

Algorithm Chains and Pipelines

ESM3081 Programming for Data Science

Seokho Kang



Algorithm Chains and Pipelines

- **Most machine learning applications require not only the application of a single algorithm, but the chaining together of many different processing steps and machine learning models.**
 - *e.g.*, feature scaling, feature engineering, ...
- **We will cover how to use the *Pipeline* class to simplify the process of building chains of transformations and models.**
 - Combining *Pipeline* and *GridSearchCV* to search over hyperparameters for all processing steps at once.

Topics

- **Pipeline Interface**
- **Automatic Model Selection**
- **AutoML**

Building Pipelines

Hyperparameter Selection with Preprocessing

- A naïve approach to search over the hyperparameters for *SVC* using *GridSearchCV*
 - When scaling the data, we used all the data points in the training set to obtain the scaling criterion.
 - We then used the scaled training data to run grid search using cross-validation.

scikit-learn Practice

• Example (*iris* dataset) with data scaling

```
In [23]: from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, StratifiedKFold, GridSearchCV
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

iris = load_iris()
X_trainval, X_test, y_trainval, y_test = train_test_split(
    iris.data, iris.target, test_size=0.25, random_state=0)

scaler = StandardScaler()
scaler.fit(X_trainval)
X_trainval_scaled = scaler.transform(X_trainval)
X_test_scaled = scaler.transform(X_test)

In [24]: kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=2)
hyperparam_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
                    'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
grid_search = GridSearchCV(SVC(), hyperparam_grid, scoring='accuracy', refit=True, cv=kfold)
grid_search.fit(X_trainval_scaled, y_trainval)

print('Best score on validation set: {:.5f}'.format(grid_search.best_score_))
print('Best hyperparameters: {}'.format(grid_search.best_params_))

Best score on validation set: 0.97312
Best hyperparameters: {'C': 10, 'gamma': 0.1}

In [25]: y_test_hat = grid_search.predict(X_test_scaled)
test_score = accuracy_score(y_test, y_test_hat)
print('Test set score with best hyperparameters: {:.5f}'.format(test_score))

Test set score with best hyperparameters: 0.97368
```

The data scaler is fitted on *X_trainval*.

We specify the hyperparameter candidates to search over using a dictionary.

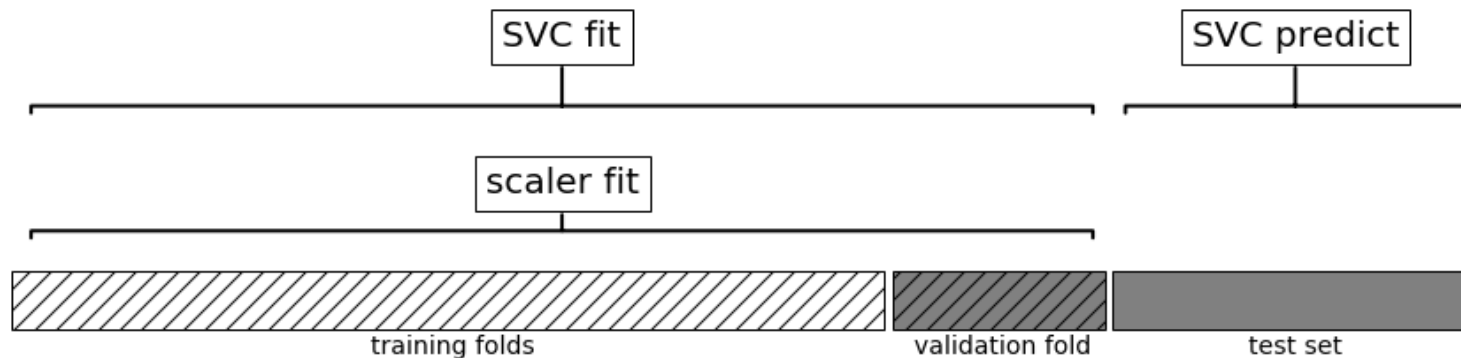
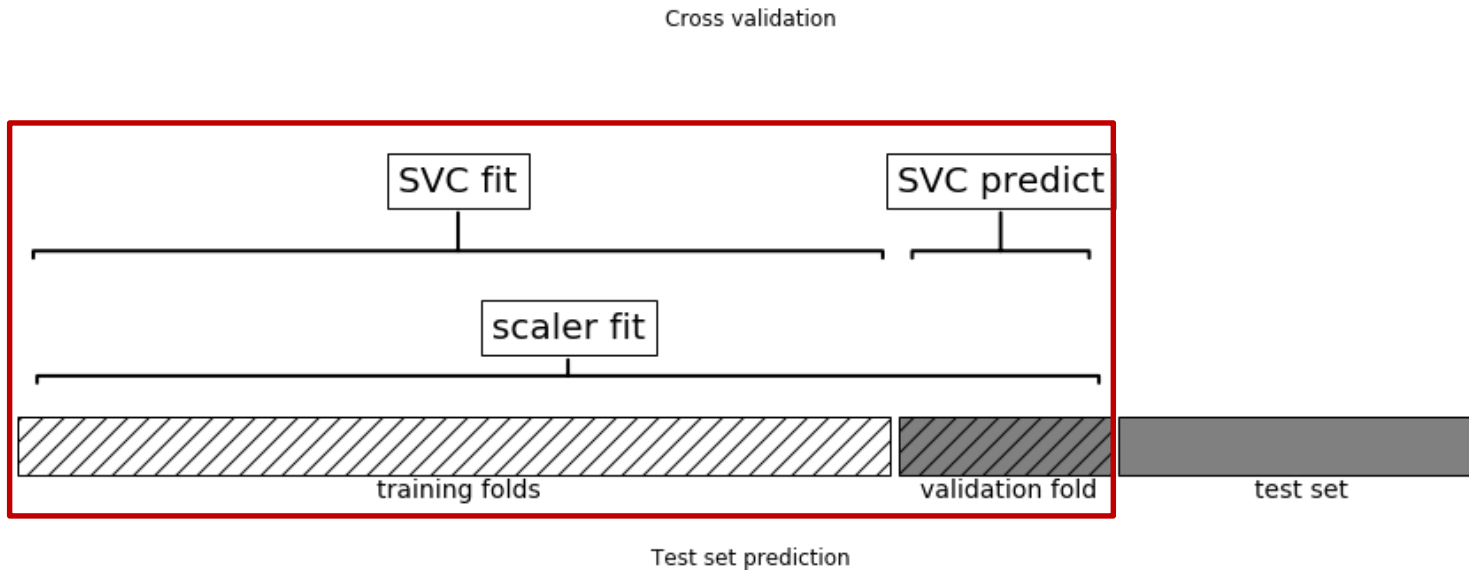
The *predict* method employs the model trained on both the training and validation data.

GridSearchCV allows the *hyperparam_grid* to be a list of dictionaries. e.g.,

```
param_grid = [{'kernel': 'rbf', 'C': [0.001, 0.01, 0.1, 1, 10, 100],
               'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
               {'kernel': 'linear', 'C': [0.001, 0.01, 0.1, 1, 10, 100]}]
```

Hyperparameter Selection with Preprocessing

- Data usage when preprocessing outside the cross-validation loop (**Improper Preprocessing**)



Pipelines

- **We used the information from the entire training set, including the validation part, to find the right scaling of the data.**
- **This is fundamentally different from how new data looks to the model.**
 - The splits in the cross-validation no longer correctly mirror how new data will look to the modeling process.
 - This will lead to overly optimistic results during cross-validation, and possibly the selection of suboptimal hyperparameters.
- **To get around this problem, the splitting of the dataset during cross-validation should be done before doing any preprocessing.**
 - To achieve this, we can use the *Pipeline* class.

scikit-learn Practice

<https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html>

sklearn.pipeline.Pipeline

```
class sklearn.pipeline.Pipeline(steps, *, memory=None, verbose=False)
```

[\[source\]](#)

Pipeline of transforms with a final estimator.

Sequentially apply a list of transforms and a final estimator. Intermediate steps of the pipeline must be 'transforms', that is, they must implement fit and transform methods. The final estimator only needs to implement fit. The transformers in the pipeline can be cached using `memory` argument.

The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters. For this, it enables setting parameters of the various steps using their names and the parameter name separated by a '_', as in the example below. A step's estimator may be replaced entirely by setting the parameter with its name to another estimator, or a transformer removed by setting it to 'passthrough' or `None`.

Read more in the [User Guide](#).

New in version 0.5.

Parameters:

steps : list

List of (name, transform) tuples (implementing fit/transform) that are chained, in the order in which they are chained, with the last object an estimator.

memory : str or object with the joblib.Memory interface, default=None

Used to cache the fitted transformers of the pipeline. By default, no caching is performed. If a string is given, it is the path to the caching directory. Enabling caching triggers a clone of the transformers before fitting. Therefore, the transformer instance given to the pipeline cannot be inspected directly. Use the attribute `named_steps` or `steps` to inspect estimators within the pipeline. Caching the transformers is advantageous when fitting is time consuming.

scikit-learn Practice

<https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html>

Methods

<code>decision_function</code> (self, X)	Apply transforms, and decision_function of the final estimator
<code>fit</code> (self, X[, y])	Fit the model
<code>fit_predict</code> (self, X[, y])	Applies fit_predict of last step in pipeline after transforms.
<code>fit_transform</code> (self, X[, y])	Fit the model and transform with the final estimator
<code>get_params</code> (self[, deep])	Get parameters for this estimator.
<code>predict</code> (self, X, <code>**predict_params</code>)	Apply transforms to the data, and predict with the final estimator
<code>predict_log_proba</code> (self, X)	Apply transforms, and predict_log_proba of the final estimator
<code>predict_proba</code> (self, X)	Apply transforms, and predict_proba of the final estimator
<code>score</code> (self, X[, y, sample_weight])	Apply transforms, and score with the final estimator
<code>set_params</code> (self, <code>**kwargs</code>)	Set the parameters of this estimator.

scikit-learn Practice

- Example (*iris* dataset) with data scaling

```
In [2]: from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

iris = load_iris()
X_trainval, X_test, y_trainval, y_test = train_test_split(
    iris.data, iris.target, test_size=0.25, random_state=0)
```

We use the Pipeline class to express the workflow for training an SVM after scaling the data with StandardScaler

```
In [3]: from sklearn.pipeline import Pipeline

pipe = Pipeline([('scaler', StandardScaler()), ('svm', SVC())])
```

```
In [4]: from sklearn.model_selection import StratifiedKFold, GridSearchCV

kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=2)
hyperparam_grid = {'svm__C': [0.001, 0.01, 0.1, 1, 10, 100],
                    'svm__gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
grid_search = GridSearchCV(pipe, hyperparam_grid, scoring='accuracy', refit=True, cv=kfold)
grid_search.fit(X_trainval, y_trainval)

print("Best score on validation set: {:.5f}".format(grid_search.best_score_))
print("Best hyperparameters: {}".format(grid_search.best_params_))
```

Using a pipeline in a grid search works the same way as using any other estimator.

```
Best score on validation set: 0.97312
Best hyperparameters: {'svm__C': 10, 'svm__gamma': 0.1}
```

When defining the hyperparameter grid, we need to specify for each hyperparameter which step of the pipeline it belongs to.

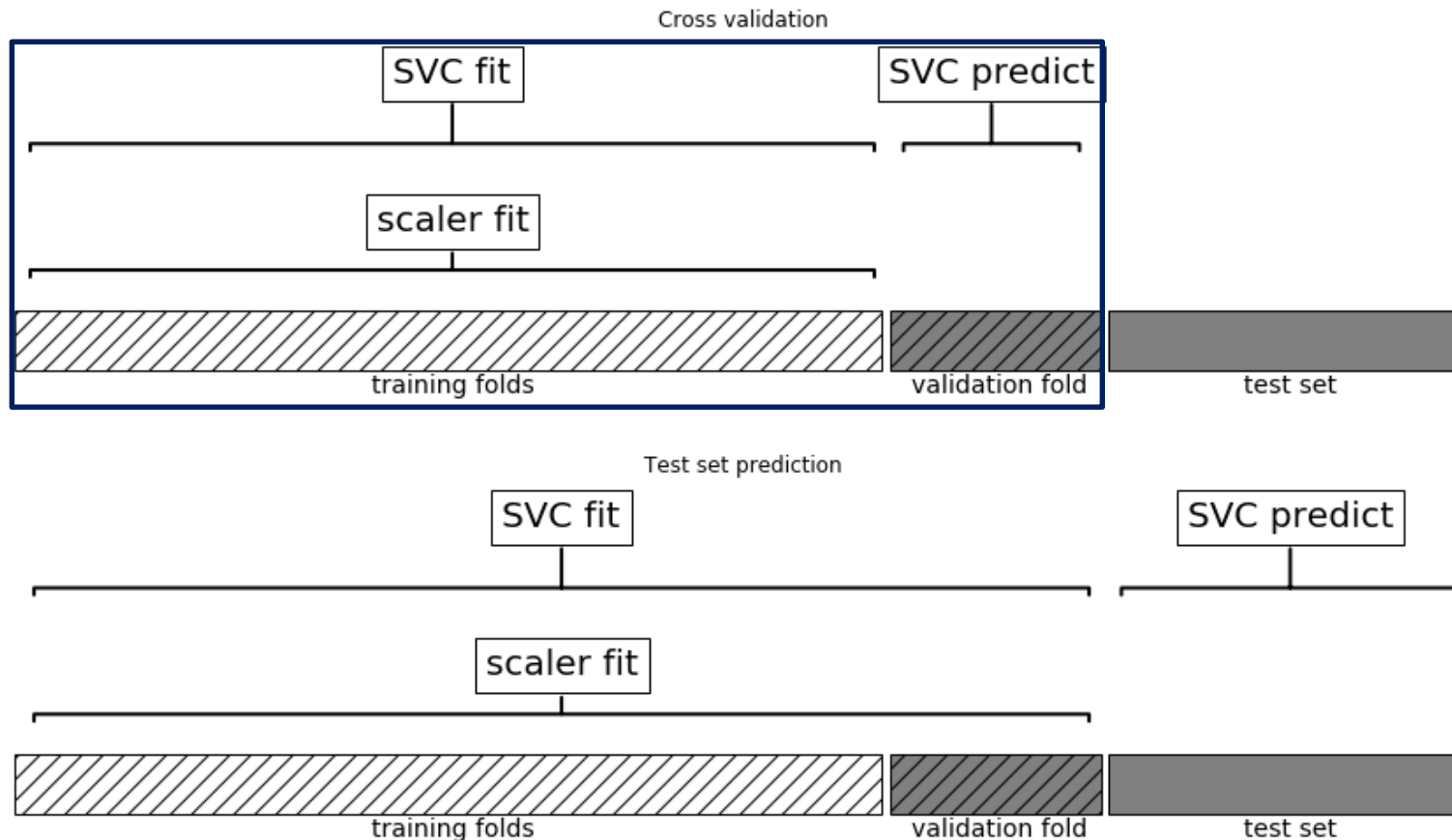
Syntax: the step name, followed by __ (a double underscore), followed by the hyperparameter name

```
In [5]: y_test_hat = grid_search.predict(X_test)
test_score = accuracy_score(y_test, y_test_hat)
print("Test set score with best hyperparameters: {:.5f}".format(test_score))
```

```
Test set score with best hyperparameters: 0.97368
```

Discussion

- Data usage when preprocessing inside the cross-validation loop with a pipeline



Discussion

- Using the pipeline, we can encapsulate all the processing steps in the machine learning workflow in a single estimator.
- We can reduce the code needed for our “preprocessing + classification” process.
- In contrast to the grid search we did before, now for each split in the cross-validation, the scaler is refit with only the training part.

General Pipeline Interface

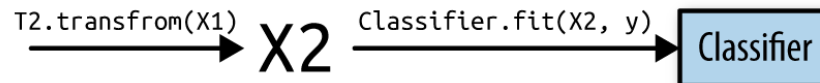
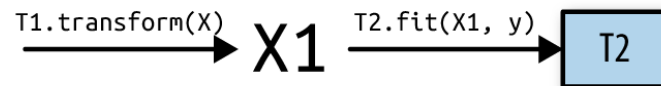
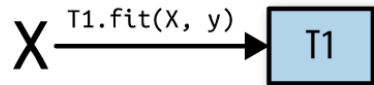
Pipeline Interface

- The **Pipeline** class is not restricted to preprocessing and classification, but can in fact join any number of estimators together.
 - For example, you could build a pipeline containing feature engineering, feature selection, feature scaling, and classification, for a total of four steps.
 - The last step could be regression or clustering instead of classification.
- **The only requirement** for estimators in a pipeline is
 - All but the last step need to have a transform method, so they can produce a new representation of the data that can be used in the next step.
 - The last step of a pipeline is only required to have a fit method.

Pipeline Interface

- Overview of the pipeline training and prediction process
 - **Training process:** the pipeline calls fit and then transform on each step in turn, with the input given by the output of the transform method of the previous step. For the last step in the pipeline, just fit is called.

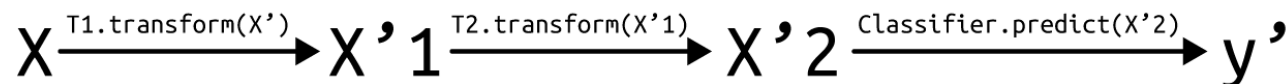
pipe.fit(X, y)



Example with two transformers, T1 and T2, and a classifier

- **Prediction process:** the pipeline transforms the data using all but the last step, and then predicts on the last step.

pipe.predict(X')



Pipeline Creation

- There is a convenience function *make_pipeline* that will create a pipeline for us and automatically name each step based on its class.
 - **Example syntax for *make_pipeline***

```
from sklearn.pipeline import make_pipeline
# standard syntax
pipe_long = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC(C=100))])
# abbreviated syntax
pipe_short = make_pipeline(MinMaxScaler(), SVC(C=100))
```

```
print("Pipeline steps:\n{}".format(pipe_short.steps))
```

Pipeline steps:

```
[('minmaxscaler', MinMaxScaler(copy=True, feature_range=(0, 1))),
 ('svc', SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
            decision_function_shape='ovr', degree=3, gamma='auto',
            kernel='rbf', max_iter=-1, probability=False,
            random_state=None, shrinking=True, tol=0.001,
            verbose=False))]
```


Accessing Attributes in a Pipeline

- You can inspect attributes of one of the steps of the pipeline.
 - *pipeline.steps* is a list of tuples, so *pipeline.steps[0][1]* is the first estimator, *pipeline.steps[1][1]* is the second estimator, and so on.
 - The easiest way to access each step in a pipeline is via the *named_steps* attribute, which is a dictionary from the step names to the estimators.
 - **Example syntax for *named_step* attribute**

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
```

```
pipe = make_pipeline(StandardScaler(), PCA(n_components=2), StandardScaler())
print("Pipeline steps:\n{}".format(pipe.steps))
```

Pipeline steps:

```
[('standardscaler-1', StandardScaler(copy=True, with_mean=True, with_std=True)),
 ('pca', PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
            svd_solver='auto', tol=0.0, whiten=False)),
 ('standardscaler-2', StandardScaler(copy=True, with_mean=True, with_std=True))]
```

fit the pipeline defined before to the cancer dataset

```
pipe.fit(cancer.data)
```

extract the first two principal components from the "pca" step

```
components = pipe.named_steps["pca"].components_
```

```
print("components.shape: {}".format(components.shape))
```

```
components.shape: (2, 30)
```

Automatic Model Selection

Grid-Searching Which Model To Use

- One of the main reasons to use Pipeline is for doing grid searches.
- Combining GridSearchCV and Pipeline makes it possible to search over the actual steps being performed in the pipeline.
 - This leads to an even bigger search space.
 - Trying all possible solutions is usually not a viable machine learning strategy, thus it should be considered carefully.
 - **Example:** Comparing an SVC, an MLPClassifier and a RandomForestClassifier
 - The SVC might need the data to be scaled, so we also search over whether to use StandardScaler, MinMaxScaler or None(no preprocessing).
 - The MLPClassifier might also need the data to be scaled.
 - For the RandomForestClassifier, we know that no preprocessing is necessary.
 - Because they have different hyperparameters to tune, and need different preprocessing, we can make use of the list of search grids

scikit-learn Practice

- Example (*breast_cancer* dataset)

```
In [6]: from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedKFold, GridSearchCV
```

```
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_state=0)
```

We start by defining the pipeline. We can instantiate this pipeline using any estimators.

```
In [7]: pipe = Pipeline([('preprocessing', None), ('classifier', SVC())])
hyperparam_grid = [
    {'classifier': [SVC()], 'preprocessing': [StandardScaler(), MinMaxScaler(), None],
     'classifier__gamma': [0.001, 0.01, 0.1, 1, 10, 100],
     'classifier__C': [0.001, 0.01, 0.1, 1, 10, 100]},
    {'classifier': [MLPClassifier(solver='lbfgs')], 'preprocessing': [StandardScaler(), MinMaxScaler(), None],
     'classifier__hidden_layer_sizes': [(10,), (20,), (50,), (100,)],
     'classifier__activation': ['tanh', 'relu']},
    {'classifier': [RandomForestClassifier(n_estimators=100)],
     'preprocessing': [None], 'classifier__max_features': [1, 2, 3]}]
kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
```

We make use of the list of search grids.

```
grid = GridSearchCV(pipe, hyperparam_grid, scoring='accuracy', refit=True, cv=kfold)
grid.fit(X_train, y_train)
```

```
print("Best hyperparams:\n{}".format(grid.best_params_))
print("Best cross-validation score: {:.5f}".format(grid.best_score_))
print("Test-set score: {:.5f}".format(grid.score(X_test, y_test)))
```

Now we instantiate and run the grid search as usual.

scikit-learn Practice

- Example (*breast_cancer* dataset)

```
Best hyperparams: {  
    'classifier': SVC(C=10, gamma=0.01),  
        'classifier__C': 10,  
        'classifier__gamma': 0.01,  
    'preprocessing': StandardScaler()  
}
```

Best cross-validation score: 0.97882

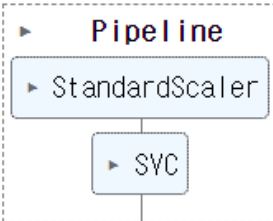
Test-set score: 0.97902

scikit-learn Practice

- Example (*breast_cancer* dataset)
 - Accessing Attributes in a Pipeline

```
In [8]: grid.best_estimator_
```

```
Out[8]:
```

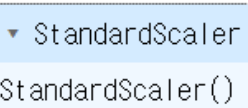


```
  ▸ Pipeline
    ▸ StandardScaler
      ▸ SVC
```

The diagram shows a dashed box labeled 'Pipeline'. Inside, there is a box labeled 'StandardScaler' which is connected by a vertical line to a box labeled 'SVC'.

```
In [9]: grid.best_estimator_.named_steps['preprocessing']
```

```
Out[9]:
```

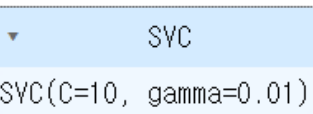


```
▼ StandardScaler
StandardScaler()
```

The dropdown menu shows 'StandardScaler' with a downward arrow. Below it, the text 'StandardScaler()' is visible.

```
In [10]: grid.best_estimator_.named_steps['classifier']
```

```
Out[10]:
```



```
▼ SVC
SVC(C=10, gamma=0.01)
```

The dropdown menu shows 'SVC' with a downward arrow. Below it, the text 'SVC(C=10, gamma=0.01)' is visible.

The best model found by GridSearchCV is stored in *grid.best_estimator_*.

We can access each step in a pipeline via the *named_steps* attribute.

scikit-learn Practice

- Example (*iris* dataset)

```
In [11]: from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedKFold, GridSearchCV

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, random_state=0)
```

We start by defining the pipeline. We can instantiate this pipeline using any estimators.

```
In [12]: pipe = Pipeline([('preprocessing', None), ('classifier', SVC())])
hyperparam_grid = [
    {'classifier': [SVC()], 'preprocessing': [StandardScaler(), MinMaxScaler(), None],
     'classifier__gamma': [0.001, 0.01, 0.1, 1, 10, 100],
     'classifier__C': [0.001, 0.01, 0.1, 1, 10, 100]},
    {'classifier': [MLPClassifier(solver='lbfgs')], 'preprocessing': [StandardScaler(), MinMaxScaler(), None],
     'classifier__hidden_layer_sizes': [(10,), (20,), (50,), (100,)],
     'classifier__activation': ['tanh', 'relu']},
    {'classifier': [RandomForestClassifier(n_estimators=100)],
     'preprocessing': [None], 'classifier__max_features': [1, 2, 3]}]
kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)

grid = GridSearchCV(pipe, hyperparam_grid, scoring='accuracy', refit=True, cv=kfold)
grid.fit(X_train, y_train)

print('Best estimator:\n{}'.format(grid.best_estimator_))
print('Best hyperparams:\n{}'.format(grid.best_params_))
print('Best cross-validation score: {:.5f}'.format(grid.best_score_))
print('Test-set score: {:.5f}'.format(grid.score(X_test, y_test)))
```

We make use of the list of search grids.

Now we instantiate and run the grid search as usual.

scikit-learn Practice

- Example (*iris* dataset)

```
Best hyperparams:{  
    'classifier': SVC(C=100, gamma=0.01),  
        'classifier__C': 100,  
        'classifier__gamma': 0.01,  
    'preprocessing': None  
}
```

Best cross-validation score: 0.98221

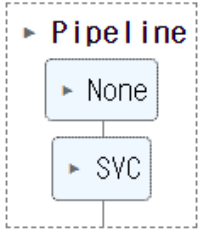
Test-set score: 0.97368

scikit-learn Practice

- **Example (*iris* dataset)**
 - Accessing Attributes in a Pipeline

```
In [13]: grid.best_estimator_
```

```
Out[13]:
```

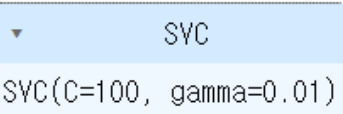


```
  Pipeline
  └── None
      SVC
```

```
In [14]: grid.best_estimator_[0]
```

```
In [15]: grid.best_estimator_[1]
```

```
Out[15]:
```



```
  SVC
  SVC(C=100, gamma=0.01)
```

The best model found by GridSearchCV is stored in `grid.best_estimator_`

We can also access each step in a pipeline using index

Discussion

- **We can search over the data preprocessing strategies using the outcome of a supervised task like regression or classification.**
- **Searching over preprocessing hyperparameters together with model hyperparameters is a very powerful strategy.**
 - Keep in mind that GridSearchCV tries *all possible combinations* of the specified hyperparameters.
 - Adding more hyperparameters to the grid exponentially increases the number of models that need to be built.

Summary and Outlook

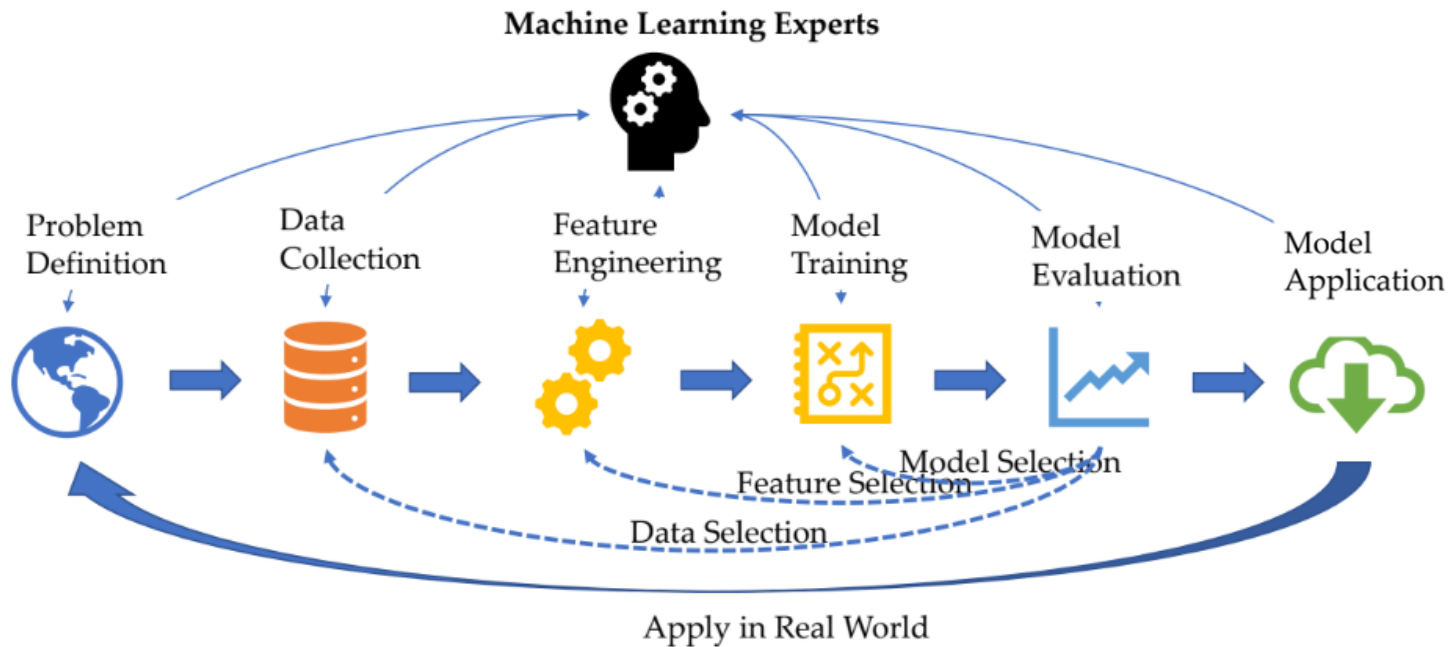
Summary and Outlook

- **The Pipeline class is a general-purpose tool to chain together multiple processing steps in a machine learning workflow.**
- **Using pipelines allows us to encapsulate multiple steps into a single Python object that adheres to the familiar scikit-learn interface of fit, predict, and transform.**
 - Real-world applications of machine learning involves a sequence of processing steps.
 - Choosing the right combination of feature engineering, preprocessing, and models is somewhat of an art, and often requires some trial and error.
 - Using pipelines, this “trying out” of many different processing steps is quite simple.
 - Also, the Pipeline class allows writing more succinct code.
- **We now possess all the required skills and know the necessary mechanisms to apply machine learning in practice.**

Automated Machine Learning

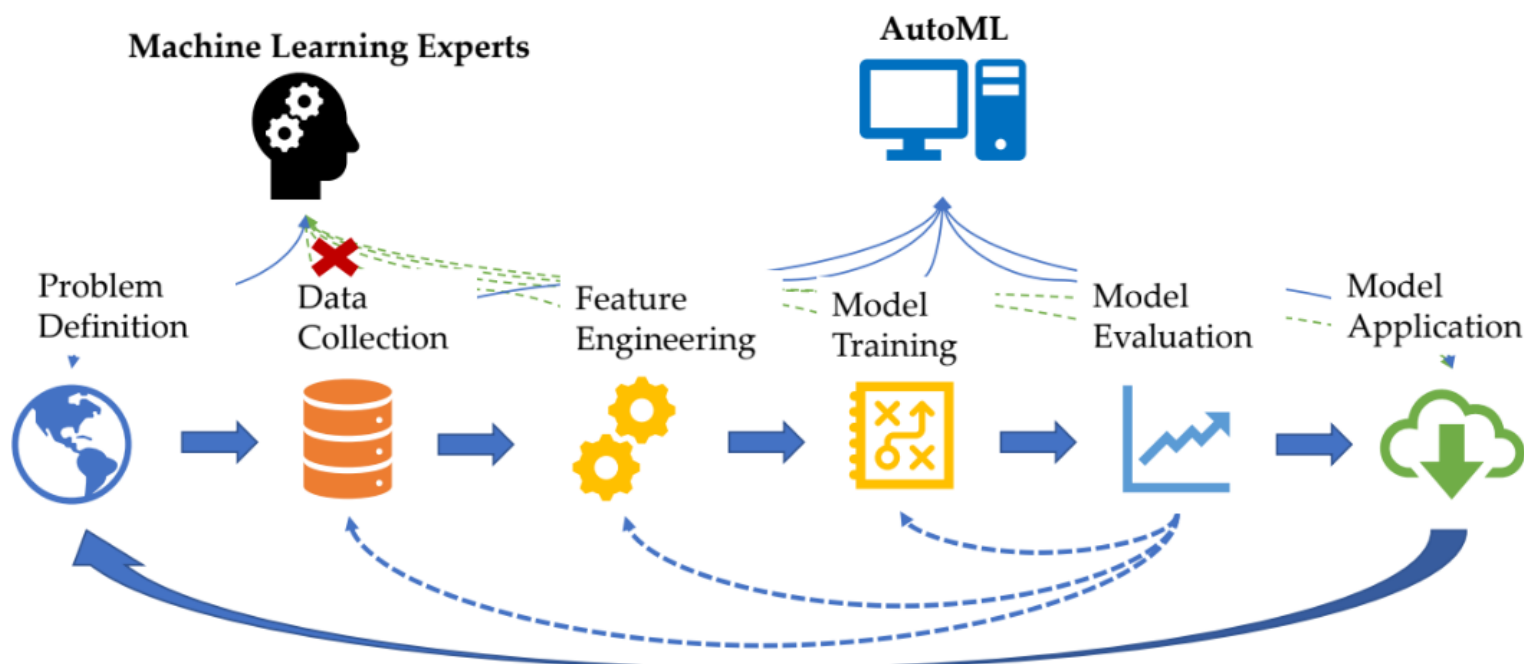
AutoML

- The success of machine learning (ML) in real-world applications crucially relies on human ML experts to perform the following tasks



AutoML

- As the complexity of these tasks is often beyond non-ML-experts, the rapid growth of machine learning applications has created a demand for off-the-shelf machine learning methods that can be used easily and without expert knowledge.
- Automated Machine Learning (AutoML) is the process of automating end-to-end the process of applying machine learning to real-world problems.



AutoML

approaches used to
explore the search space



**configurations that can be
considered**

(feature engineering, preprocessing,
learning algorithms, hyperparameters, etc.)

**performances of
candidates**

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., & Hutter, F. (2015). Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems* (pp. 2962-2970).

Efficient and Robust Automated Machine Learning

Matthias Feurer
Jost Tobias Springenberg

Aaron Klein
Manuel Blum

Katharina Eggensperger
Frank Hutter

Department of Computer Science
University of Freiburg, Germany

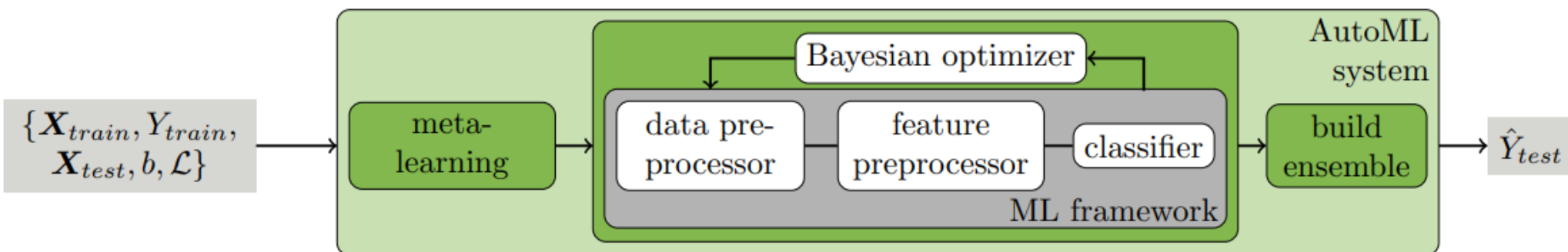
{feurerm, kleinaa, eggensp, springj, mblum, fh}@cs.uni-freiburg.de

Abstract

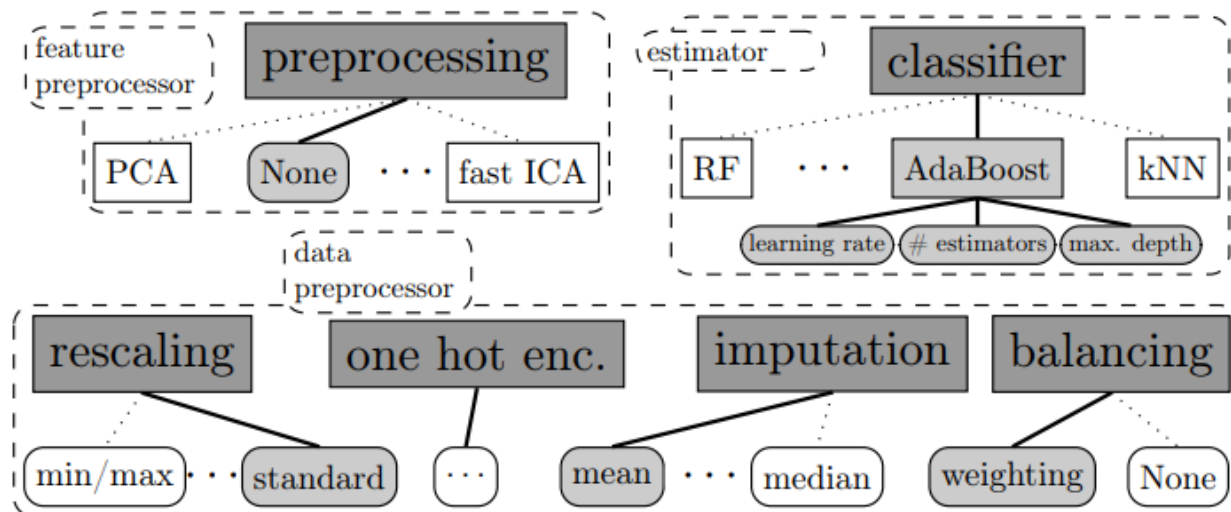
The success of machine learning in a broad range of applications has led to an ever-growing demand for machine learning systems that can be used off the shelf by non-experts. To be effective in practice, such systems need to automatically choose a good algorithm and feature preprocessing steps for a new dataset at hand, and also set their respective hyperparameters. Recent work has started to tackle this *automated machine learning (AutoML)* problem with the help of efficient Bayesian optimization methods. Building on this, we introduce a robust new AutoML system based on scikit-learn (using 15 classifiers, 14 feature preprocessing methods, and 4 data preprocessing methods, giving rise to a structured hypothesis space with 110 hyperparameters). This system, which we dub **AUTO-SKLEARN**, improves on existing AutoML methods by automatically taking into account past performance on similar datasets, and by constructing ensembles from the models evaluated during the optimization. Our system won the first phase of the ongoing ChaLearn AutoML challenge, and our comprehensive analysis on over 100 diverse datasets shows that it substantially outperforms the previous state of the art in AutoML. We also demonstrate the performance gains due to each of our contributions and derive insights into the effectiveness of the individual components of **AUTO-SKLEARN**.

auto-sklearn

- Pipeline



- Search Space



auto-sklearn

<https://automl.github.io/auto-sklearn/>

auto-sklearn

auto-sklearn is an automated machine learning toolkit and a drop-in replacement for a scikit-learn estimator:

```
>>> import autosklearn.classification
>>> cls = autosklearn.classification.AutoSklearnClassifier()
>>> cls.fit(X_train, y_train)
>>> predictions = cls.predict(X_test)
```

auto-sklearn frees a machine learning user from algorithm selection and hyperparameter tuning. It leverages recent advantages in *Bayesian optimization*, *meta-learning* and *ensemble construction*. Learn more about the technology behind *auto-sklearn* by reading our paper published at [NIPS 2015](#).

NEW: Auto-sklearn 2.0

Auto-sklearn 2.0 includes latest research on automatically configuring the AutoML system itself and contains a multitude of improvements which speed up the fitting the AutoML system.

auto-sklearn 2.0 works the same way as regular *auto-sklearn* and you can use it via

```
>>> from autosklearn.experimental.askl2 import AutoSklearn2Classifier
```

A paper describing our advances is available on [arXiv](#).

auto-sklearn

<https://automl.github.io/auto-sklearn/>

Example

```
>>> import autosklearn.classification
>>> import sklearn.model_selection
>>> import sklearn.datasets
>>> import sklearn.metrics
>>> if __name__ == "__main__":
>>>     X, y = sklearn.datasets.load_digits(return_X_y=True)
>>>     X_train, X_test, y_train, y_test = \
>>>         sklearn.model_selection.train_test_split(X, y, random_state=1)
>>>     automl = autosklearn.classification.AutoSklearnClassifier()
>>>     automl.fit(X_train, y_train)
>>>     y_hat = automl.predict(X_test)
>>>     print("Accuracy score", sklearn.metrics.accuracy_score(y_test, y_hat))
```

This will run for one hour and should result in an accuracy above 0.98.

AutoGluon

<https://auto.gluon.ai/>

AutoGluon

Search

GET STARTED

- Install
- Tabular Quick Start
- Multimodal Quick Start
- Time Series Quick Start

TUTORIALS

- Tabular
- Multimodal
- Time Series
- Cloud Training and Deployment

Fast and Accurate ML in 3 Lines of Code

[Get Started](#)

Quick Prototyping
Build machine learning solutions on raw data in a few lines of code.

State-of-the-art Techniques
Automatically utilize SOTA models without expert knowledge.

Easy to Deploy
Move from experimentation to production.

Customizable
Extensible with custom feature processing.

Quick Examples

Tabular

Predict the `class` column in a data table:

```
from autogluon.tabular import TabularDataset, TabularPredictor

data_root = 'https://autogluon.s3.amazonaws.com/datasets/Inc/'
train_data = TabularDataset(data_root + 'train.csv')
test_data = TabularDataset(data_root + 'test.csv')

predictor = TabularPredictor(label='class').fit(train_data=train_data)
predictions = predictor.predict(test_data)
```

Google Cloud AutoML

<https://cloud.google.com/automl>

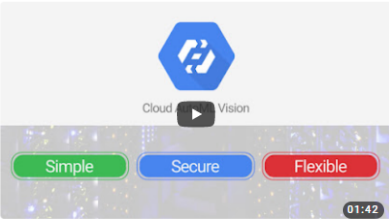
Cloud AutoML

최소한의 수고와 머신러닝 전문성만으로 간편하게 고품질 커스텀 머신러닝 모델을 학습시키세요.







[무료로 사용해 보기](#)[문서 보기](#)

커스텀 머신러닝 모델 학습

Cloud AutoML은 머신러닝 전문 지식이 부족한 개발자도 비즈니스 니즈에 맞게 고품질 모델을 학습시킬 수 있게 해 줍니다. 몇 분 만에 커스텀 머신러닝 모델을 빌드해 보세요.



Cloud AutoML 소개

카테고리	제품	특징
플랫폼	 AutoML이 포함된 통합 AI Platform(미리보기) 완벽한 단일 통합 플랫폼에서 강력한 도구 세트를 사용하여 이미지, 동영상, 텍스트, 테이블 형식 데이터를 위한 커스텀 모델을 빌드하고 배포합니다.	<ul style="list-style-type: none">데이터 세트 준비 및 저장동급 최고의 머신러닝 모델 빌드모델 검증 및 세부 조정대규모 모델 배포
시각	 AutoML Vision 클라우드나 예지에서 객체 감지와 이미지 분류를 통해 유용한 정보를 도출합니다. 지금 사용해 보기	<ul style="list-style-type: none">REST 및 RPC API 사용객체와 객체의 위치 및 개수 인식커스텀 라벨을 사용한 이미지 분류예제 ML 모델 배포
	 AutoML Video Intelligence(베타) 강력한 콘텐츠 탐색 기능과 매력적인 동영상 환경을 지원합니다. 지금 사용해 보기	<ul style="list-style-type: none">커스텀 라벨을 사용한 동영상 주석 처리스트리밍 동영상 분석장면 변화 감지객체 감지 및 추적
언어	 AutoML Natural Language 머신러닝을 통해 텍스트의 구조와 의미를 드러냅니다. 지금 사용해 보기	<ul style="list-style-type: none">통합된 REST API커스텀 항목 추출커스텀 감정 분석대규모 데이터 세트 지원
	 AutoML Translation 언어를 동적으로 감지하고 각 언어로 번역합니다. 지금 사용해 보기	<ul style="list-style-type: none">통합된 REST 및 gRPC API50개 언어 조합 지원커스텀 모델을 사용한 번역
구조화된 데이터	 AutoML Tables(베타) 구조화된 데이터에서 최신 머신러닝 모델을 자동으로 빌드하고 배포합니다. 지금 사용해 보기	<ul style="list-style-type: none">다양한 테이블 형식의 데이터 기본 요소 처리순위론 모델 빌드순위론 모델 배포 및 확장

AutoKeras

<https://autokeras.com/>

AutoKeras

Home

Installation

Tutorials >

Extensions >


Docker

Contributing Guide

Documentation >

Benchmarks

About

AutoKeras

AutoKeras: An AutoML system based on Keras. It is developed by [DATA Lab](#) at Texas A&M University. The goal of AutoKeras is to make machine learning accessible to everyone.

Learning resources

- A short example.

```
import autokeras as ak

clf = ak.ImageClassifier()
clf.fit(x_train, y_train)
results = clf.predict(x_test)
```

- Official website tutorials.
- The book of *Automated Machine Learning in Action*.
- The LiveProjects of *Image Classification with AutoKeras*.

Table of contents

Learning resources


Installation


Community

Contributing Code

Cite this work

Acknowledgements

Automated Machine Learning in Action

Five-Project Series
Image Classification with AutoKeras

Supported Tasks

AutoKeras supports several tasks with an extremely simple interface. You can click the links below to see the detailed tutorial for each task.

Supported Tasks:

[Image Classification](#)

[Image Regression](#)

[Text Classification](#)

[Text Regression](#)

[Structured Data Classification](#)

[Structured Data Regression](#)

Coming Soon: Time Series Forecasting, Object Detection, Image Segmentation.

H2O Driverless AI

<https://h2o.ai/platform/ai-cloud/make/h2o-driverless-ai/>

H2O.ai

Platform

Solutions

Customers

Partners

Resources

Events

Company

Request Live Demo


On-Demand Demos


Sign In

H2O Driverless AI

Democratizing AI with Automated Machine Learning

Request Demo





Hear directly from our customers.

Rajesh Malla, Head of Data Engineering at Resolution Life insurance, takes the stage at H2O World Sydney 2022 to discuss AI transformation and how Resolution Life uses H2O Driverless AI to predict claim triage and other insurance AI use cases.

Watch Video

