

Solving Heterogeneous AutoML Problems with AutoGOAL

Suilan Estevez-Velarde

SESTEVEZ@MATCOM.UH.CU

School of Math and Computer Science, University of Havana, Cuba

Alejandro Piad-Morffis

APIAD@MATCOM.UH.CU

School of Math and Computer Science, University of Havana, Cuba

Yoan Gutiérrez

YGUTIERREZ@DLSI.UA.ES

University Institute for Computing Research (IUII), DLSI, University of Alicante, Spain

Andrés Montoyo

MONTOMO@DLSI.UA.ES

University Institute for Computing Research (IUII), DLSI, University of Alicante, Spain

Rafael Muñoz

RAFAEL@DLSI.UA.ES

University Institute for Computing Research (IUII), DLSI, University of Alicante, Spain

Yudivian Almeida-Cruz

YUDY@MATCOM.UH.CU

School of Math and Computer Science, University of Havana, Cuba

Abstract

This paper introduces a web demo that showcases the main characteristics of the AutoGOAL framework. AutoGOAL is a framework in Python for automatically finding the best way to solve a given task. It has been designed mainly for automatic machine learning (AutoML) but it can be used in any scenario where several possible strategies are available to solve a given computational task. This paper presents an overview of the framework's design and an experimental evaluation in several machine learning problems. The accompanying software demo is available online and full source code is available under a FOSS license¹.

1. Introduction

The field of machine learning has applications across a wide range of computational problems in different domains. However, given the vast quantity of resources and technologies available, often one of the most difficult challenges is to select the best combination of them when a specific problem is faced. Making the right selection requires both technical expertise and experience in the problem domain. On one hand, technical expertise on managing different technologies allows experts to understand the strengths and limitations of each technique, while on the other hand, domain expertise provides key insights gained from previous experiences. Researchers often spend a significant amount of time and computational resources exploring multiple approaches in search of optimal configurations. This entails testing marginally different approaches, features and hyper-parameter values. The field of automatic machine learning (AutoML) has risen to prominence as a principled alternative for finding optimal or close to optimal solutions to complex machine learning problems (Hutter et al., 2018). Several software libraries have been created, which leverage existing machine learning technologies and provide AutoML features built on them. However, despite the recent success of AutoML, several challenges still remain.

1. <https://autogoal.github.io>

Most existing AutoML tools focus on a specific family of algorithms (such as neural networks) or a specific problem setting (such as supervised learning from tabular data). Furthermore, these tools are often designed as user-friendly interfaces to a back-end machine learning framework (e.g., the main components in *AutoSklearn* (Feurer et al., 2015) are based on the Transformer-Estimator API in *scikit-learn*). However, in practical scenarios, researchers need to combine technologies from different frameworks which are not always designed to interface with each other. Hence, even using modern AutoML tools, a significant effort is necessary to develop an interface code for connecting the output of algorithms from one machine learning framework into algorithms in a different framework.

This work presents AutoGOAL, a software library for AutoML that can seamlessly combine technologies and resources from different frameworks. To unify disparate APIs into a single interface, AutoGOAL proposes a novel graph-based representation for machine learning pipelines. Furthermore, a search strategy based on probabilistic grammatical evolution is used to discover optimal machine learning pipelines (Estevez-Velarde et al., 2019) whose components can be from different back-end libraries. The key features of AutoGOAL are:

Ease of use: AutoGOAL provides high-level classes that non-experts can use as black-box AutoML solutions, compatible with several data types including text, images and structured (tabular) data.

Multiple domains: AutoGOAL comes prepackaged with 133 plus adapters of existing algorithms, from 7 different back-end libraries, for multiple problems including text preprocessing, feature extraction, dataset augmentation, dimensionality reduction, as well as supervised and unsupervised learning techniques.

Extensibility: AutoGOAL proposes a simple programming interface that developers can implement to create an automatically discoverable adapter for an existing technology from any machine learning framework.

This paper focuses on the engineering design of AutoGOAL by introducing a demo application using the library to solve several machine learning problems, organised as follow: Section 2 describes AutoGOAL from the perspective of a user of the library. Section 3 describes the library’s design. Section 4 presents experimental results of the application of AutoGOAL to several different machine learning problems. Section 5 describes a demo application built with AutoGOAL. Finally, Section 6 presents the conclusions and recommendations for future work.

2. User Interaction

AutoGOAL can be used as a software library in the Python programming language. This library is oriented towards two machine learning user profiles: non-experts and experts. Non-expert users will interact with the High-Level API through a Python class —**AutoML**— that abstracts the complete AutoML process in a black-box interface. In the case of expert users, they will interact with the Low-Level API, by declaring in their own experimental setup the elements that are suitable for optimisation (e.g., valid ranges for parameter values, different strategies to evaluate).

```

from autogoal.ml import AutoML
from autogoal.datasets import haha
# import lines for semantic datatypes

automl = AutoML(
    # problem-specific input and output (semantic datatypes)
    input=List(Sentences()),
    output=CategoricalVector(),
    # additional parameters for timeout, memory, iterations, etc.
)

X, y = haha.load() # load problem-specific dataset
automl.fit(X, y)   # run optimisation

```

Figure 1: Example source code for running AutoGOAL on a specific dataset, in this case an NLP problem.

High-Level API (non-experts): This API allows AutoGOAL to be used as a black-box classification or regression algorithm with an interface similar to the *scikit-learn* library (Pedregosa et al., 2011). Behind this interface, a complete process including preprocessing, feature selection, dimensionality reduction, and learning is performed. The user must define a dataset for training and evaluation, a metric to optimise (which defaults to *accuracy*) and the type of input and output data. In many cases AutoGOAL can automatically infer the input and output type from the dataset. Input and output types can vary from tabular data to complex types such as images, natural language text with different semantic structures, and combinations thereof. Figure 1 shows an illustrative example source code, specifically in the context of a text classification problem.

Low-Level API (experts): This API is designed for users with more experience that need control over the AutoML process. For this type of user, AutoGOAL provides a simple language for defining a grammar that describes the solution space. This is done using an object oriented approach where the user defines a Python class for each component of the solution (e.g., each algorithm) and annotates the parameters of these classes with attributes that describe the space of possible values, which can be primitive value types (i.e., numeric, string, etc.) and instances of other classes, recursively. Based on the annotations, AutoGOAL can automatically construct all possible ways in which the user classes can be instantiated. An example code of this process is available in Appendix A.

3. Implementation details

This section presents the overall architecture of the AutoGOAL library. For reference purposes, Figure 2 illustrates the most relevant components of AutoGOAL, ranging from the High-Level API down to the actual implementation of algorithm adapters and the interfaces to external resources and back-end libraries.

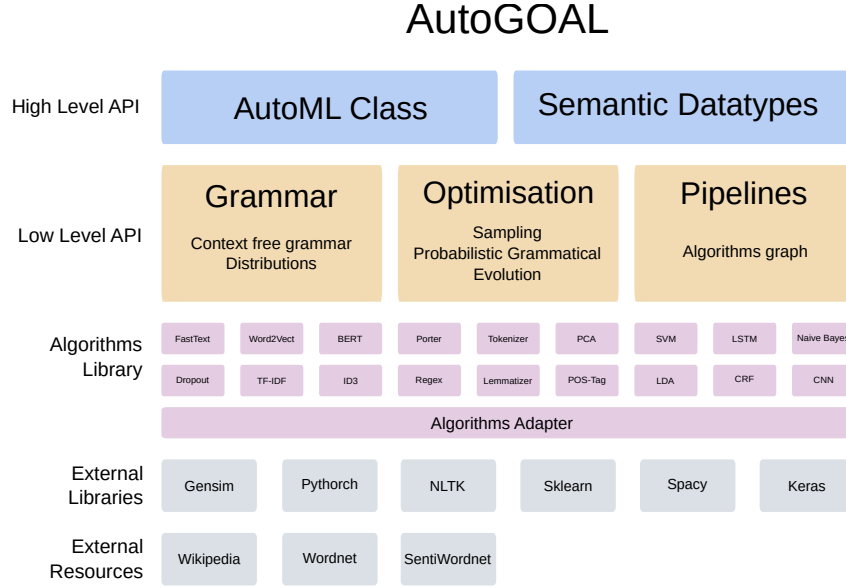


Figure 2: Overall architecture of the AutoGOAL framework.

The core of the AutoGOAL library is the Low-Level API, composed of the following elements (see Figure 2): a probabilistic context-free grammar module (*Grammar*); a sampling and optimisation module (*Optimisation*); and, a pipeline discovery module (*Pipelines*). This Low-Level API allows users to: provide their own implementations of machine learning algorithms; declare their optimisable elements (e.g., hyperparameters); and, define how they can be connected in complex multi-step pipelines.

The *Grammar* module provides a set of type annotations that are used for defining the hyperparameter space of an arbitrary technique or algorithm. Each technique is represented as a Python class, and the corresponding hyperparameters are represented as annotated arguments of the `__init__` method, either primitive values (e.g., numeric, string, etc.) or instances of other classes, recursively annotated. Given a collection of annotated classes, this module automatically infers a context-free grammar that describes the space of all possible instances of those classes.

The *Optimisation* module provides sampling strategies that traverse a context-free grammar and recursively construct one specific instance following the annotations. Two optimisation strategies are implemented: random search and probabilistic grammatical evolution (O’Neill and Ryan, 2001). The latter performs a sampling/update cycle that selects the best performing instances according to some predefined metric (e.g., accuracy on a development set) and iteratively updates the internal probabilistic model of the sampler.

The *Pipelines* module provides an abstraction for algorithms to communicate with each other via an Facade pattern, i.e., the implementation of a method `run` with type-annotated input and output. Classes implementing this pattern are automatically connected in a

graph of algorithms where each path represents a possible pipeline for solving a specific problem (defined by the input and output datatypes).

The High-Level API that provides the `AutoML` class (see Listing 1) and the *Semantic Datatypes* (see Appendix A) is built on top of this architecture, knitting together all the components of AutoGOAL. Users can interact with the High-Level API for a black-box AutoML solution, or can directly interact with the internal components for a more fine-grained control or when requiring custom-made AutoML solutions.

AutoGOAL also provides an Algorithms Library of pre-made adapters for existing machine learning technologies from External Libraries and Resources. A total of 133 algorithms from 7 different back-end libraries² are provided, several of which are semi-automatically created by code introspection, and the rest are manually added by the library developers. This library is undergoing continuous expansion. AutoGOAL can be installed as a Python package independently of any machine learning library. As such, it is a lightweight framework that provides all the construction blocks but none of the predefined adapters. Users can optionally install any one of the supported back-end libraries and AutoGOAL will automatically discover it and register the corresponding adapters, which will be available for use under the High-Level API. Additionally, a Docker image is provided with all optional dependencies and back-end libraries already installed³.

4. Evaluation

AutoGOAL has been evaluated in different domains and compared to other AutoML tools, including Auto-Weka (Thornton et al., 2013), TPOT (Olson and Moore, 2016), Auto-Sklearn (Feurer et al., 2015), and ML-Plan (Mohr et al., 2018), see Table 1. AutoGOAL is compared with other AutoML approaches in classic datasets (Dua and Graff, 2017) but can be applied to more complex domains such as text classification in Haha (Chiruzzo et al., 2019) and entity recognition in MEDDOCAN (Lara-Clares and Garcia-Serrano, 2019).

In terms of performance, AutoGOAL achieves comparative results with other AutoML tools in classic datasets. However, AutoGOAL’s main strength lies in its ability to combine different tools for solving complex problems beyond structured supervised learning. In addition to structured datasets, AutoGOAL can be applied seamlessly to natural language processing, using virtually the same code, by specifying the input and output datatypes. In these domains AutoGOAL performs comparable to state-of-the-art solutions hand-crafted by human experts, while requiring considerably less expertise and effort.

5. Demo description

To show the simplicity and versatility of AutoGOAL, an online demo application is provided⁴. It is important to bear in mind that AutoGOAL is a source code library and not an application with a user interface. The application shown in this section is simply a demonstrative example of the sort of systems that could be easily built on top of AutoGOAL, and at the same time serves as an interactive introduction to the library’s main concepts and use cases.

2. Including *scikit-learn*, *nlTK*, *gensim*, *spacy*, *keras*, *pytorch*, among others.

3. <https://hub.docker.com/repository/docker/autogoal/autogoal>

4. <https://autogoal.github.io/demo>

Dataset	Cars	Credit G.	Abalone	Shuttle	Yeast	Dorothea	Gisette	HAHA	MEDD.
ML-Plan (Weka)	1.27	25.54	73.72	0.01	39.37	6.49	2.92	-	-
Auto-WEKA	0.66	26.50	73.46	0.12	39.72	-	3.90	-	-
ML-Plan (Sklearn)	0.34	24.56	73.77	0.02	39.52	8.69	2.76	-	-
Auto-Sklearn-v	1.38	25.95	82.92	0.02	40.51	6.32	2.56	-	-
Auto-Sklearn-we	1.26	25.39	80.59	0.02	38.99	6.02	2.24	-	-
TPOT	0.37	23.91	73.14	0.02	38.47	-	-	-	-
AutoGOAL	0.60	27.01	74.33	0.11	39.94	5.97	2.25	21.1	3.99

Table 1: Comparison of AutoGOAL and other AutoML systems for 9 classic machine learning datasets in terms of accuracy except for MEDDOCAN (F_1). Values for other systems were obtained from Mohr et al. (2018).

The demo is divided in two different sections. To illustrate how to apply AutoGOAL as a black box AutoML solution in a variety of problems, the first section shows how to apply AutoGOAL to each of the machine learning problems described in section 4. The second section shows several internal details of AutoGOAL, allowing the researcher to explore the array of built-in custom implementations in the library and understand their internal structure. Due to space restrictions, screenshots of the demo are available in Appendix C.

6. Conclusion

In this paper we presented AutoGOAL, a new tool for AutoML that allows resources from different machine learning libraries to be combined and applied to different domains with little effort. AutoGOAL greatly simplifies the application of machine learning for non-expert users while providing powerful low-level components for experts to effectively optimise complex machine learning pipelines. The framework has been designed with extensibility as a priority, enabling the addition of new algorithms from any conceivable machine learning library by conforming to a simple interface. To demonstrate its usefulness, AutoGOAL is applied to different domains—including classic numeric datasets, text classification, and entity recognition—achieving competitive results with the state of the art. The software is provided freely for the research community along with a vast library of algorithms already implemented.

Acknowledgements

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. 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)*, 2019.
- Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL <http://archive.ics.uci.edu/ml>.
- Suilan Estevez-Velarde, Yoan Gutiérrez, Andrés Montoyo, and Yudivián Almeida-Cruz. 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 2019. Association for Computational Linguistics. URL <https://www.aclweb.org/anthology/P19-1428>.
- Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.
- Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors. *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2018. In press, available at <http://automl.org/book>.
- Alicia Lara-Clares and Ana Garcia-Serrano. Key phrases annotation in medical documents: Meddocan 2019 anonymization task. 2019.
- Felix Mohr, Marcel Wever, and Eyke Hüllermeier. ML-Plan: Automated machine learning via hierarchical planning. *Machine Learning*, 107(8):1495–1515, sep 2018. ISSN 1573-0565. doi: 10.1007/s10994-018-5735-z. URL <https://doi.org/10.1007/s10994-018-5735-z>.
- Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on Automatic Machine Learning*, pages 66–74, 2016.
- Michael O’Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.
- 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. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 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, 2013.

Appendix A. Example usage of the Low-Level API

This section illustrates how to use the Low-Level API to define new adapters that can be automatically used by AutoGOAL.

The Low-Level API provides utilities to annotate the types of parameters in class constructors and methods, thus indicating the valid range for their values. The user defines class adapters for a back-end component to AutoGOAL’s API, e.g., an algorithm from *scikit-learn*. As an example, the following listing shows the definition of classes that wrap *scikit-learn* algorithms and define the hyperparameter search space by annotating the parameters of interest in the class constructor.

```
class LR(sklearn.linear_model.LogisticRegression):
    def __init__(
        self,
        penalty: Categorical("l1", "l2"),
        C: Continuous(0.1, 10)
    ):
        super().__init__(penalty=penalty, C=C)

    # TODO: Implementation of the 'run' method
    # ...

class SVM(sklearn.svm.SVC):
    # ...

class DecisionTree(sklearn.tree.DecisionTreeClassifier):
    # ...
```

Additionally, each class must define a `run` method, where the input and output values are annotated with semantic types (see Appendix B). This allows AutoGOAL to detect which components can be connected. Supervised learning algorithms from *scikit-learn* receive as input both the feature matrix and the classes (during training). The `run` methods in these classes act as an adapter between AutoGOAL’s API and the *scikit-learn* API. A simplified implementation is shown in the following listing.

```
class LR(sklearn.linear_model.LogisticRegression):
    # ...
    def run(self, input: Tuple(MatrixContinuous, CategoricalVector))
        -> CategoricalVector:
        # code to account for training vs. evaluation mode
        # not shown for simplicity
        if training:
            X, y = input
            self.fit(X, y)
            return y
        else:
            return self.predict(X)
```

This API can be used at any level of detail. For example, an algorithm for obtaining *word2vec* representations from individual words, using *gensim*, can be implemented in the same manner. Notice that this algorithm runs at the level of tokens, contrary to the *scikit-learn* adapters, which run at the level of a complete dataset.


```

class Word2VecEmbedding:
    def __init__(self):
        # load word2vec model from gensim API
        self.model = gensim.downloader.load("glove-twitter-25")

    def run(self, input: Word) -> ContinuousVector:
        try:
            return self.model.get_vector(input)
        except:
            return np.zeros(25)

```

As a final example, this API can also represent feature extraction techniques that use external resources, such as Wikipedia, WordNet, etc. The following listing shows the implementation of a Wikipedia summary extractor, which uses a Python library to access Wikipedia, search for a word and return the summary of the first match.

```

class WikipediaSummary:
    def run(self, input: Word)-> Summary:
        try:
            return wikipedia.summary(input)
        except:
            return ""

```

The previous examples are just illustrative of the variety of tasks that can be represented in AutoGOAL via the Low-Level API. AutoGOAL contains hundreds of implementations similar to the ones shown in the previous examples. A pipeline for a specific problem is constructed by connecting several algorithms with compatible input/output types, where compatibility is determined by the inheritance relationship in the Semantic Datatype hierarchy (see Appendix B).

If all the intermediate components are available, AutoGOAL can automatically find pipelines for a variety of input/output types. The `AutoML` class can be customised to define a list of possible classes (which adhere to the `run` protocol) to use, instead of defaulting to using the entire library of AutoGOAL's adapters. This allows the user to exclude some algorithms and include novel adapters implemented from custom back-end libraries. AutoGOAL also allows the specification of a resource budget (i.e., time and memory) for each pipeline and for the whole optimisation process, different score metrics, and other parameters.

```

automl = AutoML(
    input= # ....
    output= # ...
    # explicitly define the list of available algorithms
    registry = [
        LR, SVM, DecisionTree, Word2VecEmbedding, # ...
    ]
    # fine-grained configuration of the experimental setup
    score_metric=f1_score, # custom evaluation metric
    search_kwargs=dict(
        search_timeout=3600,
        evaluation_timeout=60,
        memory_limit=1024 ** 3,
    )
)

```

One of the most powerful characteristics of AutoGOAL is that it can automatically perform tuple and list construction and deconstruction. To illustrate the importance of this feature, consider that *scikit-learn*-based adapters require feature matrices defined at a dataset level, but tokenizers work at sentence level and the *word2vec* encoder works at token level. This is however not a problem since AutoGOAL is capable of automatically “lifting” an algorithm with input **Tin** and output **Tout** to an algorithm of input **List(Tin)** and output **List(Tout)**. Hence, if the input is **List(Sentence)**, then algorithms that work on sentence level, such as tokenizers, will be automatically lifted to work on lists of sentences, obtaining as output a **List(List(Word))**. Each internal token can then be converted to a *word2vec* representation and the lists of vectors can be automatically assembled into feature matrices. This mechanism is completely transparent to the user.

Appendix B. Semantic datatypes

The *Semantic Datatype* hierarchy is a collection of Python classes that represent all possible input and output types in AutoGOAL. The collection is extensible, allowing the user to define new semantic types when necessary. These classes don’t hold any data and are not instantiated. They are only used to annotate the input and output types of **run** methods, allowing the *Pipeline* module (see Figure 2) to detect valid pipelines. Compatibility among types is determined by the inheritance relationship in this hierarchy. For example, if an algorithm *A* produces **MatrixContinuousDense** and another algorithm *B* receives **Matrix**, then the output of *A* can be passed to the input of *B*, which means that those two algorithms can be used in a pipeline. Figure 3 shows the current hierarchy.

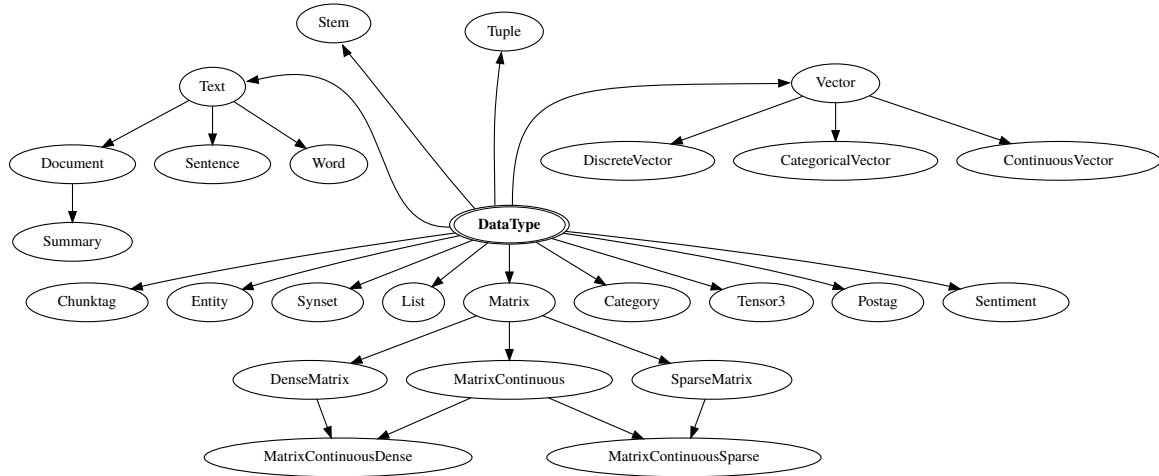


Figure 3: Diagram of Semantic Datatypes in the AutoGOAL framework.

Appendix C. Screenshots of the demo application

The demo presented with this paper is a showcase of AutoGOAL’s main characteristics. The following images show screenshots of different features of the demo. Figure 4 shows the first section of the demo, where the user can select one of several included datasets and apply the

black-box AutoML solver. Figure 5 shows the optimisation process during execution. Figure 6 shows the second section where the user can browse the Algorithms Library. Finally, Figure 7 shows a graph representing all the pipelines for a specific input/output combination.

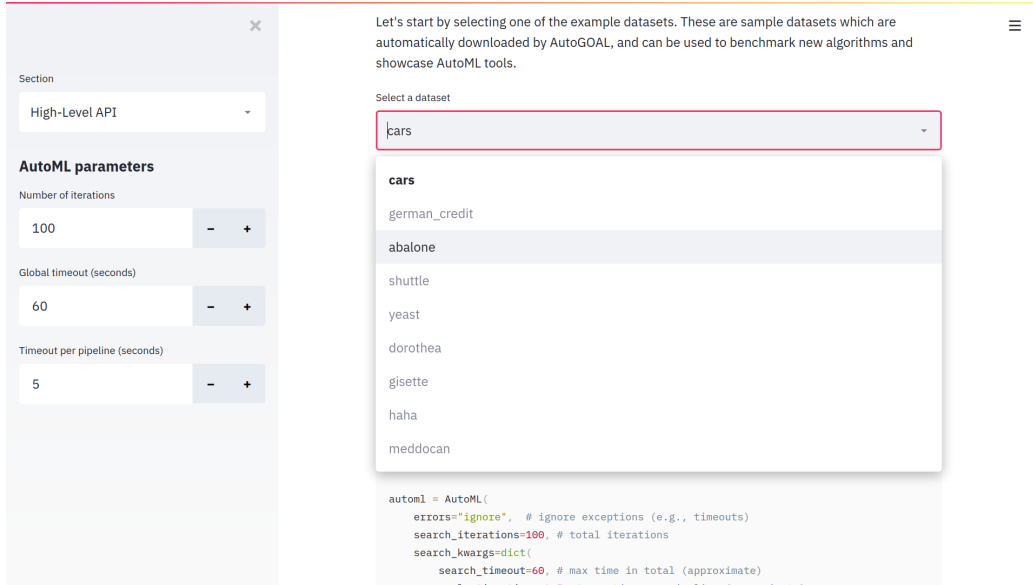


Figure 4: Screenshot of the AutoGOAL demo, selecting datasets.

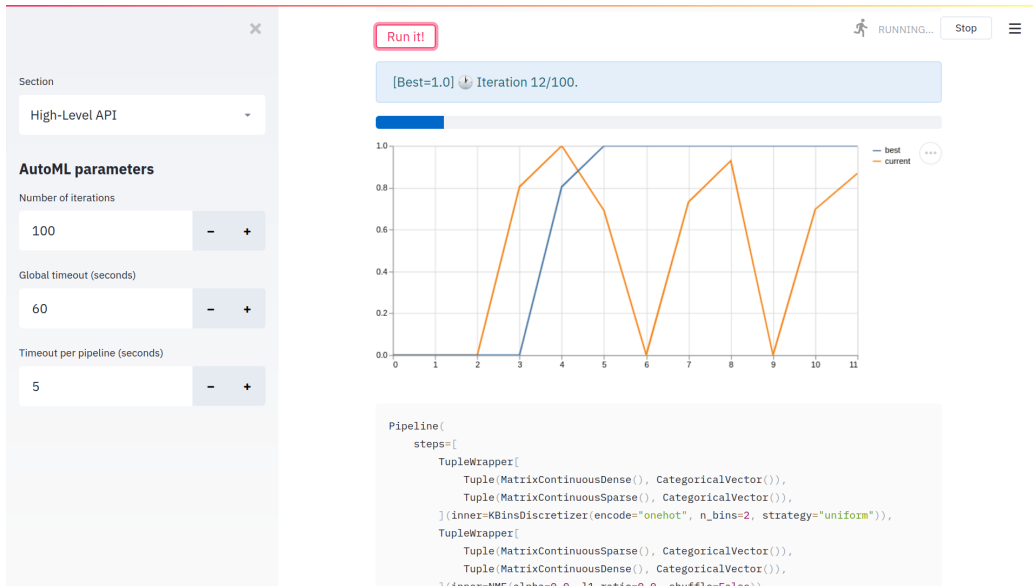
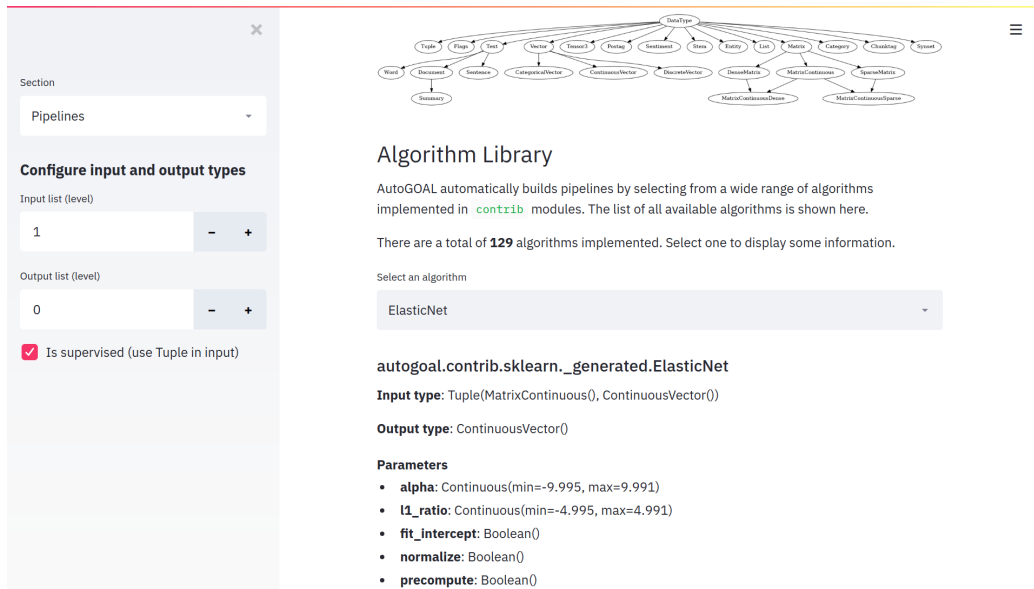


Figure 5: Screenshot of the AutoGOAL demo, optimisation process.



Section
Pipelines

Configure input and output types

Input list (level)
1

Output list (level)
0

☒ Is supervised (use Tuple in input)

Algorithm Library

AutoGOAL automatically builds pipelines by selecting from a wide range of algorithms implemented in `contxib` modules. The list of all available algorithms is shown here.

There are a total of **129** algorithms implemented. Select one to display some information.

Select an algorithm
ElasticNet

autogoal.contrib.sklearn._generated.ElasticNet

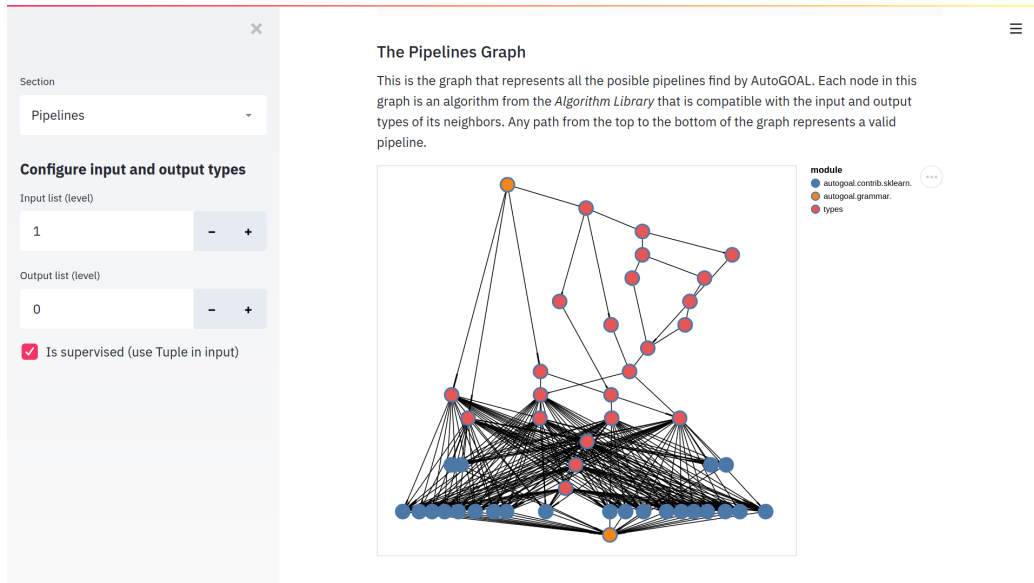
Input type: Tuple(MatrixContinuous(), ContinuousVector())

Output type: ContinuousVector()

Parameters

- alpha:** Continuous(min=-9.995, max=9.991)
- l1_ratio:** Continuous(min=-4.995, max=4.991)
- fit_intercept:** Boolean()
- normalize:** Boolean()
- precompute:** Boolean()

Figure 6: Screenshot of the AutoGOAL demo, exploring the Algorithms Library.



Section
Pipelines

Configure input and output types

Input list (level)
1

Output list (level)
0

☒ Is supervised (use Tuple in input)

The Pipelines Graph

This is the graph that represents all the posible pipelines find by AutoGOAL. Each node in this graph is an algorithm from the *Algorithm Library* that is compatible with the input and output types of its neighbors. Any path from the top to the bottom of the graph represents a valid pipeline.

module

- autogoal.contrib.sklearn
- autogoal.grammar
- types

Figure 7: Screenshot of the AutoGOAL demo, exploring a graph of pipelines.