# Automatic Discovery of Heterogeneous Machine Learning Pipelines: An Application to Natural Language Processing

**Suilan Estevez-Velarde**[1], **Yoan Gutiérrez**[2,3],
**Andrés Montoyo**[2,3], and **Yudivián Almeida-Cruz**[1]

[1]School of Math and Computer Science, University of Havana, Cuba
{sestevez,yudy}@matcom.uh.cu
[2]University Institute for Computing Research (IUII), University of Alicante, Spain
[3]Department of Languages and Computing Systems, University of Alicante, Spain
{ygutierrez,montoyo}@dlsi.ua.es

## Abstract

This paper presents AutoGOAL, a system for automatic machine learning (AutoML) that uses heterogeneous techniques. In contrast with existing AutoML approaches, our contribution can automatically build machine learning pipelines that combine techniques and algorithms from different frameworks, including shallow classifiers, natural language processing tools, and neural networks. We define the heterogeneous AutoML optimization problem as the search for the best sequence of algorithms that transforms specific input data into the desired output. This provides a novel theoretical and practical approach to AutoML. Our proposal is experimentally evaluated in diverse machine learning problems and compared with alternative approaches, showing that it is competitive with other AutoML alternatives in standard benchmarks. Furthermore, it can be applied to novel scenarios, such as several NLP tasks, where existing alternatives cannot be directly deployed. The system is freely available and includes in-built compatibility with a large number of popular machine learning frameworks, which makes our approach useful for solving practical problems with relative ease and effort.

## 1 Introduction

The continued development of new machine learning algorithms and techniques, and the widely available tools and datasets have brought new opportunities and challenges for researchers and practitioners in both academia and industry. Selecting the best possible strategy to solve a machine learning problem is increasingly difficult partly because it requires long experimentation times and deep technical knowledge. In this scenario, Automatic Machine Learning (AutoML) has risen to prominence as it provides tools based on specific technologies to efficiently search large spaces of machine learning pipelines, such as Auto-Weka (Thornton et al., 2013), Auto-Sklearn (Feurer et al., 2015) and Auto-Keras (Jin et al., 2018). However, practical problems often require combining and comparing heterogeneous algorithms implemented with different underlying technologies. Natural language processing is one scenario where the space of possible techniques to apply varies widely between different tasks, from preprocessing to representation and actual classification. Performing AutoML in an heterogeneous scenario like this is complex because the necessary solution could comprise non-compatible tools and libraries. This would require all algorithms to agree on a common protocol that enables the output of an algorithm to be shared as inputs to any other.

Table 1 contrasts several existing AutoML systems with the system proposed in this research in terms of their capabilities of dealing with heterogeneous scenarios. This evaluation does not attempt to compare AutoML systems in terms of their overall performance, capacity or applicability, but rather with respect to the specific issue of dealing with multiple, heterogeneous algorithms directly. Several popular AutoML systems are based on specific machine learning libraries, such as Auto-Sklearn (Feurer et al., 2015), Auto-Weka (Thornton et al., 2013) and Auto-Keras (Jin et al., 2018), which restrict their use to the

| | | Other AutoML Systems | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **Features** | **Auto-Sklearn** | **Auto-Weka** | **Recipe** | **TPOT** | **Auto-Keras** | **ML-Plan** | **Hyperopt** | **HML-Opt** | **AutoGOAL** |
| 1 | Supports multiple libraries | | | ≈ | ✓ | | ✓ | ≈ | ✓ | ✓ |
| 2 | Models different ML problems | | | ✓ | | | ✓ | ✓ | ✓ | ✓ |
| 3 | Probabilistic space description | ✓ | ✓ | | | | | ✓ | ✓ | ✓ |
| 4 | Automatically extensible by design | ✓ | ✓ | | | ✓ | | ✓ | | ✓ |
| 5 | Automatic pipeline discovery | | | | | | | | | ✓ |
| 6 | Year | 2015 | 2013 | 2017 | 2018 | 2016 | 2018 | 2014 | 2019 | 2020 |

Table 1: Comparison of several existing AutoML systems with our proposal (AutoGOAL) in terms of their capabilities of dealing with heterogeneous machine learning problems. Entries marked with $\approx$ indicate that the system's design enables the given capability but it is not implemented.

scenarios for which the underlying libraries are designed (e.g., supervised learning in the case of Auto-Sklearn). Other approaches such as RECIPE (de Sá et al., 2017) and Hyperopt (Komer et al., 2014) have extensible designs, which in principle allow them to include any machine learning library, but their current implementations are based on specific technologies. On the other hand, systems like TPOT (Olson and Moore, 2016), ML-Plan (Mohr et al., 2018) and HML-Opt (Estévez-Velarde et al., 2020) do provide concrete implementations of machine learning pipelines spanning more than one machine learning library.

Besides supporting multiple libraries, several AutoML systems can be extended to deal with more than one type of machine learning problem, given suitable evaluation metrics. As an example, Hyperot is a general-purpose hyperparameter tuning system which provides low-level building blocks that can be tailored to any machine learning pipeline. However, most of the systems that provide this level of flexibility require a large degree of customization and do not support an automatic discovery of the relevant pipelines for a given problem. This flexibility also comes with a price in terms of extensibility. Systems such as Auto-Sklearn and Auto-Keras benefit from a unified underlying API. This allows researchers to extend the capabilities of the AutoML system simply by extending the underlying API, without actually dealing with implementation details of the AutoML system itself. In contrast, more flexible systems like RECIPE and HML-Opt, both based on context-free grammars to describe the space of possible pipelines, require that researchers modify their internal grammars to add new algorithms.

Another interesting feature for this research is the use of probabilistic models for describing the space of possible pipelines. AutoML systems based on bayesian optimization (e.g., Auto-Sklearn, Auto-Weka) or probabilistic evolutionary optimization (e.g., HML-Opt) construct an internal representation of the pipeline space that can be interpreted as assigning a probability distribution to every pipeline. These systems do not necessarily allow their internal models to be easily accessed. However, this feature could be useful in itself as a description of the pipeline space for researchers to gather additional insights by analyzing which regions of this space have higher o lower probabilities.

Based on the previous considerations, this research focuses on the design of an AutoML system that allows a large degree of extensibility, incorporating algorithms from any underlying machine learning library. At the same time, the system defines a unified protocol that enables automatic pipeline discovery without requiring user intervention.

Concretely, we propose AutoGOAL, a system for heterogeneous AutoML in which the user describes the input and output of a specific machine problem as well as a performance metric, and the system automatically finds the best (or close to best) pipeline of algorithms that solves the problem. This system can deal with different machine learning problems by concatenating and composing algorithms from several libraries, such as `Scikit-learn`, `NLTK`, `Keras`, and `Gensim`. It is also flexible, allowing the user to introduce new algorithms that seamlessly and automatically integrate within the existing pipelines. This is achieved by defining a schema that involves a set of semantic data types and a common protocol that all algorithms implement, allowing their intercommunication.

The most important contributions of this research are:

- We describe a system for Automatic Machine Learning (AutoML) that seamlessly combines heterogeneous technologies and can be applied to a wide variety of machine learning scenarios.
- We propose an extensible and modular design, based on type annotations, that allows the valid pipelines for a given problem to be automatically discovered by providing only the desired input and output types.
- Our approach is competitive with state-of-the-art AutoML systems in standard benchmarks, and can be deployed in novel scenarios that cannot be dealt with by alternative approaches unless considerable customization is implemented, specifically regarding natural language processing tasks.
- AutoGOAL is available as a Python library with pre-packaged implementations of over 100 algorithms from popular machine learning frameworks[1].

This rest of the paper is organized as follows. Section 2 presents a formal definition of the heterogeneous AutoML problem that this paper addresses. Section 3 presents our main contribution, highlighting the key design decisions and important technical details. In Section 4, we present three experimental case studies in diverse machine learning problems to illustrate the flexibility of our system when dealing with heterogeneous scenarios. Finally, Section 5 discusses the most relevant insights and Section 6 presents the main conclusions of the research.

## 2 Problem Statement

In this section, we formally define the problem of Heterogeneous AutoML that this paper addresses. Our definition includes not only supervised learning problems, but more general scenarios including paradigms such as unsupervised learning, information retrieval, etc.

We consider $A$ as the space of all "atomic" machine learning algorithms, i.e., specific techniques such as logistic regression, k-means, or tf-idf representation, which can intervene in any machine learning process. An algorithm $a \in A$ is represented for this purpose as a function $a : T_{in} \to T_{out}$ that maps inputs $x \in T_{in}$ to corresponding outputs in $T_{out}$. $T_i$ are all the possible types of data that machine learning algorithms deal with, e.g., natural language text, embedding vectors, categories, word stems, etc. For example, tokenization algorithms can be seen as any $a : T_{in} \to T_{out} \in A$ such that $T_{in}$ is `Sentence` and $T_{out}$ is `List[Word]`, where these names have a semantic interpretation that corresponds to their usual usage in machine learning.

We define $S$ as the space of all possible "pipelines", i.e., sequences of algorithms from $A$. A pipeline $p =< a^1, \ldots, a^n >\in S$ can thus be seen as a special case of algorithm that applies each $a^{(i)}$ sequentially to the output of the previous algorithm, or formally as the composition of the corresponding algorithms, i.e., $p(x) = a^n(a^{n-1}(\ldots a^1(x)\ldots))$. It follows that $p$ has input $T^1_{in}$ and output $T^n_{out}$. Not all possible algorithms can be combined sequentially, but only those $a^i$ y $a^j$ such that their corresponding output and input types are compatible, e.g., $T^i_{out} = T^j_{in}$. In general we can define a partial ordering function $T_i \leq T_j$ that captures the notion of "compatible" types. Let $S'$ be the space of all "valid" pipelines in the previous sense.

To select the best pipeline we require a metric $\varphi(p) : p \in S'$ that can evaluate each pipeline and allows the comparison of any two valid pipelines. Finally, we define the problem of Heterogeneous AutoML as the optimization problem of finding the best pipeline (given an arbitrary $\varphi$) that transforms a specific input $x \in T^*_{in}$ to a desired $y \in T^*_{out}$. Formally:

$$\begin{array}{rl} \text{argmax} & \{\, \varphi(p) \,|\, p \in S' \,\} \\ \text{s.t:} & p : T^*_{in} \to T^*_{out} \end{array}$$

[1] `https://autogoal.github.io`

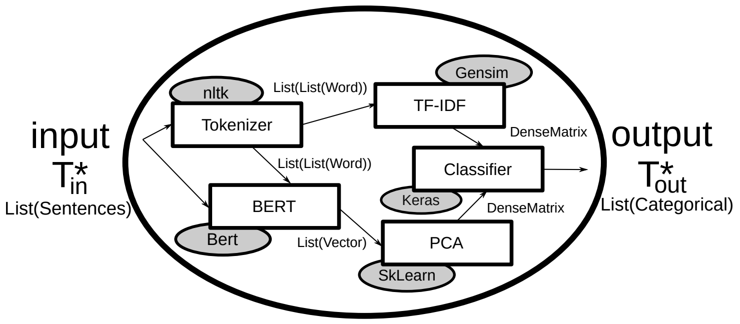


Figure 1: The automatic pipeline discovery process, given specific (example) inputs and outputs. Algorithms and type names are only for illustrative purposes. The current version of AutoGOAL contains 133 algorithm wrappers from 7 different machine learning frameworks, and 26 semantic data type definitions.

This definition applies directly to supervised and unsupervised learning scenarios. In supervised learning, the input will consist of some data in feature space (e.g., documents, images, numerical features) plus a prediction target (e.g., a vector of classes or values to perform regression). In unsupervised learning only the features data is assumed as input. In both cases, the output consists of some prediction target, e.g., classes for supervised learning and cluster labels for unsupervised learning.

## 3 Computational Implementation

The Heterogeneous AutoML problem can be computationally represented as a graph $G_A$ where all known algorithms $a^i$ are nodes, and edges exist between all pairs $a^i, a^j$ such that $T^i_{out} \leq T^j_{in}$, i.e., their corresponding output and input types are compatible. To solve the optimization problem, we insert two nodes in this graph, $Input$ and $Output$, connected correspondingly to all algorithms compatible with the specific input $T^*_{in}$ and all algorithms that output the desired $T^*_{out}$. A pipeline that solves our problem is any path in $G_A$ between $Input$ and $Output$.

A suitable computational implementation of this process requires solving the following problems:

- Defining each algorithm, and their respective input and output, such that it is computationally feasible to determine if two algorithms can be connected, and constructing the graph.
- Designing an optimization strategy that can effectively search in the space of all pipelines, algorithms and their hyperparameters, given restricted computational resources.

This research proposes a theoretical solution to the aforementioned problems, and a concrete computational implementation in the Python programming language, i.e., the AutoGOAL library. Figure 1 illustrates one key design idea of AutoGOAL, i.e., the automatic discovery of machine learning pipelines in a graph of heterogeneous algorithms.

### 3.1 Component Definition

As explained in Section 2, an algorithm is represented as a function that transforms a given input of type $T_{in}$ to an output of type $T_{out}$. The computational representation of the possible data types consists of a class hierarchy in which object inheritance directly represents the $\leq$ relation for type compatibility. The data types have a semantic interpretation beyond their underlying computational structure. For example, a `string` in computational terms can either be a `Document`, a `Sentence` or a `Word`. At the moment of writing, 26 distinct semantic data types are defined, including several types for natural language data, such as tokens and word stems, and several types that represent tensorial data with different semantic interpretations[2]. These semantic data types power the mechanism for automatic pipeline discovery.

Each algorithm is implemented as a class with a `run(input:Tin) -> Tout` method that performs the corresponding processing, potentially wrapping an underlying implementation from a machine learning

[2]See `https://autogoal.github.io/guide/datatypes.png`

library. The class constructor has arguments with annotations that provide valid ranges from which a random sampling process can guarantee to create an instance of the algorithm with valid hyperparameters.

Solving a specific machine learning problem with out approach requires the user to define desired input and output types (respectively $T^*_{in}$ and $T^*_{out}$) as instances of the semantic data types available in the system (see Figure 3 for an example). The graph $G_A$ of all possible algorithms is constructed automatically via source code introspection. All class definitions that contain a `run()` method with suitable annotations are considered nodes of the graph. Edges are automatically added between all nodes (i.e., algorithms) $a_i, a_j$ such that the corresponding annotations are compatible, i.e., $T^{(i)}_{out} \leq T^{(j)}_{in}$. Two "virtual" nodes are inserted in the graph, $Input$ and $Output$, connected correspondingly to all algorithms compatible with the specific input $T^*_{in}$ and all algorithms that output the desired $T^*_{out}$. Once the graph is defined, two depth-first walks are performed one starting at the $Input$ node, and the other at the $Output$ but taking edges in reverse order. Any algorithm reached in both walks is part of a possible pipeline, and the remaining algorithms are discarded. After this process is completed, any random path between $T^*_{in}$ and $T^*_{out}$ is guaranteed to produce a valid pipeline that meets the user requirements.

At the moment of writing, the computational prototype includes a total of 133 algorithms with suitable annotations[3]. The source code for these algorithms has been semi-automatically generated via code introspection from popular machine learning frameworks such as `Scikit-learn` (Pedregosa et al., 2011), `keras` (Chollet and others, 2015), `NLTK` (Loper and Bird, 2002), `Gensim` (Khosrovian et al., 2008), and `Pytorch` (Paszke et al., 2019), and some manual implementations, such as searching terms in knowledge bases like `Wikipedia` and `WordNet` (Miller, 1995). Type annotations enable the seamless and automatic discovery of pipelines that, for example, use `NLTK` for tokenization, `Gensim` to convert tokens to word embeddings, `Scikit-learn` for dimensionality reduction and then a `Keras`-based neural network for classification (see Figure 1).

## 3.2 Sampling and Optimization Process

Given graph $G_A$ built as described in Section 3.1 for a specific machine problem, our proposal can potentially discover all valid pipelines that solve the problem via random sampling.

### 3.2.1 Sampling

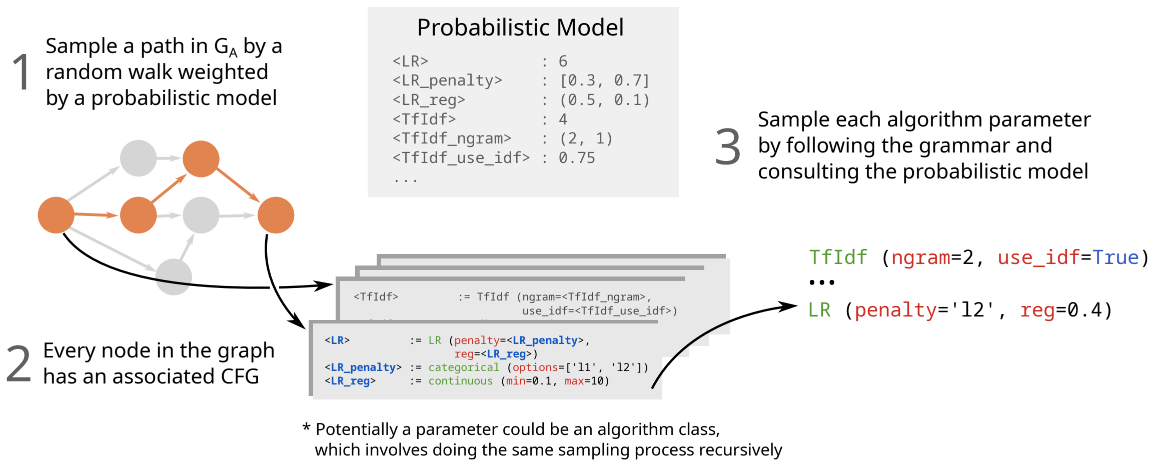
The first step in the sampling process consists of sampling a random path in $G_A$. Sampling starts in the $Input$ node, selecting a random neighbor node that has not been already added to the pipeline, until $Output$ is reached. Since by construction, every node in $G_A$ belongs to some path between $Input$ and $Output$, if all edges have non-zero probability of being traversed, it is guaranteed that this process will eventually end in $Output$.

A random path sampled from $G_A$ is, computationally speaking, a sequence of class definitions, one for each algorithm that will compose the pipeline under construction. Afterwards, from each class an instance is sampled by a process guided by the annotations in the corresponding constructor. For this purpose, a probabilistic context free grammar is inferred for each class. This grammar contains productions for each of the possible hyperparameters (i.e., arguments in the class constructor). Hyperparameters annotated as basic types (discrete, continuous, categorical and boolean) yield productions that simply produce a random value from a suitable distribution. Hyperparameters annotated as other classes will recursively build a corresponding grammar, taking care of correctly solving recursive direct and indirect dependencies (e.g., class A has parameters of type A or a type that recursively depends on type A). This mechanism enables a rich definition of algorithms, beyond simple valued hyperparameters. For example, a sentence vectorization algorithm depends on a tokenization algorithm and a stopword removal algorithm, each of which can recursively depend upon simpler building blocks. From the software design point of view, $G_A$ can be considered as a graph where nodes are Factory Methods (Gamma et al., 1993) that will automatically construct instances of the corresponding algorithms.

The above sampling process is guided by a probabilistic model $\sigma$ that attaches specific parameters to the distributions used in every step. Every algorithm $a_i$ in $G_A$ is assigned an un-normalized weight $w_i$, which

[3]See `https://autogoal.github.io/guide/predefined/#bundled-algorithms`

is used to select a random neighbor during path sampling following a multinomial Bernoulli distribution. Every production in each of the classes' grammars is also assigned a set of parameters according to the distribution used (e.g., mean and variance for numerical values, normalized weights for categorical values, etc.). These sampling parameters are indexed by the class-hyperparameter name, which means that any instance of the same algorithm in a given pipeline shares the same sampling parameters for a given hyperparameter (e.g., the mean and variance of the regularization factor of all logistic regressions instances is shared). Figure 2 illustrates the sampling process.


1 Sample a path in $G_A$ by a random walk weighted by a probabilistic model
2 Every node in the graph has an associated CFG
3 Sample each algorithm parameter by following the grammar and consulting the probabilistic model
Probabilistic Model

```
<LR>              : 6
<LR_penalty>      : [0.3, 0.7]
<LR_reg>          : (0.5, 0.1)
<TfIdf>           : 4
<TfIdf_ngram>     : (2, 1)
<TfIdf_use_idf>   : 0.75
...
```

```
<TfIdf>         := TfIdf (ngram=<TfIdf_ngram>,
                          use_idf=<TfIdf_use_idf>)
```

```
<LR>         := LR (penalty=<LR_penalty>,
                    reg=<LR_reg>)
<LR_penalty> := categorical (options=['l1', 'l2'])
<LR_reg>     := continuous (min=0.1, max=10)
```

```
TfIdf (ngram=2, use_idf=True)
...
LR (penalty='l2', reg=0.4)
```

* Potentially a parameter could be an algorithm class, which involves doing the same sampling process recursively

Figure 2: Illustrative representation of the sampling process for one pipeline from the algorithm graph $G_A$ using a probabilistic model and a context-free grammar defined for each algorithm.

### 3.2.2 Optimization Strategy

For optimization purposes, the probabilistic model $\sigma$ is initialized with neutral parameters for all distribution, e.g., uniform weights for categorical distribution, centered mean and maximum variance for all continuous distributions, $p = 0.5$ for binomial distributions, etc. A probabilistic grammatical evolution strategy (O'Neill and Ryan, 2001) is used to incrementally adjust $\sigma$ so as to maximize the probability of producing the best pipeline. This optimization procedure is adapted from the strategy used in HML-Opt (Estevez-Velarde et al., 2019) to optimize pipelines based on context-free grammars, and extended to also allow optimization over graph structures. This process involves a cycle of: random generation of $n$ pipelines; evaluating each in the machine learning problem at hand (using the fitness function $\varphi(p)$ provided by the user, see Section 2); and, selecting the $k < n$ best pipeline. To avoid overfitting the optimization process in the training set, a small number $M$ of cross-validation evaluations is recommended (e.g., $M \in [1, 5]$]). In any case, once optimization is finished, the best pipeline found must be evaluated on an independent test set for fair comparisons. We define $\varphi(p) = 0$ whenever $p$ results in a runtime error, memory overflow or timeout. This makes the optimization process gradually penalize pipelines that produce runtime errors, eventually generating only pipelines that perform within predefined memory and time constraints.

In each cycle, from the selection of the $k$ best pipelines, a marginal probabilistic model $\sigma^*$ is constructed by taking the sample values of the actual hyperparameter values generated. The marginal model $\sigma^*$ and the original model $\sigma$ are merged using an interpolation factor $\alpha \in [0, 1]$ which provides a balance between exploration and exploitation. The best pipeline sampled in every cycle is compared with the global best and updated accordingly. This cycle is repeated until a desired number of iterations is reached, or a given timeout is passed, or after a number of iterations where no improvement is found. In every iteration, the probabilistic model $\sigma$ slowly converges to a saturated model that maximizes the probability of producing the best pipeline. When the optimization process is completed, the best pipeline found is returned as the potentially best solution for the machine learning problem at hand.

The aforementioned process optimizes pipelines at several levels. First, it converges towards selecting the sequences of algorithms in graph $G_A$ that result in higher fitness. Furthermore, each of these algorithms' hyperparameters are incrementally adjusted. This is recursively performed for any internal algorithm they use. Since all the sampling parameters are stored in a shared probabilistic model, examining this model provides high-level insights for the entire AutoML process. Besides obtaining the best performing pipeline, the probabilistic model $\sigma$ enables a comparison of the average benefit of any given algorithm. It is possible to detect that certain types of algorithms are better on average for a given problem, such as linear classifiers or neural networks. This information can be used to design better optimization strategies, for example, by initializing a probabilistic model not uniformly but according to probabilities learned from past experiments. This strategy will be explored in future research.

## 4 Results and Discussion

In this section, we evaluate the adaptability and performance of AutoGOAL in dealing with heterogeneous machine learning problems. For this purpose, we select three distinct problems with increasing levels of complexity, respectively: classic supervised benchmarks, Section 4.1; text classification, Section 4.2; and, named entity recognition, Section 4.3.

Regarding the first group in Section 4.1, our system is evaluated for seven classic supervised learning benchmarks in the AutoML literature. The purpose of this experimentation is to show that AutoGOAL is competitive with other AutoML systems. Nevertheless, our purpose is not to supersede other AutoML tools, but rather to provide solutions in novel problem settings where existing systems cannot be directly deployed. For this reason, Sections 4.2 and 4.3 deal with more challenging problems in natural language processing tasks. In these more complex scenarios, AutoGOAL will be compared with state-of-the-art solutions that are carefully crafted by domain experts, since existing AutoML tools cannot be directly applied in these scenarios.

Source code and data to reproduce these experimental results is available online[4]. In terms of code, the three experiments follow the same logic, as exemplified in Figure 3.

```
# import library, datasets, etc.
from autogoal.ml import AutoML

automl = AutoML(
    # problem-specific input and output
    input=List(Sentences()),
    output=CategoricalVector()
)

# load problem-specific dataset
X, y = dataset.load()
automl.fit(X, y)
```

Figure 3: Example source code for running AutoGOAL on a specific dataset. For clarity, only key aspects are included. Extra code such as irrelevant imports, logging, time and memory constraints, etc., are not shown. The `input` and `output` types used in this example are specific for supervised NLP problems and must be adapted to the problem definition.

### 4.1 Comparison with Alternative AutoML Systems

This section compares AutoGOAL with other AutoML systems for the purpose of demonstrating that it achieves a similar performance in a set of seven classic benchmarks in the AutoML literature, taken from the UCI repository (Dua and Graff, 2017). This experimental setup is based on the results reported by *ML-Plan* (Mohr et al., 2018): the mean error across 20 independent executions is measured in each dataset, with a random 70% of the data for optimization and the remaining 30% as a hold-out test set, where the final error is measured. Each execution has a timeout of one hour or $10,000$ iterations (whichever

[4]`https://autogoal.github.io/examples/`

is reached first), and performs $M = 3$ Monte Carlo cross-validation evaluations (70%/30%) for each pipeline. In all cases, the population size is 100 with a selection of the best 20 and a learning rate of 0.05.

Table 2 shows the average result obtained by each AutoML system in the corresponding dataset. In all datasets, we obtain a result that is not statistically different to the other systems, according to a $t$-test between our results and the best and worst result in each dataset ($p$-value $< 0.05$). The main takeaway from these results is that our proposal performs comparably to other systems in these datasets.

| **Dataset** | **Cars** | **Credit G.** | **Abalone** | **Shuttle** | **Yeast** | **Dorothea** | **Gisette** |
|---|---|---|---|---|---|---|---|
| ML-Plan (Weka) | 1.27 + 0.56 | 25.54 + 1.28 | 73.72 + 1.23 | 0.01 + 0.01 | 39.37 + 2.54 | 6.49 + 1.23 | 2.92 + 0.27 |
| Auto-WEKA | 0.66 + 0.38 | 26.50 + 2.32 | 73.46 + 1.08 | 0.12 + 0.06 | 39.72 + 2.29 | - | 3.90 + 0.40 |
| ML-Plan (Sklearn) | 0.34 + 0.51 | 24.56 + 2.53 | 73.77 + 1.11 | 0.02 + 0.01 | 39.52 + 2.56 | 8.69 + 1.54 | 2.76 + 0.36 |
| Auto-Sklearn-v | 1.38 + 0.67 | 25.95 + 1.89 | 82.92 + 8.38 | 0.02 + 0.01 | 40.51 + 2.17 | 6.32 + 1.16 | 2.56 + 0.36 |
| Auto-Sklearn-we | 1.26 + 0.53 | 25.39 + 0.88 | 80.59 + 8.32 | 0.02 + 0.01 | 38.99 + 2.28 | 6.02 + 1.01 | 2.24 + 0.33 |
| TPOT | 0.37 + 0.33 | 23.91 + 2.22 | 73.14 + 1.02 | 0.02 + 0.02 | 38.47 + 2.36 | - | - |
| **AutoGOAL** | 0.60 + 0.68 | 27.01 + 3.64 | 74.33 + 0.76 | 0.11 + 0.04 | 39.94 + 2.67 | 5.97 + 1.07 | 2.25 + 0.30 |

Table 2: Comparison of our proposal (AutoGOAL) and other Auto-ML systems for seven classic machine learning datasets in terms of mean $0/1$ loss and its standard deviation. Values for other systems were obtained from *ML-Plan* (Mohr et al., 2018).

### 4.2 Evaluation in the HAHA Challenge

This section compares our approach with human-designed pipelines in the HAHA (Humor Analysis based on Human Annotation) challenge (Chiruzzo et al., 2019). This is a text classification problem presented at IberLEF 2019, with the objective of classifying Spanish-language tweets as humorous or not. The results of this task are measured in terms of $F_1$ of the humorous class. The corpus contains $30,000$ manually classified tweets of which $24,000$ are for training and $6,000$ for testing.

The results are summarized in Table 3 and compared with official results from the HAHA challenge. Given the complexity of this problem, it is harder for an AutoML system to optimize in this space. AutoGOAL was executed on this dataset for a total of 115 iterations (approximately 20 hours total time) after which it was manually stopped. Each pipeline evaluation is relatively slow (around 10 minutes) and thus convergence is slower. However, despite using no problem-specific strategies (other than defining the input as `List[Sentence]` and the output as `CategoricalVector`), our system manages to outperform 14 of the 18 participants in the challenge ($\sim 78\%$). The best pipeline automatically found is composed of two steps: a pre-processing strategy using a BERT transformer (Devlin et al., 2019);

| **Competitors** | F1 | **Competitors** | F1 |
|---|---|---|---|
| adilism | 82.1 | LaSTUS/TALN | 75.9 |
| Kevin-Hiromi | 81.6 | Taha | 75.7 |
| bfarzin | 81.0 | LadyHeidy | 72.5 |
| jamestjw | 79.8 | Aspie96 | 71.1 |
| INGEOTEC | 78.8 | OFAI–UKP | 66.0 |
| BLAIR GMU | 78.4 | acattle | 64.0 |
| UO UPV2 | 77.3 | jmeaney | 63.6 |
| vaduvabogdan | 77.2 | garain | 59.3 |
| UTMN | 76.0 | Amrita CEN | 49.5 |
| **AutoGOAL** | **78.9** | | |

Table 3: Comparison of our proposal (AutoGOAL) with the official results of the HAHA corpus.

and a neural network with 2 recurrent nodes (one BiLSTM layer and one LSTM layer) followed by 2 time-distributed dense layers, with a total of $12,915,169$ trainable parameters.

### 4.3 Evaluation in the MEDDOCAN Challenge

This section compares our approach with human-designed pipelines in the MEDDOCAN (Medical Document Anonymity) challenge (Lara-Clares and Garcia-Serrano, 2019). This is a novel entity recognition problem presented at IberLEF 2019, with the objective of detecting privacy-sensitive elements in Spanish-language medical documents. The corpus is composed of $1,000$ clinical case studies, where $750$ are used for training and $250$ for testing. The results of this task are measured in terms of a micro-averaged $F_1$ across all entity classes.

| **Competitors** | $F_1$ | **Competitors** | $F_1$ |
|---|---|---|---|
| lukas.lange | 96.96 | ccolon | 93.22 |
| Fadi | 96.32 | sohrab | 93.11 |
| nperez | 96.01 | Jordi | 91.84 |
| FSL | 95.95 | plubeda | 90.38 |
| mhjabreel | 95.83 | m.domrachev | 90.00 |
| lsi uned | 94.33 | lsi2 uned | 89.97 |
| jiangdehuan | 94.01 | vcotik | 89.67 |
| jimblair | 93.75 | VSP | 86.00 |
| **AutoGOAL** | **96.01** | | |

Table 4: Comparison of our proposal (AutoGOAL) with the official results of the MEDDOCAN corpus.

The results of this experiment are summarized in Table 4 and compared with official results from the MEDDOCAN challenge. Similar to the previous experiment, since each pipeline evaluation is considerably slow, a total runtime of $48$ hours was allocated for this experiment. The complexity of this challenge is higher than the previous experiments, since it involves token-level instead of sentence-level classification. However, in this scenario, the best pipeline found also uses a BERT transformer and a neural network with a Bi-LSTM and time-distributed dense layers. Despite its simplicity, this pipeline is highly competitive with hand-crafted solutions that contain several domain-specific heuristics, outperforming 13 out of 16 state-of-the-art approaches.

### 4.4 Experimental Resources and Computational Cost

The experimental comparisons in Section 4 were performed in a computational infrastructure consisting of a 16-core Intel i9-9900K CPU with a clock speed of 3.60GHz, 128 GB of total RAM memory and two NVIDIA Titan RTX GPUs with 48 GB of combined graphics memory.

In terms of the computational cost that AutoGOAL adds to the intrinsic cost of training and evaluating an arbitrary algorithm, we need to consider three stages in the pipeline optimization process. First, there is an initial setup cost associated with building the algorithm graph $G_A$ and the corresponding grammars. There is an additional cost associated to every pipeline evaluated, for sampling and bookkeeping purposes. Finally, there is a cost at the end of each generation in the optimization algorithm (i.e., after every batch of $n$ pipeline evaluations) for updating the probabilistic model.

Formally, the cost of building the algorithm graph $G_A$ is determined by the number $N$ of algorithms and the total number of inter-connections (i.e., nodes and edges). At most, this cost is $O(N^2)$ if all algorithms are used in a specific problem, which is never the case. In practice, for the current number of algorithms available, this process takes approximately 10 seconds on commodity hardware. The cost of training and evaluating each algorithm is determined by the intrinsic cost of executing the algorithm in the underlying library. The associated cost of invoking the library and feeding the training data is negligible. The cost of sampling a new pipeline is proportional to its length, which is also negligible compared to the actual training cost. Finally, the cost associated to updating the probabilistic model in each generation is proportional to the total number of hyperparameters under optimization, i.e., the total number of productions involved in all the algorithm grammars, which currently takes less than 1 second on commodity hardware. In general, the overhead cost of executing AutoGOAL is minimal compared to the inherent cost of training and evaluating each algorithm.

## 5 Discussion

Regarding the experimental results, it is important to note that our system is deployed using the same code in all the experiments (see Figure 3), varying only the definition of the input and output types. For example, setting up HAHA and MEDDOCAN challenges took approximately one and four hours respectively, fundamentally devoted to preparing the corpora in a suitable format. The total running time in our approach is considerably higher than running a single hand-crafted pipeline, since several different pipelines are trained and tested (e.g., 20 total hours on the HAHA dataset). However, the time necessary for setting up on a novel problem is considerably smaller than coding a hand-crafted solution from scratch, since several pre-made components are provided. Hence, there is a trade-off between computational cost and human cost, which allows researchers to focus more on the conceptualization and modeling, i.e., deciding what is the space of possible pipelines to evaluate, and less on actually implementing those pipelines.

AutoGOAL enables researchers and practitioners to quickly develop strong baselines in diverse machine learning problems. In some scenarios, the solution provided by ours and other AutoML systems could already be good enough. However, AutoML systems should not attempt only to replace human experts, but rather to serve as complementary tools that allow researchers to quickly obtain better baselines and insights on the most promising strategies. We envision a future in which humans and computers work together to solve the most challenging and complex problems in Artificial Intelligence, each bringing their core competences. Much like compilers brought a significant improvement in the efficiency of software development, we think AutoML opens the door to revolutionizing the way machine learning research and practice is performed.

## 6 Conclusions

This paper presents AutoGOAL, a system for Automatic Machine Learning (AutoML) that seamlessly combines heterogeneous technologies and can be applied to a wide variety of machine learning scenarios. Our proposal is competitive with other AutoML frameworks on standard benchmarks, and it can be applied to novel scenarios for which several existing AutoML tools are not directly applicable. An experimental evaluation in two different NLP problems shows that this system can obtain competitive results with human-designed pipelines, without any domain or problem-specific considerations. A computational prototype of this system with pre-packaged implementations of over 100 algorithms from popular machine learning frameworks is made available for the research community.

## Acknowledgements

**Funding:** This research has been supported by a Carolina Foundation grant in agreement with University of Alicante and University of Havana. Moreover, it has also been partially funded by both aforementioned universities, the Generalitat Valenciana (*Conselleria d'Educació, Investigació, Cultura i Esport*) and the Spanish Government through the projects LIVING-LANG (`RTI2018-094653-B-C22`) and SIIA (`PROMETEO/2018/089`, `PROMETEU/2018/089`).

## References

Luis Chiruzzo, S Castro, Mathias Etcheverry, Diego Garat, Juan José Prada, and Aiala Rosá. 2019. Overview of haha at iberlef 2019: Humor analysis based on human annotation. In *Proceedings of the Iberian Languages Evaluation Forum (IberLEF 2019). CEUR Workshop Proceedings, CEUR-WS, Bilbao, Spain (9 2019).*

François Chollet et al. 2015. Keras. `https://keras.io`.

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*, pages 246–261. Springer.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*.

Dheeru Dua and Casey Graff. 2017. UCI machine learning repository.

Suilan Estevez-Velarde, Yoan Gutiérrez, Andrés Montoyo, and Yudivián Almeida-Cruz. 2019. AutoML strategy based on grammatical evolution: A case study about knowledge discovery from text. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4356–4365, Florence, Italy, July. Association for Computational Linguistics.

Suilan Estévez-Velarde, Yoan Gutiérrez, Yudivián Almeida-Cruz, and Andrés Montoyo. 2020. General-purpose hierarchical optimisation of machine learning pipelines with grammatical evolution. *Information Sciences*, 543:58–71.

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*, pages 2962–2970.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1993. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer.

Haifeng Jin, Qingquan Song, and Xia Hu. 2018. Auto-Keras: Efficient Neural Architecture Search with network morphism.

Keyvan Khosrovian, Dietmar Pfahl, and Vahid Garousi. 2008. Gensim 2.0: a customizable process simulation model for software process evaluation. In *International conference on software process*, pages 294–306. Springer.

Brent Komer, James Bergstra, and Chris Eliasmith. 2014. Hyperopt-sklearn: automatic hyperparameter configuration for scikit-learn. In *ICML workshop on AutoML*, pages 2825–2830. Citeseer.

Alicia Lara-Clares and Ana Garcia-Serrano. 2019. Key phrases annotation in medical documents: Meddocan 2019 anonymization task.

Edward Loper and Steven Bird. 2002. NLTK: the natural language toolkit. *arXiv preprint cs/0205028*.

George A. Miller. 1995. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, November.

Felix Mohr, Marcel Wever, and Eyke Hüllermeier. 2018. ML-Plan: Automated machine learning via hierarchical planning. *Machine Learning*, 107(8):1495–1515, sep.

Randal S Olson and Jason H Moore. 2016. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on Automatic Machine Learning*, pages 66–74.

Michael O'Neill and Conor Ryan. 2001. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

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

Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855. ACM.