sktools Documentation

Release 0.1.4

David Masip Bonet

Jun 17, 2021

Contents:

sktools provides tools to extend sklearn, like several feature engineering based transformers.

CHAPTER 1

Installation

To install sktools, run this command in your terminal:

```
$ pip install sktools
```

CHAPTER 2

Documentation

Can be found in https://sktools.readthedocs.io

CHAPTER 3

Usage

```
from sktools import IsEmptyExtractor

from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

...

mod = Pipeline([
    ("impute-features", IsEmptyExtractor()),
    ("model", LogisticRegression())
])

...
```

CHAPTER 4

Features

Here's a list of features that sktools currently offers:

- `sktools.encoders.NestedTargetEncoder` performs target encoding suited for variables with nesting.
- `sktools.encoders.QuantileEncoder` performs target aggregation using a quantile instead of the mean.
- `sktools.preprocessing.CyclicFeaturizer` converts numeric to cyclical features via sine and cosine transformations.
- `sktools.impute.IsEmptyExtractor` creates binary variables indicating if there are missing values.
- `sktools.matrix_denser.MatrixDenser` transformer that converts sparse matrices to dense.
- `sktools.quantilegroups.GroupedQuantileTransformer` creates quantiles of a feature by group.
- `sktools.quantilegroups.PercentileGroupFeaturizer` creates features regarding how an instance compares with a quantile of its group.
- `sktools.quantilegroups.MeanGroupFeaturizer` creates features regarding how an instance compares with the mean of its group.
- `sktools.selectors.TypeSelector` gets variables matching a type.
- `sktools.selectors.ItemsSelector` allows to manually choose some variables.
- `sktools.ensemble.MedianForestRegressor` applies the median instead of the mean when aggregating trees predictions.
- `sktools.linear_model.QuantileRegression` sklearn style wrapper for quantile regression.
- `sktools.model_selection.BootstrapFold` bootstrap cross-validator.
- `sktools.GradientBoostingFeatureGenerator` Automated feature generation through gradient boosting.

CHAPTER 5

Contributing

Fork/clone, in a fresh environment, run:

```
$ pip install -e ".[dev]"
```

To check if the unit tests are ok, run

```
$ make test
```

CHAPTER 6

License

MIT license

CHAPTER 7

Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

7.1 Installation

7.1.1 Stable release

To install sktools, run this command in your terminal:

```
$ pip install sktools
```

This is the preferred method to install sktools, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

7.1.2 From sources

The sources for sktools can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/david26694/sktools
```

Or download the tarball:

```
$ curl -OJL https://github.com/david26694/sktools/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

7.2 sktools

7.2.1 sktools package

Submodules

sktools.encoders module

Nested target encoder

class sktools.encoders.**NestedTargetEncoder**(*verbose=0*, *cols=None*, *drop_invariant=False*, *feature_mapping={}*, *return_df=True*, *handle_unknown='value'*, *handle_missing='value'*, *random_state=None*, *randomized=False*, *sigma=0.05*, *m_prior=1.0*, *m_parent=1.0*)

Bases: sklearn.base.BaseEstimator, category_encoders.utils.TransformerWithTargetMixin

Estimate of likelihood for nested data.

This is a generalization of the m-probability estimate. The main difference is that instead of using a global prior, it can use a more fine-tuned prior. This only works for nested data. For instance, I have individuals who live in counties, that are inside states. If I want to estimate the likelihood encoding for a county, it is better to use as prior the estimate for the state instead of the global estimate.

verbose: int integer indicating verbosity of the output. 0 for none.

cols: list a list of columns to encode, if None, all string columns will be encoded.

drop_invariant: bool boolean for whether or not to drop encoded columns with 0 variance.

feature_mapping: dict dictionary representing the child - parent relationship. keys are children.

return_df: bool boolean for whether to return a pandas DataFrame from transform (otherwise it will be a numpy array).

handle_missing: str options are 'return_nan', 'error' and 'value', defaults to 'value', which returns the prior probability.

handle_unknown: str options are 'return_nan', 'error' and 'value', defaults to 'value', which returns the prior probability.

randomized: bool, adds normal (Gaussian) distribution noise into training data in order to decrease overfitting (testing data are untouched).

sigma: float standard deviation (spread or "width") of the normal distribution.

m_prior: float this is the "m" in the m-probability estimate for the global mean. Higher value of m results into stronger shrinking. It is used whenever we estimate a likelihood using the global mean as a prior. M is non-negative.

m_parent: float this is the "m" in the m-probability estimate. Higher value of m results into stronger shrinking. It is used whenever we estimate a likelihood using the parent mean as a prior. M is non-negative.

```
>>> from sktools import NestedTargetEncoder
>>> import pandas as pd
>>> X = pd.DataFrame(
>>>     {
>>>         "child": ["a", "a", "b", "b", "b", "c", "c", "d", "d", "d"],
```

(continues on next page)

(continued from previous page)

```
>>>         "parent": ["e", "e", "e", "e", "e", "f", "f", "f", "f", "f",]
>>>     }
>>> )
>>> y = pd.Series([1, 2, 3, 1, 2, 4, 4, 5, 4, 4.5])
>>> ne = NestedTargetEncoder(feature_mapping={"child": "parent"}, m_prior=0)
>>> ne.fit_transform(X, y)
      child  parent
0  2.016667     1.8
1  2.016667     1.8
2  2.262500     1.8
3  2.262500     1.8
4  2.262500     1.8
5  3.683333     4.3
6  3.683333     4.3
7  4.137500     4.3
8  4.137500     4.3
9  4.137500     4.3
```

fit (*X*, *y*, ***kwargs*)

Fit encoder according to X and binary y.

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples]] Binary target values.

self [encoder] Returns self.

get_feature_names ()

Returns the names of all transformed / added columns.

feature_names: list A list with all feature names transformed or added. Note: potentially dropped features are not included!

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

Perform the transformation to new categorical data.

When the data are used for model training, it is important to also pass the target in order to apply leave one out.

X : array-like, shape = [n_samples, n_features] y : array-like, shape = [n_samples] when transform by leave one out

None, when transform without target information (such as transform test set)

p [array, shape = [n_samples, n_numeric + N]] Transformed values with encoding applied.

class sktools.encoders.**QuantileEncoder** (*verbose=0*, *cols=None*, *drop_invariant=False*, *return_df=True*, *handle_missing='value'*, *handle_unknown='value'*, *quantile=0.5*, *m=1.0*)

Bases: sklearn.base.BaseEstimator, category_encoders.utils.TransformerWithTargetMixin

Quantile Encoding for categorical features.

This a statistically modified version of target MEstimate encoder where selected features are replaced the statistical quantile instead than the mean. Replacing with the median is a particular case where self.quantile = 0.5. In comparison to MEstimateEncoder it has two tunable parameter *m* and *quantile*

verbose: int integer indicating verbosity of the output. 0 for none.

quantile: int integer indicating statistical quantile. ´0.5´ for median.

m: int integer indicating the smoothing parameter. 0 for no smoothing.

cols: list a list of columns to encode, if None, all string columns will be encoded.

drop_invariant: bool boolean for whether or not to drop columns with 0 variance.

return_df: bool boolean for whether to return a pandas DataFrame from transform (otherwise it will be a numpy array).

handle_missing: str options are 'error', 'return_nan' and 'value', defaults to 'value', which returns the target quantile.

handle_unknown: str options are 'error', 'return_nan' and 'value', defaults to 'value', which returns the target quantile.

```
>>> from sktools import QuantileEncoder
>>> import pandas as pd
>>> from sklearn.datasets import load_boston
>>> bunch = load_boston()
>>> y = bunch.target
>>> X = pd.DataFrame(bunch.data, columns=bunch.feature_names)
>>> enc = QuantileEncoder(cols=['CHAS', 'RAD']).fit(X, y)
>>> numeric_dataset = enc.transform(X)
>>> print(numeric_dataset.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 13 columns):
CRIM       506 non-null float64
ZN         506 non-null float64
INDUS      506 non-null float64
CHAS       506 non-null float64
NOX        506 non-null float64
RM         506 non-null float64
AGE        506 non-null float64
DIS        506 non-null float64
RAD        506 non-null float64
TAX        506 non-null float64
PTRATIO    506 non-null float64
B          506 non-null float64
LSTAT      506 non-null float64
dtypes: float64(13)
memory usage: 51.5 KB
None
```

fit (*X*, *y*, ***kwargs*)

Fit encoder according to X and y.

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples]] Target values.

self [encoder] Returns self.

fit_quantile_encoding (*X*, *y*)

get_feature_names()
Returns the names of all transformed / added columns.

feature_names: list A list with all feature names transformed or added. Note: potentially dropped features are not included!

quantile_encode(*X_in*)

transform(*X*, *y=None*, *override_return_df=False*)
Perform the transformation to new categorical data.

X : array-like, shape = [n_samples, n_features] y : array-like, shape = [n_samples] when transform by leave one out

None, when transform without target info (such as transform test set)

p [array, shape = [n_samples, n_numeric + N]] Transformed values with encoding applied.

class sktools.encoders.**SummaryEncoder**(*cols*, *quantiles*, *m=1.0*)
Bases: sklearn.base.BaseEstimator, category_encoders.utils.TransformerWithTargetMixin

fit(*X*, *y*)

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

sktools.ensemble module

class sktools.ensemble.**MedianForestRegressor**(**args*, ***kwargs*)
Bases: object

Random forest with median aggregation

Very similar to random forest regressor, but aggregating using the median instead of the mean. Can improve the mean absolute error a little.

```
>>> from sktools import MedianForestRegressor
>>> from sklearn.datasets import load_boston
>>> boston = load_boston()['data']
>>> y = load_boston()['target']
>>> mf = MedianForestRegressor()
>>> mf.fit(boston, y)
>>> mf.predict(boston)[0:10]
array([24. , 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, 18.9])
```

fit(*X*, *y*)

predict(*X*)

sktools.imputer module

class sktools.imputer.**IsEmptyExtractor**(*keep_trivial=False*, *cols=None*)
Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

Transformer that adds columns indicating wether columns have NaN values in a row

keep_trivial: If a column doesn't have NaN, don't add the column

cols: List of columns to transform. If None, all columns are transformed. It only works if input is a DataFrame

```
>>> from sktools import IsEmptyExtractor
>>> import pandas as pd
>>> import numpy as np
>>> X = pd.DataFrame(
>>>     {
>>>         "x": ["a", "b", np.nan],
>>>         "y": ["c", np.nan, "d"]
>>>     }
>>> )
>>> IsEmptyExtractor().fit_transform(X)
     x    y   x_na   y_na
0    a    c  False  False
1    b  NaN  False   True
2  NaN    d   True  False
```

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

transform(*X*)
For each column, it creates a new one indicating if that column is na

transform_data_frame(*X*)
Transform method in case of receiving a pandas data frame

sktools.linear_model module

class sktools.linear_model.**QuantileRegression**(*quantile=0.5*, *add_intercept=True*)
Bases: object

Quantile regression wrapper

It can work on sklearn pipelines

```
>>> from sktools import QuantileRegression
>>> from sklearn.datasets import load_boston
>>> boston = load_boston()['data']
>>> y = load_boston()['target']
>>> qr = QuantileRegression(quantile=0.9)
>>> qr.fit(boston, y)
>>> qr.predict(boston)[0:5].round(2)
array([34.87, 28.98, 34.86, 32.67, 32.52])
```

fit(*X*, *y*)

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

preprocess(*X*)

sktools.matrix_denser module

Main module.

class sktools.matrix_denser.**MatrixDenser**
Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

Converts matrix to dense.

Useful when doing an union between dense and sparse matrices.

```
>>> from sktools import MatrixDenser
>>> from scipy.sparse import csr_matrix
>>> import numpy as np
>>> sparse_matrix = csr_matrix((3, 4), dtype=np.int8)
>>> dense_matrix = MatrixDenser().fit_transform(sparse_matrix)
>>> print(dense_matrix)
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```

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

transform(*X*)

sktools.model_selection module

class sktools.model_selection.**BootstrapFold**(*n_bootstraps=10*, *size_fraction=1*)

Bases: object

Create folds based on bootsrapping

For each fold, create a bootstrap sample, training data is the bootstrapped data. The test data is the rest of the data, the data that is not in the bootstrap sample

The average size of the test data is 1/e of the total data.

n_bootstraps: int number of folds of our cross-validation setting

size_fraction: float fraction of the training data being sampled. The lower, the bigger the test set

```
>>> import numpy as np
>>> from sktools.model_selection import BootstrapFold
>>> X = np.array([
>>>     np.random.randint(1, 3, 1000),
>>>     np.random.randint(0, 2, 1000)]
>>> ).T
>>> loo = BootstrapFold(10, size_fraction=1)
>>> for train_index, test_index in loo.split(X):
>>>     print(f"Train length: {len(train_index)} Test length: {len(test_index)}")
Train length: 1000 Test length: 393
Train length: 1000 Test length: 367
Train length: 1000 Test length: 372
Train length: 1000 Test length: 377
Train length: 1000 Test length: 361
Train length: 1000 Test length: 356
Train length: 1000 Test length: 366
Train length: 1000 Test length: 369
Train length: 1000 Test length: 390
Train length: 1000 Test length: 365
```

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

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

Generator to iterate over the indices :param X: Array to split on :param y: Always ignored, exists for compatibility :param groups: Always ignored, exists for compatibility

sktools.preprocessing module

class sktools.preprocessing.**CyclicFeaturizer**(*cols*, *period_mapping=None*)
Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

Cyclic featurizer

Given some numeric columns, applies sine and cosine transformations to obtain cyclic features. This is specially suited to month of the year, day of the week, day of the month, hour of the day, etc, where the plain numeric representation doesn't work very well.

cols [list] columns to be encoded using sine and cosine transformations. Should be numeric columns

period_mapping [dict] keys should be names of cols and values should be tuples indicating minimum and maximum values

```
>>> from sktools import CyclicFeaturizer
>>> import pandas as pd
>>> df = pd.DataFrame(
>>>     {
>>>         "posted_at": pd.date_range(
>>>             start="1/1/2018", periods=365 * 3, freq="d"
>>>         ),
>>>         "created_at": pd.date_range(
>>>             start="1/1/2018", periods=365 * 3, freq="h"
>>>         )
>>>     }
>>> )
>>> df["month_posted"] = df.posted_at.dt.month
>>> df["hour_created"] = df.created_at.dt.hour
>>> transformed_df = CyclicFeaturizer(
>>>     cols=["month_posted", "hour_created"]
>>> ).fit_transform(df)
```

fit(*X*)

transform(*X*)

class sktools.preprocessing.**GradientBoostingFeatureGenerator**(*stack_to_X=True*, *add_probs=False*, *regression=False*, ***kwargs*)
Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

Feature generator from a gradient boosting.

Gradient boosting decision trees are a powerful and very convenient way to implement non-linear and tuple transformations. We treat each individual tree as a categorical feature that takes as value the index of the leaf an instance ends up falling in and then perform one hot encoding for these features.

Parameters

stack_to_X: bool, default = True Generates leaves features using the fitted self.gbm and saves them in R. If *stack_to_X is True* then *.transform* returns the original features with 'R' appended as columns. If *stack_to_X is False* then *.transform* returns only the leaves features from 'R'

add_probs: bool, default = False If *add_probs is True* then the created features are appended a probability [0,1]. If *add_probs is False* features are binary

```
>>> from sktools import GradientBoostingFeatureGenerator
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification()
>>> mf = GradientBoostingFeatureGenerator()
>>> mf.fit(X, y)
>>> mf.transform(X)
```

https://research.fb.com/wp-content/uploads/2016/11/practical-lessons-from-predicting-clicks-on-ads-at-facebook.pdf

fit(*X*, *y*)

transform(*X*)

R contains the matrix with the encoded leaves. The shape depends upon the parameters. P contains a two columns array with the probability.

sktools.quantilegroups module

Grouped Quantile Featurizer

class sktools.quantilegroups.**GroupedQuantileTransformer**(*feature_mapping*, *handle_missing='value'*, *n_quantiles=1000*, *subsample=100000*, *random_state=None*, *copy=True*)

Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

Computes the group quantile of a numeric feature with respect to a categorical feature.

For instance, if each datum is an apartment, and we have both the price and the city, this feature tries to model how expensive is an apartment in its city. The most expensive apartment in the city will score 1, and the cheapest will score 0.

It is equivalent at what it is done in: https://stackoverflow.com/questions/33899369/ranking-order-per-group-in-pandas

feature_mapping: dict mapping from numeric variables to categories that want to be used as groups.

n_quantiles: int number of quantiles per category.

subsample: int Maximum number of samples used to estimate the quantiles for computational efficiency.

random_state: Any Determines random number generation for subsampling and smoothing noise. Please see `subsample` for more details. Pass an int for reproducible results across multiple function calls. See :term:'Glossary '

copy: bool Set to False to perform inplace transformation and avoid a copy (if the input is already a numpy array).

```
>>> from sktools import GroupedQuantileTransformer
>>> import pandas as pd
>>> X = pd.DataFrame(
>>>         {
>>>             "price": [1, 2, 3, 3, 2, 10, 0],
>>>             "city": ["a", "a", "a", "b", "b", None, None],
>>>         }
>>>     )
>>> featurizer = GroupedQuantileTransformer(feature_mapping={"price": "city"})
```

(continues on next page)

(continued from previous page)

```
>>> print(featurizer.fit_transform(X).columns)
Index(['price', 'city', 'price_quantile_city'], dtype='object')
```

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

transform(*X*)

class sktools.quantilegroups.**MeanGroupFeaturizer**(*feature_mapping*, *create_features=True*, *handle_missing='value'*, *handle_unknown='value'*)

Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

Creates features establishing a relationship between a numeric and a categorical feature, by using the mean of the numeric feature in each cateogry.

For instance, if each datum is an apartment, and we have both the price and the city, the features model how expensive is an apartment with respect to the mean in the city.

feature_mapping: dict mapping from numeric variables to categories that want to be used as groups.

percentile: int percentile used to compute features

create_features: bool If false, it just computes percentiles by category

handle_missing: str options are 'return_nan' and 'value', defaults to 'value', which uses the global quantile.

handle_unknown: str options are 'return_nan' and 'value', defaults to 'value', which uses the global quantile.

```
>>> from sktools import MeanGroupFeaturizer
>>> import pandas as pd
>>> X = pd.DataFrame(
>>>         {
>>>             "price": [1, 2, 3, 3, 2, 10, 0],
>>>             "city": ["a", "a", "a", "b", "b", None, None],
>>>         }
>>>     )
>>> featurizer = MeanGroupFeaturizer(
>>>     feature_mapping={"price": "city"}
>>> )
>>> print(featurizer.fit_transform(X).columns)
Index(['price', 'city', 'mean_price_city', 'diff_mean_price_city',
       'relu_diff_mean_price_city', 'ratio_mean_price_city'],
      dtype='object')
```

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

transform(*X*)

class sktools.quantilegroups.**PercentileGroupFeaturizer**(*feature_mapping*, *percentile=50*, *create_features=True*, *handle_missing='value'*, *handle_unknown='value'*)

Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

Creates features establishing a relationship between a numeric and a categorical feature, by using a given percentile of the numeric feature in each cateogry.

For instance, if each datum is an apartment, and we have both the price and the city, if we use the percentile 50 the features model how expensive is an apartment with respect to the median in the city.

feature_mapping: dict mapping from numeric variables to categories that want to be used as groups.

percentile: int percentile used to compute features

create_features: bool If false, it just computes percentiles by category

handle_missing: str options are 'return_nan' and 'value', defaults to 'value', which uses the global quantile.

handle_unknown: str options are 'return_nan' and 'value', defaults to 'value', which uses the global quantile.

```
>>> from sktools import PercentileGroupFeaturizer
>>> import pandas as pd
>>> X = pd.DataFrame(
>>>         {
>>>             "price": [1, 2, 3, 3, 2, 10, 0],
>>>             "city": ["a", "a", "a", "b", "b", None, None],
>>>         }
>>>     )
>>> featurizer = PercentileGroupFeaturizer(
>>>     feature_mapping={"price": "city"}
>>> )
>>> print(featurizer.fit_transform(X).columns)
Index(['price', 'city', 'p50_price_city', 'diff_p50_price_city',
       'relu_diff_p50_price_city', 'ratio_p50_price_city'],
      dtype='object')
```

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

transform(*X*)

sktools.selectors module

class sktools.selectors.**ItemSelector**(*key*)

Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

For data grouped by feature, select subset of data at a provided key.

The data is expected to be stored in a 2D data structure, where the first index is over features and the second is over samples. i.e.

```
>>> len(data[key]) == n_samples
```

Please note that this is the opposite convention to scikit-learn feature matrices (where the first index corresponds to sample).

ItemSelector only requires that the collection implement getitem (data[key]). Examples include: a dict of lists, 2D numpy array, Pandas DataFrame, numpy record array, etc.

```
>>> data = {'a': [1, 5, 2, 5, 2, 8],
            'b': [9, 4, 1, 4, 1, 3]}
>>> ds = ItemSelector(key='a')
>>> data['a'] == ds.transform(data)
```

ItemSelector is not designed to handle data grouped by sample. (e.g. a list of dicts). If your data is structured this way, consider a transformer along the lines of *sklearn.feature_extraction.DictVectorizer*.

key [hashable, required] The key corresponding to the desired value in a mappable.

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

transform(*X*)

class sktools.selectors.**TypeSelector**(*dtype*)
Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

Transformer that filters a type of columns of a given data frame. This can be useful if we want to treat numeric and object columns differently

dtype [required] The type we want to filter

```
>>> from sktools import TypeSelector
>>> import pandas as pd
>>> X = pd.DataFrame(
>>>         {
>>>             "price": [1., 2., 3.],
>>>             "city": ["a", "a", "b"]
>>>         }
>>>     )
>>> selector = TypeSelector(
>>>     dtype='float'
>>> )
>>> print(selector.fit_transform(X))
    price
0    1.0
1    2.0
2    3.0
```

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

transform(*X*)

Module contents

Top-level package for sktools.

7.3 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

7.3.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/david26694/sktools/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

sktools could always use more documentation, whether as part of the official sktools docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/david26694/sktools/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

7.3.2 Get Started!

Ready to contribute? Here's how to set up *sktools* for local development.

1. Fork the *sktools* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/sktools.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv sktools
$ cd sktools/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 sktools tests
$ python setup.py test or pytest
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

7.3.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.5, 3.6, 3.7 and 3.8, and for PyPy. Check https://travis-ci.com/david26694/sktools/pull_requests and make sure that the tests pass for all supported Python versions.

7.3.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_sktools
```

7.3.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bump2version patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

7.4 Credits

7.4.1 Development Lead

- David Masip Bonet <david26694@gmail.com>

7.4.2 Contributors

None yet. Why not be the first?

7.5 History

7.5.1 0.1.4 (2021-03-20)

- Gradient boosting feature regressor

7.5.2 0.1.3 (2020-07-13)

- Bootstrap cross-validation
- Cyclic featurizer

7.5.3 0.1.2 (2020-06-24)

- L1 linear model and random forest
- Quantile encoder refactor

7.5.4 0.1.1 (2020-06-10)

- Refactor code, add group featurizers

7.5.5 0.1.0 (2020-04-19)

- First release on PyPI.

7.6 Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

S

T