nyaggle

Release 0.1.6

nyanp

Sep 10, 2023

CONTENTS:

CHAPTER

ONE

INSTALLATION

You can install nyaggle via pip:

```
pip install nyaggle   # Install core parts of nyaggle
```

nyaggle does not install the following packages by default:

- catboost
- lightgbm
- xgboost
- mlflow
- pytorch

Modules which depends on these packages won't work until you also install them. For example, `run_experiment` with `algorithm_type='xgb'`, `'lgbm'` and `'cat'` options won't work until you also install xgboost, lightgbm and catboost respectively.

If you want to install everything required in nyaggle, This command can be used:

```
pip install nyaggle[all]  # Install everything
```

If you use `lang=ja` option in `BertSentenceVecorizer`, you also need to intall MeCab and mecab-python3 package to your environment.

CHAPTER

TWO

TUTORIAL

2.1 Tracking your machine learning experiments with run_experiment

2.1.1 Concept

In a typical tabular data competition, you may probably repeat evaluating your idea by cross-validation with logging the parameters and results to track your experiments.

The `nyaggle.experiment.run_experiment` is an API for such situations. If you are using LightGBM as your model, the code will be quite simple:

```
import pandas as pd
from nyaggle.experiment import run_experiment
from nyaggle.experiment import make_classification_df

INPUT_DIR = '../input'
target_column = 'target'

X_train = pd.read_csv(f'{INPUT_DIR}/train.csv')
X_test = pd.read_csv(f'{INPUT_DIR}/test.csv')
sample_df = pd.read_csv(f'{INPUT_DIR}/sample_submission.csv')  # OPTIONAL

y = X_train[target_column]
X_train = X_train.drop(target_column, axis=1)

lightgbm_params = {
    'max_depth': 8
}

result = run_experiment(lightgbm_params,
                        X_train,
                        y,
                        X_test,
                        sample_submission=sample_df)
```

The `run_experiment` API performs cross-validation and stores artifacts to the logging directory. You will see the output files stored as follows:

```
output
└── 20200130123456          # yyyymmssHHMMSS
    ├── params.json         # Parameters
```

(continues on next page)

(continued from previous page)

```
    ├── metrics.json        # Metrics (single fold & overall CV score)
    ├── oof_prediction.npy  # Out of fold prediction
    ├── test_prediction.npy # Test prediction
    ├── 20200130123456.csv  # Submission csv file
    ├── importances.png     # Feature importance plot
    ├── log.txt             # Log file
    └── models              # The trained models for each fold
        ├── fold1
        ├── fold2
        ├── fold3
        ├── fold4
        └── fold5
```

Hint: The default validation strategy is a 5-fold CV. You can change this behavior by passing `cv` parameter (see API reference in detail).

If you want to use XGBoost, CatBoost or other sklearn estimators, specify the type of algorithm:

```
# CatBoost
catboost_params = {
    'eval_metric': 'Logloss',
    'loss_function': 'Logloss',
    'depth': 8,
    'task_type': 'GPU'
}
result = run_experiment(catboost_params,
                        X_train,
                        y,
                        X_test,
                        algorithm_type='cat')

# XGBoost
xgboost_params = {
    'objective': 'reg:linear',
    'max_depth': 8
}
result = run_experiment(xgboost_params,
                        X_train,
                        y,
                        X_test,
                        algorithm_type='xgb')

# sklearn estimator
from sklearn.linear_model import Ridge
rigde_params = {
    'alpha': 1.0
}
result = run_experiment(rigde_params,
                        X_train,
                        y,
                        X_test,
```

(continues on next page)

(continued from previous page)

```
                          algorithm_type=Ridge)
```

Hint: The parameter will be passed to the constructor of sklearn API (e.g. `LGBMClassifier`).

2.1.2 Collaborating with mlflow

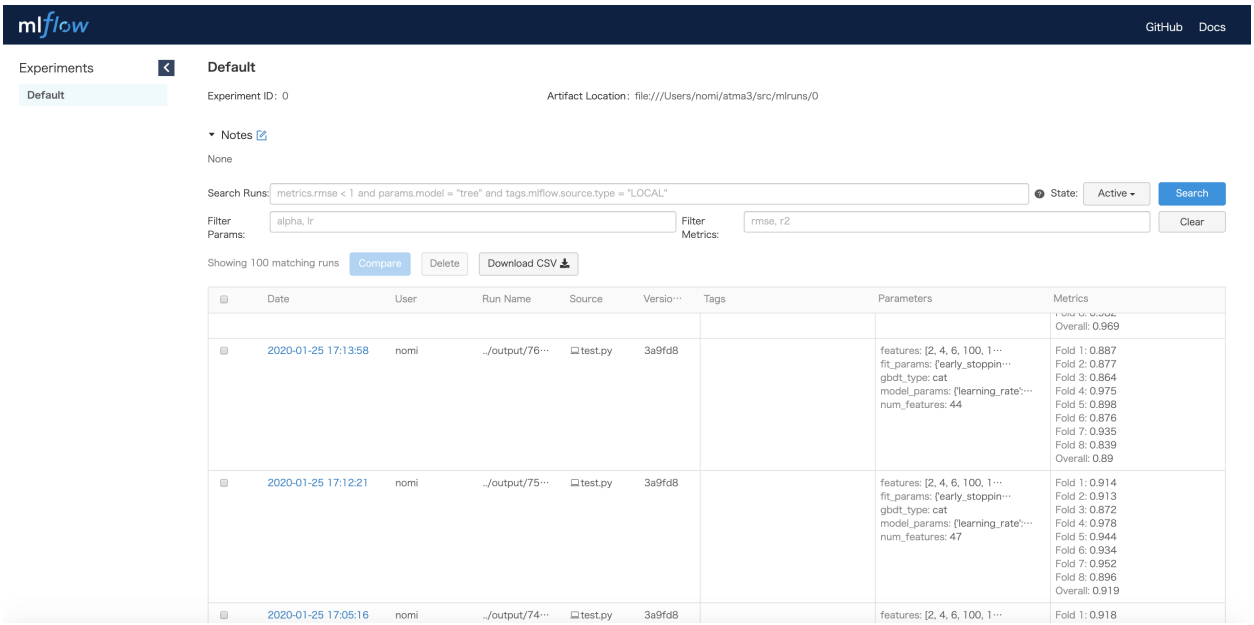
If you want GUI dashboard to manage your experiments, you can use `run_experiment` with mlflow by just setting `with_mlfow = True` (you need to install mlflow beforehand).

```
result = run_experiment(params,
                        X_train,
                        y,
                        X_test,
                        with_mlflow=True)
```

In the same directory as the script executed, run

```
mlflow ui
```

and view it at http://localhost:5000 . On this page, you can see the list of experiments with CV scores and parameters.



If you want to customize the behavior of logging, you can call `run_experiment` in the context of mlflow run. If there is an active run, `run_experiment` will use the currently active run instead of creating a new one.

```
mlflow.set_tracking_uri('gs://ok-i-want-to-use-gcs')

with mlflow.start_run(run_name='your-favorite-run-name'):
    mlflow.log_param('something-you-want-to-log', 42)

    result = run_experiment(params,
```

(continues on next page)

(continued from previous page)

```
                                X_train,
                                y,
                                X_test,
                                with_mlflow=True)
```

2.1.3 What does `run_experiment` not do?

`run_experiment` can be considered as a mere cross-validation API with logging functionality. Therefore, you have to choose model parameters and perform feature engineering yourself.

2.2 Feature management using feature_store

2.2.1 Concept

Feature engineering is one of the most important parts of Kaggle. If you do a lot of feature engineering, it is time-consuming to calculate features each time you build a model.

Many skilled Kagglers save their features to local disk as binary (npy, pickle or feather) to manage their features[1][2][3][4].

`feature_store` provides simple helper APIs for feature management.

```
import pandas as pd
import nyaggle.feature_store as fs

def make_feature_1(df: pd.DataFrame) -> pd.DataFrame:
    return ...

def make_feature_2(df: pd.DataFrame) -> pd.DataFrame:
    return ...

# feature 1
feature_1 = make_feature_1(df)

# feature 2
feature_2 = make_feature_2(df)

# name can be str or int
fs.save_feature(feature_1, "my_feature_1")
fs.save_feature(feature_2, 42, '../my_favorite_feature_store')  # change directory to be␣
↪saved
```

`save_feature` stores dataframe as a feather format under the feature directory (`./features` by default). If you want to load the feature, just call `load_feature` by name.

```
feature_1_restored = fs.load_feature("my_feature_1")
feature_2_restored = fs.load_feature(999)
```

[1] https://www.kaggle.com/c/avito-demand-prediction/discussion/59881
[2] https://github.com/flowlight0/talkingdata-adtracking-fraud-detection
[3] https://www.kaggle.com/c/talkingdata-adtracking-fraud-detection/discussion/55581
[4] https://amalog.hateblo.jp/entry/kaggle-feature-management

To merge all features into the main dataframe, call `load_features` with the main dataframe you want to merge with.

```
train = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')
base_df = pd.concat([train, test])

df_with_features = fs.load_features(base_df, ["my_feature_1", "magic_1", "leaky_1"])
```

Note: `load_features` assumes that the stored feature values are concatenated in the order [train, test].

If you don't like separating your feature engineering code into the independent module, `cached_feature` decorator provides cache functionality. The function with this decorator automatically saves the return value using `save_feature` on the first call, and returns the result of `load_feature` on subsequent calls instead of executing the function body.

```
import pandas as pd
import nyaggle.feature_store as fs

@fs.cached_feature("my_feature_1")
def make_feature_1(df: pd.DataFrame) -> pd.DataFrame:
    ...
    return result

# saves automatically to features/my_feature_1.f
feature_1 = make_feature_1(df)

# loads from saved binary instead of calling make_feature_1
feature_1 = make_feature_1(df)
```

Note: The function decorated by `cached_feature` must return pandas DataFrame.

2.2.2 Use with `run_experiment`

If you pass `feature_list` and `feature_directory` parameters to `run_experiment` API, nyaggle will combine specified features to the given dataframe before performing cross-validation.

List of features is logged as parameters (and of course can be seen in mlflow ui), that makes your experiment cycle much simpler.

```
import pandas as pd
import nyaggle.feature_store as fs
from nyaggle.experiment import run_experiment

run_experiment(params,
               X_train,
               y,
               X_test,
               feature_list=["my_feature_1", "magic_1", "leaky_1"],
               feature_directory="../my_features")
```

2.2.3 Reference

2.3 Advanced usage

2.3.1 Using low-level experiment API

While nyaggle provides `run_experiment` as a high-level API, `Experiment` class can be used as a low-level API that provides primitive functionality for logging experiments.

It is useful when you want to track something other than CV, or need to implement your own CV logic.

```
from nyaggle.experiment import Experiment


with Experiment(logging_directory='./output/') as exp:
    # log key-value pair as a parameter
    exp.log_param('lr', 0.01)
    exp.log_param('optimizer', 'adam')

    # log text
    exp.log('blah blah blah')

    # log metric
    exp.log_metric('CV', 0.85)

    # log numpy ndarray
    exp.log_numpy('predicted', predicted)

    # log pandas dataframe
    exp.log_dataframe('submission', sub, file_format='csv')

    # log any file
    exp.log_artifact('path-to-your-file')


# you can continue logging from existing result
with Experiment.continue_from('./output') as exp:
    ...
```

If you are familiar with mlflow tracking, you may notice that these APIs are similar to mlflow. `Experiment` can be treated as a thin wrapper if you pass `mlflow=True` to the constructor.

```
from nyaggle.experiment import Experiment

with Experiment(logging_directory='./output/', with_mlflow=True) as exp:
    # logging as you want, and you can see the result in mlflow ui
    ...
```

2.3.2 Logging extra parameters to run_experiment

By using `inherit_experiment` parameter, you can mix any additional logging with the results `run_experiment` will create. In the following example, nyaggle records the result of `run_experiment` under the same experiment as the parameter and metrics written outside of the function.

```
from nyaggle.experiment import Experiment, run_experiment

with Experiment(logging_directory='./output/') as exp:

    exp.log_param('my extra param', 'bar')

    run_experiment(..., inherit_experiment=exp)

    exp.log_metrics('my extra metrics', 0.999)
```

2.3.3 Tracking seed averaging experiment

If you train a bunch of models with different seeds to ensemble them, tracking individual models with mlflow will make GUI filled up with these results and make it difficult to manage. A nested run functionality of mlflow is useful to display multiple models together in one result.

```
import mlflow
from nyaggle.ensemble import averaging
from nyaggle.util import make_submission_df

mlflow.start_run()
base_logging_dir = './seed-avg/'
results = []

for i in range(3):
    mlflow.start_run(nested=True)  # use nested-run to place each experiments under the␣
↪parent run
    params['seed'] = i

    result = run_experiment(params,
                            X_train,
                            y_train,
                            X_test,
                            logging_directory=base_logging_dir+f'seed_{i}',
                            with_mlflow=True)
    results.append(result)

    mlflow.end_run()


ensemble = averaging([result.test_prediction for result in results])
sub = make_submission_df(ensemble.test_prediction, pd.read_csv('sample_submission.csv'))
sub.to_csv('ensemble_sub.csv', index=False)
```

CHAPTER

THREE

API REFERENCE

3.1 nyaggle.ensemble

nyaggle.ensemble.**averaging**(*test_predictions*, *oof_predictions=None*, *y=None*, *weights=None*, *eval_func=None*, *rank_averaging=False*)

Perform averaging on model predictions.

Parameters

- **test_predictions** (List[ndarray]) – List of predicted values on test data.
- **oof_predictions** (Optional[List[ndarray]]) – List of predicted values on out-of-fold training data.
- **y** (Optional[Series]) – Target value
- **weights** (Optional[List[float]]) – Weights for each predictions
- **eval_func** (Optional[Callable]) – Evaluation metric used for calculating result score. Used only if oof_predictions and y are given.
- **rank_averaging** (bool) – If True, predictions will be converted to rank before averaging.

Return type
EnsembleResult

Returns

Namedtuple with following members

- **test_prediction:**
  numpy array, Average prediction on test data.
- **oof_prediction:**
  numpy array, Average prediction on Out-of-Fold validation data. None if oof_predictions = None.
- **score:**
  float, Calculated score on Out-of-Fold data. None if eval_func is None.

nyaggle.ensemble.**averaging_opt**(*test_predictions*, *oof_predictions*, *y*, *eval_func*, *higher_is_better*, *weight_bounds=(0.0, 1.0)*, *rank_averaging=False*, *method=None*)

Perform averaging with optimal weights using scipy.optimize.

Parameters

- **test_predictions** (List[ndarray]) – List of predicted values on test data.

- **oof_predictions** (Optional[List[ndarray]]) – List of predicted values on out-of-fold training data.
- **y** (Optional[Series]) – Target value
- **eval_func** (Optional[Callable[[ndarray, ndarray], float]]) – Evaluation metric f(y_true, y_pred) used for calculating result score. Used only if oof_predictions and y are given.
- **higher_is_better** (bool) – Determine the direction of optimize eval_func.
- **weight_bounds** (Tuple[float, float]) – Specify lower/upper bounds of each weight.
- **rank_averaging** (bool) – If True, predictions will be converted to rank before averaging.
- **method** (Optional[str]) – Type of solver. If None, SLSQP will be used.

Return type
EnsembleResult

Returns

Namedtuple with following members

- **test_prediction:**
  numpy array, Average prediction on test data.
- **oof_prediction:**
  numpy array, Average prediction on Out-of-Fold validation data. None if oof_predictions = None.
- **score:**
  float, Calculated score on Out-of-Fold data. None if eval_func is None.

nyaggle.ensemble.**stacking**(*test_predictions*, *oof_predictions*, *y*, *estimator=None*, *cv=None*, *groups=None*, *type_of_target='auto'*, *eval_func=None*)

Perform stacking on predictions.

Parameters

- **test_predictions** (List[ndarray]) – List of predicted values on test data.
- **oof_predictions** (List[ndarray]) – List of predicted values on out-of-fold training data.
- **y** (Series) – Target value
- **estimator** (Optional[BaseEstimator]) – Estimator used for the 2nd-level model. If None, the default estimator (auto-tuned linear model) will be used.
- **cv** (Union[int, Iterable, BaseCrossValidator, None]) – int, cross-validation generator or an iterable which determines the cross-validation splitting strategy.
  - None, to use the default KFold(5, random_state=0, shuffle=True),
  - integer, to specify the number of folds in a (Stratified)KFold,
  - CV splitter (the instance of BaseCrossValidator),
  - An iterable yielding (train, test) splits as arrays of indices.
- **groups** (Optional[Series]) – Group labels for the samples. Only used in conjunction with a "Group" cv instance (e.g., GroupKFold).
- **type_of_target** (str) – The type of target variable. If auto, type is inferred by sklearn.utils.multiclass.type_of_target. Otherwise, binary, continuous, or multiclass are supported.

- **eval_func** (Optional[Callable]) – Evaluation metric used for calculating result score. Used only if oof_predictions and y are given.

Return type
EnsembleResult

Returns

Namedtuple with following members

- **test_prediction:**
  numpy array, Average prediction on test data.
- **oof_prediction:**
  numpy array, Average prediction on Out-of-Fold validation data. None if oof_predictions = None.
- **score:**
  float, Calculated score on Out-of-Fold data. None if eval_func is None.

3.2 nyaggle.experiment

class nyaggle.experiment.**Experiment**(*logging_directory*, *custom_logger=None*, *with_mlflow=False*, *if_exists='error'*)

Minimal experiment logger for Kaggle

This module provides minimal functionality for tracking experiments. The output files are laid out as follows:

```
<logging_directory>/
    log.txt       <== Output of log
    metrics.json  <== Output of log_metric(s), format: name,score
    params.json   <== Output of log_param(s), format: key,value
    mlflow.json   <== mlflow's run_id, experiment_id and artifact_uri (logged if␣
↪with_mlflow=True)
```

You can add numpy array and pandas dataframe under the directory through log_numpy and log_dataframe.

Parameters

- **logging_directory** (str) – Path to directory where output is stored.
- **custom_logger** (Optional[Logger]) – A custom logger to be used instead of default logger.
- **with_mlflow** (bool) – If True, mlflow tracking is used. One instance of nyaggle.experiment.Experiment corresponds to one run in mlflow. Note that all output files are located both logging_directory and mlflow's directory (mlruns by default).
- **if_exists** (str) – How to behave if the logging directory already exists.
  - error: Raise a ValueError.
  - replace: Delete logging directory before logging.
  - append: Append to exisitng experiment.
  - rename: Rename current directory by adding "_1", "_2"… prefix

Example

```
>>> import numpy as np
>>> import pandas as pd
>>> from nyaggle.experiment import Experiment
>>>
>>> with Experiment(logging_directory='./output/') as exp:
>>>     # log key-value pair as a parameter
>>>     exp.log_param('lr', 0.01)
>>>     exp.log_param('optimizer', 'adam')
>>>
>>>     # log text
>>>     exp.log('blah blah blah')
>>>
>>>     # log metric
>>>     exp.log_metric('CV', 0.85)
>>>
>>>     # log dictionary with flattening keys
>>>     exp.log_dict('params', {'X': 3, 'Y': {'Z': 'foobar'}})
>>>
>>>     # log numpy ndarray, pandas dafaframe and any artifacts
>>>     exp.log_numpy('predicted', np.zeros(1))
>>>     exp.log_dataframe('submission', pd.DataFrame(), file_format='csv')
>>>     exp.log_artifact('path-to-your-file')
```

get_logger()

Get logger used in this experiment.

Return type
Logger

Returns
logger object

get_run()

Get mlflow's currently active run, or None if `with_mlflow = False`.

Returns
active Run

log(*text*)

Logs a message on the logger for the experiment.

Parameters
text (str) – The message to be written.

log_artifact(*src_file_path*)

Make a copy of the file under the logging directory.

Parameters
src_file_path (str) – Path of the file. If path is not a child of the logging directory, the file will be copied. If `with_mlflow` is True, `mlflow.log_artifact` will be called (then another copy will be made).

log_dataframe(*name*, *df*, *file_format='feather'*)

Log a pandas dataframe under the logging directory.

Parameters

- **name** (str) – Name of the file. A `.f` or `.csv` extension will be appended to the file name if it does not already have one.
- **df** (DataFrame) – A dataframe to be saved.
- **file_format** (str) – A format of output file. `csv` and `feather` are supported.

log_dict(*name*, *value*, *separator='.'*)

Logs a dictionary as parameter with flatten format.

Parameters

- **name** (str) – Parameter name
- **value** (Dict) – Parameter value
- **separator** (str) – Separating character used to concatanate keys

Examples

```
>>> with Experiment('./') as e:
>>>     e.log_dict('a', {'b': 1, 'c': 'd'})
>>>     print(e.params)
{ 'a.b': 1, 'a.c': 'd' }
```

log_metric(*name*, *score*)

Log a metric under the logging directory.

Parameters

- **name** (str) – Metric name.
- **score** (float) – Metric value.

log_metrics(*metrics*)

Log a batch of metrics under the logging directory.

Parameters
metrics (Dict) – dictionary of metrics.

log_numpy(*name*, *array*)

Log a numpy ndarray under the logging directory.

Parameters

- **name** (str) – Name of the file. A .npy extension will be appended to the file name if it does not already have one.
- **array** (ndarray) – Array data to be saved.

log_param(*key*, *value*)

Logs a key-value pair for the experiment.

Parameters

- **key** – parameter name
- **value** – parameter value

log_params(*params*)

Logs a batch of params for the experiments.

Parameters
params (Dict) – dictionary of parameters

start()

Start a new experiment.

stop()

Stop current experiment.

nyaggle.experiment.**add_leaderboard_score**(*logging_directory*, *score*)

Record leaderboard score to the existing experiment directory.

Parameters

- **logging_directory** (str) – The directory to be added
- **score** (float) – Leaderboard score

nyaggle.experiment.**find_best_lgbm_parameter**(*base_param*, *X*, *y*, *cv=None*, *groups=None*, *time_budget=None*, *type_of_target='auto'*)

Search hyperparameter for lightgbm using optuna.

Parameters

- **base_param** (Dict) – Base parameters passed to lgb.train.
- **X** (DataFrame) – Training data.
- **y** (Series) – Target
- **cv** (Union[int, Iterable, BaseCrossValidator, None]) – int, cross-validation generator or an iterable which determines the cross-validation splitting strategy.
- **groups** (Optional[Series]) – Group labels for the samples. Only used in conjunction with a "Group" cv instance (e.g., GroupKFold).
- **time_budget** (Optional[int]) – Time budget for tuning (in seconds).
- **type_of_target** (str) – The type of target variable. If auto, type is inferred by sklearn.utils.multiclass.type_of_target. Otherwise, binary, continuous, or multiclass are supported.

Return type
Dict

Returns
The best parameters found

nyaggle.experiment.**run_experiment**(*model_params*, *X_train*, *y*, *X_test=None*, *logging_directory='output/{time}'*, *if_exists='error'*, *eval_func=None*, *algorithm_type='lgbm'*, *fit_params=None*, *cv=None*, *groups=None*, *categorical_feature=None*, *sample_submission=None*, *submission_filename=None*, *type_of_target='auto'*, *feature_list=None*, *feature_directory=None*, *inherit_experiment=None*, *with_auto_hpo=False*, *with_auto_prep=False*, *with_mlflow=False*)

Evaluate metrics by cross-validation and stores result (log, oof prediction, test prediction, feature importance plot and submission file) under the directory specified.

One of the following estimators are used (automatically dispatched by type_of_target(y) and gbdt_type).

- LGBMClassifier
- LGBMRegressor
- CatBoostClassifier
- CatBoostRegressor

The output files are laid out as follows:

```
<logging_directory>/
    log.txt                 <== Logging file
    importance.png          <== Feature importance plot generated by nyaggle.util.
↪plot_importance
    oof_prediction.npy      <== Out of fold prediction in numpy array format
    test_prediction.npy     <== Test prediction in numpy array format
    submission.csv          <== Submission csv file
    metrics.json            <== Metrics
    params.json             <== Parameters
    models/
        fold1               <== The trained model in fold 1
        ...
```

Parameters

- **model_params** (Dict[str, Any]) – Parameters passed to the constructor of the classifier/regressor object (i.e. LGBMRegressor).
- **X_train** (DataFrame) – Training data. Categorical feature should be casted to pandas categorical type or encoded to integer.
- **y** (Series) – Target
- **X_test** (Optional[DataFrame]) – Test data (Optional). If specified, prediction on the test data is performed using ensemble of models.
- **logging_directory** (str) – Path to directory where output of experiment is stored.
- **if_exists** (str) – How to behave if the logging directory already exists.
  - error: Raise a ValueError.
  - replace: Delete logging directory before logging.
  - append: Append to exisitng experiment.
  - rename: Rename current directory by adding "_1", "_2"… prefix
- **fit_params** (Union[Dict[str, Any], Callable, None]) – Parameters passed to the fit method of the estimator. If dict is passed, the same parameter except eval_set passed for each fold. If callable is passed, returning value of `fit_params(fold_id, train_index, test_index)` will be used for each fold.
- **eval_func** (Optional[Callable]) – Function used for logging and calculation of returning scores. This parameter isn't passed to GBDT, so you should set objective and eval_metric separately if needed. If `eval_func` is None, `roc_auc_score` or `mean_squared_error` is used by default.
- **gbdt_type** – Type of gradient boosting library used. "lgbm" (lightgbm) or "cat" (catboost)
- **cv** (Union[int, Iterable, BaseCrossValidator, None]) – int, cross-validation generator or an iterable which determines the cross-validation splitting strategy.

  - None, to use the default `KFold(5, random_state=0, shuffle=True)`,
  - integer, to specify the number of folds in a `(Stratified)KFold`,
  - CV splitter (the instance of `BaseCrossValidator`),
  - An iterable yielding (train, test) splits as arrays of indices.
- **groups** (`Optional[Series]`) – Group labels for the samples. Only used in conjunction with a "Group" cv instance (e.g., `GroupKFold`).
- **sample_submission** (`Optional[DataFrame]`) – A sample dataframe alined with test data (Usually in Kaggle, it is available as sample_submission.csv). The submission file will be created with the same schema as this dataframe.
- **submission_filename** (`Optional[str]`) – The name of submission file will be created under logging directory. If `None`, the basename of the logging directory will be used as a filename.
- **categorical_feature** (`Optional[List[str]]`) – List of categorical column names. If `None`, categorical columns are automatically determined by dtype.
- **type_of_target** (`str`) – The type of target variable. If `auto`, type is inferred by `sklearn.utils.multiclass.type_of_target`. Otherwise, `binary`, `continuous`, or `multiclass` are supported.
- **feature_list** (`Optional[List[Union[int, str]]]`) – The list of feature ids saved through nyaggle.feature_store module.
- **feature_directory** (`Optional[str]`) – The location of features stored. Only used if feature_list is not empty.
- **inherit_experiment** (`Optional[`*`Experiment`*`]`) – An experiment object which is used to log results. if not `None`, all logs in this function are treated as a part of this experiment.
- **with_auto_prep** (`bool`) – If True, the input datasets will be copied and automatic preprocessing will be performed on them. For example, if `gbdt_type = 'cat'`, all missing values in categorical features will be filled.
- **with_auto_hpo** (`bool`) – If True, model parameters will be automatically updated using optuna (only available in lightgbm).
- **with_mlflow** (`bool`) – If True, mlflow tracking is used. One instance of `nyaggle.experiment.Experiment` corresponds to one run in mlflow. Note that all output mlflow's directory (`mlruns` by default).

Returns

Namedtuple with following members

- **oof_prediction:**
  numpy array, shape (len(X_train),) Predicted value on Out-of-Fold validation data.
- **test_prediction:**
  numpy array, shape (len(X_test),) Predicted value on test data. `None` if X_test is `None`
- **metrics:**
  list of float, shape(nfolds+1) `metrics[i]` denotes validation score in i-th fold. `metrics[-1]` is overall score.
- **models:**
  list of objects, shape(nfolds) Trained models for each folds.

- **importance:**
  list of pd.DataFrame, feature importance for each fold (type="gain").
- **time:**
  Training time in seconds.
- **submit_df:**
  The dataframe saved as submission.csv

3.3 nyaggle.feature_store

nyaggle.feature_store.**cached_feature**(*feature_name*, *directory='./features/'*, *ignore_columns=None*)

Decorator to wrap a function which returns pd.DataFrame with a memorizing callable that saves dataframe using `feature_store.save_feature`.

Parameters

- **feature_name** (`Union[int, str]`) – The name of the feature (used in `save_feature`).
- **directory** (`str`) – The directory where the feature is stored.
- **ignore_columns** (`Optional[List[str]]`) – The list of columns that will be dropped from the loaded dataframe.

Example

```
>>> from nyaggle.feature_store import cached_feature
>>>
>>> @cached_feature('x')
>>> def make_feature_x(param) -> pd.DataFrame:
>>>     print('called')
>>>     ...
>>>     return df
>>>
>>> x = make_feature_x(...)  # if x.f does not exist, call the function and save␣
↪result to x.f
"called"
>>> x = make_feature_x(...)  # load from file in the second time
```

nyaggle.feature_store.**load_feature**(*feature_name*, *directory='./features/'*, *ignore_columns=None*)

Load feature as pandas DataFrame.

Parameters

- **feature_name** (`Union[int, str]`) – The name of the feature (used in `save_feature`).
- **directory** (`str`) – The directory where the feature is stored.
- **ignore_columns** (`Optional[List[str]]`) – The list of columns that will be dropped from the loaded dataframe.

Return type
`DataFrame`

Returns
The feature dataframe

nyaggle.feature_store.**load_features**(*base_df*, *feature_names*, *directory='./features/'*, *ignore_columns=None*, *create_directory=True*, *rename_duplicate=True*)

Load features and returns concatenated dataframe

Parameters

- **base_df** (Optional[DataFrame]) – The base dataframe. If not None, resulting dataframe will consist of base and loaded feature columns.
- **feature_names** (List[Union[int, str]]) – The list of feature names to be loaded.
- **directory** (str) – The directory where the feature is stored.
- **ignore_columns** (Optional[List[str]]) – The list of columns that will be dropped from the loaded dataframe.
- **create_directory** (bool) – If True, create directory if not exists.
- **rename_duplicate** (bool) – If True, duplicated column name will be renamed automatically (feature name will be used as suffix). If False, duplicated columns will be as-is.

Return type
DataFrame

Returns
The merged dataframe

nyaggle.feature_store.**save_feature**(*df*, *feature_name*, *directory='./features/'*, *with_csv_dump=False*, *create_directory=True*, *reference_target_variable=None*, *overwrite=False*)

Save pandas dataframe as feather-format

Parameters

- **df** (DataFrame) – The dataframe to be saved.
- **feature_name** (Union[int, str]) – The name of the feature. The output file will be {feature_name}.f.
- **directory** (str) – The directory where the feature will be stored.
- **with_csv_dump** (bool) – If True, the first 1000 lines are dumped to csv file for debug.
- **create_directory** (bool) – If True, create directory if not exists.
- **reference_target_variable** (Optional[Series]) – If not None, instant validation will be made on the feature.
- **overwrite** (bool) – If False and file already exists, RuntimeError will be raised.

3.4 nyaggle.feature

class nyaggle.feature.category_encoder.**KFoldEncoderWrapper**(*base_transformer*, *cv=None*, *return_same_type=True*, *groups=None*)

KFold Wrapper for sklearn like interface

This class wraps sklearn's TransformerMixIn (object that has fit/transform/fit_transform methods), and call it as K-fold manner.

Parameters

- **base_transformer** (BaseEstimator) – Transformer object to be wrapped.
- **cv** (Union[int, Iterable, BaseCrossValidator, None]) – int, cross-validation generator or an iterable which determines the cross-validation splitting strategy.
  - None, to use the default KFold(5, random_state=0, shuffle=True),
  - integer, to specify the number of folds in a (Stratified)KFold,
  - CV splitter (the instance of BaseCrossValidator),
  - An iterable yielding (train, test) splits as arrays of indices.
- **groups** (Optional[Series]) – Group labels for the samples. Only used in conjunction with a "Group" cv instance (e.g., GroupKFold).
- **return_same_type** (bool) – If True, *transform* and *fit_transform* return the same type as X. If False, these APIs always return a numpy array, similar to sklearn's API.

fit(*X*, *y*)

Fit models for each fold.

Parameters

- **X** (DataFrame) – Data
- **y** (Series) – Target

Returns

returns the transformer object.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit models for each fold, then transform X

Parameters

- **X** (Union[DataFrame, ndarray]) – Data
- **y** (Optional[Series]) – Target
- **fit_params** – Additional parameters passed to models

Return type

Union[DataFrame, ndarray]

Returns

Transformed version of X. It will be pd.DataFrame If X is *pd.DataFrame* and return_same_type is True.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters

deep (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params – Parameter names mapped to their values.

Return type

dict

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters
****params** (*dict*) – Estimator parameters.

Returns
self – Estimator instance.

Return type
estimator instance

`transform`(*X*)

Transform X

Parameters
X (Union[DataFrame, ndarray]) – Data

Return type
Union[DataFrame, ndarray]

Returns
Transformed version of X. It will be pd.DataFrame If X is *pd.DataFrame* and return_same_type is True.

`class` nyaggle.feature.category_encoder.**TargetEncoder**(*cv=None*, *groups=None*, *cols=None*, *drop_invariant=False*, *handle_missing='value'*, *handle_unknown='value'*, *min_samples_leaf=20*, *smoothing=10.0*, *return_same_type=True*)

Target Encoder

KFold version of category_encoders.TargetEncoder in https://contrib.scikit-learn.org/categorical-encoding/targetencoder.html.

Parameters

- **cv** (Union[Iterable, BaseCrossValidator, None]) – int, cross-validation generator or an iterable which determines the cross-validation splitting strategy.
  - None, to use the default KFold(5, random_state=0, shuffle=True),
  - integer, to specify the number of folds in a (Stratified)KFold,
  - CV splitter (the instance of BaseCrossValidator),
  - An iterable yielding (train, test) splits as arrays of indices.
- **groups** (Optional[Series]) – Group labels for the samples. Only used in conjunction with a "Group" cv instance (e.g., GroupKFold).
- **cols** (Optional[List[str]]) – A list of columns to encode, if None, all string columns will be encoded.
- **drop_invariant** (bool) – Boolean for whether or not to drop columns with 0 variance.
- **handle_missing** (str) – Options are 'error', 'return_nan' and 'value', defaults to 'value', which returns the target mean.

- **handle_unknown** (str) – Options are 'error', 'return_nan' and 'value', defaults to 'value', which returns the target mean.
- **min_samples_leaf** (int) – Minimum samples to take category average into account.
- **smoothing** (float) – Smoothing effect to balance categorical average vs prior. Higher value means stronger regularization. The value must be strictly bigger than 0.
- **return_same_type** (bool) – If True, transform and fit_transform return the same type as X. If False, these APIs always return a numpy array, similar to sklearn's API.

fit(*X*, *y*)

Fit models for each fold.

Parameters

- **X** (DataFrame) – Data
- **y** (Series) – Target

Returns
returns the transformer object.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit models for each fold, then transform X

Parameters

- **X** (Union[DataFrame, ndarray]) – Data
- **y** (Optional[Series]) – Target
- **fit_params** – Additional parameters passed to models

Return type
Union[DataFrame, ndarray]

Returns
Transformed version of X. It will be pd.DataFrame If X is *pd.DataFrame* and return_same_type is True.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters
deep (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns
params – Parameter names mapped to their values.

Return type
dict

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as Pipeline). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Parameters
****params** (*dict*) – Estimator parameters.

Returns
self – Estimator instance.

Return type
estimator instance

`transform`(*X*)

Transform X

Parameters
X (Union[DataFrame, ndarray]) – Data

Return type
Union[DataFrame, ndarray]

Returns
Transformed version of X. It will be pd.DataFrame If X is *pd.DataFrame* and return_same_type is True.

`class` nyaggle.feature.nlp.`BertSentenceVectorizer`(*lang='en'*, *n_components=None*, *text_columns=None*, *pooling_strategy='reduce_mean'*, *use_cuda=False*, *tokenizer=None*, *model=None*, *return_same_type=True*, *column_format='{col}_{idx}'*)

Sentence Vectorizer using BERT pretrained model.

Extract fixed-length feature vector from English/Japanese variable-length sentence using BERT.

Parameters

- **lang** (str) – Type of language. If set to "jp", Japanese BERT model is used (you need to install MeCab).
- **n_components** (Optional[int]) – Number of components in SVD. If *None*, SVD is not applied.
- **text_columns** (Optional[List[str]]) – List of processing columns. If *None*, all object columns are regarded as text column.
- **pooling_strategy** (str) – The pooling algorithm for generating fixed length encoding vector. 'reduce_mean' and 'reduce_max' use average pooling and max pooling respectively to reduce vector from (num-words, emb-dim) to (emb_dim). 'reduce_mean_max' performs 'reduce_mean' and 'reduce_max' separately and concat them. 'cls_token' takes the first element (i.e. [CLS]).
- **use_cuda** (bool) – If *True*, inference is performed on GPU.
- **tokenizer** (Optional[PreTrainedTokenizer]) – The custom tokenizer used instead of default tokenizer
- **model** – The custom pretrained model used instead of default BERT model
- **return_same_type** (bool) – If True, *transform* and *fit_transform* return the same type as X. If False, these APIs always return a numpy array, similar to sklearn's API.
- **column_format** (str) – Name of transformed columns (used if returning type is pd.DataFrame)

`fit`(*X*, *y=None*)

Fit SVD model on training data X.

Parameters

- **X** (Union[DataFrame, ndarray]) – Data
- **y** – Ignored

fit_transform(*X*, *y=None*, ***fit_params*)

Fit SVD model on training data X and perform feature extraction and dimensionality reduction using BERT pre-trained model and trained SVD model.

Parameters

- **X** (Union[DataFrame, ndarray]) – Data
- **y** – Ignored

transform(*X*, *y=None*)

Perform feature extraction and dimensionality reduction using BERT pre-trained model and trained SVD model.

Parameters

- **X** (Union[DataFrame, ndarray]) – Data
- **y** – Ignored

nyaggle.feature.groupby.**aggregation**(*input_df*, *group_key*, *group_values*, *agg_methods*)

Aggregate values after grouping table rows by a given key.

Parameters

- **input_df** (DataFrame) – Input data frame.
- **group_key** (str) – Used to determine the groups for the groupby.
- **group_values** (List[str]) – Used to aggregate values for the groupby.
- **agg_methods** (List[Union[str, FunctionType]]) – List of function or function names, e.g. ['mean', 'max', 'min', numpy.mean]. Do not use a lambda function because the name attribute of the lambda function cannot generate a unique string of column names in <lambda>.

Return type
Tuple[DataFrame, List[str]]

Returns
Tuple of output dataframe and new column names.

3.5 nyaggle.hyper_parameters

nyaggle.hyper_parameters.**get_hyperparam_byname**(*name*, *gbdt_type='lgbm'*, *with_metadata=False*)

Get a hyperparameter by parameter name

Parameters

- **name** (str) – The name of parameter (e.g. "ieee-2019-10th").
- **gbdt_type** (str) – The type of gbdt library. lgbm, cat, xgb can be used.
- **with_metadata** (bool) – When set to True, parameters are wrapped by metadata dictionary which contains information about source URL, competition name etc.

Return type
Dict

Returns
A hyperparameter dictionary.

nyaggle.hyper_parameters.**list_hyperparams**(*gbdt_type='lgbm'*, *with_metadata=False*)

List all hyperparameters

Parameters

- **gbdt_type** (str) – The type of gbdt library. lgbm, cat, xgb can be used.
- **with_metadata** (bool) – When set to True, parameters are wrapped by metadata dictionary which contains information about source URL, competition name etc.

Return type
List[Dict]

Returns
A list of hyper-parameters used in Kaggle gold medal solutions

3.6 nyaggle.util

nyaggle.util.**is_instance**(*obj*, *class_path_str*)

Acts as a safe version of isinstance without having to explicitly import packages which may not exist in the users environment. Checks if obj is an instance of type specified by class_path_str. :type obj: :param obj: Some object you want to test against :type obj: Any :type class_path_str: Union[str, List, Tuple] :param class_path_str: A string or list of strings specifying full class paths

Example: *sklearn.ensemble.RandomForestRegressor*

Returns
bool

Return type
True if isinstance is true and the package exists, False otherwise

nyaggle.util.**make_submission_df**(*test_prediction*, *sample_submission=None*, *y=None*)

Make a dataframe formatted as a kaggle competition style.

Parameters

- **test_prediction** (ndarray) – A test prediction to be formatted.
- **sample_submission** (Optional[DataFrame]) – A sample dataframe alined with test data (Usually in Kaggle, it is available as sample_submission.csv). The submission file will be created with the same schema as this dataframe.
- **y** (Optional[Series]) – Target variables which is used for inferring the column name. Ignored if sample_submission is passed.

Return type
DataFrame

Returns
The formatted dataframe

nyaggle.util.**plot_importance**(*importance*, *path=None*, *top_n=100*, *figsize=None*, *title=None*)

Plot feature importance and write to image

Parameters

- **importance** (DataFrame) – The dataframe which has "feature" and "importance" column
- **path** (Optional[str]) – The file path to be saved
- **top_n** (int) – The number of features to be visualized
- **figsize** (Optional[Tuple[int, int]]) – The size of the figure
- **title** (Optional[str]) – The title of the plot

Example

```
>>> import pandas as pd
>>> import lightgbm as lgb
>>> from nyaggle.util import plot_importance
>>> from sklearn.datasets import make_classification
```

```
>>> X, y = make_classification()
>>> X = pd.DataFrame(X, columns=['col{}'.format(i) for i in range(X.shape[1])])
>>> booster = lgb.train({'objective': 'binary'}, lgb.Dataset(X, y))
>>> importance = pd.DataFrame({
>>>     'feature': X.columns,
>>>     'importance': booster.feature_importance('gain')
>>> })
>>> plot_importance(importance, 'importance.png')
```

3.7 nyaggle.validation

class nyaggle.validation.**Nth**(*n*, *base_validator*)

Returns N-th fold of the base validator

This validator wraps the base validator to take n-th (1-origin) fold.

Parameters

- **n** (int) – The number of folds to be taken.
- **base_validator** (BaseCrossValidator) – The base validator to be wrapped.

Example

```
>>> import numpy as np
>>> import pandas as pd
>>> from sklearn.model_selection import KFold
>>> from nyaggle.validation import Nth
```

```
>>> # take the 3rd fold
>>> folds = Nth(3, KFold(5))
>>> folds.get_n_splits()
1
```

get_n_splits(*X=None*, *y=None*, *groups=None*)

Returns the number of splitting iterations in the cross-validator

split(*X*, *y=None*, *groups=None*)

Generate indices to split data into training and test set.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Training data, where *n_samples* is the number of samples and *n_features* is the number of features.
- **y** (*array-like of shape (n_samples,)*) – The target variable for supervised learning problems.
- **groups** (*array-like of shape (n_samples,), default=None*) – Group labels for the samples used while splitting the dataset into train/test set.

Yields

- **train** (*ndarray*) – The training set indices for that split.
- **test** (*ndarray*) – The testing set indices for that split.

class nyaggle.validation.**Skip**(*n*, *base_validator*)

Skips the first N folds and returns the remaining folds

This validator wraps the base validator to skip first n folds.

Parameters

- **n** (int) – The number of folds to be skipped.
- **base_validator** (BaseCrossValidator) – The base validator to be wrapped.

Example

```
>>> import numpy as np
>>> import pandas as pd
>>> from sklearn.model_selection import KFold
>>> from nyaggle.validation import Skip
```

```
>>> # take the last 2 folds out of 5
>>> folds = Skip(3, KFold(5))
>>> folds.get_n_splits()
2
```

get_n_splits(*X=None*, *y=None*, *groups=None*)

Returns the number of splitting iterations in the cross-validator

split(*X*, *y=None*, *groups=None*)

Generate indices to split data into training and test set.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Training data, where *n_samples* is the number of samples and *n_features* is the number of features.
- **y** (*array-like of shape (n_samples,)*) – The target variable for supervised learning problems.

- **groups** (*array-like of shape (n_samples,), default=None*) – Group labels for the samples used while splitting the dataset into train/test set.

Yields

- **train** (*ndarray*) – The training set indices for that split.
- **test** (*ndarray*) – The testing set indices for that split.

class nyaggle.validation.**SlidingWindowSplit**(*source*, *train_from*, *train_to*, *test_from*, *test_to*, *n_windows*, *stride*)

Sliding window time series cross-validator

Time Series cross-validator which provides train/test indices based on the sliding window to split variable interval time series data. Splitting for each fold will be as follows:

```
Folds  Training data                                    Testing data
1      ((train_from-(N-1)*stride, train_to-(N-1)*stride), (test_from-(N-1)*stride,␣
↪test_to-(N-1)*stride))
...    ...                                              ...
N-1    ((train_from-stride,       train_to-stride),       (test_from-stride,      ␣
↪test_to-stride))
N      ((train_from,              train_to),              (test_from,             ␣
↪test_to))
```

This class is compatible with sklearn's `BaseCrossValidator` (base class of `KFold`, `GroupKFold` etc).

Parameters

- **source** (`Union[Series, str]`) – The column name or series of timestamp.
- **train_from** (`Union[datetime, str]`) – Start datetime for the training data in the base split.
- **train_to** (`Union[datetime, str]`) – End datetime for the training data in the base split.
- **test_from** (`Union[datetime, str]`) – Start datetime for the testing data in the base split.
- **test_to** (`Union[datetime, str]`) – End datetime for the testing data in the base split.
- **n_windows** (`int`) – The number of windows (or folds) in the validation.
- **stride** (`timedelta`) – Time delta between folds.

class nyaggle.validation.**StratifiedGroupKFold**(*n_splits=3*, *shuffle=False*, *random_state=None*)

Stratified K-Folds cross-validator with grouping

Provides train/test indices to split data in train/test sets. This cross-validation object is a variation of GroupKFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class. Read more in the User Guide.

Parameters

n_splits (`int`) – Number of folds. Must be at least 2.

Example

```
>>> from pprint import pprint
>>> rng = np.random.RandomState(0)
>>> groups = [1, 1, 3, 4, 2, 2, 7, 8, 8]
>>> y      = [1, 1, 1, 1, 2, 2, 2, 3, 3]
>>> X = np.empty((len(y), 0))
>>> self = StratifiedGroupKFold(random_state=rng)
>>> skf_list = list(self.split(X=X, y=y, groups=groups))
>>> pprint(skf_list)
[
    (np.array([2, 3, 4, 5, 6]), np.array([0, 1, 7, 8])),
    (np.array([0, 1, 2, 7, 8]), np.array([3, 4, 5, 6])),
    (np.array([0, 1, 3, 4, 5, 6, 7, 8]), np.array([2])),
]
```

split(*X*, *y*, *groups=None*)

Generate indices to split data into training and test set.

class nyaggle.validation.**Take**(*n*, *base_validator*)

Returns the first N folds of the base validator

This validator wraps the base validator to take first n folds.

Parameters

- **n** (int) – The number of folds.
- **base_validator** (BaseCrossValidator) – The base validator to be wrapped.

Example

```
>>> import numpy as np
>>> import pandas as pd
>>> from sklearn.model_selection import KFold
>>> from nyaggle.validation import Take
```

```
>>> # take the first 3 folds out of 5
>>> folds = Take(3, KFold(5))
>>> folds.get_n_splits()
3
```

get_n_splits(*X=None*, *y=None*, *groups=None*)

Returns the number of splitting iterations in the cross-validator

split(*X*, *y=None*, *groups=None*)

Generate indices to split data into training and test set.

Parameters

- **X** – Training data.
- **y** – Target.
- **groups** – Group indices.

Yields
The training set and the testing set indices for that split.

class nyaggle.validation.**TimeSeriesSplit**(*source*, *times=None*)

Time Series cross-validator

Time Series cross-validator which provides train/test indices to split variable interval time series data. This class provides low-level API for time series validation strategy. This class is compatible with sklearn's `BaseCrossValidator` (base class of `KFold`, `GroupKFold` etc).

Parameters

- **source** (`Union[Series, str]`) – The column name or series of timestamp.
- **times** (`Optional[List[Tuple[Tuple[Union[datetime, str], Union[datetime, str]], Tuple[Union[datetime, str], Union[datetime, str]]]]]`) – Splitting window, where times[i][0] and times[i][1] denotes train and test time interval in (i-1)th fold respectively. Each time interval should be pair of datetime or str, and the validator generates indices of rows where timestamp is in the half-open interval [start, end). For example, if `times[i][0] = ('2018-01-01', '2018-01-03')`, indices for (i-1)th training data will be rows where timestamp value meets `2018-01-01 <= t < 2018-01-03`.

Example

```
>>> import numpy as np
>>> import pandas as pd
>>> from nyaggle.validation import TimeSeriesSplit
>>> df = pd.DataFrame()
>>> df['time'] = pd.date_range(start='2018/1/1', periods=5)
```

```
>>> folds = TimeSeriesSplit('time',
>>>                         [(('2018-01-01', '2018-01-02'), ('2018-01-02', '2018-01-
↪04')),
>>>                          (('2018-01-02', '2018-01-03'), ('2018-01-04', '2018-01-
↪06'))])
```

```
>>> folds.get_n_splits()
2
```

```
>>> splits = folds.split(df)
```

```
>>> train_index, test_index = next(splits)
>>> train_index
[0]
>>> test_index
[1, 2]
```

```
>>> train_index, test_index = next(splits)
>>> train_index
[1]
>>> test_index
[3, 4]
```

add_fold(*train_interval*, *test_interval*)

Append 1 split to the validator.

Parameters

- **train_interval** (Tuple[Union[datetime, str], Union[datetime, str]]) – start and end time of training data.
- **test_interval** (Tuple[Union[datetime, str], Union[datetime, str]]) – start and end time of test data.

get_n_splits(*X=None*, *y=None*, *groups=None*)

Returns the number of splitting iterations in the cross-validator

split(*X*, *y=None*, *groups=None*)

Generate indices to split data into training and test set.

Parameters

- **X** – Training data.
- **y** – Ignored.
- **groups** – Ignored.

Yields

The training set and the testing set indices for that split.

nyaggle.validation.**adversarial_validate**(*X_train*, *X_test*, *importance_type='gain'*, *estimator=None*, *categorical_feature=None*, *cv=None*)

Perform adversarial validation between X_train and X_test.

Parameters

- **X_train** (DataFrame) – Training data
- **X_test** (DataFrame) – Test data
- **importance_type** (str) – The type of feature importance calculated.
- **estimator** (Optional[BaseEstimator]) – The custom estimator. If None, LGBMClassifier is automatically used. Only LGBMModel or CatBoost instances are supported.
- **categorical_feature** (Optional[List[str]]) – List of categorical column names. If None, categorical columns are automatically determined by dtype.
- **cv** (Union[int, Iterable, BaseCrossValidator, None]) – Cross validation split. If None, the first fold out of 5 fold is used as validation.

Return type

ADVResult

Returns

Namedtuple with following members

- **auc:**
  float, ROC AUC score of adversarial validation.
- **importance:**
  pandas DataFrame, feature importance of adversarial model (order by importance)

Example

```
>>> from sklearn.model_selection import train_test_split
>>> from nyaggle.testing import make_regression_df
>>> from nyaggle.validation import adversarial_validate
```

```
>>> X, y = make_regression_df(n_samples=8)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y)
>>> auc, importance = cross_validate(X_train, X_test)
>>>
>>> print(auc)
0.51078231
>>> importance.head()
feature importance
col_1   231.5827204
col_5   207.1837266
col_7   188.6920685
col_4   174.5668498
col_9   170.6438643
```

nyaggle.validation.**cross_validate**(*estimator*, *X_train*, *y*, *X_test=None*, *cv=None*, *groups=None*, *eval_func=None*, *logger=None*, *on_each_fold=None*, *fit_params=None*, *importance_type='gain'*, *early_stopping=True*, *type_of_target='auto'*)

Evaluate metrics by cross-validation. It also records out-of-fold prediction and test prediction.

Parameters

- **estimator** (Union[BaseEstimator, List[BaseEstimator]]) – The object to be used in cross-validation. For list inputs, `estimator[i]` is trained on i-th fold.
- **X_train** (Union[DataFrame, ndarray]) – Training data
- **y** (Union[Series, ndarray]) – Target
- **X_test** (Union[DataFrame, ndarray, None]) – Test data (Optional). If specified, prediction on the test data is performed using ensemble of models.
- **cv** (Union[int, Iterable, BaseCrossValidator, None]) – int, cross-validation generator or an iterable which determines the cross-validation splitting strategy.
  - None, to use the default `KFold(5, random_state=0, shuffle=True)`,
  - integer, to specify the number of folds in a `(Stratified)KFold`,
  - CV splitter (the instance of `BaseCrossValidator`),
  - An iterable yielding (train, test) splits as arrays of indices.
- **groups** (Optional[Series]) – Group labels for the samples. Only used in conjunction with a "Group" cv instance (e.g., `GroupKFold`).
- **eval_func** (Optional[Callable]) – Function used for logging and returning scores
- **logger** (Optional[Logger]) – logger
- **on_each_fold** (Optional[Callable[[int, BaseEstimator, DataFrame, Series], None]]) – called for each fold with (idx_fold, model, X_fold, y_fold)
- **fit_params** (Union[Dict[str, Any], Callable, None]) – Parameters passed to the fit method of the estimator

- **importance_type** (str) – The type of feature importance to be used to calculate result. Used only in LGBMClassifier and LGBMRegressor.
- **early_stopping** (bool) – If True, eval_set will be added to fit_params for each fold. early_stopping_rounds = 100 will also be appended to fit_params if it does not already have one.
- **type_of_target** (str) – The type of target variable. If auto, type is inferred by sklearn.utils.multiclass.type_of_target. Otherwise, binary, continuous, or multiclass are supported.

Return type
CVResult

Returns

Namedtuple with following members

- **oof_prediction (numpy array, shape (len(X_train),)):**
  The predicted value on put-of-Fold validation data.
- **test_prediction (numpy array, hape (len(X_test),)):**
  The predicted value on test data. None if X_test is None.
- **scores (list of float, shape (nfolds+1,)):**
  scores[i] denotes validation score in i-th fold. scores[-1] is the overall score. *None* if eval is not specified.
- **importance (list of pandas DataFrame, shape (nfolds,)):**
  importance[i] denotes feature importance in i-th fold model. If the estimator is not GBDT, empty array is returned.

Example

```
>>> from sklearn.datasets import make_regression
>>> from sklearn.linear_model import Ridge
>>> from sklearn.metrics import mean_squared_error
>>> from nyaggle.validation import cross_validate
```

```
>>> X, y = make_regression(n_samples=8)
>>> model = Ridge(alpha=1.0)
>>> pred_oof, pred_test, scores, _ =         >>>     cross_validate(model,
>>>                     X_train=X[:3, :],
>>>                     y=y[:3],
>>>                     X_test=X[3:, :],
>>>                     cv=3,
>>>                     eval_func=mean_squared_error)
>>> print(pred_oof)
[-101.1123267 ,   26.79300693,   17.72635528]
>>> print(pred_test)
[-10.65095894 -12.18909059 -23.09906427 -17.68360714 -20.08218267]
>>> print(scores)
[71912.80290003832, 15236.680239881942, 15472.822033121925, 34207.43505768073]
```

PYTHON MODULE INDEX

n

INDEX

M

N

P

R

S

T