xplainable

Release 1.2.1

xplainable pty ltd

Dec 10, 2023

GETTING STARTED

CHAPTER ONE

ABOUT XPLAINABLE

1.1 Docs

See our detailed docs at xplainable docs

CHAPTER TWO

INSTALLATION

2.1 Quickstart

PyPI is the distribution channel for xplainable release versions. The best way to install it is with pip:

```
pip install xplainable
```

2.2 Optional dependencies

To use xplainable's embedded GUI in jupyter, you will need to install `xplainable` with the `gui` extra:

```
pip install xplainable[gui]
```

To use xplainable's advanced plotting functions, you will need to install `xplainable` with the `plotting` extra:

```
pip install xplainable[plotting]
```

2.3 Environment

2.3.1 Reproducible Installs

As libraries get updated, results from running your code can change, or your code can break completely. It's essential to be able to reconstruct the set of packages and versions you're using. Best practice is to:

1. use a different environment per project you're working on,
2. record package names and versions using your package installer; each has it's own metadata format for this:

- **Conda:** conda environments and environment.yml
- **Pip:** virtual environments and requirements.txt
- **Poetry:** virtual environments and pyproject.toml

CHAPTER

THREE

CLOUD CLIENT

3.1 What is Xplainable Cloud?

Xplainable Cloud is a hosted service that allows you to persist and load models and preprocessing pipelines and collaborate on them within teams and organisations. Persisted models are also able to be deployed as API endpoints in seconds. The cloud service is accessible via a web interface to manage organisations, teams, and users and provides an excellent interface for visualising model explainers and metrics. You can find more information about Xplainable Cloud at https://www.xplainable.io.

3.2 What is the Cloud Client?

The cloud client is built into the xplainable python package, allowing you to connect to Xplainable Cloud and query the API, enabling you to manage your account, models, and deployments within Python.

3.3 Initialising a session

To initialise a session, you first must generate an API key at *xplainable cloud <https://beta.xplainable.io>*.

Copyright Xplainable Pty Ltd, 2023

`xplainable.client.init.`**`initialise`**(*api_key=None*, *hostname='https://api.xplainable.io'*)

Initialise the client with an API Key.

API Keys can be generated from https://beta.xplainable.io with a valid account.

Example

```
>>> import xplainable as xp
>>> import os
>>> xp.initialise(api_key=os.environ['XP_API_KEY'])
```

Returns
The users account information.

Return type
dict

3.4 Querying the API

When you connect successfully to Xplainable Cloud, you can use the client to query the API. The client is accessible by running:

```python
import xplainable as xp
import os

# Initialise your session
xp.initialise(api_key=os.environ['XP_API_KEY'])

# Query the API
xp.client.list_models()
```

class `xplainable.client.client.`**`Client`**(*api_key*, *hostname='https://api.xplainable.io'*)

Bases: `object`

A client for interfacing with the xplainable web api (xplainable cloud).

Access models, preprocessors and user data from xplainable cloud. API keys can be generated at https://beta.xplainable.io.

Parameters
api_key (*str*) – A valid api key.

`activate_deployment`(*deployment_id*)

Activates a model deployment.

Parameters
deployment_id (*str*) – The deployment id

`add_deployment_middleware`(*deployment_id*, *func*, *name*, *description=None*)

Adds or replaces a middleware function to a deployment.

Parameters

- **deployment_id** (*str*) – The deployment id
- **func** (*function*) – The middleware function

`create_model_id`(*model*, *model_name: str*, *model_description: str*) → str

Creates a new model and returns the model id.

Parameters

- **model_name** (*str*) – The name of the model
- **model_description** (*str*) – The description of the model
- **model** (`XClassifier` | `XRegressor`) – The model to create.

Returns
The model id

Return type
int

`create_model_version`(*model*, *model_id: str*, *x: DataFrame*, *y: Series*) → str

Creates a new model version and returns the version id.

Parameters

- **model_id** (*int*) – The model id
- **partition_on** (*str*) – The partition column name
- **ruleset** (*dict | str*) – The feeature ruleset
- **health_info** (*dict*) – Feature health information
- **versions** (*dict*) – Versions of current environment

Returns
The model version id

Return type
int

`create_preprocessor_id`(*preprocessor_name: str*, *preprocessor_description: str*) → str

Creates a new preprocessor and returns the preprocessor id.

Parameters

- **preprocessor_name** (*str*) – The name of the preprocessor
- **preprocessor_description** (*str*) – The description of the preprocessor

Returns
The preprocessor id

Return type
int

`create_preprocessor_version`(*preprocessor_id: str*, *pipeline: list*, *df: DataFrame | None = None*) → str

Creates a new preprocessor version and returns the version id.

Parameters

- **preprocessor_id** (*int*) – The preprocessor id
- **pipeline** (*xplainable.preprocessing.pipeline.Pipeline*) – pipeline

Returns
The preprocessor version id

Return type
int

`deactivate_deployment`(*deployment_id*)

Deactivates a model deployment.

Parameters
deployment_id (*str*) – The deployment id

`delete_deployment_middleware`(*deployment_id*)

Deletes a middleware function from a deployment.

Parameters
deployment_id (*str*) – The deployment id

`deploy`(*model_id: str*, *version_id: str*, *hostname: str = 'https://inference.xplainable.io'*, *location: str = 'syd'*, *raw_output: bool = True*) → dict

Deploys a model partition to xplainable cloud.

The hostname should be the url of the inference server. For example: https://inference.xplainable.io

Parameters

- **hostname** (*str*) – The host name for the inference server
- **model_id** (*int*) – The model id
- **version_id** (*int*) – The version id
- **partition_id** (*int*) – The partition id
- **raw_output** (*bool, optional*) – returns a dictionary

Returns
deployment status and details.

Return type
dict

generate_deploy_key(*description: str*, *deployment_id: str*, *days_until_expiry: float = 90*, *clipboard: bool = True*, *surpress_output: bool = False*) → None

Generates a deploy key for a model deployment.

Parameters

- **description** (*str*) – Description of the deploy key use case.
- **deployment_id** (*str*) – The deployment id.
- **days_until_expiry** (*float*) – The number of days until the key expires.
- **surpress_output** (*bool*) – Surpress output. Defaults to False.

Returns
No key is returned. The key is copied to the clipboard.

Return type
None

generate_example_deployment_payload(*deployment_id*)

Generates an example deployment payload for a deployment.

Parameters
deployment_id (*str*) – The deployment id.

get_user_data() → dict

Retrieves the user data for the active user.

Returns
User data

Return type
dict

list_deployments()

Lists all deployments of the active user's team.

Returns
Dictionary of deployments.

Return type
dict

list_model_versions(*model_id: int*) → list

Lists all versions of a model.

Parameters
model_id (*int*) – The model id

Returns
Dictionary of model versions.

Return type
dict

`list_models()` → list

Lists all models of the active user's team.

Returns
Dictionary of saved models.

Return type
dict

`list_preprocessor_versions`(*preprocessor_id: int*) → list

Lists all versions of a preprocessor.

Parameters
`preprocessor_id` (*`int`*) – The preprocessor id

Returns
Dictionary of preprocessor versions.

Return type
dict

`list_preprocessors()` → list

Lists all preprocessors of the active user's team.

Returns
Dictionary of preprocessors.

Return type
dict

`load_classifier`(*model_id: int*, *version_id: int*, *model=None*)

Loads a binary classification model by model_id

Parameters

- **`model_id`** (*`str`*) – A valid model_id
- **`version_id`** (*`str`*) – A valid version_id
- **`model`** (`PartitionedClassifier`) – An existing model to add partitions

Returns
The loaded xplainable classifier

Return type
xplainable.PartitionedClassifier

`load_preprocessor`(*preprocessor_id: int*, *version_id: int*, *gui_object: bool = False*, *response_only: bool = False*)

Loads a preprocessor by preprocessor_id and version_id.

Parameters

- **`preprocessor_id`** (*`int`*) – The preprocessor id
- **`version_id`** (*`int`*) – The version id
- **`response_only`** (*`bool, optional`*) – Returns the preprocessor metadata.

Returns
The loaded pipeline

Return type
xplainable.preprocessing.pipeline.Pipeline

`load_regressor`(*model_id: int*, *version_id: int*, *model=None*)

Loads a regression model by model_id and version_id

Parameters

- `model_id` (*str*) – A valid model_id
- `version_id` (*str*) – A valid version_id
- `model` (`PartitionedRegressor`) – An existing model to add partitions to

Returns
The loaded xplainable regressor

Return type
xplainable.PartitionedRegressor

CHAPTER

FOUR

CLASSIFICATION – BINARY

4.1 Using the GUI

Training an `XClassifier` model with the embedded xplainable GUI is easy. Run the following lines of code, and you can configure and optimise your model within the GUI to minimise the amount of code you need to write.

4.1.1 Example – GUI

```
import xplainable as xp
import pandas as pd
import os

# Initialise your session
xp.initialise(api_key=os.environ['XP_API_KEY'])

# Load your data
data = pd.read_csv('data.csv')

# Train your model (this will open an embedded gui)
model = xp.classifier(data)
```

4.2 Using the Python API

You can also train an xplainable classification model programmatically. This works in a very similar way to other popular machine learning libraries.

You can import the `XClassifier` class and train a model as follows:

4.2.1 Example – XClassifier()

```python
from xplainable.core.models import XClassifier
from sklearn.model_selection import train_test_split
import pandas as pd

# Load your data
data = pd.read_csv('data.csv')
x, y = data.drop('target', axis=1), data['target']
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

# Train your model
model = XClassifier()
model.fit(x_train, y_train)

# Predict on the test set
y_pred = model.predict(x_test)
```

4.2.2 Example – PartitionedClassifier()

```python
from xplainable.core.models import PartitionedClassifier
from xpainable.core.models import XClassifier
import pandas as pd
from sklearn.model_selection import train_test_split

# Load your data
data = pd.read_csv('data.csv')
train, test = train_test_split(data, test_size=0.2)

# Instantiate the partitioned model
partitioned_model = PartitionedClassifier(partition_on='partition_column')

# Train the base model
base_model = XClassifier()
base_model.fit(
      train.drop(columns=['target', 'partition_column']),
      train['target']
      )

# Add the base model to the partitioned model (call this '__dataset__')
partitioned_model.add_partition(base_model, '__dataset__')

# Iterate over the unique values in the partition column
```

(continues on next page)

(continued from previous page)

```
for partition in train['partition_column'].unique():
      # Get the data for the partition
      part = train[train['partition_column'] == partition]
      x_train, y_train = part.drop('target', axis=1), part['target']

      # Fit the embedded model
      model = XClassifier()
      model.fit(x, y)

      # Add the model to the partitioned model
      partitioned_model.add_partition(model, partition)

# Prepare the test data
x_test, y_test = test.drop('target', axis=1), test['target']

# Predict on the partitioned model
y_pred = partitioned_model.predict(x_test)
```

4.2.3 Classifier Classes

Copyright Xplainable Pty Ltd, 2023

class xplainable.core.ml.classification.**PartitionedClassifier**(*partition_on: str | None = None, *args, **kwargs*)

Bases: BasePartition

Partitioned XClassifier model.

This class is a wrapper for the XClassifier model that allows for individual models to be trained on subsets of the data. Each model can be used in isolation or in combination with the other models.

Individual models can be accessed using the partitions attribute.

Example

```
>>> from xplainable.core.models import PartitionedClassifier
>>> import pandas as pd
>>> from sklearn.model_selection import train_test_split
```

```
>>> data = pd.read_csv('data.csv')
>>> train, test = train_test_split(data, test_size=0.2)
```

```
>>> # Train your model (this will open an embedded gui)
>>> partitioned_model = PartitionedClassifier(partition_on='partition_column')
```

```
>>> # Iterate over the unique values in the partition column
>>> for partition in train['partition_column'].unique():
>>>         # Get the data for the partition
>>>         part = train[train['partition_column'] == partition]
>>>         x_train, y_train = part.drop('target', axis=1), part['target']
```

(continues on next page)

(continued from previous page)

```
>>>         # Fit the embedded model
>>>         model = XClassifier()
>>>         model.fit(x_train, y_train)
>>>         # Add the model to the partitioned model
>>>         partitioned_model.add_partition(model, partition)
```

```
>>> # Prepare the test data
>>> x_test, y_test = test.drop('target', axis=1), test['target']
```

```
>>> # Predict on the partitioned model
>>> y_pred = partitioned_model.predict(x_test)
```

Parameters
`partition_on` (*str, optional*) – The column to partition on.

`add_partition`(*model*, *partition: str*)

Adds a partition to the model.

All partitions must be of the same type.

Parameters

- `model` (`XClassifier | XRegressor`) – The model to add.
- `partition` (*str*) – The name of the partition to add.

`drop_partition`(*partition: str*)

Removes a partition from the model.

Parameters
`partition` (*str*) – The name of the partition to drop.

`explain`(*partition: str = '__dataset__'*)

Generates a global explainer for the model.

Parameters
`partition` (*str*) – The partition to explain.

Raises
`ImportError` – If user does not have altair installed.

`predict`(*x*, *use_prob=False*, *threshold=0.5*)

Predicts the target for each row in the data across all partitions.

The partition_on columns will be used to determine which model to use for each observation. If the partition_on column is not present in the data, the '__dataset__' model will be used.

Parameters
`x` (*pd.DataFrame | np.ndarray*) – The x variables to predict.

Returns
The predicted targets

Return type
np.array

`predict_proba`(*x*)

Predicts the probability for each row in the data across all partitions.

The partition_on columns will be used to determine which model to use for each observation. If the partition_on column is not present in the data, the '__dataset__' model will be used.

Parameters
`x` (*pd.DataFrame | np.ndarray*) – The x variables to predict.

Returns
The predicted probabilities

Return type
np.array

`predict_score`(*x: DataFrame* | *ndarray*, *proba: bool = False*)

Predicts the score for each row in the data across all partitions.

The partition_on columns will be used to determine which model to use for each observation. If the partition_on column is not present in the data, the '__dataset__' model will be used.

Parameters
`x` (*pd.DataFrame | np.ndarray*) – The x variables to predict.

Returns
The predicted scores

Return type
np.array

`class xplainable.core.ml.classification.XClassifier`(*max_depth=8*, *min_info_gain=0.0001*, *min_leaf_size=0.0001*, *ignore_nan=False*, *weight=1*, *power_degree=1*, *sigmoid_exponent=0*, *tail_sensitivity: float = 1.0*, *map_calibration: bool = True*)

Bases: `BaseModel`

Xplainable Classification model for transparent machine learning.

XClassifier offers powerful predictive power and complete transparency for classification problems on tabular data. It is designed to be used in place of black box models such as Random Forests and Gradient Boosting Machines when explainabilty is important.

XClassifier is a feature-wise ensemble of decision trees. Each tree is constructed using a custom algorithm that optimises for information with respect to the target variable. The trees are then weighted and normalised against one another to produce a variable step function for each feature. The summation of these functions produces a score that can be explained in real time. The score is a float value between 0 and 1 and represents the likelihood of the positive class occuring. The score can also be mapped to a probability when probability is important.

When the fit method is called, the specified params are set across all features. Following the initial fit, the update_feature_params method may be called on a subset of features to update the params for those features only. This allows for a more granular approach to model tuning.

Example

```
>>> from xplainable.core.models import XClassifier
>>> import pandas as pd
>>> from sklearn.model_selection import train_test_split
```

```
>>> data = pd.read_csv('data.csv')
>>> x = data.drop(columns=['target'])
>>> y = data['target']
>>> x_train, x_test, y_train, y_test = train_test_split(
>>>     x, y, test_size=0.2, random_state=42)
```

```
>>> model = XClassifier()
>>> model.fit(x_train, y_train)
```

```
>>> model.predict(x_test)
```

Parameters

- **max_depth** (*int, optional*) – The maximum depth of each decision tree.
- **min_info_gain** (*float, optional*) – The minimum information gain required to make a split.
- **min_leaf_size** (*float, optional*) – The minimum number of samples required to make a split.
- **alpha** (*float, optional*) – Sets the number of possible splits with respect to unique values.
- **weight** (*float, optional*) – Activation function weight.
- **power_degree** (*float, optional*) – Activation function power degree.
- **sigmoid_exponent** (*float, optional*) – Activation function sigmoid exponent.
- **map_calibration** (*bool, optional*) – Maps the associated probability for each possible feature score.

constructs_from_json(*data*)

constructs_to_json()

convert_to_model_profile_categories(*x*)

evaluate(*x: DataFrame | ndarray*, *y: Series | array*, *use_prob: bool = False*, *threshold: float = 0.5*)

Evaluates the model performance.

Parameters

- **x** (*pd.DataFrame | np.ndarray*) – The x variables to predict.
- **y** (*pd.Series | np.array*) – The target values.
- **use_prob** (*bool, optional*) – Use probability instead of score.
- **threshold** (*float, optional*) – The threshold to use for classification.

Returns

The model performance metrics.

Return type
dict

`explain`(*label_rounding=5*)

`property feature_importances: dict`

Calculates the feature importances for the model decision process.

Returns
The feature importances.

Return type
dict

`fit`(*x: DataFrame | ndarray*, *y: Series | array*, *id_columns: list = []*, *column_names: list | None = None*, *target_name: str = 'target'*, *alpha=0.1*) → *XClassifier*

Fits the model to the data.

Parameters

- **x** (*pd.DataFrame | np.ndarray*) – The x variables used for training.
- **y** (*pd.Series | np.array*) – The target values.
- **id_columns** (*list, optional*) – id_columns to ignore from training.
- **column_names** (*list, optional*) – column_names to use for training if using a np.ndarray
- **target_name** (*str, optional*) – The name of the target column if using a np.array
- **alpha** (*float*) – Controlls the number of possible splits with respect to unique values.

Returns
The fitted model.

Return type
XClassifier

`get_construct_from_column_name`(*column_name: str*)

`local_explainer`(*x*, *subsample*)

`property params: ConstructorParams`

Returns the parameters of the model.

Returns
The default model parameters.

Return type
ConstructorParams

`predict`(*x: DataFrame | ndarray*, *use_prob: bool = False*, *threshold: float = 0.5*, *remap: bool = True*) → array

Predicts the target for each row in the data.

Parameters

- **x** (*pd.DataFrame | np.ndarray*) – The x variables to predict.
- **use_prob** (*bool, optional*) – Use probability instead of score.
- **threshold** (*float, optional*) – The threshold to use for classification.
- **remap** (*bool, optional*) – Remap the target values to their original values.

Returns
The predicted targets

Return type
np.array

`predict_explain`(*x*)

Predictions with explanations.

Parameters
`x` (*array-like*) – data to predict

Returns
prediction and explanation

Return type
pd.DataFrame

`predict_proba`(*x: DataFrame* | *ndarray*) → array

Predicts the probability for each row in the data.

Parameters
`x` (*pd.DataFrame | np.ndarray*) – The x variables to predict.

Returns
The predicted probabilities

Return type
np.array

`predict_score`(*x: DataFrame* | *ndarray*) → array

Predicts the score for each row in the data.

Parameters
`x` (*pd.DataFrame | np.ndarray*) – The x variables to predict.

Returns
The predicted scores

Return type
np.array

`property profile: dict`

Returns the model profile.

The model profile contains more granular information about the model and how it makes decisions. It is the primary property for interpreting a model and is used by the xplainable client to render the model.

Returns
The model profile.

Return type
dict

`set_params`(*default_parameters: ConstructorParams*) → None

Sets the parameters of the model. Generally used for model tuning.

Parameters
`default_parameters` (*ConstructorParams*) – default constructor parameters

Returns
None

`update_feature_params`(*features: list*, *max_depth=None*, *min_info_gain=None*, *min_leaf_size=None*, *ignore_nan=None*, *weight=None*, *power_degree=None*, *sigmoid_exponent=None*, *tail_sensitivity=None*, *x: DataFrame | ndarray | None = None*, *y: Series | array | None = None*, **args*, ***kwargs*) → *XClassifier*

Updates the parameters for a subset of features.

XClassifier allows you to update the parameters for a subset of features for a more granular approach to model tuning. This is useful when you identify under or overfitting on some features, but not all.

This also referred to as 'refitting' the model to a new set of params. Refitting parameters to an xplainable model is extremely fast as it has already pre-computed the complex metadata required for training. This can yeild huge performance gains compared to refitting traditional models, and is particularly powerful when parameter tuning. The desired result is to have a model that is well calibrated across all features without spending considerable time on parameter tuning.

Parameters

- **features** (*list*) – The features to update.
- **max_depth** (*int*) – The maximum depth of each decision tree in the subset.
- **min_info_gain** (*float*) – The minimum information gain required to make a split in the subset.
- **min_leaf_size** (*float*) – The minimum number of samples required to make a split in the subset.
- **ignore_nan** (*bool*) – Whether to ignore nan/null/empty values
- **weight** (*float*) – Activation function weight.
- **power_degree** (*float*) – Activation function power degree.
- **sigmoid_exponent** (*float*) – Activation function sigmoid exponent.
- **tail_sensitivity** (*float*) – Adds weight to divisive leaf nodes in the subset.
- **x** (*pd.DataFrame | np.ndarray, optional*) – The x variables used for training. Use if map_calibration is True.
- **y** (*pd.Series | np.array, optional*) – The target values. Use if map_calibration is True.

Returns

The refitted model.

Return type

XClassifier

4.3 Hyperparameter Optimisation

You can optimise `XClassifier` models automatically using the embedded GUI or programmatically using the Python API. The speed of hyperparameter optimisation with xplainable is much faster than traditional methods due to the concept of **rapid refits** first introduced by xplainable. You can find documentation on rapid refits in the advanced_concepts/rapid_refitting section.

The hyperparameter optimisation process uses a class called `XParamOptimiser` which is based on Bayesian optimisation using the Hyperopt library. Xplainable's wrapper has pre-configured optimisation objectives and an easy way to set the search space for each parameter. You can find more details in the XParamOptimiser docs.

4.3.1 Example

```
from xplainable.core.models import XClassifier
from xplainable.core.optimisation.bayesian import XParamOptimiser
from sklearn.model_selection import train_test_split
import pandas as pd

# Load your data
data = pd.read_csv('data.csv')
x, y = data.drop('target', axis=1), data['target']
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

# Find optimised params
optimiser = XParamOptimiser(n_trials=200, n_folds=5, early_stopping=40)
params = optimiser.optimise(x_train, y_train)

# Train your optimised model
model = XClassifier(**params)
model.fit(x_train, y_train)
```

4.3.2 Optimiser Class

class xplainable.core.optimisation.bayesian.**XParamOptimiser**(*metric='roc-auc'*, *n_trials=30*, *n_folds=5*, *early_stopping=30*, *shuffle=False*, *subsample=1*, *alpha=0.01*, *max_depth_space=[4, 10, 2]*, *min_leaf_size_space=[0.005, 0.05, 0.005]*, *min_info_gain_space=[0.005, 0.05, 0.005]*, *ignore_nan_space=[False, True]*, *weight_space=[0, 1.2, 0.05]*, *power_degree_space=[1, 3, 2]*, *sigmoid_exponent_space=[0.5, 1, 0.1]*, *verbose=True*, *random_state=1*)

Bases: `object`

Baysian optimisation for hyperparameter tuning XClassifier models.

This optimiser is built on top of the Hyperopt library. It has pre-configured optimisation objectives and an easy way to set the search space for each parameter.

The accepted metrics are:

- 'macro-f1'
- 'weighted-f1'
- 'positive-f1'
- 'negative-f1'
- 'macro-precision'
- 'weighted-precision'
- 'positive-precision'

- ‘negative-precision’
- ‘macro-recall’
- ‘weighted-recall’
- ‘positive-recall’
- ‘negative-recall’
- ‘accuracy’
- ‘brier-loss’
- ‘log-loss’
- ‘roc-auc’

Parameters

- `metric` (*str, optional*) – Optimisation metric. Defaults to ‘roc-auc’.
- `n_trials` (*int, optional*) – Number of trials to run. Defaults to 30.
- `n_folds` (*int, optional*) – Number of folds for CV split. Defaults to 5.
- `early_stopping` (*int, optional*) – Stops early if no improvement after n trials.
- `shuffle` (*bool, optional*) – Shuffle the CV splits. Defaults to False.
- `subsample` (*float, optional*) – Subsamples the training data.
- `alpha` (*float, optional*) – Sets the alpha of the model.
- `max_depth_space` (*list, optional*) – Sets the max_depth search space.
- `min_leaf_size_space` (*list, optional*) – Sets the min_leaf_size search space.
- `min_info_gain_space` (*list, optional*) – Sets the min_info_gain search space.
- `ignore_nan_space` (*list, optional*) – Sets the ignore_nan search space.
- `weight_space` (*list, optional*) – Sets the weight search space.
- `power_degree_space` (*list, optional*) – Sets the power_degree search space.
- `sigmoid_exponent_space` (*list, optional*) – Sets the sigmoid_exponent search space.
- `verbose` (*bool, optional*) – Sets output amount. Defaults to True.
- `random_state` (*int, optional*) – Random seed. Defaults to 1.

`optimise`(*x: DataFrame*, *y: Series*, *id_columns: list = []*, *verbose: bool = True*, *callback=None*)

Get an optimised set of parameters for an XClassifier model.

Parameters

- `x` (*pd.DataFrame*) – The x variables used for prediction.
- `y` (*pd.Series*) – The true values used for validation.
- `id_columns` (*list, optional*) – ID columns in dataset. Defaults to [].
- `verbose` (*bool, optional*) – Sets output amount. Defaults to True.
- `callback` (*any, optional*) – Callback for progress tracking.
- `return_model` (*bool, optional*) – Returna model, else returns params

Returns
The optimised set of parameters.

Return type
dict

CHAPTER

FIVE

CLASSIFICATION – MULTI-CLASS

5.1 Important!

Multi-Class is still being developed and is yet to be available in a release version of xplainable. Please check back soon for updates.

The following documentation is a preview of the functionality that will be available in an upcoming release of xplainable.

5.2 Using the GUI

Training a classification model with the embedded xplainable GUI is easy. Run the following lines of code, and you can configure and optimise your model within the GUI to minimise the amount of code you need to write.

5.2.1 Example – GUI

```python
import xplainable as xp
import pandas as pd
import os

# Initialise your session
xp.initialise(api_key=os.environ['XP_API_KEY'])

# Load your data
data = pd.read_csv('data.csv')

# Train your model (this will open an embedded gui)
model = xp.multiclass_classifier(data)
```

5.3 Using the Python API

You can also train a multi-class classification model programmatically. This works in a very similar way to other popular machine learning libraries.

You can import the `XMultiClassifier` class and train a model as follows:

5.3.1 Example – XMultiClassifier()

```python
from xplainable.core.models import XMultiClassifier
from sklearn.model_selection import train_test_split
import pandas as pd

# Load your data
data = pd.read_csv('data.csv')
x, y = data.drop('target', axis=1), data['target']
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

# Train your model
model = XMultiClassifier()
model.fit(x_train, y_train)

# Predict on the test set
y_pred = model.predict(x_test)
```

5.3.2 Example – PartitionedMultiClassifier()

```python
from xplainable.core.models import PartitionedMultiClassifier
from xpainable.core.models import XMultiClassifier
import pandas as pd
from sklearn.model_selection import train_test_split

# Load your data
data = pd.read_csv('data.csv')
train, test = train_test_split(data, test_size=0.2)

# Instantiate the partitioned model
partitioned_model = PartitionedMultiClassifier(partition_on='partition_column')

# Train the base model
base_model = XMultiClassifier()
base_model.fit(
      train.drop(columns=['target', 'partition_column']),
      train['target']
      )

# Add the base model to the partitioned model (call this '__dataset__')
partitioned_model.add_partition(base_model, '__dataset__')

# Iterate over the unique values in the partition column
```

(continues on next page)

(continued from previous page)

```
for partition in train['partition_column'].unique():
      # Get the data for the partition
      part = train[train['partition_column'] == partition]
      x_train, y_train = part.drop('target', axis=1), part['target']

      # Fit the embedded model
      model = XMultiClassifier()
      model.fit(x, y)

      # Add the model to the partitioned model
      partitioned_model.add_partition(model, partition)

# Prepare the test data
x_test, y_test = test.drop('target', axis=1), test['target']

# Predict on the partitioned model
y_pred = partitioned_model.predict(x_test)
```

CHAPTER

SIX

REGRESSION

6.1 Using the GUI

Training an `XRegressor` model with the embedded xplainable GUI is easy. Run the following lines of code, and you can configure and optimise your model within the GUI to minimise the amount of code you need to write.

6.1.1 Examples

GUI

```python
import xplainable as xp
import pandas as pd
import os

# Initialise your session
xp.initialise(api_key=os.environ['XP_API_KEY'])

# Load your data
data = pd.read_csv('data.csv')

# Train your model (this will open an embedded gui)
model = xp.regressor(data)
```

6.2 Using the Python API

You can also train an xplainable regression model programmatically. This works in a very similar way to other popular machine learning libraries.

You can import the `XRegressor` class and train a model as follows:

6.2.1 Examples

XRegressor

```
from xplainable.core.models import XRegressor
from sklearn.model_selection import train_test_split
import pandas as pd

# Load data
data = pd.read_csv('data.csv')
x, y = data.drop('target', axis=1), data['target']
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

# Train model
model = XRegressor()
model.fit(x_train, y_train)

# Optimise the model
model.optimise_tail_sensitivity(x_train, y_train)

# <-- Add XEvolutionaryNetwork here -->

# Predict on the test set
y_pred = model.predict(x_test)
```

PartitionedRegressor

```
from xplainable.core.models import PartitionedRegressor
from xpainable.core.models import XRegressor
import pandas as pd
from sklearn.model_selection import train_test_split

# Load your data
data = pd.read_csv('data.csv')
train, test = train_test_split(data, test_size=0.2)

# Instantiate the partitioned model
partitioned_model = PartitionedRegressor(partition_on='partition_column')

# Train the base model
base_model = XRegressor()
base_model.fit(
      train.drop(columns=['target', 'partition_column']),
      train['target']
      )

# Optimise the model
base_model.optimise_tail_sensitivity(
      train.drop('target', axis=1), train['target'])

# <-- Add XEvolutionaryNetwork here -->

# Add the base model to the partitioned model (call this '__dataset__')
partitioned_model.add_partition(base_model, '__dataset__')
```

(continues on next page)

(continued from previous page)

```
# Iterate over the unique values in the partition column
for partition in train['partition_column'].unique():
      # Get the data for the partition
      part = train[train['partition_column'] == partition]
      x_train, y_train = part.drop('target', axis=1), part['target']

      # Fit the embedded model
      model = XRegressor()
      model.fit(x_train, y_train)

      # Optimise the model
      model.optimise_tail_sensitivity(x_train, y_train)

      # <-- Add XEvolutionaryNetwork here -->

      # Add the model to the partitioned model
      partitioned_model.add_partition(model, partition)

# Prepare the test data
x_test, y_test = test.drop('target', axis=1), test['target']

# Predict on the partitioned model
y_pred = partitioned_model.predict(x_test)
```

6.2.2 Classes – Regressors

Copyright Xplainable Pty Ltd, 2023

`class xplainable.core.ml.regression.`**`PartitionedRegressor`**(*partition_on=None*, **args*, ***kwargs*)

Bases: `BasePartition`

Partitioned XRegressor model.

This class is a wrapper for the XRegressor model that allows for individual models to be trained on subsets of the data. Each model can be used in isolation or in combination with the other models.

Individual models can be accessed using the partitions attribute.

Example

```
>>> from xplainable.core.models import PartitionedRegressor
>>> import pandas as pd
>>> from sklearn.model_selection import train_test_split
```

```
>>> data = pd.read_csv('data.csv')
>>> train, test = train_test_split(data, test_size=0.2)
```

```
>>> # Train your model (this will open an embedded gui)
>>> partitioned_model = PartitionedClassifier(partition_on='partition_column')
>>> # Iterate over the unique values in the partition column
>>> for partition in train['partition_column'].unique():
>>>         # Get the data for the partition
>>>         part = train[train['partition_column'] == partition]
>>>         x_train, y_train = part.drop('target', axis=1), part['target']
>>>         # Fit the embedded model
>>>         model = XRegressor()
>>>         model.fit(x_train, y_train)
>>>         model.optimise_tail_sensitivity(x_train, y_train)
>>>         # <-- Add XEvolutionaryNetwork here -->
>>>         # Add the model to the partitioned model
>>>         partitioned_model.add_partition(model, partition)
```

```
>>> # Prepare the test data
>>> x_test, y_test = test.drop('target', axis=1), test['target']
```

```
>>> # Predict on the partitioned model
>>> y_pred = partitioned_model.predict(x_test)
```

Parameters
`partition_on` (*str, optional*) – The column to partition on.

add_partition(*model*, *partition: str*)

Adds a partition to the model.

All partitions must be of the same type.

Parameters

- **model** (XClassifier | XRegressor) – The model to add.
- **partition** (*str*) – The name of the partition to add.

drop_partition(*partition: str*)

Removes a partition from the model.

Parameters
partition (*str*) – The name of the partition to drop.

explain(*partition: str = '__dataset__'*)

Generates a global explainer for the model.

Parameters
partition (*str*) – The partition to explain.

Raises
ImportError – If user does not have altair installed.

predict(*x*) → array

Predicts the target value for each row in the data across all partitions.

The partition_on columns will be used to determine which model to use for each observation. If the partition_on column is not present in the data, the '__dataset__' model will be used.

Parameters
x (*pd.DataFrame | np.ndarray*) – The x variables to predict.

Returns
The predicted target values

Return type
np.array

`class xplainable.core.ml.regression.`**`XRegressor`**`(`*max_depth=8*, *min_info_gain=0.0001*, *min_leaf_size=0.0001*, *ignore_nan=False*, *weight=1*, *power_degree=1*, *sigmoid_exponent=0*, *tail_sensitivity: float = 1.0*, *prediction_range: tuple = (-inf, inf)*`)`

Bases: `BaseModel`

Xplainable Regression model for transparent machine learning.

XRegressor offers powerful predictive power and complete transparency for regression problems on tabular data. It is designed to be used in place of black box models such as Random Forests and Gradient Boosting Machines when explainabilty is important.

XRegressor is a feature-wise ensemble of decision trees. Each tree is constructed using a custom algorithm that optimises for information with respect to the target variable. The trees are then weighted and normalised against one another to produce a variable step function for each feature. The summation of these functions produces a score that can be explained in real time. The bounds of the prediction can be set using the prediction_range parameter.

When the fit method is called, the specified params are set across all features. Following the initial fit, the update_feature_params method may be called on a subset of features to update the params for those features only. This allows for a more granular approach to model tuning.

Important note on performance:
XRegressor alone can be a weak predictor. There are a number of ways to get the most out of the model in terms of predictive power: - use the optimise_tail_sensitivity method - fit an XEvolutionaryNetwork to the model. This will iteratively optimise the weights of the model to produce a much more accurate predictor. You can find more information on this in the XEvolutionaryNetwork documentation at xplainable/core/optimisation/genetic.py.

Example

```
>>> from xplainable.core.models import XRegressor
>>> import pandas as pd
>>> from sklearn.model_selection import train_test_split
```

```
>>> data = pd.read_csv('data.csv')
>>> x = data.drop(columns=['target'])
>>> y = data['target']
>>> x_train, x_test, y_train, y_test = train_test_split(
>>>     x, y, test_size=0.2, random_state=42)
```

```
>>> model = XRegressor()
>>> model.fit(x_train, y_train)
```

```
>>> # This will be a weak predictor
>>> model.predict(x_test)
>>> # For a strong predictor, apply optimisations
>>> model.optimise_tail_sensitivity(x_train, y_train)
>>> # Add evolutionary network here
>>> ...
```

Parameters

- **max_depth** (*int*) – The maximum depth of each decision tree.
- **min_leaf_size** (*float*) – The minimum number of samples required to make a split.
- **min_info_gain** (*float*) – The minimum information gain required to make a split.
- **tail_sensitivity** (*float*) – Adds weight to divisive leaf nodes.
- **prediction_range** (*tuple*) – The lower and upper limits for predictions.

constructs_from_json(*data*)

constructs_to_json()

convert_to_model_profile_categories(*x*)

evaluate(*x: DataFrame | ndarray, y: Series | ndarray*) → dict

Evaluates the model performance.

Parameters

- **x** (*pd.DataFrame | np.ndarray*) – The x variables to predict.
- **y** (*pd.Series | np.array*) – The target values.

Returns
The model performance metrics.

Return type
dict

explain(*label_rounding=5*)

property feature_importances: dict

Calculates the feature importances for the model decision process.

Returns
The feature importances.

Return type
dict

fit(*x: DataFrame | ndarray, y: Series | ndarray, id_columns: list = [], column_names: list | None = None, target_name: str = 'target', alpha=0.1*) → *XRegressor*

Fits the model to the data.

Parameters

- **x** (*pd.DataFrame | np.ndarray*) – The x variables used for training.
- **y** (*pd.Series | np.array*) – The target values.
- **id_columns** (*list, optional*) – id_columns to ignore from training.
- **column_names** (*list, optional*) – column_names to use for training if using a np.ndarray

- **target_name** (*str, optional*) – The name of the target column if using a np.ndarray
- **alpha** (*float, optional*) – Sets the number of possible splits with respect to unique values.

Returns
The fitted model.

Return type
XRegressor

get_construct_from_column_name(*column_name: str*)

local_explainer(*x*, *subsample*)

optimise_tail_sensitivity(*X: DataFrame* | *ndarray*, *y: Series* | *ndarray*) → *XRegressor*

Optimises the tail_sensitivity parameter at a global level.

Parameters

- **X** (*pd.DataFrame | np.ndarray*) – The x variables to fit.
- **y** (*pd.Series | np.ndarray*) – The target values.

Returns
The optimised model.

Return type
XRegressor

property params: ConstructorParams

Returns the parameters of the model.

Returns
The default model parameters.

Return type
ConstructorParams

predict(*x: DataFrame* | *ndarray*) → array

Predicts the target value for each row in the data.

Parameters
x (*pd.DataFrame | np.ndarray*) – The x variables to predict.

Returns
The predicted target values.

Return type
np.array

predict_explain(*x*)

Predictions with explanations.

Parameters
x (*array-like*) – data to predict

Returns
prediction and explanation

Return type
pd.DataFrame

`property profile: dict`

Returns the model profile.

The model profile contains more granular information about the model and how it makes decisions. It is the primary property for interpreting a model and is used by the xplainable client to render the model.

Returns
The model profile.

Return type
dict

`set_params`(*default_parameters: ConstructorParams*) → None

Sets the parameters of the model. Generally used for model tuning.

Parameters
`default_parameters` (*ConstructorParams*) – default constructor parameters

Returns
None

`update_feature_params`(*features: list*, *max_depth=None*, *min_info_gain=None*, *min_leaf_size=None*, *ignore_nan=None*, *weight=None*, *power_degree=None*, *sigmoid_exponent=None*, *tail_sensitivity=None*, **args*, ***kwargs*) → *XRegressor*

Updates the parameters for a subset of features.

XRegressor allows you to update the parameters for a subset of features for a more granular approach to model tuning. This is useful when you identify under or overfitting on some features, but not all.

This also refered to as 'refitting' the model to a new set of params. Refitting parameters to an xplainable model is extremely fast as it has already pre-computed the complex metadata required for training. This can yeild huge performance gains compared to refitting traditional models, and is particularly powerful when parameter tuning. The desired result is to have a model that is well calibrated across all features without spending considerable time on parameter tuning.

It's important to note that if a model has been further optimised using an XEvolutionaryNetwork, the optimised feature_params will be overwritten by this method and will need to be re-optimised.

Parameters

- `features` (*list*) – The features to update.
- `max_depth` (*int*) – The maximum depth of each decision tree in the subset.
- `min_info_gain` (*float*) – The minimum information gain required to make a split in the subset.
- `min_leaf_size` (*float*) – The minimum number of samples required to make a split in the subset.
- `ignore_nan` (*bool*) – Whether to ignore nan/null/empty values
- `weight` (*float*) – Activation function weight.
- `power_degree` (*float*) – Activation function power degree.
- `sigmoid_exponent` (*float*) – Activation function sigmoid exponent.
- `tail_sensitivity` (*float*) – Adds weight to divisive leaf nodes in the subset.

Returns
The refitted model.

Return type
XRegressor

6.2.3 Classes – Regression Optimisation

Regression Optimisers can optimise `XRegressor` model weights to a specific metric. They are used on top of pre-trained models and can be a powerful tool for optimising models for maximum predictive power while maintaining complete transparency.

Example:

```python
from xplainable.core.optimisation.genetic import XEvolutionaryNetwork
from xplainable.core.optimisation.layers import Tighten, Evolve
import pandas as pd
from sklearn.model_selection import train_test_split

# Load your data
data = pd.read_csv('data.csv')
x, y = data.drop('target', axis=1), data['target']
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

# Train your model
model = XRegressor()
model.fit(x_train, y_train)
model.optimise_tail_sensitivity(x_train, y_train)

# Create an optimiser
optimiser = XEvolutionaryNetwork(model)
optimiser.fit(x_train, y_train)

# Add a layers to tighten the model
optimiser.add_layer(Tighten())
optimiser.add_layer(Evolve())
optimiser.add_layer(Evolve())
optimiser.add_layer(Tighten())

# Optimise the model weights in place
optimiser.optimise()

# Predict on the test set
y_pred = model.predict(x_test)
```

class `xplainable.core.optimisation.genetic.`**`XEvolutionaryNetwork`**(*model: XRegressor, apply_range: bool = False*)

Bases: `object`

A layer-based optimisation framework for XRegressor models.

XEvolutionaryNetwork is a novel optimisation framework for XRegressor models that allows for flexibility and depth. It is inspired by deep learning frameworks, but is applied over additive models for weight optimisation.

It works by taking a pre-trained XRegressor model and fitting it, along with the training data, to an evolutionary network. The evolutionary network consists of a series of layers, each of which is responsible for optimising the model weights given a set of constraints.

What are layers?:
There are currently two types of layers: Tighten() and Evolve().

More information on each layer can be found in their respective documentation.

There is no limit to the number of layers that can be added to the network, and each layer can be customised for specific objectives. Like other machine learning methods, the network can be prone to over-fitting, so it is recommended to use a validation set to monitor performance.

An XEvolutionaryNetwork can be stopped mid-training and resumed at any time. This is useful for long-running optimisations and iterative work. You can track the remaining and completed layers using the *future_layers* and *completed_layers* attributes.

Parameters

- **`model`** (`XRegressor`) – The model to optimise.
- **`apply_range`** (*`bool`*) – Whether to apply the model's prediction range to the output.

`add_layer`(*layer*, *idx: int | None = None*)

Adds a layer to the network.

Parameters

- **`layer`** (`Tighten | Evolve`) – The layer to add.
- **`idx`** (*`int, optional`*) – The index to add the layer at.

`clear_layers`()

Removes all layers from the network.

`drop_layer`(*idx: int*)

Removes a layer from the network.

Parameters
`idx` (*`int`*) – The index of the layer to remove.

`fit`(*x: DataFrame | ndarray*, *y: Series | ndarray*, *subset: list = []*) → *XEvolutionaryNetwork*

Fits the model and data to the evolutionary network.

Parameters

- **`x`** (*`pd.DataFrame | np.ndarray`*) – The data to fit.
- **`y`** (*`pd.Series | np.ndarray`*) – The target to fit.
- **`subset`** (*`list, optional`*) – A list of columns to subset for feature level optimisation.

Returns
The fitted network.

Return type
XEvolutionaryNetwork

`optimise`(*callback=None*) → *XEvolutionaryNetwork*

Sequentially runs the layers in the network.

Parameters
`callback` (*`any, optional`*) – Callback for progress tracking.

Returns
The evolutionary network.

Return type
XEvolutionaryNetwork

Copyright Xplainable Pty Ltd, 2023

`class xplainable.core.optimisation.layers.`**`BaseLayer`**(*metric='mae'*)

Bases: `object`

Base class for optimisation layers.

Parameters
`metric` (*`str, optional`*) – Metric to optimise on. Defaults to 'mae'.

`class xplainable.core.optimisation.layers.`**`Evolve`**(*mutations: int = 100*, *generations: int = 50*, *max_generation_depth: int = 10*, *max_severity: float = 0.5*, *max_leaves: int = 20*, *early_stopping: int | None = None*)

Bases: *`BaseLayer`*

Evolutionary algorithm to optimise XRegressor leaf weights.

The Evolve layer uses a genetic algorithm to optimise the leaf weights of an XRegressor model. The algorithm works by mutating the leaf weights of the model and scoring the resulting predictions. The best mutations are then selected to reproduce and mutate again. This process is repeated until the maximum number of generations is reached, or the early stopping threshold is reached.

Parameters

- **`mutations`** (*`int, optional`*) – The number of mutations to generate per generation.
- **`generations`** (*`int, optional`*) – The number of generations to run.
- **`max_generation_depth`** (*`int, optional`*) – The maximum depth of a generation.
- **`max_severity`** (*`float, optional`*) – The maximum severity of a mutation.
- **`max_leaves`** (*`int, optional`*) – The maximum number of leaves to mutate.
- **`early_stopping`** (*`int, optional`*) – Stop early if no improvement after n iters.

`property params: dict`

Returns the parameters of the layer.

Returns
The layer parameters.

Return type
dict

`transform`(*xnetwork: XEvolutionaryNetwork*, *x: ndarray*, *y: array*, *callback=None*)

Optimises an XRegressor profile given the set of parameters.

Parameters

- **`xnetwork`** (`XEvolutionaryNetwork`) – The evolutionary network.
- **`x`** (*`np.ndarray`*) – The input variables used for prediction.
- **`y`** (*`np.array`*) – The target values.
- **`callbacks`** (*`list`*) – Callback function for progress tracking.

Returns
The original x data to pass to the next layer. np.ndarray: The final optimised chromosome to pass to the next layer.

Return type
np.ndarray

`class xplainable.core.optimisation.layers.`**`Tighten`**(*iterations: int = 100*, *learning_rate: float = 0.03*, *early_stopping: int | None = None*)

Bases: *`BaseLayer`*

A leaf boosting algorithm to optimise XRegressor leaf node weights.

The Tighten layer uses a novel leaf boosting algorithm to optimise the leaf weights of an XRegressor model. The algorithm works by iteratively identifying the leaf node that will have the greatest impact on the overall model score, and then incrementally increasing or decreasing the leaf node weight to improve the model score. This process is repeated until the maximum number of iterations is reached, or the early stopping threshold is reached.

Args:

iterations (int): The number of iterations to run. learning_rate (float): How fast the model learns. Between 0.001 - 1 early_stopping (int): Stop early if no improvement after n iters.

`property params: dict`

Returns the parameters of the layer.

Returns

The layer parameters.

Return type

dict

`transform`(*xnetwork:* XEvolutionaryNetwork, *x: ndarray*, *y: array*, *callback=None*) → tuple

Optimises an XRegressor profile given the set of parameters.

Parameters

- **`x`** (*`np.ndarray`*) – The input variables used for prediction.
- **`y`** (*`np.array`*) – The target values.
- **`callback`** (*`any`*) – Callback function for progress tracking.

Returns

The optimised feature score map.

Return type

dict

CHAPTER

SEVEN

PREPROCESSING

Xplainable offers a preprocessing module that allows you to build reproducible preprocessing pipelines. The module aims to rapidly develop and deploy pipelines in production environments and play friendly with ipywidgets.

The preprocessing module is built on the `XPipeline` class from xplainable and is used similarly to the scikit-learn Pipeline class. All transformers in the pipeline are expected to have a fit and transform method, along with an inverse_transform method.

To create custom transformers, you can inherit from the `XBaseTransformer` class. You can render these custom transformers in the embedded xplainable GUI, which allows you to build pipelines without writing any code. You can find documentation on how to embed them in the GUI in the advanced_concepts/custom_transformers section.

7.1 Using the GUI

Xplainable offers a GUI for making preprocessing pipelines easy and reproducible. You can start the GUI by running a few simple lines.

7.1.1 Example

```python
import xplainable as xp
import pandas as pd
from sklearn.model_selection import train_test_split

# Load data
data = pd.read_csv('data.csv')
train, test = train_test_split(data, test_size=0.2, random_state=42)

# Instantiate the preprocessor object
pp = xp.Preprocessor()

# Open the GUI and build pipeline
pp.preprocess(train)

# Apply the pipeline on new data
test_transformed = pp.transform(test)
```

7.2 Using the Python API

You can develop preprocessing pipelines using the Python API with `XPipeline`. The following example shows how to build a pipeline.

7.2.1 Example

```python
from xplainable.preprocessing import transformers as xtf
from xplainable.preprocessing.pipeline import XPipeline
from sklearn.model_selection import train_test_split
import pandas as pd

# Load data
data = pd.read_csv('data.csv')
train, test = train_test_split(data, test_size=0.2, random_state=42)

# Instantiate a pipeline
pipeline = XPipeline()

# Add stages for specific features
pipeline.add_stages([
    {"feature": "age", "transformer": xtf.Clip(lower=18, upper=99)},
    {"feature": "balance", "transformer": xtf.LogTransform()}
])

# add stages on multiple features
pipeline.add_stages([
    {"transformer": xtf.FillMissing({'job': 'mode', 'age': 'mean'})},
    {"transformer": xtf.DropCols(columns=['duration', 'campaign'])}
])

# Share a single transformer across multiple features.
# Note this can only be applied when no fit method is required.
upper_case = xtf.ChangeCase(case='upper')

pipeline.add_stages([
    {"feature": "job", "transformer": upper_case},
    {"feature": "month", "transformer": upper_case}
])

# Fit and transform the data
train_transformed = pipeline.fit_transform(train)

# Apply transformations on new data
test_transformed = pipeline.transform(test)

# Inverse transform (only applies to configured features)
test_inv_transformed = pipeline.inverse_transform(test_transform)
```

7.2.2 XPipeline

Copyright Xplainable Pty Ltd, 2023

class `xplainable.preprocessing.pipeline.`**`XPipeline`**

Bases: `object`

Pipeline builder for xplainable transformers.

Parameters
`stages` (*`list`*) – list containing xplainable pipeline stages.

`add_stages`(*stages: list*) → *XPipeline*

Adds multiple stages to the pipeline.

Parameters
`stages` (*`list`*) – list containing xplainable pipeline stages.

Returns
self

Return type
XPipeline

`drop_stage`(*stage: int*) → *XPipeline*

Drops a stage from the pipeline.

Parameters
`stage` (*`int`*) – index of the stage to drop.

Returns
self

Return type
XPipeline

`fit`(*x: DataFrame*) → *XPipeline*

Sequentially iterates through pipeline stages and fits data.

Parameters
`x` (*`pd.DataFrame`*) – A non-empty DataFrame to fit.

Returns
The fitted pipeline.

Return type
XPipeline

`fit_transform`(*x: DataFrame, start: int = 0*)

Runs the fit method followed by the transform method.

Parameters

- **`x`** (*`pd.DataFrame`*) – A non-empty DataFrame to fit.
- **`start`** (*`int`*) – index of the stage to start fitting from.

Returns
The transformed dataframe.

Return type
pd.DataFrame

`get_blueprint`()

Returns a blueprint of the pipeline.

Returns
A list containing the pipeline blueprint.

Return type
list

`inverse_transform`(*x: DataFrame*)

Iterates through pipeline stages applying inverse transformations.

Parameters
x (*pd.DataFrame*) – A non-empty DataFrame to inverse transform.

Returns
The inverse transformed dataframe.

Return type
pd.DataFrame

`transform`(*x: DataFrame*)

Iterates through pipeline stages applying transformations.

Parameters
x (*pd.DataFrame*) – A non-empty DataFrame to transform.

Returns
The transformed dataframe.

Return type
pd.DataFrame

`transform_generator`(*x*)

transform generator

7.2.3 Base Transformer

Copyright Xplainable Pty Ltd, 2023

`class xplainable.preprocessing.transformers.base.XBaseTransformer`

Bases: `object`

Base class for all transformers.

This base class is used as a template for all xplainable transformers. It contains the basic methods that all transformers should have, and is used to enforce a consistent API across all transformers.

the __call__ method is used to allow the transformers to be called inside the xplainable gui in jupyter, but does not need to be called.

`fit`(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform`(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform`(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors`()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`transform`(*x: Series | DataFrame*)

Placeholder for transformation operation. Intended to be overridden.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

7.2.4 Categorical Transformers

Copyright Xplainable Pty Ltd, 2023

class `xplainable.preprocessing.transformers.categorical.ChangeCase`(*case='lower'*)

Bases: *XBaseTransformer*

Changes the case of a string.

Parameters
case (*str*) – 'upper' or 'lower'

`fit`(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform`(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform`(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors`()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['categorical']`

`transform`(*ser: Series*) → Series

Changes the case of a string.

Parameters
ser (*pd.Series*) – The series to transform.

Returns
The transformed series.

Return type
pd.Series

class xplainable.preprocessing.transformers.categorical.`Condense`(*pct=0.8*, *categories=[]*)

Bases: *XBaseTransformer*

Condenses a feature into categories that make up x pct of obserations.

Parameters
`pct` (*int*) – The minumum pct of observations the categories should cover.

`fit`(*ser: Series*) → *Condense*

Determines the categories that make up x pct of obserations.

Parameters
`ser` (*pandas.Series*) – The series in which to analyse.

Raises
`TypeError` – If the series is not of type string.

Returns
The fitted transformer.

Return type
Condense

`fit_transform`(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
`x` (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform`(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
`x` (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors`()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['categorical']`

`transform`(*ser: Series*) → Series

Condenses a feature into categories that make up x pct of obserations.

Parameters
`ser` (*pd.Series*) – The series to transform.

Raises
`ValueError` – If the series is not of type string.

Returns
The transformed series.

Return type
pd.Series

`class xplainable.preprocessing.transformers.categorical.`**`DetectCategories`**`(`*max_categories=10*, *category_list=[]*`)`

Bases: *`XBaseTransformer`*

Auto-detects categories from a string column.

Parameters
`max_categories` (*`int`*) – The maximum number of categories to extract.

`fit``(`*ser: Series*`)` → *DetectCategories*

Identifies the top categories from a text series.

Parameters
`ser` (*`pandas.Series`*) – The series in which to analyse.

Returns
The fitted transformer.

Return type
DetectCategories

`fit_transform``(`*x: Series | DataFrame*`)`

Fit and transforms data on a series or dataframe.

Parameters
`x` (*`pd.Series | pd.DataFrame`*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform``(`*x: Series | DataFrame*`)`

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
`x` (*`pd.Series | pd.DataFrame`*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors``()`

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['categorical']`

transform(*ser: Series*) → Series

Detects categories from a string column.

Parameters
ser (*pd.Series*) – The series to transform.

Raises
TypeError – If the series is not of type string.

Returns
The transformed series.

Return type
pd.Series

class xplainable.preprocessing.transformers.categorical.**FillMissingCategorical**(*fill_with='missing'*)

Bases: *XBaseTransformer*

Fills missing values with a specified value.

Parameters
fill_with (*str*) – Text to fill with.

fit(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

fit_transform(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

inverse_transform(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors``()`

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['categorical']`

`transform``(`*ser: Series*`)` → Series

Fills missing values with a specified value.

Parameters
`ser` (*`pd.Series`*) – The series to transform.

Returns
The transformed series.

Return type
pd.Series

`class xplainable.preprocessing.transformers.categorical.`**`MapCategories`**`(`*category_values={}*`)`

Bases: *`XBaseTransformer`*

Maps all categories of a string column to new values

`fit``(`**args*, ***kwargs*`)`

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform``(`*x: Series* | *DataFrame*`)`

Fit and transforms data on a series or dataframe.

Parameters
`x` (*`pd.Series | pd.DataFrame`*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform``(`*x: Series* | *DataFrame*`)`

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
`x` (*`pd.Series | pd.DataFrame`*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

raise_errors()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

supported_types = ['categorical']

transform(*ser: Series*) → Series

Maps all categories of a string column to new values

Parameters
ser (*pd.Series*) – The series to transform.

Returns
The transformed series.

Return type
pd.Series

class xplainable.preprocessing.transformers.categorical.**MergeCategories**(*merge_from=[]*, *merge_to=''*)

Bases: *XBaseTransformer*

Merges specified categories in a series into one category.

Parameters

- **merge_from** (*list*) – List of categories to merge from.
- **merge_to** (*str*) – The category to merge to.

fit(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

fit_transform(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

inverse_transform(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
`x` (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors()`

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['categorical']`

`transform`(*ser: Series*) → Series

Merges specified categories in a series into one category.

Parameters
`ser` (*pd.Series*) – The series to transform.

Returns
The transformed series.

Return type
pd.Series

`class xplainable.preprocessing.transformers.categorical.ReplaceCategory`(*target=None, replace_with=''*)

Bases: *XBaseTransformer*

Replaces a category in a series with specified value.

Parameters

- `target` – The target value to replace.
- `replace_with` – The value to insert in place.

`fit`(**args, **kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform`(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
`x` (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform`(*x: Series* | *DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors`()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['categorical']`

`transform`(*ser: Series*) → Series

Replaces a category in a series with specified value.

Parameters
ser (*pd.Series*) – The series to transform.

Returns
The transformed series.

Return type
pd.Series

class xplainable.preprocessing.transformers.categorical.**ReplaceWith**(*target=None*, *replace_with=None*)

Bases: *XBaseTransformer*

Replaces specified value in series

Parameters
case (*str*) – 'upper' or 'lower'

`case`

The case the string will convert to.

Type
str

`fit`(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

fit_transform(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

inverse_transform(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

raise_errors()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

supported_types = ['categorical']

transform(*ser: Series*) → Series

Replaces specified value in series

Parameters
ser (*pd.Series*) – The series to transform.

Returns
The transformed series.

Return type
pd.Series

class xplainable.preprocessing.transformers.categorical.**TextContains**(*selector=None*, *value=None*)

Bases: *XBaseTransformer*

Flags series values that contain, start with, or end with a value.

Parameters

- **selector** (*str*) – The type of search to make.
- **value** (*str*) – The value to search.

`fit`(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform`(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform`(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors`()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['categorical']`

`transform`(*ser: Series*) → Series

Flags series values that contain, start with, or end with a value.

Parameters
ser (*pd.Series*) – The series to transform.

Returns
The transformed series.

Return type
pd.Series

class `xplainable.preprocessing.transformers.categorical.TextRemove`(*numbers=False*, *characters=False*, *uppercase=False*, *lowercase=False*, *special=False*, *whitespace=False*, *stopwords=False*, *text=None*, *custom_regex=None*)

Bases: *XBaseTransformer*

Remove specified values from a str type series.

This transformer cannot be inverse_transformed and does not require fitting.

Parameters

- **numbers** (*bool, optional*) – Removes numbers from string.
- **characters** (*bool, optional*) – Removes characters from string.
- **uppercase** (*bool, optional*) – Removes uppercase characters from string.
- **lowercase** (*bool, optional*) – Removes lowercase characters from string.
- **special** (*bool, optional*) – Removes special characters from string.
- **whitespace** (*bool, optional*) – Removes whitespace from string.
- **stopwords** (*bool, optional*) – Removes stopwords from string.
- **text** (*str, optional*) – Removes specific text match from string.
- **custom_regex** (*str, optional*) – Removes matching regex text from string.

`fit`(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform`(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform`(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
`x` (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors()`
Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['categorical']`

`transform`(*ser: Series*) → Series
Removes specified values from a str type series.

Parameters
`ser` (*pd.Series*) – The series to transform.

Returns
The transformed series.

Return type
pd.Series

`class xplainable.preprocessing.transformers.categorical.`**`TextSlice`**(*start=None*, *end=None*, *action='keep'*)

Bases: *XBaseTransformer*

Selects slice from categorical column string.

Parameters

- **`start`** (*int*) – Starting character.
- **`end`** (*int*) – Ending character.
- **`action`** (*str*) – [keep, drop] selected slice.

`fit`(**args*, ***kwargs*)
No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform`(*x: Series | DataFrame*)
Fit and transforms data on a series or dataframe.

Parameters
`x` (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

inverse_transform(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters

x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:

raise_errors (decorator): Raises detailed errors.

raise_errors()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

supported_types = ['categorical']

transform(*ser: Series*) → Series

Selects slice from categorical column string.

Parameters

ser (*pd.Series*) – The series to transform.

Returns

The transformed series.

Return type

pd.Series

class xplainable.preprocessing.transformers.categorical.**TextTrim**(*selector=None*, *n=0*, *action='keep'*)

Bases: *XBaseTransformer*

Drops or keeps first/last n characters of a categorical column.

Parameters

- **selector** (*str*) – [first, last].
- **n** (*int*) – Number of characters to identify.
- **action** (*str*) – [keep, drop] the identified characters.

fit(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:

raise_errors (decorator): Raises detailed errors.

fit_transform(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform`(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors()`

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['categorical']`

`transform`(*ser: Series*) → Series

Drops or keeps first/last n characters of a categorical column.

Parameters
ser (*pd.Series*) – The series to transform.

Returns
The transformed series.

Return type
pd.Series

7.2.5 Numeric Transformers

Copyright Xplainable Pty Ltd, 2023

class `xplainable.preprocessing.transformers.numeric.Clip`(*lower=None*, *upper=None*)

Bases: *XBaseTransformer*

Clips numeric values to a specified range.

Parameters

- **lower** (*float*) – The lower threshold value.
- **upper** (*float*) – The upper threshold value.

`fit`(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform`(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform`(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors`()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['numeric']`

`transform`(*ser: Series*) → Series

Parameters
ser (*pd.Series*) – The series to transform.

Returns
The transformed series.

Return type
pd.Series

class `xplainable.preprocessing.transformers.numeric.FillMissingNumeric`(*fill_with='mean'*, *fill_value=None*)

Bases: *XBaseTransformer*

Fills missing values with a specified strategy.

Parameters
: **`fill_with`** (*`str`*) – The strategy ['mean', 'median', 'mode'].

`fit`(*ser: Series*) → *FillMissingNumeric*
: Calculates the fill value from a series.

  Parameters
  : **`ser`** (*`pandas.Series`*) – The series to analyse.

  Returns
  : The fitted transformer.

  Return type
  : *FillMissingNumeric*

`fit_transform`(*x: Series | DataFrame*)
: Fit and transforms data on a series or dataframe.

  Parameters
  : **`x`** (*`pd.Series | pd.DataFrame`*) – Series or df to fit & transform.

  Returns
  : The transformed series or df.

  Return type
  : pandas.Series

`inverse_transform`(*x: Series | DataFrame*)
: No inverse transform is available for this transformer.

  This is a default inverse method in case no inverse transform is available.

  The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

  Parameters
  : **`x`** (*`pd.Series | pd.DataFrame`*) – To be specified by transformer.

  Decorators:
  : raise_errors (decorator): Raises detailed errors.

`raise_errors`()
: Decorator to raise detailed errors in transformer functions.

  This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['numeric']`

`transform`(*ser: Series*) → Series

  Parameters
  : **`ser`** (*`pd.Series`*) – The series to transform.

  Returns
  : The transformed series.

  Return type
  : pd.Series

`class xplainable.preprocessing.transformers.numeric.`**`LogTransform`**

Bases: *`XBaseTransformer`*

Log transforms a given numeric series.

`fit`(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform`(*x: Series* | *DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform`(*ser: Series*) → Series

Parameters
ser (*pd.Series*) – The series to inverse transform.

Returns
The inverse transformed series.

Return type
pd.Series

`raise_errors`()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['numeric']`

`transform`(*ser: Series*) → Series

Parameters
ser (*pd.Series*) – The series to transform.

Returns
The transformed series.

Return type
pd.Series

`class xplainable.preprocessing.transformers.numeric.`**`MinMaxScale`**(*min_value=None*, *max_value=None*)

Bases: *`XBaseTransformer`*

Scales a numeric series between 0 and 1.

fit(*ser: Series*) → *MinMaxScale*

Extracts the min and max value from a series.

Parameters
`ser` (*pandas.Series*) – The series in which to analyse.

Returns
The fitted transformer.

Return type
MinMaxScale

fit_transform(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
`x` (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

inverse_transform(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
`x` (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

raise_errors()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

supported_types = ['numeric']

transform(*ser: Series*) → Series

Scale a numeric series between 0 and 1.

Parameters
`ser` (*pd.Series*) – The series to transform.

Returns
The transformed series.

Return type
pd.Series

7.2.6 Mixed Transformers

Copyright Xplainable Pty Ltd, 2023

class `xplainable.preprocessing.transformers.mixed.`**`SetDType`**(*to_type=None*)

Bases: *`XBaseTransformer`*

Changes the data type of a specified column.

`fit`(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform`(*x: Series* | *DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform`(*x: Series* | *DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors`()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['numeric', 'categorical']`

`transform`(*ser: Series*) → Series

Changes the data type of a specified column.

Parameters
ser (*pd.Series*) – The series to transform.

Returns
The transformed series.

Return type
pd.Series

`class xplainable.preprocessing.transformers.mixed.Shift`(*step=0*)

Bases: `XBaseTransformer`

Shifts a series up or down n steps.

Parameters
step (*str*) – The number of steps to shift.

`fit`(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform`(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform`(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors()`

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['categorical', 'numeric']`

`transform`(*ser: Series*) → Series

Shifts a series up or down n steps.

Parameters
ser (*pd.Series*) – The series to transform.

Returns
The transformed series.

Return type
pd.Series

7.2.7 Dataset Transformers

Copyright Xplainable Pty Ltd, 2023

`class xplainable.preprocessing.transformers.dataset.`**`ChangeCases`**`(`*columns=[]*, *case='lower'*`)`

Bases: *XBaseTransformer*

Changes the case of all specified categorical columns.

Parameters

- **columns** (*list*) – To apply the case change to.
- **case** (*str*) – 'upper' or 'lower'.

`fit``(`**args*, ***kwargs*`)`

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform``(`*x: Series | DataFrame*`)`

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform``(`*x: Series | DataFrame*`)`

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors()`

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['dataset']`

`transform`(*df: DataFrame*) → DataFrame

Changes the case of all specified categorical columns.

Parameters
`df` (*pd.DataFrame*) – The dataset to transform.

Returns
The transformed dataset.

Return type
pd.DataFrame

`class xplainable.preprocessing.transformers.dataset.ChangeNames`(*col_names={}*)

Bases: *XBaseTransformer*

Changes names of columns in a dataset

Parameters
`col_names` (*dict*) – Dictionary of old and new column names.

`fit`(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform`(*x: Series* | *DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
`x` (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform`(*x: Series* | *DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
`x` (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors()`

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['dataset']`

`transform`(*df: DataFrame*) → DataFrame

Changes names of columns in a dataset

Parameters
`df` (*pd.DataFrame*) – The dataset to transform.

Returns
The transformed dataset.

Return type
pd.DataFrame

class xplainable.preprocessing.transformers.dataset.**DateTimeExtract**(*target=None*, *year=False*, *month=False*, *day=False*, *weekday=False*, *day_name=False*, *hour=False*, *minute=False*, *second=False*, *drop=False*)

Bases: *XBaseTransformer*

Extracts Datetime values from datetime object.

Parameters

- **target** (*str*) – The datetime column to extract from.
- **year** (*bool*) – Extracts year if True.
- **month** (*bool*) – Extracts month if True.
- **day** (*bool*) – Extracts day if True.
- **weekday** (*bool*) – Extracts weekday if True.
- **day_name** (*bool*) – Extracts day name if True.
- **hour** (*bool*) – Extracts hour if True.
- **minute** (*bool*) – Extracts minute if True.
- **second** (*bool*) – Extracts second if True.
- **drop** (*bool*) – Drops original datetime column if True.

`fit`(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

fit_transform(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

inverse_transform(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

raise_errors()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

supported_types = ['dataset']

transform(*df: DataFrame*) → DataFrame

Extracts Datetime values from datetime object.

Parameters
df (*pd.DataFrame*) – The dataset to transform.

Returns
The transformed dataset.

Return type
pd.DataFrame

class xplainable.preprocessing.transformers.dataset.**DropCols**(*columns=None*)

Bases: *XBaseTransformer*

Drops specified columns from a dataset.

Parameters
columns (*str*) – The columns to be dropped.

fit(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

fit_transform(*x: Series* | *DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

inverse_transform(*x: Series* | *DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

raise_errors()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

supported_types = ['dataset']

transform(*df: DataFrame*) → DataFrame

Drops specified columns from a dataset.

Parameters
df (*pd.DataFrame*) – The dataset to transform.

Returns
The transformed dataset.

Return type
pd.DataFrame

class xplainable.preprocessing.transformers.dataset.**DropNaNs**(*subset=None*)

Bases: *XBaseTransformer*

Drops nan rows from a dataset.

Parameters
subset (*list, optional*) – A subset of columns to apply the transfomer.

`fit`(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform`(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform`(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors`()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['dataset']`

`transform`(*df: DataFrame*) → DataFrame

Drops nan rows from a dataset.

Parameters
df (*pd.DataFrame*) – The dataset to transform.

Returns
The transformed dataset.

Return type
pd.DataFrame

class xplainable.preprocessing.transformers.dataset.**FillMissing**(*fill_with={}*, *fill_values={}*)

Bases: *XBaseTransformer*

Fills missing values of all columns with a specified value/strategy.

`fit`(*df: DataFrame*) → *FillMissing*

Calculates the fill_value for all columns in the dataset.

The fill values are based on a specified strategy for each column.

Parameters
`df` (*pd.DataFrame*) – The dataset to fit

Returns
The fitted transformer.

Return type
FillMissing

`fit_transform`(*x: Series* | *DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
`x` (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform`(*x: Series* | *DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
`x` (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors`()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['dataset']`

`transform`(*df: DataFrame*) → DataFrame

Fills missing values of all columns with a specified value/strategy.

Parameters
`df` (*pd.DataFrame*) – The dataset to transform.

Returns
The transformed dataset.

Return type
pd.DataFrame

class xplainable.preprocessing.transformers.dataset.**GroupbyShift**(*columns=None*, *step=0*, *as_new=True*, *col_names=[]*, *group_by=None*, *order_by=None*, *descending=None*)

Bases: *XBaseTransformer*

Shifts a series up or down n steps within specified group.

Parameters

- **target** (*str*) – The target feature to shift.
- **step** (*int*) – The number of steps to shift.
- **as_new** (*bool*) – Creates new column if True.
- **group_by** (*str*) – The column to group by.
- **order_by** (*str*) – The column to order by.
- **descending** (*bool*) – Orders the value descending if True.

fit(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

fit_transform(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

inverse_transform(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors()`

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['dataset']`

`transform`(*df: DataFrame*) → DataFrame

Shifts a series up or down n steps within specified group.

Parameters
`df` (*pd.DataFrame*) – The dataset to transform.

Returns
The transformed dataset.

Return type
pd.DataFrame

`class xplainable.preprocessing.transformers.dataset.GroupedSignalSmoothing`(*target=None*, *group_by=None*, *order_by=None*, *descending=None*)

Bases: *XBaseTransformer*

Smooths signal data within specified group.

Parameters

- **target** (*str*) – The target feature to shift.
- **as_new** (*bool*) – Creates new column if True.
- **group_by** (*str*) – The column to group by.
- **order_by** (*str*) – The column to order by.
- **descending** (*bool*) – Orders the value descending if True.

`fit`(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform`(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
`x` (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform`(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
`x` (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors`()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['dataset']`

`transform`(*df: DataFrame*) → DataFrame

Smooths signal data within specified group.

Parameters
`df` (*pd.DataFrame*) – The dataset to transform.

Returns
The transformed dataset.

Return type
pd.DataFrame

class `xplainable.preprocessing.transformers.dataset.Operation`(*columns=[]*, *operation=None*, *alias: str | None = None*, *drop: bool = False*)

Bases: *XBaseTransformer*

Applies operation to multiple columns (in order) into new feature.

Parameters

- `columns` (*list*) – Column names to add.
- `alias` (*str*) – Name of newly created column.
- `drop` (*bool*) – Drops original columns if True

`fit`(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

fit_transform(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters

x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns

The transformed series or df.

Return type

pandas.Series

inverse_transform(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters

x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:

raise_errors (decorator): Raises detailed errors.

raise_errors()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

supported_types = ['dataset']

transform(*df: DataFrame*) → DataFrame

Applies operation to multiple columns (in order) into new feature.

Parameters

df (*pd.DataFrame*) – The dataset to transform.

Returns

The transformed dataset.

Return type

pd.DataFrame

class xplainable.preprocessing.transformers.dataset.**OrderBy**(*order_by=None*, *ascending=True*)

Bases: *XBaseTransformer*

Orders the dataset by the values of a given series.

Parameters

- **order_by** (*str*) – The series to order by.
- **ascending** (*bool*) – Orders in ascending order if True.

fit(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform`(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform`(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors`()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['dataset']`

`transform`(*df: DataFrame*) → DataFrame

Orders the dataset by the values of a given series.

Parameters
df (*pd.DataFrame*) – The dataset to transform.

Returns
The transformed dataset.

Return type
pd.DataFrame

class xplainable.preprocessing.transformers.dataset.**RollingOperation**(*groupby=None, orderby=None, direction=None, columns=[], window=None, operation=None, drop: bool = False*)

Bases: *XBaseTransformer*

Applies operation to multiple columns (in order) into new feature.

Parameters

- **columns** (*list*) – Column names to add.
- **alias** (*str*) – Name of newly created column.
- **drop** (*bool*) – Drops original columns if True

fit(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

fit_transform(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

inverse_transform(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

raise_errors()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

supported_types = ['dataset']

transform(*df: DataFrame*) → DataFrame

Applies operation to multiple columns (in order) into new feature.

Parameters
df (*pd.DataFrame*) – The dataset to transform.

Returns
The transformed dataset.

Return type
pd.DataFrame

`class xplainable.preprocessing.transformers.dataset.`**`SetDTypes`**`(`*types={}*`)`

Bases: *XBaseTransformer*

Sets the data type of all columns in the dataset.

Parameters
types (*dict*) – Dictionary of column names and data types.

`fit``(`**args*, ***kwargs*`)`

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform``(`*x: Series | DataFrame*`)`

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform``(`*x: Series | DataFrame*`)`

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors``()`

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['dataset']`

`transform``(`*df: DataFrame*`)` → DataFrame

Sets the data type of all columns in the dataset.

Parameters
df (*pd.DataFrame*) – The dataset to transform.

Returns
The transformed dataset.

Return type
pd.DataFrame

class `xplainable.preprocessing.transformers.dataset.`**`TextSplit`**(*target=None*, *separator=None*, *max_splits=0*)

Bases: *XBaseTransformer*

Splits a string column into multiple columns on a specified separator.

Parameters

- **target** (*str*) – The columns to split.
- **separator** (*str*) – The separator to split on.
- **max_splits** (*int*) – The maximum number of splits to make.

fit(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

fit_transform(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
x (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

inverse_transform(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
x (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

raise_errors()

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

supported_types = ['dataset']

`transform`(*df: DataFrame*) → DataFrame

Splits a string column into multiple columns on a specified separator.

Parameters
`df` (*pd.DataFrame*) – The dataset to transform.

Returns
The transformed dataset.

Return type
pd.DataFrame

`class xplainable.preprocessing.transformers.dataset.TextTrimMulti`(*column=''*, *selector=None*, *n=0*, *action='keep'*, *drop_col=False*, *alias=''*)

Bases: *XBaseTransformer*

Drops or keeps first/last n characters of a categorical column.

Parameters

- **selector** (*str*) – [first, last].
- **n** (*int*) – Number of characters to identify.
- **action** (*str*) – [keep, drop] the identified characters.

`fit`(**args*, ***kwargs*)

No fit is required for this transformer.

This is a default fit method in case no fit is needed. This method is used to allow the transformer to be used in a pipeline, and is intended to be overridden by transformers that require fitting.

Decorators:
raise_errors (decorator): Raises detailed errors.

`fit_transform`(*x: Series | DataFrame*)

Fit and transforms data on a series or dataframe.

Parameters
`x` (*pd.Series | pd.DataFrame*) – Series or df to fit & transform.

Returns
The transformed series or df.

Return type
pandas.Series

`inverse_transform`(*x: Series | DataFrame*)

No inverse transform is available for this transformer.

This is a default inverse method in case no inverse transform is available.

The input parameter is either a pd.Series or a pd.DataFrame, depending on the transformer. Documentation for each individual transformer should specify which type of input is expected in this method when it is being overridden.

Parameters
`x` (*pd.Series | pd.DataFrame*) – To be specified by transformer.

Decorators:
raise_errors (decorator): Raises detailed errors.

`raise_errors()`

Decorator to raise detailed errors in transformer functions.

This decorator is used to wrap the transformer methods and raise any errors that occur during processing. This is done to allow the gui to catch the errors and display them.

`supported_types = ['dataset']`

`transform`(*df: DataFrame*) → DataFrame

Drops or keeps first/last n characters of a categorical column.

Parameters
`df` (*pd.DataFrame*) – The dataset to transform.

Returns
The transformed dataset.

Return type
pd.DataFrame

CHAPTER

EIGHT

NLP

Documentation Coming Soon.

CHAPTER NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX