sapientml

Release 0.4.9

The SapientML Authors

Nov 22, 2023

USERS:

Generative AutoML for Tabular Data

SapientML is an AutoML technology that can learn from a corpus of existing datasets and their human-written pipelines, and efficiently generate a high-quality pipeline for a predictive task on a new dataset.

CHAPTER

ONE

INSTALLATION

1.1 Install with pip

sapientml needs to be installed just like any other python package into your documentation building environment:

```
pip install sapientml
```

Note: If you wish to use plugins created by a third-party, please follow the guidelines provided by them.

CHAPTER

TWO

USAGE

sapientml generates source code to train and predict a machine learning model from a CSV-formatted dataset and requirements of a machine learning task to be solved.

2.1 SapientML class

sapientml provides `SapientML` class that provides the top level API of SapientML. In the constructor of `SapientML`, you firstly need to set `target_columns` as a requirement of the task. `target_columns` specifies which the task is to predict. Second, you can set `task_type` from `classification` or `regression` as a type of machine learning task. You can also skip setting `task_type` and in that case SapientML automatially suggests task type by looking into values of the target columns.

```
from sapientml import SapientML

cls = SapientML(
    target_columns=["survived"],
    task_type=None, # suggested automatically from the target columns
)
```

As well as model classes of the other well-known libraries like **scikit-learn**, `SapientML` provides `fit` and `predict` to conduct model training and prediction by using generated code.

```
import pandas as pd
from sklearn.metrics import f1_score
from sklearn.model_selection import train_test_split

train_data = pd.read_csv("https://github.com/sapientml/sapientml/files/12481088/titanic.
↪csv")
train_data, test_data = train_test_split(train_data)
y_true = test_data["survived"].reset_index(drop=True)
test_data.drop(["survived"], axis=1, inplace=True)

cls.fit(train_data, output_dir="./outputs")
y_pred = cls.predict(test_data)

print(f"F1 score: {f1_score(y_true, y_pred)}")
```

2.2 Generated source code

After calling *fit*, you can get generated source code at `./outputs` folder. Here is the example of files generated by `fit`:

```
outputs
├── 1_script.py
├── 2_script.py
├── 3_script.py
├── final_predict.py
├── final_script.out.json
├── final_script.py
├── final_train.py
└── lib
    └── sample_dataset.py
```

`1_script.py`, `2_script.py`, and `3_script.py` are scripts of the hold-out validation using the preprocessors and the top-3 most plausible models. `final_script.py` is the script that selects the model actually achieved the highest score of the top-3 models, and `final_script.out.json` contains its score. `final_train.py` is the script for training the selected model, and `final_predict.py` is the the script for prediction using the model trained by `final_train.py`. `lib` folder contains modules that the above scripts uses.

2.3 Using generated code as a model

After calling `fit`, you can also get `cls.model`, which is a `GeneratedModel` instance that contains generated source code and `.pkl` files of preprocessers and a actual machine learning model. The instance also asts a usual model providing `fit` and `predict`.

```
cls.fit(train_data)
model = cls.model # obtains GeneratedModel instance
```

You can get the set of source code and `.pkl` files by referring `model.files` or by looking into `./outputs` folder after calling `model.save("./model")`. Here is the example of files contained in `GeneratedModel`:

```
model
├── final_predict.py
├── final_train.py
├── lib
│   └── sample_dataset.py
├── model.pkl
├── ordinalEncoder.pkl
├── simpleimputer-numeric.pkl
└── simpleimputer-string.pkl
```

The actual behavior of `model.fit` is a subprocess executing `final_train.py`. Beware that `model.fit(another_train_data)` is not retraining the existing model but buiding a new one. `model.predict` creates a subprocess executing `final_predict.py` as well.

`SapientML` provides a utility function to restore the `SapientML` instance from generated model.

```
import pickle

cls.fit(train_data)
```

(continues on next page)

(continued from previous page)

```
with open("model.pkl", "wb") as f:
    pickle.dump(sml.model, f)

with open("model.pkl", "rb") as f:
    model = pickle.load(f)
sml = SapientML.from_pretrained(model)
```

CHAPTER

THREE

CONFIGURATION

The constructor of `SapientML` class consumes various parameters depending on plugin installation. Here we show the parameters you can assign at the constructor of `SapientML` in cases of each `model_type` assigned.

3.1 Model types

sapientml provides the plugin mechanism for generating source code that is different from the original algorithm of **sapientml** in utilizing machine learning models and preprocessing components. Each plugin has a unique `model_type`, and users can choose one of them as a parameter of the constructor of `SapientML` class. The default value of `model_type` is `sapientml`, which is provided by **sapientml_core** plugin.

3.1.1 Parameters for `sapientml`

target_columns (list[str])
Names of target columns.

task_type (classification', 'regression', or None) = None
Identifies the task type from classification or regression, or automatically suggests it if set to `None`

adaptation_metric (str) = 'f1' if task_type is 'classification', 'r2' if 'regression'
Metric for evaluation. `f1`, `auc`, `ROC_AUC`, `accuracy`, `Gini`, `LogLoss`, `MCC` (Matthews correlation coefficient), `QWK` (Quadratic weighted kappa) are available for classification. `r2`, `RMSLE`, `RMSE`, `MAE` are available for regression.

split_method ('random', 'time', or 'group') = 'random'
Method of train-test split. `random` uses random split. `time` requires `split_column_name`. This sorts the data rows based on the column, and then splits data. `group` requires `split_column_name`. This splits the data so that rows with the same value of `split_column_name` are not placed in both training and test data.

split_seed (int) = 17
Random seed for train-test split. Ignored when `split_method='time'`.

split_train_size (float) = 0.75
The ratio of training size to input data. Ignored when `split_method='time'`.

split_column_name (str or None) = None
Name of the column used to split. Ignored when `split_method='random'`

time_split_num (int) = 5
Passed to `n_splits` of `TimeSeriesSplit`. Valid only when `split_method='time'`.

time_split_index (int) = 4
The index of the split from `TimeSeriesSplit`. Valid only when `split_method='time'`.

split_stratification (bool or None) = None
To perform stratification in train-test split. Valid only when `task_type='classification'`.

initial_timeout (int) = 600
Timelimit to execute each generated script. Ignored when `hyperparameter_tuning=True` and `hyperparameter_tuning_timeout` is set.

timeout_for_test (int) = 0
Timelimit to execute test script (final_script) and Visualization.

cancel (CancellationToken or None) = None
Object to interrupt evaluations.

project_name (str or None) = None
Project name.

debug (bool) = False
Debug mode or not.

use_pos_list (list[str]) = [“”, “”, “”, “”, “”]
List of parts-of-speech to be used during text analysis. This variable is used for japanese texts analysis. Select the part of speech below. “”, “”, “”, “”, “”.

use_word_stemming (bool) = True
Specify whether or not word stemming is used. This variable is used for japanese texts analysis.

n_models (int) = 3
Number of output models to be tried.

seed_for_model (int) = 42
Random seed for models such as `RandomForestClassifier`.

id_columns_for_prediction (list[str] or None) = None
Name of the dataframe columns that outputs the prediction result.

use_word_list (list[str], dict[str, list[str]], or None) = None
List of words to be used as features when generating explanatory variables from text. If dict type is specified, key must be a column name and value must be a list of words.

hyperparameter_tuning (bool) = False
On/Off of hyperparameter tuning.

hyperparameter_tuning_n_trials (int) = 10
The number of trials of hyperparameter tuning.

hyperparameter_tuning_timeout (int) = 0
Time limit for hyperparameter tuning in each generated script. Ignored when `hyperparameter_tuning` is `False`.

hyperparameter_tuning_random_state (int) = 1023
Random seed for hyperparameter tuning.

predict_option (‘default’ or ‘probability’) = ‘default’
Specify predict method (default: `predict()`, probability: `predict_proba()`.)

permutation_importance (bool) = True
On/Off of outputting permutation importance calculation code.

add_explanation (bool) = False
If `True`, outputs ipynb files including EDA and explanation.

CHAPTER

FOUR

SETUP

4.1 Creating a development environment in your host

Python >=*3.10,<3.13* is required.

Clone sapientml and core. If you need to modify preprocess and loaddata, please clone them as well.

```
mkdir AutoML
cd AutoML
git clone https://github.com/sapientml/sapientml.git
git clone https://github.com/sapientml/core.git

# optional
git clone https://github.com/sapientml/preprocess.git
git clone https://github.com/sapientml/loaddata.git
```

Setup an environment in the **sapientml** repository folder.

```
cd /path/to/AutoML/sapientml
python -m venv venv
. venv/bin/activate
pip install poetry
poetry install
pre-commit install
pip install -e ../core

# optional
pip install -e ../preprocess
pip install -e ../loaddata
```

For ubuntu, *poetry install* may fail. If so, try the following command:

```
PYTHON_KEYRING_BACKEND="keyring.backends.null.Keyring" poetry install
```

As sapientml and core are interdependent. Use below command to integrate.

```
pip install -e /path/to/AutoML/core
deactivate
```

Now download corpus inside **sapientml_core**.

```
. venv/bin/activate
cd /path/to/AutoML/core/sapientml_core
pip install dvc
wget https://github.com/sapientml/sapientml/files/13432403/sapientml-corpus-0.1.3.zip
unzip sapientml-corpus-0.1.3.zip
mv sapientml-corpus-0.1.3 corpus
cd corpus
bash ./scripts/pull.sh
rm -f sapientml-corpus-0.1.3.zip
deac
```

After successfull installation, the following directory structure should reflect.

```
AutoML/
├── core/
│   ├── sapientml_core/
│       ├── corpus/
│       │   ├── clean-notebooks/
│       │   ├── annotated-notebooks/
│       │   └── dataset/
│       ├── design/
│       └── training/
│
└── sapientml/
    └──sapientml/
```

CHAPTER

FIVE

TRAINING FROM A CORPUS

5.1 SapientML local training

5.1.1 1. Execution Method

Please refer to *this page* to finish the setup of development environment first. We assume that at this point, **corpus** is downloaded and stored at the **sapientml_core** location, all the pipelines in the corpus is already clean using program slicing and there exists a label file such as *annotated-notebooks/annotated-notebooks-1140.csv* that has all the components for each pipeline.

- After successfull setup, the following directory structure should reflect.

```
AutoML/
├── core/
│   ├── sapientml_core/
│       ├── corpus/
│       │   ├── clean-notebooks/
│       │   ├── annotated-notebooks/
│       │   └── dataset/
│       ├── design
│       └── training
│
└── sapientml/
    └──sapientml/
```

Create sample main.py

- Create a driver code inside sapientml which runs each training step.
- We have to explicitly call the train method from the SapientMLGenerator class in order to train sapientml by considering datasets taken from corpus.

```
from sapientml_core.generator import SapientMLGenerator

print("Training started")
print("=================")
cls = SapientMLGenerator()
cls.train(<tag>, <num_parallelization>)
print("=================")
print("Training ended")
```

- By executing the above driver code, a folder **.cache** is created inside **sapientml_core** and output files from local training are stored here.
- Argument **tag** is passed to each step to determine the cache folder name. For example, *./.cache/2.5.1-test* is created as the cache folder if *tag* is set as "2.5.1-test", then all artifacts of local training will be stored in that folder. Otherwise if **tag** is not set, all artifacts will be stored in **.cache**.
- Also the argument **num_parallelization** is used for parallellizing the execution process and its default value is 200.

5.1.2 2. Local training process overview

- Step-1 : Denoise dataset
- Step-2 : Augment the corpus
- Step-3 : Extract meta-features
- Step-4 : Train the models
- Step-5 : Create dataflow model

5.1.3 3. Explanation of each process in local training

Step-1 : Denoise Dataset

Step-1A : static_analysis_of_columns

- *core/sapientml_core/training/denoising/static_analysis_of_columns.py* fetches all project list or pipeline details.
- Parse pipeline and fetch target, dropped, renamed column names.
- We use **libcst** library for parsing the column api details.

Note: This script can traverse an Abstract Syntax Tree (AST) using the LibCST library and retrieve many useful information such as column names, API names, strings, assignments, and so on.

Output : static_info.json

- It will create the directory *.cache/<tag>/static_info.json.*
- It gives informations about pre-processing components operations.

Example:

```
{
    "script0011.py": {
        "drop_api": [
            "Age",
            "Balance",
            "CreditScore",
            "EstimatedSalary",
            "RowNumber",
            "CustomerId",
```

(continues on next page)

(continued from previous page)

```
            "Surname",
            "Tenure",
            "HasCrCard"
        ],
        "rename_api": [],
        "target": "Exited"
    },
```

Step-1B : dataset_snapshot_extractor

- *core/sapientml_core/training/denoising/dataset_snapshot_extractor.py* fetches all project list or all pipeline details.
- Parse pipelines and instruments a given pipeline with code snippets to collect snapshots of dataset.
- We use **ast** library for parse and update the code.
- We use **machinery** library for the implementation of the import statement in updated pipeline.
- Execute the instrumented version of the pipeline to store the snapshot of the dataset after each line in the pipeline.

Output : dataset-snapshots

- It will create the directory .cache/<tag>/dataset-snapshots/.
- A JSON file for each pipeline that stores the snapshot of column names of the dataframe after important statements in *.cache/<tag>/dataset-snapshots* as shown below.
- It is a dictionary that contains line number as a key and a list of column names as value.

Example:

```
[
    {
        "4": [
            [
                "RowNumber",
                "CustomerId",
                "Surname",
                "CreditScore",
                "Geography",
                "Gender",
                "Age",
                "Tenure",
                "Balance",
                "NumOfProducts",
                "HasCrCard",
                "IsActiveMember",
                "EstimatedSalary",
                "Exited"
            ],
            "data",
            "<class 'pandas.core.frame.DataFrame'>"
```

(continues on next page)

(continued from previous page)

```
        ]
    }
]
```

Step-1C : determine_used_features

- *core/sapientml_core/training/denoising/determine_used_features.py* takes the outputs of **static_info.json** and **dataset-snapshots** from Step-1A and Step-1B as input.
- Fetch summary for each pipeline from dataset_snapshot(json) created in step 1b.
- **The summary consist of following information:**
  - pipeline name
  - used_cols
  - unmapped_cols
  - new_cols
  - target
  - deleted
  - status

Output : feature_analysis_summary.json

- It will create the JSON file *.cache/<tag>/feature_analysis_summary.json*
- It contains summary for all pipelines.

Example:

```
{
 "script0011.py": {
    "pipeline": "script0011.py",
    "used_cols": [
        "EstimatedSalary",
        "Exited",
        "Age",
        "CreditScore",
        "NumOfProducts",
        "Gender",
        "Geography",
        "Balance",
        "IsActiveMember"
    ],
    "unmapped_cols": [],
    "new_cols": [],
    "target": "Exited",
    "deleted": [
        "Tenure",
        "Surname",
```

(continues on next page)

(continued from previous page)

```
        "HasCrCard",
        "RowNumber",
        "CustomerId"
    ],
    "status": "FINALIZED"
},
```

Step-2 : Corpus Augmentation

Step-2A : mutation_runner

- *core/sapientml_core/training/augmentation/mutation_runner.py* mutates each pipeline in the corpus, runs the mutated version, and store all the details in *.cache/<tag>/exec_info* directory.
- In the first run, this step is expected to take a long time depending on the number of the pipelines in the corpus. From the subsequent runs, mutation is only run for the new notebooks, i.e., if the mutated results are not found locally for those notebooks.
- We use *ast* library for parsing and analyse the components in pipeline.
- It executes the mutated pipelines and store the results and logs.

Output: exec_info

- It will create the directory *.cache/<tag>/exec_info*
- It will contain the information of all the mutated pipeleines i.e., it replaces the model in the original pipeline with a pre-defined list of models(21 models).

Step-2B : mutation_results

- *core/sapientml_core/training/augmentation/mutation_results.py* combines all the results in a CSV file and selects the best model.
- It fetches the accuracy score of mutated corpus for all pipelines.
- And saves it in *.cache/{tag(if any)}/mutation_results.csv* file.

Output : mutation_results.csv

file_	random forest,	extra tree	lightgbm	xgboost	catboost	gradient boosting,	adaboost	decision tree	svm	linear svm	logistic/linear regression	lasso	sgd	mlp,	multinomial nb,	gaussian nb,	bernb,	original,	best_models
script	0.82	0.83	0.85	0.85	0.85	0.86	0.85	0.82	0.85	0.84	0.85	0.0	0.84	0.85	0.84	0.81	0.81	0.85	['gradient boosting']

- From the above we can say that the gradient boosting model is the best model as it has greater accuracy than the rest of the models.

Step-3 : Extraction of Meta-Features and Pipeline Components

- ***core/sapientml_core/training/meta_feature_extractor.py* extracts the meta-features for all the projects, In other words it fetches all the pipeline details. This will save all the meta-features at *.cache/<tag>/* in form of two CSV files:**
    1. one for pre-processing components (pp_metafeatures_training.csv).
    2. another for the model components (model_metafeatures_training.csv).
- **There are two modes of extracting meta-features. "clean" is active in default. This setting can be modified directly in the source code**
    1. "as-is"
    2. "clean"
- **as-is** computes meta-features based on all the meta-features in the dataset.
- **clean** mode only uses the meta-features that are used in the pipeline. Features which are already used in the pipeline are pre-computed and stored in the *.cache/<tag>/feature_analysis_summary.json file*.

Output : pp_metafeatures_training.csv, model_metafeatures_training.csv

- Fetch meta features related to model and save to *.cache/<tag>/pp_metafeatures_training.csv*.
- Fetch meta features related to preprocess component and save to *.cache/<tag>/model_meta_features_trainer.csv*.

Step-4 : Training Meta-Models for Skeleton Predictor

Step-4A: Training of pre-processing components (pp_model_trainer)

- *core/sapientml_core/training/pp_model_trainer.py* is in charge of training the meta-models for pre-processing components.
- It takes *.cache/<tag>/pp_metafeatures_training.csv* as input and trains a decision tree for each pre-processing component.

Output : pp_models.pkl

- *.cache/<tag>/pp_models.pkl* is a machine learning model pickle file for selecting pre-processing components.

Step-4B: Training of Model components (meta_model_trainer)

- *core/sapientml_core/training/meta_model_trainer.py* is in charge of training the meta-model that predicts and ranks the model components for the pipeline. Currently it is an ensemble model that uses **LogisticRegression** and **SVM** as the base classifiers and ranks the predicted model based on the average of their probability scores.

Output: mp_model_1.pkl, mp_model_2.pkl

- *.cache/<tag>/mp_model_1.pkl* is a **LogisticRegression** model pickle file for selecting pre-processing components.
- *.cache/<tag>/mp_model_2.pkl* is a **svm** model pickle file for selecting pre-processing components.

Step-5 : Construct the Data Flow Model

Step-5A : dependent_api_extractor

- *core/sapientml_core/training/dataflowmodel/dependent_api_extractor.py* will get the API/labels that are dependent on each other. A label is dependent on each other when they are applied on the same column.
- It gets all the annotated pipelines in the corpus.
- It reads Annotated_notebook csv and store the labels with respect to filename and line number
- If same label exists take a count and store as a dictionary data in final_dependency_list i.e {'a b':1, 'c d':3, 'e f':2}
- It sorts the items and store all the list of dependent labels/APIs in dependent_labels.json file.

Output : dependent_labels.json

- A JSON file stored in *.cache/<tag>/dependent_labels.json* containing the list of dependent APIs.

Example:

```
{
"['PREPROCESS:Category:get_dummies:pandas', 'PREPROCESS:DeleteColumns:drop:pandas']": 79,
"['PREPROCESS:ConvertStr2Date:to_datetime:pandas', 'PREPROCESS:DeleteColumns:drop:pandas
↪']": 27,
"['PREPROCESS:MissingValues:fillna:pandas', 'PREPROCESS:DeleteColumns:drop:pandas']": 16,
"['PREPROCESS:Scaling:log:numpy', 'PREPROCESS:DeleteColumns:drop:pandas']": 12,
}
```

- In the above sample json file. The first line shows that they call *get_dummies* preprocessor first and then *DeleteColumns* preprocessor next and this pair is dependent on each other.
- The number denotes the count of this dependent_labels executed as we have multiple pipelines.

Step-5B : determine_label_order

- *core/sapientml_core/training/dataflowmodel/determine_label_order.py* will determine the order of the components.
- If there is any order exists. It will extract the order of two APIs/labels A and B.
- There is an order between A –> B if A and B are dependent on each other based on 'dependent_api_extractor.py' and A is always followed by B in all piplelines and there is NO case in the corpus where B is followed by A.
- Based on the previous step output file *.cache/<tag>/dependent_labels.json*, An output json file *.cache/<tag>/label_orders.json* is created.

Output: label_orders.json

- A JSON file stored in *.cache/<tag>/label_orders.json* containing the order of labels in a pair-wise form.

Example:

```
[
"PREPROCESS:MissingValues:fillna:pandas#PREPROCESS:GenerateColumn:groupby:pandas",
"PREPROCESS:TypeChange:astype:pandas#PREPROCESS:MissingValues:fillna:pandas",
"PREPROCESS:MissingValues:interpolate:sklearn#PREPROCESS:CONVERT_NUM2NUM:where:numpy",
"PREPROCESS:TypeChange:astype:pandas#PREPROCESS:GenerateColumn:date:pandas",
"PREPROCESS:MissingValues:fillna:pandas#PREPROCESS:TypeChange:astype:pandas",
"PREPROCESS:MissingValues:fillna:pandas#PREPROCESS:Category:get_dummies:pandas",
]
```

5.1.4 4. How to use training output

- After **label_orders.json** is produced, it is copied into *core/sapientml_core/adaptation/artifacts/label_order.json* so that SapientML can use it. Please note that the **dataflow model** is a very important artifact. So make sure that the updated **dataflow model** is correct before replacing the existing one. Generally, it should not be updated unless there is no new pre-processing components.
- **Replace models (*core/sapientml_core/models/*) folder files with the respective files generated in .cache file.**
  - Replace **core/sapientml_core/models/feature_importance.json** with **.cache/<tag>/feature_importance.json**.
  - Replace **core/sapientml_core/models/pp_models.pkl** with **.cache/<tag>/pp_models.pkl**
  - Replace **core/sapientml_core/models/mp_model_1.pkl** with **.cache/<tag>/mp_model_1.pkl**
  - Replace **core/sapientml_core/models/mp_model_2.pkl** with **.cache/<tag>/mp_model_2.pkl**

CHAPTER

SIX

API

6.1 Main class

class sapientml.**SapientML**(*target_columns: list[str]*, *task_type: Literal['classification', 'regression'] | None = None*, *adaptation_metric: str | None = None*, *split_method: Literal['random', 'time', 'group'] = 'random'*, *split_seed: int = 17*, *split_train_size: float = 0.75*, *split_column_name: str | None = None*, *time_split_num: int = 5*, *time_split_index: int = 4*, *split_stratification: bool | None = None*, *model_type: str = 'sapientml'*, ***kwargs*)

The constructor of SapientML.

You can pass all the keyword arguments for configurations of PipelineGenerators and CodeBlockGenerators from plugins.

Parameters

- **target_columns** (*list[str]*) – Names of target columns.
- **task_type** (*'classification', 'regression' or None*) – Specify task type classification, regression.
- **adaptation_metric** (*str*) – Metric for evaluation. Classification: 'f1', 'auc', 'ROC_AUC', 'accuracy', 'Gini', 'LogLoss', 'MCC'(Matthews correlation coefficient), 'QWK'(Quadratic weighted kappa). Regression: 'r2', 'RMSLE', 'RMSE', 'MAE'.
- **split_method** (*'random', 'time', or 'group'*) – Method of train-test split. 'random' uses random split. 'time' requires 'split_column_name'. This sorts the data rows based on the column, and then splits data. 'group' requires 'split_column_name'. This splits the data so that rows with the same value of 'split_column_name' are not placed in both training and test data.
- **split_seed** (*int*) – Random seed for train-test split. Ignored when split_method='time'.
- **split_train_size** (*float*) – The ratio of training size to input data. Ignored when split_method='time'.
- **split_column_name** (*str*) – Name of the column used to split. Ignored when split_method='random'
- **time_split_num** (*int*) – Passed to TimeSeriesSplit's n_splits. Valid only when split_method='time'.
- **time_split_index** (*int*) – The index of the split from TimeSeriesSplit. Valid only when split_method='time'.
- **split_stratification** (*bool*) – To perform stratification in train-test split. Valid only when task_type='classification'.

fit(*training_data: DataFrame | str*, *validation_data: DataFrame | str | None = None*, *test_data: DataFrame | str | None = None*, *save_datasets_format: Literal['csv', 'pickle'] = 'pickle'*, *csv_encoding: Literal['UTF-8', 'SJIS'] = 'UTF-8'*, *csv_delimiter: str = ','*, *ignore_columns: list[str] | None = None*, *output_dir: str = './outputs'*, *codegen_only: bool = False*)

Generate ML scripts for input data.

Parameters

- **training_data** (*pandas.DataFrame or str*) – Training dataframe. When str, this is regarded as a file path.
- **validation_data** (*pandas.DataFrame, str or None*) – Validation dataframe. When str, this is regarded as file paths. When None, validation data is extracted from training data by split.
- **test_data** (*pandas.DataFrame, str, or None*) – Test dataframes. When str, they are regarded as file paths. When None, test data is extracted from training data by split.
- **save_datasets_format** (*'csv' or 'pickle'*) – Data format when the input dataframes are written to files. Ignored when all inputs are specified as file path.
- **csv_encoding** (*'UTF-8' or 'SJIS'*) – Encoding method when csv files are involved. Ignored when only pickle files are involved.
- **csv_delimiter** (*str*) – Delimiter to read csv files.
- **ignore_columns** (*list[str]*) – Column names which must not be used and must be dropped.
- **output_dir** (*str*) – Output directory.
- **codegen_only** (*bool*) – Do not conduct fit() of GeneratedModel if True.

Returns

self – SapientML object itself.

Return type

SapientML

static from_pretrained(*model*)

The factory method of SapientML from a pretrained model built by source code previously generated by SapientML.

model must be either pickle filename, pickle bytes-like object, or deserialized object

Parameters

model (*str, bytes-like object, or GeneratedModel*) – A pretrained model built by source code previously generated by SapientML.

Returns

sml – a new SapientML instance loaded from the pretrained model.

Return type

SapientML

predict(*test_data: DataFrame*)

Predicts the output of the test_data.

Parameters

test_data (*pd.DataFrame*) – Dataframe used for predicting the result.

Returns

result – It returns the prediction_result.csv result in dataframe format.

Return type
pd.DataFrame

class sapientml.**GeneratedModel**(*input_dir: PathLike*, *save_datasets_format: Literal['csv', 'pickle']*, *timeout: int*, *csv_encoding: Literal['UTF-8', 'SJIS']*, *csv_delimiter: str*, *params: dict*)

The constructor of GeneratedModel. Instantiating this class by yourself is not intended.

Parameters

- **input_dir** (*PathLike*) – Directory path containing training/prediction scripts and trained models.
- **save_datasets_format** (*'csv' or 'pickle'*) – Data format when the input dataframes are written to files. Ignored when all inputs are specified as file path.
- **timeout** (*int*) – Timeout for the execution of training and prediction.
- **csv_encoding** (*'UTF-8' or 'SJIS'*) – Encoding method when csv files are involved. Ignored when only pickle files are involved.
- **csv_delimiter** (*str*) – Delimiter to read csv files.

fit(*X: DataFrame*, *y: DataFrame | Series | None = None*)

Generate ML scripts for input data.

Parameters

- **X** (*pandas.DataFrame*) – Training dataframe. Contains target values if *y* is *None*.
- **y** (*pandas.DataFrame or pandas.Series*) – The target values.

Returns
self – GeneratedModel object itself

Return type
GeneratedModel

predict(*X: DataFrame*)

Predicts the output of the test_data and store in the prediction_result.csv.

Parameters
X (*pd.DataFrame*) – Dataframe used for predicting the result.

Returns
result_df – It returns the prediction_result.csv result in dataframe format.

Return type
pd.DataFrame

save(*output_dir: PathLike*)

Save generated code to *output_dir* folder

Parameters
output_dir (*Path-like object*) – Training dataframe.

Returns
self – GeneratedModel object itself

Return type
GeneratedModel

6.2 Config parameters

class sapientml.**Config**(*, *initial_timeout: int = 600*, *timeout_for_test: int = 0*, *cancel: CancellationToken | None = None*, *project_name: str | None = None*, *debug: bool = False*)

Configuration arguments for sapientml.generator.CodeBlockGenerator and/or sapientml.generator.PipelineGenerator.

initial_timeout

Timelimit to execute each generated script. Ignored when hyperparameter_tuning=True and hyperparameter_tuning_timeout is set.

Type
int

timeout_for_test

Timelimit to execute test script (final_script) and Visualization.

Type
int

cancel

Object to interrupt evaluations.

Type
CancellationToken, optional

project_name

Project name.

Type
str, optional

debug

Debug mode or not.

Type
bool

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses *__pydantic_self__* instead of the more common *self* for the first arg to allow *self* as a field name.

class sapientml_core.**SapientMLConfig**(*, *initial_timeout: int = 600*, *timeout_for_test: int = 0*, *cancel: CancellationToken | None = None*, *project_name: str | None = None*, *debug: bool = False*, *n_models: int = 3*, *seed_for_model: int = 42*, *id_columns_for_prediction: list[str] | None = None*, *use_word_list: list[str] | dict[str, list[str]] | None = None*, *hyperparameter_tuning: bool = False*, *hyperparameter_tuning_n_trials: int = 10*, *hyperparameter_tuning_timeout: int = 0*, *hyperparameter_tuning_random_state: int = 1023*, *predict_option: Literal['default', 'probability'] = 'default'*, *permutation_importance: bool = True*, *add_explanation: bool = False*)

Configuration arguments for SapientMLGenerator.

`n_models`

Number of output models to be tried.

Type
int, default 3

`seed_for_model`

Random seed for models such as RandomForestClassifier.

Type
int, default 42

`id_columns_for_prediction`

Name of the dataframe columns that outputs the prediction result.

Type
Optional[list[str]], default None

`use_word_list`

List of words to be used as features when generating explanatory variables from text. If dict type is specified, key must be a column name and value must be a list of words.

Type
Optional[Union[list[str], dict[str, list[str]]]], default None

`hyperparameter_tuning`

On/Off of hyperparameter tuning.

Type
bool, default False

`hyperparameter_tuning_n_trials`

The number of trials of hyperparameter tuning.

Type
int, default 10

`hyperparameter_tuning_timeout`

Time limit for hyperparameter tuning in each generated script. Ignored when hyperparameter_tuning is False.

Type
int, default 0

`hyperparameter_tuning_random_state`

Random seed for hyperparameter tuning.

Type
int, default 1023

`predict_option`

Specify predict method (default: predict(), probability: predict_proba().)

Type
Literal[“default”, “probability”], default “default”

`permutation_importance`

On/Off of outputting permutation importance calculation code.

Type
bool, default True

add_explanation

If True, outputs ipynb files including EDA and explanation.

Type
bool, default False

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses *__pydantic_self__* instead of the more common *self* for the first arg to allow *self* as a field name.

postinit()

Set initial_timeout and hyperparameter_tuning_timeout.

If initial_timeout is set as None and hyperparameter_tuning is false, set initial_timeout as INITIAL_TIMEOUT.

For hyperparameter_tuning_timeout, if both initial_timeout and hyperparameter_tuning_timeout are set as None, set hyperparameter_tuning_timeout as INITIAL_TIMEOUT.

If initial_timeout is set and hyperparameter_tuning is True, and hyperparameter_tuning_timeout is None :

Set the hyperparameter_tuning_timeout to unlimited.(hyperparameter_tuning_timeout = self.initial_timeout.) Since initial_timeout always precedes hyperparameter_tuning_timeout, it can be expressed that there is no time limit for hyperparameters during actual execution.

class sapientml_preprocess.**PreprocessConfig**(*, *initial_timeout: int = 600*, *timeout_for_test: int = 0*, *cancel: CancellationToken | None = None*, *project_name: str | None = None*, *debug: bool = False*, *use_pos_list: list[str] | None = ['', '', '', '', '']*, *use_word_stemming: bool = True*)

Configuration arguments for sapientml_preprocess.Preprocess class.

use_pos_list

List of parts-of-speech to be used during text analysis. This variable is used for japanese texts analysis. Select the part of speech below. “”, “”, “”, “”, “”.

Type
Optional[list[str]]

use_word_stemming

Specify whether or not word stemming is used. This variable is used for japanese texts analysis.

Type
bool default True

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses *__pydantic_self__* instead of the more common *self* for the first arg to allow *self* as a field name.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX