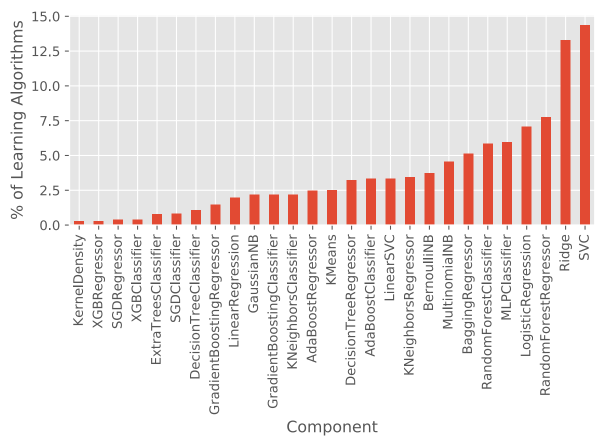
12.1 Learned Search Space

Table 3 shows CrowdLearn's performance on the first collection of datasets from Kaggle and Mulan. The first column identifies the dataset. The 4 following columns correspond to the maximum test set performance for the top 1, 3, 5 and 10 programs generated by CrowdLearn. This ranking of programs is based on a held-out validation dataset that is disjoint from the test set. Because *Autosklearn* fails to run out-of-the-box on these benchmarks, we compare to the *dummy* strategy, which predicts the most common label for classification and the mean for regression problems. The final column in the table indicates the evaluation metric: $F1$ for classification problems and R^2 for regression problems. For these experiments, we set CrowdLearn's search depth bound to 2, the greedy search bound to 30 programs per depth, and a timeout per API component call of 60 seconds.

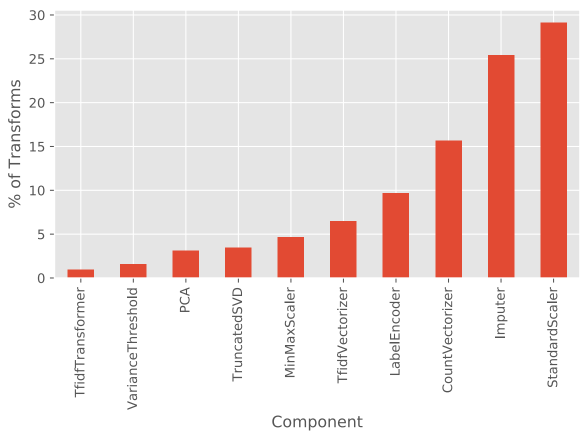
CrowdLearn outperforms the dummy strategy in all cases. *Autosklearn* fails to execute on the Kaggle datasets as these datasets have columns of different datatypes, such as string columns. For example, the `spooky-author-identification` dataset contains two columns with text: the first column is a string identifier, and the second column contains free-form text. *Autosklearn* fails to run on this dataset and raises a `ValueError`. Other datasets produce similar errors.

Autosklearn's errors on these datasets result from a search space [5] that does not include an effective transformation to convert strings to a real-valued vector. *Autosklearn*'s space includes a one-hot-encoding of text, where each string is mapped to a binary vector of length equal to the number of distinct strings. But a one-hot-encoding transform would not be informative on a free-form text vector as each string is likely to be unique. In CrowdLearn's case, the set of transforms extracted from existing code include a TF-IDF transformation and a transform to convert strings to simple token frequency.

Autosklearn fails to produce an ensemble for the Mulan datasets because their system does not handle multivariate-regression [4], a result of a missing method implementation in *sklearn*'s API necessary for *Autosklearn*'s optimization process. CrowdLearn, in contrast, handles multivariate regression without modifications to its implementation as many of the underlying regression algorithms in *sklearn* can handle multivariate outputs. CrowdLearn's simple success criterion (does a candidate program execute without errors) and ranking algorithms (program likelihood and performance on the



(a) Distribution of learning algorithms components in generated pipelines



(b) Distribution of transformation components in generated pipelines

Figure 4. Distribution of learning algorithm and transform components in the top ten generated pipelines for our evaluation datasets. 2% of pipelines generated contained 4 components, 33% of contained 3 components, 38% contained two components, and 26% contained a single component (the learning algorithm). Our algorithm produces a bounded number of programs of each depth up to a depth bound.

Dataset	Top 1	Top 3	Top 5	Top 10	Dummy	Metric
detecting-insults-in-social-commentary	0.77	0.77	0.77	0.77	0.42	F1
enb	0.98	0.98	0.98	0.98	-0.00	R^2
housing-prices	0.83	0.85	0.85	0.85	-0.00	R^2
jura	0.72	0.73	0.75	0.75	-0.01	R^2
mercedes-benz	0.50	0.50	0.50	0.50	-0.00	R^2
sentiment-analysis-on-movie-reviews	0.50	0.50	0.50	0.50	0.13	F1
sf1	-0.01	0.02	0.02	0.02	-0.03	R^2
sf2	0.10	0.11	0.11	0.11	-0.00	R^2
spooky-author-identification	0.84	0.84	0.84	0.84	0.19	F1
titanic	0.82	0.82	0.82	0.82	0.38	F1

Table 3. Performance on the first collection of benchmarks. *Autosklearn* fails to run on these benchmarks out-of-the-box as a result of their manually-defined search space. CrowdLearn produces candidate programs that execute succesfully as a result of extracting relevant transformations from the crowdsourced examples, its simple success criteria, which simply checks if programs produced are executable, and its ranking algorithms.

held-out validation data set) allows our system to produce a pipeline with no special handling.

12.2 Comparing to Kaggle User Programs

Table 4 shows the performance for the top ranked pipeline produced by CrowdLearn for 3 Kaggle datasets with open leaderboards. The first column shows the dataset. The second column shows CrowdLearn's submission percentile (where higher is better). The third column shows the corresponding pipeline's Kaggle score. The final column shows the top Kaggle score for that dataset. CrowdLearn outperformed 29%, 51% and 91% of submissions to the *housing-prices*, *spooky-author-identification*, and *titanic* leaderboards, respectively, as of this writing.

12.3 Comparative Predictive Performance

Table 5 shows a comparison of the predictive performance of CrowdLearn's pipelines and *Autosklearn*'s ensembles. The first column indicates the corresponding datasets. The next four columns show the test set performance for the top 1, 3, 5, and 10 pipelines generated by CrowdLearn. These pipelines are ranked based on performance on a held-out validation dataset. The next column shows *Autosklearn*'s test set performance. The last column indicates the evaluation metric

Dataset	Submission Percentile	CrowdLearn Score	Top User Score
housing-prices	29.08	0.16	0.00
spooky-author-identification	51.55	0.47	0.13
titanic	91.51	0.81	1.00

Table 4. Submissions using CrowdLearn to open Kaggle leaderboards. The top-ranked CrowdLearn program for each dataset outperformed 29%, 51% and 91% of existing submissions as of this writing.

used: $F1$ for classification and R^2 for regression. CrowdLearn produces a pipeline comparable to *Autosklearn* in its top 10 programs for all our benchmark datasets.

```
1 import sklearn
2 import xgboost
3 import runtime_helpers
4 import sklearn.svm.classes
5 import sklearn.preprocessing.data
6
7 _t0 = sklearn.preprocessing.data.StandardScaler().
      fit(X_train)
8 X_train = _t0.transform(X_train)
9 _m1 = sklearn.svm.classes.SVC().fit(X_train,
      y_train)
```

Figure 5. We show the training portion of the top-ranked pipeline produced by CrowdLearn for the `breast_cancer` classification dataset. Prior to fitting an SVM classifier with RBF kernel, the pipeline transforms the input so that each column has zero mean and unit variance. This matches the suggested use described in *sklearn*'s component documentation [31].

12.4 Search Times

Table 6 shows the search times for generating CrowdLearn's final set of programs for the comparative performance evaluation. The first column indicates the dataset and the second column shows the corresponding search time in seconds. CrowdLearn produced the final set of programs for each these benchmark datasets in under 20 minutes. On average, CrowdLearn completed its search in under 10 minutes. For the `iris` dataset, CrowdLearn produced the final set of programs in under 5 minutes. Recall that these programs produced comparable performance to *Autosklearn*'s ensembles, which were produced with a default execution time budget of 1 hour per benchmark dataset.

12.5 Recommended Transforms

Figure 5 presents the top-ranked program generated for the `breast_cancer` classification dataset. Lines 7 to 8 scale the input data so that each column has zero mean and unit variance. Line 9 fits a support vector machine with an RBF kernel. This matches the use case suggested by *sklearn*'s API documentation for `StandardScaler` [31], the transformation component used by the pipeline.

13 Related Work

We discuss related work in the areas of automated machine learning, component-based program synthesis and code mining.

13.1 Automated Machine Learning

Autosklearn [13], an automated machine learning tool, uses Sequential Model-based Algorithm Configuration [18] to explore the space of possible supervised learning pipelines. Their search is initialized using a collection of pipelines that performed well on similar data during offline experiments. The space of configurations is determined by a fixed instantiation of components (from *sklearn*'s API) and the tuneable parameters for each of these components.

TPOT [27] is an automated machine learning system that produces tree-based pipelines, which combine different data processing and modeling operations. Pipelines are evolved using genetic programming. The evolution process maximizes an objective function, such as accuracy in the case of classification, and can be extended to a multi-objective variant that aims to minimize pipeline complexity. The output is the single best pipeline.

Recipe [11] uses genetic programming to evolve supervised classification pipelines generated from a grammar of possible configurations. This grammar-driven approach allows *Recipe* to avoid executing potentially invalid candidate pipelines.

`Autoweka` [22] provides end-to-end supervised learning pipeline construction for both classification and regression tasks. The configuration search process uses the same underlying technique as *Autosklearn* but instantiates components using the Java-based `Weka` [17] library.

In contrast to all these systems, CrowdLearn does not have pre-defined components (beyond the instantiation to *sklearn* and *xgboost*), and instead extracts the relevant set of components based on existing supervised learning programs crowdsourced through Kaggle. Manually defined configuration search spaces limit these systems' ability to handle unexpected input data. The programs generated by CrowdLearn proved to be more flexible, handling benchmark tasks that these other systems fail to handle appropriately: text data in covariates and multivariate regression.

Dataset	Top 1	Top 3	Top 5	Top 10	Autosklearn	Metric
airfoil-self-noise	0.92	0.93	0.93	0.93	0.95	R^2
boston	0.86	0.87	0.87	0.87	0.87	R^2
diabetes	0.44	0.45	0.46	0.46	0.49	R^2
1049	0.72	0.74	0.74	0.74	0.75	F1
1120	0.85	0.86	0.86	0.86	0.87	F1
179	0.78	0.79	0.79	0.79	0.79	F1
38	0.92	0.92	0.92	0.92	0.93	F1
389	0.81	0.81	0.81	0.81	0.81	F1
772	0.43	0.47	0.48	0.50	0.50	F1
917	0.84	0.84	0.84	0.84	0.90	F1
banknote-authentication	1.00	1.00	1.00	1.00	1.00	F1
breast_cancer	0.97	0.97	0.97	0.97	0.97	F1
digits	0.98	0.98	0.98	0.98	0.98	F1
iris	0.87	0.99	0.99	0.99	0.96	F1
website-phishing	0.86	0.87	0.87	0.88	0.88	F1

Table 5. We compare the predictive performance of pipelines produced by CrowdLearn and ensembles produced by *Autosklearn*. Regression tasks are evaluated using R^2, while classification tasks are evaluated using macro-averaged F1. CrowdLearn produces pipelines comparable to *Autosklearn* in its top 10 candidates with shorter search times.

Dataset	Time (seconds)
1049	396.25
1120	540.69
179	937.05
38	541.03
389	426.40
772	384.35
917	455.93
airfoil-self-noise	354.61
banknote-authentication	298.75
boston	366.02
breast_cancer	313.55
diabetes	252.48
digits	444.09
iris	223.90
website-phishing	273.98

Table 6. CrowdLearn search time for generated programs. These programs displayed comparable predictive performance to the pipelines produced by *Autosklearn*, which ran for a default execution time of 1 hour per benchmark.

To incrementally build pipelines, CrowdLearn uses a probabilistic model of program likelihood. Existing systems do not consider program likelihood. *Autosklearn* initializes the pipelines based on a meta-learning approach that chooses from a set of possible pipelines based on dataset similarity, but does not explicitly model pipeline modifications as a function of program likelihood. *TPOT* and *Recipe* modify pipelines based on cross-validation performance in their genetic evolution process.

In our experiments, we compare to *Autosklearn* as both our systems share an overlap of techniques: *sklearn*-component instantiation, meta-learning techniques, and output multiple pipelines.

13.2 Component-Based Program Synthesis and Code Mining

`Prospector` [24] is a component-based program synthesis tool that uses *jungloids*, a simple representation of unary class-based operations, to drive synthesis based on a user-provided query. The jungloid mining algorithm takes existing programs and statically extracts an overapproximation of the jungloids in the program corpus.

`Morpheus` [12] is a component-based synthesis system that produces R programs for table transformations. `Morpheus` takes an input/output example and a set of components, with an over-approximated specification, and uses enumerative search, along with SMT-based pruning, to produce a program to perform the table transformation. During synthesis, possible programs are ranked based on a language model over relevant source code snippets.

Like `Prospector`, CrowdLearn mines possible components from a small set of existing crowdsourced examples. But CrowdLearn uses dynamic trace slicing as the conditional probability model for program likelihood uses an abstraction of the input data to each component call. This use of dynamic program information also differs from `Morpheus`'s model, which is trained on a sequence of static source code tokens.

There is existing work on building statistical models for source code and using them to extract programming constructs [2]. For example, Bayesian techniques have been applied to probabilistic grammars to mine idioms from example repositories [3]. Statistical models of code have also been used to generate program synthesis candidate sketches [26]. Deep learning techniques have also been used to mine code repositories for clones [36].

In contrast to some of these approaches, CrowdLearn uses a much smaller set of example programs (500) to build its conditional probability model. CrowdLearn learns from both the API components called and their input data by instrumenting example programs to obtain dynamic program traces, while many existing approaches focus on applying static analysis techniques. For constructing pipelines for supervised learning, the characteristics of the data are key for choosing appropriate components [15].

14 Conclusion

We presented CrowdLearn, a new system that processes an existing corpus of crowdsourced machine learning programs to learn how to generate effective pipelines for solving supervised machine learning problems. CrowdLearn uses a probabilistic model of program likelihood, conditioned on the current sequence of pipeline components and on the characteristics of the input data to the next component in the pipeline, to predict candidate pipelines. Our results highlight the effectiveness of this technique in leveraging existing crowdsourced programs to generate pipelines that work well on a range of supervised learning problems.

References

[1] Shawkat Ali and Kate A Smith-Miles. 2006. A meta-learning approach to automatic kernel selection for support vector machines. *Neurocomputing* 70, 1 (2006), 173–186.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2017. A Survey of Machine Learning for Big Code and Naturalness. *arXiv preprint arXiv:1709.06182* (2017).

[3] Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 472–483.

[4] Autosklearn. 2017. Github Repository Issue 292. (2017). https://github.com/automl/auto-sklearn/issues/292

[5] Autosklearn. 2017. Github Repository (pipeline components). (2017). https://github.com/automl/auto-sklearn/tree/master/autosklearn/pipeline/components

[6] Leonard E Baum and Ted Petrie. 1966. Statistical inference for probabilistic functions of finite state Markov chains. *The annals of mathematical statistics* 37, 6 (1966), 1554–1563.

[7] Christopher M Bishop. 2006. *Pattern recognition and machine learning*. springer.

[8] Allan Borodin, Ran El-Yaniv, and Vincent Gogan. 2004. Can we learn to beat the best stock. In *Advances in Neural Information Processing Systems*. 345–352.

[9] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.

[10] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. *CoRR* abs/1603.02754 (2016). arXiv:1603.02754 http://arxiv.org/abs/1603.02754

[11] 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.

[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*. ACM, 422–436.

[13] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*. 2962–2970.

[14] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[15] Christophe Giraud-Carrier, Ricardo Vilalta, and Pavel Brazdil. 2004. Introduction to the special issue on meta-learning. *Machine learning* 54, 3 (2004), 187–193.

[16] Google. 2017. Kaggle Website. (2017). https://www.kaggle.com/

[17] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.

[18] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. *LION* 5 (2011), 507–523.

[19] Kaggle. 2015. Titanic: Machine Learning from Disaster (Start here! Predict survival on the Titanic and get familiar with ML basics). (2015). https://www.kaggle.com/c/titanic

[20] Kaggle. 2017. Meta-Kaggle. (2017). https://www.kaggle.com/kaggle/meta-kaggle/data

[21] Mary Beth Kery, Amber Horvath, and Brad A Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists.. In *CHI*. 1265–1276.

[22] Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. 2016. Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA. *Journal of Machine Learning Research* 17 (2016), 1–5.

[23] M. Lichman. 2013. UCI Machine Learning Repository. (2013). http://archive.ics.uci.edu/ml

[24] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid mining: helping to navigate the API jungle. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 48–61.

[25] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014). http://dl.acm.org/citation.cfm?id=2600239.2600241

[26] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Bayesian Sketch Learning for Program Synthesis. *arXiv preprint arXiv:1703.05698* (2017).

[27] 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 2016 on Genetic and Evolutionary Computation Conference*. ACM, 485–492.

[28] Patrick Pantel, Dekang Lin, et al. 1998. Spamcop: A spam classification & organization program. In *Proceedings of AAAI-98 Workshop on Learning for Text Categorization*. 95–98.

[29] John W Ratcliff and David E Metzener. 1988. Pattern-matching-the gestalt approach. *Dr Dobbs Journal* 13, 7 (1988), 46.

[30] Matthias Reif, Faisal Shafait, Markus Goldstein, Thomas Breuel, and Andreas Dengel. 2014. Automatic classifier selection for non-experts. *Pattern Analysis and Applications* 17, 1 (2014), 83–96.

[31] Scikit-Learn. 2017. sklearn.preprocessing.StandardScaler Documentation. (2017). http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

[32] Evan R Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J Franklin, and Benjamin Recht. 2017. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on.* IEEE, 535–546.

[33] Adi L Tarca, Vincent J Carey, Xue-wen Chen, Roberto Romero, and Sorin Drăghici. 2007. Machine learning and its applications to biology. *PLoS computational biology* 3, 6 (2007), e116.

[34] Grigorios Tsoumakas, Eleftherios Spyromitros-Xioufis, Jozef Vilcek, and Ioannis Vlahavas. 2011. Mulan: A java library for multi-label learning. *Journal of Machine Learning Research* 12, Jul (2011), 2411–2414.

[35] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. 2013. OpenML: Networked Science in Machine Learning. *SIGKDD Explorations* 15, 2 (2013), 49–60. https://doi.org/10.1145/2641190.2641198

[36] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 87–98.

[37] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. 2016. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.