Scikit-learn enhancement proposals Documentation

scikit-learn community

Sep 04, 2020

Under review

This repository is for structured discussions about large modifications or additions to scikit-learn.

The discussions must create an "enhancement proposal", similar Python enhancement proposal, that reflects the major arguments to keep in mind, the rational and usecases that are addressed, the problems and the major possible solution. It should be a summary of the key points that drive the decision, and ideally converge to a draft of an API or object to be implemented in scikit-learn.

The SLEPs are publicly available online on Read The Docs.

CHAPTER 1

SLEP007: Feature names, their generation and the API

Author Adrin Jalali

Status Under Review

Type Standards Track

Created 2019-04

1.1 Abstract

This SLEP proposes the introduction of the `feature_names_in_` attribute for all estimators, and the `feature_names_out_` attribute for all transformers. We here discuss the generation of such attributes and their propagation through pipelines. Since for most estimators there are multiple ways to generate feature names, this SLEP does not intend to define how exactly feature names are generated for all of them.

1.2 Motivation

`scikit-learn` has been making it easier to build complex workflows with the `ColumnTransformer` and it has been seeing widespread adoption. However, using it results in pipelines where it's not clear what the input features to the final predictor are, even more so than before. For example, after fitting the following pipeline, users should ideally be able to inspect the features going into the final predictor:

```
X, y = fetch_openml("titanic", version=1, as_frame=True, return_X_y=True)

# We will train our classifier with the following features:
# Numeric Features:
# - age: float.
# - fare: float.
# Categorical Features:
# - embarked: categories encoded as strings {'C', 'S', 'Q'}.
```

(continues on next page)

(continued from previous page)

```
# - sex: categories encoded as strings {'female', 'male'}.
# - pclass: ordinal integers {1, 2, 3}.

# We create the preprocessing pipelines for both numeric and categorical data.
numeric_features = ['age', 'fare']
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_features = ['embarked', 'sex', 'pclass']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Append classifier to preprocessing pipeline.
# Now we have a full prediction pipeline.
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('classifier', LogisticRegression())])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

clf.fit(X_train, y_train)
```

However, it's impossible to interpret or even sanity-check the `LogisticRegression` instance that's produced in the example, because the correspondence of the coefficients to the input features is basically impossible to figure out.

This proposal suggests adding two attributes to fitted estimators: `feature_names_in_` and `feature_names_out_`, such that in the abovementioned example `clf[-1].feature_names_in_` and `clf[-2].feature_names_out_` will be:

```
['num__age',
 'num__fare',
 'cat__embarked_C',
 'cat__embarked_Q',
 'cat__embarked_S',
 'cat__embarked_missing',
 'cat__sex_female',
 'cat__sex_male',
 'cat__pclass_1',
 'cat__pclass_2',
 'cat__pclass_3']
```

Ideally the generated feature names describe how a feature is generated at each stage of a pipeline. For instance, `cat__sex_female` shows that the feature has been through a categorical preprocessing pipeline, was originally the column `sex`, and has been one hot encoded and is one if it was originally `female`. However, this is not always possible or desirable especially when a generated column is based on many columns, since the generated feature names will be too long, for example in `PCA`. As a rule of thumb, the following types of transformers may generate feature names which corresponds to the original features:

- Leave columns unchanged, *e.g.* `StandardScaler`
- Select a subset of columns, *e.g.* `SelectKBest`

- create new columns where each column depends on at most one input column, *e.g* `OneHotEncoder`
- Algorithms that create combinations of a fixed number of features, *e.g.* `PolynomialFeatures`, as opposed to all of them where there are many. Note that verbosity considerations and `verbose_feature_names` as explained later can apply here.

This proposal talks about how feature names are generated and not how they are propagated.

1.3 Scope

The API for input and output feature names includes a `feature_names_in_` attribute for all estimators, and a `feature_names_out_` attribute for any estimator with a `transform` method, *i.e.* they expose the generated feature names via the `feature_names_out_` attribute.

Note that this SLEP also applies to resamplers the same way as transformers.

1.4 Input Feature Names

The input feature names are stored in a fitted estimator in a `feature_names_in_` attribute, and are taken from the given input data, for instance a `pandas` data frame. This attribute will be `None` if the input provides no feature names.

1.5 Output Feature Names

A fitted estimator exposes the output feature names through the `feature_names_out_` attribute. Here we discuss more in detail how these feature names are generated. Since for most estimators there are multiple ways to generate feature names, this SLEP does not intend to define how exactly feature names are generated for all of them. It is instead a guideline on how they could generally be generated.

As detailed bellow, some generated output features names are the same or a derived from the input feature names. In such cases, if no input feature names are provided, `x0` to `xn` are assumed to be their names.

1.5.1 Feature Selector Transformers

This includes transformers which output a subset of the input features, w/o changing them. For example, if a `SelectKBest` transformer selects the first and the third features, and no names are provided, the `feature_names_out_` will be `[x0, x2]`.

1.5.2 Feature Generating Transformers

The simplest category of transformers in this section are the ones which generate a column based on a single given column. These would simply preserve the input feature names if a single new feature is generated, such as in `StandardScaler`, which would map `'age'` to `'age'`. If an input feature maps to multiple new features, a postfix is added, so that `OneHotEncoder` might map `'gender'` to `'gender_female'` `'gender_fluid'` etc.

Transformers where each output feature depends on a fixed number of input features may generate descriptive names as well. For instance, a `PolynomialTransformer` on a small subset of features can generate an output feature name such as `x[0] * x[2] ** 3`.

And finally, the transformers where each output feature depends on many or all input features, generate feature names which has the form of `name0` to `namen`, where `name` represents the transformer. For instance, a `PCA` transformer will output `[pca0, ..., pcan]`, `n` being the number of PCA components.

1.5.3 Meta-Estimators

Meta estimators can choose to prefix the output feature names given by the estimators they are wrapping or not.

By default, `Pipeline` adds no prefix, *i.e* its `feature_names_out_` is the same as the `feature_names_out_` of the last step, and `None` if the last step is not a transformer.

`ColumnTransformer` by default adds a prefix to the output feature names, indicating the name of the transformer applied to them. If a column is in the output as a part of `passthrough`, it won't be prefixed since no operation has been applied on it.

1.6 Examples

Here we include some examples to demonstrate the behavior of output feature names:

```
100 features (no names) -> PCA(n_components=3)
feature_names_out_: [pca0, pca1, pca2]


100 features (no names) -> SelectKBest(k=3)
feature_names_out_: [x2, x17, x42]


[f1, ..., f100] -> SelectKBest(k=3)
feature_names_out_: [f2, f17, f42]


[cat0] -> OneHotEncoder()
feature_names_out_: [cat0_cat, cat0_dog, ...]


[f1, ..., f100] -> Pipeline(
                       [SelectKBest(k=30),
                        PCA(n_components=3)]
                   )
feature_names_out_: [pca0, pca1, pca2]


[model, make, numeric0, ..., numeric100] ->
    ColumnTransformer(
        [('cat', Pipeline(SimpleImputer(), OneHotEncoder()),
          ['model', 'make']),
         ('num', Pipeline(SimpleImputer(), PCA(n_components=3)),
          ['numeric0', ..., 'numeric100'])]
    )
feature_names_out_: ['cat_model_100', 'cat_model_200', ...,
                     'cat_make_ABC', 'cat_make_XYZ', ...,
                     'num_pca0', 'num_pca1', 'num_pca2']
```

However, the following examples produce a somewhat redundant feature names:

```
[model, make, numeric0, ..., numeric100] ->
    ColumnTransformer([
        ('ohe', OneHotEncoder(), ['model', 'make']),
        ('pca', PCA(n_components=3), ['numeric0', ..., 'numeric100'])
    ])
feature_names_out_: ['ohe_model_100', 'ohe_model_200', ...,
                     'ohe_make_ABC', 'ohe_make_XYZ', ...,
                     'pca_pca0', 'pca_pca1', 'pca_pca2']
```

1.7 Extensions

1.7.1 verbose_feature_names

To provide more control over feature names, we could add a boolean `verbose_feature_names` constructor argument to certain transformers. The default would reflect the description above, but changes would allow more verbose names in some transformers, say having `StandardScaler` map `'age'` to `'scale(age)'`.

In case of the `ColumnTransformer` example above `verbose_feature_names` could remove the estimator names, leading to shorter and less redundant names:

```
[model, make, numeric0, ..., numeric100] ->
    make_column_transformer(
        (OneHotEncoder(), ['model', 'make']),
        (PCA(n_components=3), ['numeric0', ..., 'numeric100']),
        verbose_feature_names=False
    )
feature_names_out_: ['model_100', 'model_200', ...,
                     'make_ABC', 'make_XYZ', ...,
                     'pca0', 'pca1', 'pca2']
```

Alternative solutions to a boolean flag could include:

- an integer: fine tuning the verbosity of the generated feature names.
- a `callable` which would give further flexibility to the user to generate user defined feature names.

These alternatives may be discussed and implemented in the future if deemed necessary.

1.8 Backward Compatibility

All estimators should implement the `feature_names_in_` and `feature_names_out_` API. This is checked in `check_estimator`, and the transition is done with a `FutureWarning` for at least two versions to give time to third party developers to implement the API.

CHAPTER 2

SLEP012: `InputArray`

Author Adrin jalali

Status Draft

Type Standards Track

Created 2019-12-20

2.1 Motivation

This proposal results in a solution to propagating feature names through transformers, pipelines, and the column transformer. Ideally, we would have:

```
df = pd.readcsv('tabular.csv')
# transforming the data in an arbitrary way
transformer0 = ColumnTransformer(...)
# a pipeline preprocessing the data and then a classifier (or a regressor)
clf = make_pipeline(transformer0, ..., SVC())

# now we can investigate features at each stage of the pipeline
clf[-1].input_feature_names_
```

The feature names are propagated throughout the pipeline and the user can investigate them at each step of the pipeline.

This proposal suggests adding a new data structure, called `InputArray`, which augments the data array `X` with additional meta-data. In this proposal we assume the feature names (and other potential meta-data) are attached to the data when passed to an estimator. Alternative solutions are discussed later in this document.

A main constraint of this data structure is that is should be backward compatible, *i.e.* code which expects a `numpy.ndarray` as the output of a transformer, would not break. This SLEP focuses on *feature names* as the only meta-data attached to the data. Support for other meta-data can be added later.

2.2 Backward/NumPy/Pandas Compatibility

Since currently transformers return a `numpy` or a `scipy` array, backward compatibility in this context means the operations which are valid on those arrays should also be valid on the new data structure.

All operations are delegated to the *data* part of the container, and the meta-data is lost immediately after each operation and operations result in a `numpy.ndarray`. This includes indexing and slicing, *i.e.* to avoid performance degradation, `__getitem__` is not overloaded and if the user wishes to preserve the meta-data, they shall do so via explicitly calling a method such as `select()`. Operations between two ‘‘InpuArray‘‘s will not try to align rows and/or columns of the two given objects.

`pandas` compatibility comes ideally as a `pd.DataFrame(inputarray)`, for which `pandas` does not provide a clean API at the moment. Alternatively, `inputarray.todataframe()` would return a `pandas.DataFrame` with the relevant meta-data attached.

2.3 Feature Names

Feature names are an object `ndarray` of strings aligned with the columns. They can be `None`.

2.4 Operations

Estimators understand the `InputArray` and extract the feature names from the given data before applying the operations and transformations on the data.

All transformers return an `InputArray` with feature names attached to it. The way feature names are generated is discussed in *SLEP007 - The Style of The Feature Names*.

2.5 Sparse Arrays

Ideally sparse arrays follow the same pattern, but since `scipy.sparse` does not provide the kinda of API provided by `numpy`, we may need to find compromises.

2.6 Factory Methods

There will be factory methods creating an `InputArray` given a `pandas.DataFrame` or an `xarray.DataArray` or simply an `np.ndarray` or an `sp.SparseMatrix` and a given set of feature names.

An `InputArray` can also be converted to a `pandas.DataFrame` using a `todataframe()` method.

`X` being an `InputArray`:

```
>>> np.array(X)
>>> X.todataframe()
>>> pd.DataFrame(X)  # only if pandas implements the API
```

And given `X` a `np.ndarray` or an `sp.sparse` matrix and a set of feature names, one can make the right `InputArray` using:

```
>>> make_inputarray(X, feature_names)
```

2.7 Alternative Solutions

Since we expect the feature names to be attached to the data given to an estimator, there are a few potential approaches we can take:

- `pandas` in, `pandas` out: this means we expect the user to give the data as a `pandas.DataFrame`, and if so, the transformer would output a `pandas.DataFrame` which also includes the [generated] feature names. This is not a feasible solution since `pandas` plans to move to a per column representation, which means `pd.DataFrame(np.asarray(df))` has two guaranteed memory copies.
- `XArray`: we could accept a `pandas.DataFrame`, and use `xarray.DataArray` as the output of transformers, including feature names. However, `xarray` has a hard dependency on `pandas`, and uses `pandas.Index` to handle row labels and aligns rows when an operation between two `xarray.DataArray` is done, which can be time consuming, and is not the semantic expected in `scikit-learn`; we only expect the number of rows to be equal, and that the rows always correspond to one another in the same order.

As a result, we need to have another data structure which we'll use to transfer data related information (such as feature names), which is lightweight and doesn't interfere with existing user code.

Another alternative to the problem of passing meta-data around is to pass that as a parameter to `fit`. This would heavily involve modifying meta-estimators since they'd need to pass that information, and extract the relevant information from the estimators to pass that along to the next estimator. Our prototype implementations showed significant challenges compared to when the meta-data is attached to the data.

CHAPTER 3

SLEP013: n_features_out_ attribute

Author Adrin Jalali

Status Under Review

Type Standards Track

Created 2020-02-12

3.1 Abstract

This SLEP proposes the introduction of a public n_features_out_ attribute for most transformers (where relevant).

3.2 Motivation

Knowing the number of features that a transformer outputs is useful for inspection purposes. This is in conjunction with *SLEP010: ``n_features_in_``*.

3.3 Solution

The proposed solution is for the n_features_out_ attribute to be set once a call to fit is done. In many cases the value of n_features_out_ is the same as some other attribute stored in the transformer, *e.g.* n_components_, and in these cases a Mixin such as ComponentsMixin can delegate n_features_out_ to those attributes.

3.3.1 Testing

A test to the common tests is added to ensure the presence of the attribute or property after calling fit.

3.4 Considerations

The main consideration is that the addition of the common test means that existing estimators in downstream libraries will not pass our test suite, unless the estimators also have the `n_features_out_` attribute.

The newly introduced checks will only raise a warning instead of an exception for the next 2 releases, so this will give more time for downstream packages to adjust.

There are other minor considerations:

- In some meta-estimators, this is delegated to the sub-estimator(s). The `n_features_out_` attribute of the meta-estimator is thus explicitly set to that of the sub-estimator, either via a `@property`, or directly in `fit()`.
- Some transformers such as `FunctionTransformer` may not know the number of output features since arbitrary arrays can be passed to `transform`. In such cases `n_features_out_` is set to `None`.

3.4.1 Copyright

This document has been placed in the public domain.[1]

3.4.2 References and Footnotes

[1] _Open Publication License: https://www.opencontent.org/openpub/

CHAPTER 4

SLEP009: Keyword-only arguments

Author Adrin Jalali

Status Accepted

Type Standards Track

Created 2019-07-13

Vote opened 2019-09-11

4.1 Abstract

This proposal discusses the path to gradually forcing users to pass arguments, or most of them, as keyword arguments only. It talks about the status-quo, and the motivation to introduce the change. It shall cover the pros and cons of the change. The original issue starting the discussion is located here.

4.2 Motivation

At the moment `sklearn` accepts all arguments both as positional and keyword arguments. For example, both of the following are valid:

```
# positional arguments
clf = svm.SVC(.1, 'rbf')
# keyword arguments
clf = svm.SVC(C=.1, kernel='rbf')
```

Using keyword arguments has a few benefits:

- It is more readable.
- For models which accept many parameters, especially numerical, it is less error-prone than positional arguments. Compare these examples:

```
cls = cluster.OPTICS(
    min_samples=5, max_eps=inf, metric=’minkowski’, p=2,
    metric_params=None, cluster_method=’xi’, eps=None, xi=0.05,
    predecessor_correction=True, min_cluster_size=None, algorithm=’auto’,
    leaf_size=30, n_jobs=None)

cls = cluster.OPTICS(5, inf, ’minkowski’, 2, None, ’xi’, None, 0.05,
                     True, None, ’auto’, 30, None)
```

- It allows adding new parameters closer the other relevant parameters, instead of adding new ones at the end of the list without breaking backward compatibility. Right now all new parameters are added at the end of the signature. Once we move to a keyword only argument list, we can change their order and put related parameters together. Assuming at some point numpydoc would support sections for parameters, these groups of parameters would be in different sections for the documentation to be more readable. Also, note that we have previously assumed users would pass most parameters by name and have sometimes considered changes to be backwards compatible when they modified the order of parameters. For example, user code relying on positional arguments could break after a deprecated parameter was removed. Accepting this SLEP would make this requirement explicit.

4.3 Solution

The official supported way to have keyword only arguments is:

```
def func(arg1, arg2, *, arg3, arg4)
```

Which means the function can only be called with `arg3` and `arg4` specified as keyword arguments:

```
func(1, 2, arg3=3, arg4=4)
```

The feature was discussed and the related PEP PEP3102 was accepted and introduced in Python 3.0, in 2006.

For the change to happen in `sklearn`, we would need to add the `*` where we want all subsequent parameters to be passed as keyword only.

4.4 Considerations

We can identify the following main challenges: familiarity of the users with the syntax, and its support by different IDEs.

4.4.1 Syntax

Partly due to the fact that the Scipy/PyData has been supporting Python 2 until recently, the feature (among other Python 3 features) has seen limited adoption and the users may not be used to seeing the syntax. The similarity between the following two definitions may also be confusing to some users:

```
def f(arg1, *arg2, arg3): pass # variable length arguments at arg2

def f(arg1, *, arg3): pass # no arguments accepted at *
```

However, some other teams are already moving towards using the syntax, such as `matplotlib` which has introduced the syntax with a deprecation cycle using a decorator for this purpose in version 3.1. The related PRs can be found here and here. Soon users will be familiar with the syntax.

4.4.2 IDE Support

Many users rely on autocomplete and parameter hints of the IDE while coding. Here is how the hint looks like in two different IDEs. For instance, for the above function, defined in VSCode, the hint would be shown as:

```
           func(arg1, arg2, *, arg3, arg4)

           param arg3
func(1, 2, |)
```

The good news is that the IDE understands the syntax and tells the user it's the `arg3`'s turn. But it doesn't say it is a keyword only argument.

`ipython`, however, suggests all parameters be given with the keyword anyway:

```
In [1]: def func(arg1, arg2, *, arg3, arg4): pass

In [2]: func(
  abs()                          arg3=
  all()                          arg4=
  any()                          ArithmeticError                >
  arg1=                          ascii()
  arg2=                          AssertionError
```

4.5 Scope

An important open question is which functions/methods and/or parameters should follow this pattern, and which parameters should be keyword only. We can identify the following categories of functions/methods:

- `__init__`
- Main methods of the API, *i.e.* `fit`, `transform`, etc.
- All other methods, *e.g.* `SpectralBiclustering.get_submatrix`
- Functions

With regard to the common methods of the API, the decision for these methods should be the same throughout the library in order to keep a consistent interface to the user.

This proposal suggests making only *most commonly* used parameters positional. The *most commonly* used parameters are defined per method or function, to be defined as either of the following two ways:

- The set defined and agreed upon by the core developers, which should cover the *easy* cases.
- A set identified as being in the top 95% of the use cases, using some automated analysis such as this one or this one.

This way we would minimize the number of warnings the users would receive, which minimizes the friction cause by the change. This SLEP does not define these parameter sets, and the respective decisions shall be made in their corresponding pull requests.

4.5.1 Deprecation Path

For a smooth transition, we need an easy deprecation path. Similar to the decorators developed in `matplotlib`, a proposed solution is available at [#13311](https://github.com/scikit-learn/scikit-learn/pull/13311), which deprecates the usage of positional arguments on selected functions and methods. With the decorator, the user sees a warning if they pass the designated keyword-only arguments as positional, and removing the decorator would result in an error. Examples (borrowing from the PR):

```
@warn_args
def dbscan(X, eps=0.5, *, min_samples=4, metric='minkowski'):
    pass


class LogisticRegression:

    @warn_args
    def __init__(self, penalty='l2', *, dual=False):

        self.penalty = penalty
        self.dual = dual
```

Calling `LogisticRegression('l2', True)` will result with a `DeprecationWarning`:

```
Should use keyword args: dual=True
```

Once the deprecation period is over, we'd remove the decorator and calling the function/method with the positional arguments after `*` would fail.

The final decorator solution shall make sure it is well understood by most commonly used IDEs and editors such as IPython, Jupiter Lab, Emacs, vim, VSCode, and PyCharm.

CHAPTER 5

SLEP010: n_features_in_ attribute

Author Nicolas Hug

Status Accepted

Type Standards Track

Created 2019-11-23

5.1 Abstract

This SLEP proposes the introduction of a public n_features_in_ attribute for most estimators (where relevant).

5.2 Motivation

Knowing the number of features that an estimator expects is useful for inspection purposes. This is also useful for implementing the feature names propagation (SLEP 8) . For example any of the scaler can easily create feature names if they know n_features_in_.

5.3 Solution

The proposed solution is to replace most calls to check_array() or check_X_y() by calls to a newly created private method:

```
def _validate_data(self, X, y=None, reset=True, **check_array_params)
    ...
```

The _validate_data() method will call check_array() or check_X_y() function depending on the y parameter.

If the reset parameter is True (default), the method will set the n_feature_in_ attribute of the estimator, regardless of its potential previous value. This should typically be used in fit(), or in the first partial_fit() call. Passing reset=False will not set the attribute but instead check against it, and potentially raise an error. This should typically be used in predict() or transform(), or on subsequent calls to partial_fit.

In most cases, the n_features_in_ attribute exists only once fit has been called, but there are exceptions (see below).

A new common check is added: it makes sure that for most estimators, the n_features_in_ attribute does not exist until fit is called, and that its value is correct. Instead of raising an exception, this check will raise a warning for the next two releases. This will give downstream packages some time to adjust (see considerations below).

Since the introduced method is private, third party libraries are recommended not to rely on it.

The logic that is proposed here (calling a stateful method instead of a stateless function) is a pre-requisite to fixing the dataframe column ordering issue: with a stateless check_array, there is no way to raise an error if the column ordering of a dataframe was changed between fit and predict. This is however out os scope for this SLEP, which only focuses on the introduction of the n_features_in_ attribute.

5.4 Considerations

The main consideration is that the addition of the common test means that existing estimators in downstream libraries will not pass our test suite, unless the estimators also have the n_features_in_ attribute.

The newly introduced checks will only raise a warning instead of an exception for the next 2 releases, so this will give more time for downstream packages to adjust.

There are other minor considerations:

- In most meta-estimators, the input validation is handled by the sub-estimator(s). The n_features_in_ attribute of the meta-estimator is thus explicitly set to that of the sub-estimator, either via a @property, or directly in fit().
- Some estimators like the dummy estimators do not validate the input (the 'no_validation' tag should be True). The n_features_in_ attribute should be set to None, though this is not enforced in the common check.
- Some estimators expect a non-rectangular input: the vectorizers. These estimators expect dicts or lists, not a n_samples * n_features matrix. n_features_in_ makes no sense here and these estimators just don't have the attribute.
- Some estimators may know the number of input features before fit is called: typically the SparseCoder, where n_feature_in_ is known at __init__ from the dictionary parameter. In this case the attribute is a property and is available right after object instantiation.

5.4.1 References and Footnotes

5.4.2 Copyright

This document has been placed in the public domain.[1]

[1] Each SLEP must either be explicitly labeled as placed in the public domain (see this SLEP as an example) or licensed under the Open Publication License.

CHAPTER 6

SLEP001: Transformers that modify their target

Summary

Transformers implement:

```
self = estimator.fit(X, y=None)
X_transform = estimator.transform(X)
estimator.fit(X, y=None).transform(X) == estimator.fit_transform(X, y)
```

Within a chain or processing sequence of estimators, many usecases require modifying y. How do we support this?

Doing many of these things is possible "by hand". The question is: how to avoid writing custom connecting logic.

Table of contents

6.1 Rational

6.1.1 Summary of the contract of transformers

- .transform(. . .) returns a data matrix X
- .transform(. . .) returns one feature vector for each sample of the input
- .fit_transform(. . .) is the same and .fit(. . .).transform(. . .)

6.1.2 Examples of usecases targetted

1. Over sampling:
   1. Class rembalancing: over sampling the minority class in unbalanced dataset
   2. Data enhancement (nudgging images for instance)
2. Under-sampling
   1. Stateless undersampling: Take one sample out of two
   2. Stateful undersampling: apply clustering and transform to cluster centers
   3. Coresets: return a smaller number of samples and associated sample weights
3. Outlier detection:
   1. Remove outlier from train set
   2. Create a special class 'y' for outliers
4. Completing y:
   1. Missing data imputation on y
   2. Semi-supervised learning (related to above)
5. Data loading / conversion
   1. Pandas in => (X, y) out
   2. Images in => patches out
   3. Filename in => (X, y) with multiple samples (very useful in combination with online learning)
   4. Database query => (X, y) out
6. Aggregate statistics over multiple samples
   1. Windowing-like functions on time-series

   In a sense, these are dodgy with scikit-learn's cross-validation API that knows nothing about sample structure. But the refactor of the CV API is really helping in this regard.

These usecases pretty much require breaking the contract of the Transformer, as detailed above.

The intuition driving this enhancement proposal is that the more the data-processing pipeline becomes rich, the more the data grow, the more the usecases above become important.

6.2 Enhancements proposed

6.2.1 Option A: meta-estimators

Proposal

This option advocates that any transformer-like usecase that wants to modify y or the number of samples should not be a transformer-like but a specific meta-estimator. A core-set object would thus look like:

- From the user perspective:

```
from sklearn.sample_shrink import BirchCoreSet
from sklearn.ensemble import RandomForest
estimator = BirchCoreSet(RandomForest())
```

- From the developer perspective:

```
class BirchCoreSet(BaseEstimator):

   def fit(self, X, y):
       # The logic here is wrong, as we need to handle y:
       super(BirchCoreSet, self).fit(X)
       X_red = self.subcluster_centers_
       self.estimator_.fit(X_red)
```

Benefits

1. No change to the existing API
2. The meta-estimator pattern is very powerful, and pretty much anything is possible.

Limitations

The different limitations listed below are variants of the same conceptual difficulty

1. It is hard to have mental models and garantees of what a meta-estimator does, as it is by definition super versatile

   This is both a problem for the beginner, that needs to learn them on an almost case-by-case basis, and for the advanced user, that needs to maintain a set of case-specific code

2. The “estimator heap” problem.

   Here the word heap is used to denote the multiple pipelines and meta-estimators. It corresponds to what we would naturally call a “data processing pipeline”, but we use “heap” to avoid confusion with the pipeline object.

   Heaps combining many steps of pipelines and meta-estimators become very hard to inspect and manipulate, both for the user, and for pipeline-management (aka “heap-management”) code. Currently, these difficulties are mostly in user code, so we don’t see them too much in scikit-learn. Here are concrete examples

   1. Trying to retrieve coefficients from a model estimated in a “heap”. Eg:

      - you know there is a lasso in your stack and you want to get it’s coef (in whatever space that resides?): `pipeline.named_steps['lasso'].coef_` is possible.
      - you want to retrieve the coef of the last step: `pipeline.steps[-1][1].coef_` is possible.

      With meta estimators this is tricky. Solving this problem requires https://github.com/scikit-learn/scikit-learn/issues/2562#issuecomment-27543186 (this enhancement proposal is not advocating to solve the problem above, but pointing it out as an illustration)

   2. DaskLearn has modified the logic of pipeline to expose it as a computation graph. The reason that it was relatively easy to do is that there was mostly one object to modify to do the dispatching, the Pipeline object.

   3. A future, out-of-core “conductor” object to fit a “heap” in out of core by connecting it to a data-store would need to have a representation of the heap. For instance, when chaining random projections with Birch coresets and finally SGD, the user would need to specify that random projections are stateless, birch needs to do one pass of the data, and SGD a few. Given this information, the conductor could orchestrate pull the data from the data source, and sending it to the various steps. Such an object is much harder to implement if the various steps are to be combined in a heap. Note that the scikit-learn pipeline can

only implement a linear "chain" like set of processing. For instance a One vs All will never be able to be implemented in a scikit-learn pipeline.

This is not a problem in non out-of-core settings, in the sense that the BirchCoreSet meta-estimator would take care of doing a pass on the data before feeding it to its sub estimator.

In conclusion, meta-estimators are harder to comprehend (problem 1) and write (problem 2).

That said, we will never get rid of meta estimators. It is a very powerful pattern. The discussion here is about extending a bit the estimator API to have a less pressing need for meta-estimators.

6.2.2 Option B: transformer-like that modify y

Two variants

1. Changing the semantics of transformers to modify y and return something more complex than a data matrix X
2. Introducing new methods (and a new type of object)

There is an emerging consensus for option 2.

``transform`` modifying y

Variant 1 above could be implementing by allowing transform to modify y. However, the return signature of transform would be unclear.

Do we modify all transformers to return a y (y=None for unsupervised transformers that are not given y?). This sounds like leading to code full of surprises and difficult to maintain from the user perspective.

We would loose the contract that the number of samples is unchanged by a transformer. This contract is very useful (eg for model selection: measuring error for each sample).

For these reasons, we feel new methods are necessary.

Proposal

Introduce a `TransModifier` type of object with the following API (names are discussed below):

- `X_new, y_new = estimator.fit_modify(X, y)`
- `X_new, y_new = estimator.trans_modify(X, y)`

Or:

- `X_new, y_new, sample_props = estimator.fit_modify(X, y)`
- `X_new, y_new, sample_props = estimator.trans_modify(X, y)`

Contracts (these are weaker contracts than the transformer:

- Neither `fit_modify` nor `trans_modify` are guarantied to keep the number of samples unchanged.
- `fit_modify` may not exist (questionnable)

Design questions and difficulties

Should there be a fit method?

In such estimators, it may not be a good idea to call fit rather than fit_modify (for instance in coreset).

How does a pipeline use such an object?

In particular at test time?

1. Should there be a transform method used at test time?
2. What to do with objects that implement both `transform` and `trans_modify`?

Creating y in a pipeline makes error measurement harder For some usecases, test time needs to modify the number of samples (for instance data loading from a file). However, these will by construction a problem for eg cross-val-score, as in supervised settings, these expect a y_true. Indeed, the problem is the following:

- To measure an error, we need y_true at the level of `sklearn.model_selection.cross_val_score` or `sklearn.model_selection.GridSearchCV`
- y_true is created inside the pipeline by the data-loading object.

It is thus unclear that the data-loading usecases can be fully integrated in the CV framework (which is not an argument against enabling them).

For our CV framework, we need the number of samples to remain constant: for each y_pred, we need a corresponding y_true.

Proposal 1: use transform at `predict` time.

1. Objects implementing both `transform` and `trans_modify` are valid
2. The pipeline's `predict` method use `transform` on its intermediate steps

The different semantics of `trans_modify` and `transform` can be very useful, as `transform` keeps untouched the notion of sample, and `y_true`.

Proposal 2 Modify the scoring framework

One option is to modify the scoring framework to be able to handle these things, the scoring gets the output of the chain of trans_modify for y. This should rely on clever code in the `score` method of pipeline. Maybe it should be controlled by a keyword argument on the pipeline, and turned off by default.

How do we deal with sample weights and other sample properties?

This discussion feeds in the `sample_props` discussion (that should be discussed in a different enhancement proposal).

The suggestion is to have the sample properties as a dictionary of arrays `sample_props`.

Example usecase useful to think about sample properties: coresets: given (X, y) return (X_new, y_new, weights) with a much smaller number of samples.

This example is interesting because it shows that TransModifiers can legitimately create sample properties.

Proposed solution:

TransModifiers always return (X_new, y_new, sample_props) where sample_props can be an empty dictionary.

Naming suggestions

In term of name choice, the rational would be to have method names that are close to 'fit' and 'transform', to make discoverability and readability of the code easier.

- Name of the object (referred in the docs): - TransModifier - TransformPipe - PipeTransformer
- Method to fit and apply on training - fit_modify - fit_pipe - pipe_fit - fit_filter
- Method to apply on new data - trans_modify - transform_pipe - pipe_transform

Benefits

- Many usecases listed above will be implemented scikit-learn without a meta-estimator, and thus will be easy to use (eg in a pipeline). Many of these are patterns that we should be encouraging.
- The API being more versatile, it will be easier to create application-specific code or framework wrappers (ala DaskLearn) that are scikit-learn compatible, and thus that can be used with the parameter-selection framework. This will be especially true for ETL (extract transform and load) pattern.

Limitations

- Introducing new methods, and a new type of estimator object. There are probably a total of **3 new methods** that will get introduced by this enhancement: fit_modify, trans_modify, and partial_fit_modify.
- Cannot solve all possible cases, and thus we will not get rid of meta-estimators.

6.3 TODO

- Implement an example doing outlier filtering
- Implement an example doing data downsampling

CHAPTER 7

SLEP002: Dynamic pipelines

Summary

Create and manipulate pipelines with ease.

Table of contents

7.1 Goals

- Being backward-compatible
- Allow interactive pipeline construction (for example in IPython)
- Support adding and replacing parts of pipeline
- Support using steps as label (y's) transformers

7.2 Design

7.2.1 Imports

In addition to `Pipeline` class some additional wrappers are proposed as part of public API:

```
from sklearn.pipeline import (Pipeline, fitted, transformer, predictor
                              label_transformer, label_predictor,
                              ignore_transform, ignore_predict)
```

7.2.2 Pipeline creation

Backward-compatible

Of course, old syntax should be supported:

```
pipe = Pipeline(steps=[('name1', estimator1), ('name2', 'estimator2)]
```

Proposed default constructor

It is not backward-compatible, but it shouldn't break most of old code:

```
pipe = Pipeline()
```

It is not yet configured, so trying to use it should fail:

```
>>> pipe.predict(...)
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
NotFittedError: This Pipeline instance is not fitted yet

>>> pipe.fit(...)
Traceback (most recent call last):
...
NotConfiguredError: This Pipeline instance is not configured yet
```

Proposed construction from iterable of dicts

Dictionaries emphasize structure:

```
pipe = Pipeline(
    steps=[
        {'name1': Estimator1()},
        {'name2': Estimator2()},
    ]
)
```

Every dict should be of length 1:

```
>>> pipe = Pipeline(
...     steps=(
...         {'name1': Estimator1(),
...          'name2': Estimator2()},
...         {},
...     ),
... )
Traceback (most recent call last):
...
TypeError: Wrong step definition
```

Proposed construction from `collections.OrderedDict`

It is probably the most natural way to create a pipeline:

```
pipe = Pipeline(
    collections.OrderedDict([
        ('name1', Estimator1()),
        ('name2', Estimator2()),
    ]),
)
```

7.2.3 Backward-compatibility notice

As user can provide object of any type as `steps` argument to constructor, there is no way to be 100% compatible, if we are going to maintain our oun type for `Pipeline.steps`. But in most cases people provide `list` object as `steps` parameter, so being backward-compatible with `list` API should be fine.

7.2.4 Adding estimators

Backward-compatible

Although not documented, but popular method of modifying (not fitted) pipelines should be supported:

```
pipe.steps.append(['name', estimator])
```

The only difference is that special handler is returned instead of `None`.

Enhanced: by indexing

Using dict-like syntax if very user-friendly:

```
pipe.steps['name'] = estimator
```

Enhanced: `add` function

Alias to previous two calls:

```
pipe.steps.add('name', estimator)
```

And also:

```
pipe.add_estimator('name', estimator)
```

Adding estimators with type specification

Estimator types will be discussed later, but some functions belong to this section:

```
pipe.add_estimator('name0', estimator0).mark_fitted()
pipe.add_transformer('name1', estimator1)  # never calls .fit (x, y -> x)
pipe.add_predictor('name2', estimator2)  # never calls .trasform (x -> y)
pipe.add_label_transformer('name3', estimator3)  # (y -> y)
pipe.add_label_predictor('name4', estimator4)  # (y -> y)
```

7.2.5 Steps (subestimators) access

Backward-compatible

Indexing by number should return `(step, estimator)` pair:

```
>>> pipe.steps[0]
('name', SomeEstimator(...))
```

Enhanced access via indexing

One should be able to retrieve any estimator with indexing by step's name:

```
>>> pipe.steps['mame']
SomeEstimator(param1=value1, param2=value2)
```

Enhanced access via attributes

Dotted access should also work if name of step is valid python name literal and there is no inference with internal methods:

```
>>> pipe.steps.name
SomeEstimator(param1=value1, param2=value2)

>>> pipe.steps.get
<bound method index of <StepsOrderedDict object at ...>>

>>> pipe.add_transformer('my transformer', estimator)
>>> pipe.steps.my transformer
File ...
pipe.steps_.my transformer
               ^
SyntaxError: invalid syntax
```

7.2.6 Replacing estimators

Backward-compatible

Replacing should only be supported via access to `.steps` attribute. This way there is no ambiguity with new/old subestimator subtype:

```
pipe = Pipeline(steps=[('name', SomeEstimator())])
pipe.steps[0] = ('name', AnotherEstimator())
```

Replace via indexing by step name

Dict-like behavior can be used too:

```
pipe = Pipeline(steps=[('name', SomeEstimator())])
pipe.steps['name'] = AnotherEstimator()
```

Replace via `replace()` function

This way one can obtain special handler:

```
pipe.steps.replace('old_step_name', 'new_step_name', NewEstimator())
pipe.steps.replace('step_name', 'new_name',
                   SomeEstimator()).mark_transformer()
```

Rename step via `rename()` function

Simple way to change step's name (doesn't affect anything except object representation):

```
pipe.steps.rename('old_name', 'new_name')
```

7.2.7 Modifying estimators

Changing estimator params should only be performed via `pipeline.set_params()`. If somebody calls `subestimator.set_params()` directly, pipeline object will have no idea about changed state. There is no easy way to control it, so docs should just warm users about it.

On the other hand, there exist not-so-easy way to at least warm users during runtime: pipeline will have to keep params of all its children and compare them with actual params during `fit` or `predict` routines and raise a warning if they do not match. This functionality may be implemented as part of some kind of debugging mode.

7.2.8 Deleting estimators

Backward-compatible

Backward-compatible way to delete a step is to `del` it via index number:

```
del pipe.steps[2]
```

Enhanced indexing

A little more user-friendly way to remove a step can be achieved using enhanced indexing:

```
pipe = Pipeline()
est1 = Estimator1()
est2 = Estimator2()

pipe.steps.add('name1', est1)
pipe.steps.add('name2', est2)

del pipe.steps['name1']
del pipe.steps[pipe.steps.index(est2)]
```

Using dict/list-like `pop()` functions

Last estimator in a chain can be deleted with any of these calls:

```
>>> pipe.steps.pop()
SomeEstimator()

>>> pipe.steps.popitem()
('some_name', SomeEstimator())
```

Likewise, first estimator in the pipeline can be removed with any of these calls:

```
>>> pipe.steps.popfront()
BeginEstimator()

>>> pipe.steps.popitemfront()
('begin', BeginEstimator)
```

Any step can be removed with `pop(step_name)` or `popitem(step_name)`.

7.2.9 Fitted flag reset

Internally `Pipeline` object should keep track on whatever it is fitted or not. It should consider itself fitted if it wasn't modified after:

- successful call to `.fit`:

```
pipe.fit(...)  # Got fitted pipeline if no exception was raised
```

- construction with list of estimators, all marked as fitted via `fitted` function:

```
pipe = pipeline.Pipeline(steps=[
    ('name1', fitted(estimator1)),
    ('name2', fitted(estimator2)(,
    ...
])
```

- adding fitted estimator to fitted pipeline:

```
pipe.steps.append(fitted(estimator1))
pipe.steps['new_step'] = fitted(estimator2)
pipe.add_transformer('some_key', estimator3).set_fitted()
```

- renaming step in fitted pipeline
- removing first or last step from fitted pipeline

7.2.10 Subestimator types

Subestimator type contains information about the way a pipeline should process a step with that subestimator.

Subestimator type can be specified:

- **By wrapping estimator with subtype constructor call:**
  - when creating pipeline:

    ```
    Pipeline([
        ('name1', transformer(estimator)),
        ('name2', predictor(estimator)),
        ('name3', label_transformer(estimator)),
        ('name4', label_predictor(estimator)),
    ])
    ```

  - when adding or replacing a step:

    ```
    pipe.steps.append(['name', label_predictor(estimator])
    pipe.steps.add('name', label_transformer(estimator))
    pipe.add_estimator('name', predictor(estimator))
    pipe.steps.replace('name', transformer(fitted(estimator)))
    pipe.steps['name'] = fitted(predictor(estimator))
    ```

- Using `pipe.add_*` methods:

```
pipe.add_transformer('transformer', Transformer())
pipe.add_predictor('predictor', Predictor())
pipe.add_label_transformer('l_transformer', LabelTransformer())
pipe.add_label_predictor('l_predictor', LabelPredictor())
```

- Using special handler methods:

  ```
  pipe.add_estimator('name1', EstimatorA()).mark_transformer()
  pipe.steps.add('name2', EstimatorB()).mark_predictor()
  pipe.steps.append(['name3', EstimatorC()]).mark_label_transformer()
  pipe.steps.replace('name4', EstimatorD()).mark_label_predictor()
  pipe.steps.replace('name4', EstimatorE()).mark('label_transformer')
  ```

Transformer

Is a default type.

It is processed like this:

```
y_new = y
if fiting:
    X_new = step_estimator.fit_transform(X, y)
else:
    X_new = step.transform(X, y)
```

Predictor

It is processed like this:

```
X_new = X
if fitting:
    y_new = step_estimator.fit_predict(X, y)
else:
    y_new = step_estimator.predict(X, y)
```

Label transformer

Processing pseudocode:

```
X_new = X
if fitting:
    y_new = step_estimator.fit_transform(y)
else:
    y_new = step_estimator.transform(y)
```

Label predictor

Processing pseudocode:

```
X_new = X
if fitting:
    y_new = step_estimator.fit_predict(y)
else:
    y_new = step_estimator.predict(y)
```

7.2.11 Special handlers and wrapper functions

Assuming estimator is already fitted

to add estimator, that was already fitted to a pipline one can use fitted function:

```
est = SomeEstimator().fit(some_data)
pipe.steps.add('prefitted', fitted(est))
```

or special hanlder method:

```
pipe.steps.add('prefitted', est).mark_fitted()
# or
pipe.steps.add('prefitted', est).mark('fitted')
```

Ignoring estimator during prediction

In some cases we only need to apply estimator only during fit-phase:

```
pipe.add_estimator('sampler', ignore_transform(Sampler()))
# or
pipe.add_estimator('sampler', Sampler()).mark_ignore_transform()
# or
pipe.add_estimator('sampler', Sampler()).mark('ignore_transform')
```

If it is `predictor` or `label_predictor`, then one should use `ignore_predict`:

```
pipe.add_estimator('cluster', ignore_predict(predictor(ClusteringEstimator())))
# or
pipe.add_estimator('cluster', predictor(ClusteringEstimator())).mark_ignore_predict()
# or
pipe.add_estimator('cluster', predictor(ClusteringEstimator())).mark('ignore_predict')
```

Setting subestimator type

As specified above setting subestimator type can be performed with special handler or special function call.

Combining multiple flags

All sorts of syntax combinations should be supported:

```
pipe.steps.add('step', fitted(predictor(Estimator())))
pipe.steps.add('step', predictor(fitted(Estimator())))
pipe.steps.add('step', predictor(Estimator())).mark_fitted()
pipe.steps.add('step', fitted(Estimator())).mark_predictor()
pipe.steps.add('step', Estimator()).mark_predictor().mark_fitted()
pipe.steps.add('step', Estimator()).mark_fitted().mark_predictor()
pipe.steps.add('step', Estimator()).mark('fitted').mark_predictor()
pipe.steps.add('step', Estimator()).mark('predictor').mark_fitted()
pipe.steps.add('step', Estimator()).mark('predictor').mark('fitted')
pipe.steps.add('step', Estimator()).mark('fitted').mark('predictor')
pipe.steps.add('step', Estimator()).mark('fitted', 'predictor')
pipe.steps.add('step', Estimator()).mark('predictor', 'fitted')
```

7.2.12 Type of steps object

This is internal type, users shouldn’r usualy mess with that. But public methods should be considered as part of pipeline API.

Attributes and methods with standard behavior

Special methods:

- `__contains__()`, `__getitem__()`, `__setitem__()`, `__delitem__()`
- `__len__()`, `__iter__()`
- `__add__()`, `__iadd__()`

Methods:

- `get()`, `index()`
- `extend()`, `insert()`
- `keys()`, `items()`, `values()`
- `clear()`, `pop()`, `popitem()`, `popfront()`, `popitemfront()`

Non-standard methods

- `replace()`
- `rename()`

Not supported arguments and methods

This type provides dict-like and list-like interfaces, but following methods and attributes are not supported:

- `fromkeys()`
- `setdefault()`
- `sort()`
- `__mul__()`, `__rmul__()`, `__imul__()`

Any attempt to use them should fail with `AttributeError` or `NotImplementedError`

Thease methods may be not supported:

- `__ge__()`, `__gt__()`
- `__le__()`, `__lt__()`

7.2.13 Serialization

- Support loading/unpickling pipelines from old scikit-learn versions
- Keep track of API version in `__getstate__` / `picklier`: all future versions should support unpickling all previous versions of enhanced pipeline
- Serialization of `.steps` attribute (without master pipeline) may be not supported.

7.3 Examples

7.3.1 Example: remove outliers

Proposed design allows to do many things, but some of them have to be done in two steps. But it shouldn't be a problem, as one can make a pipeline with those steps:

```
def make_outlier_remover(bad_value=-1):
    outlier_remover = Pipeline()
    outlier_remover.steps.add(
        'data',
        DropLinesOfXCorrespondingLabel(remove_if=bad_value),
    )
    outlier_remover.steps.add(
        'labels',
        DropLabelsIf(remove_if=bad_value),
    ).mark_label_transformer()
    return outlier_remover
```

7.3.2 Example: sample dataset

We can use previous example function for this:

```
def make_sampler(percent=75):
    sentinel = object()
    sampler = Pipeline()
    sampler.steps.add(
        'sample',
        LabelSomeRowsAs(percent=percent, label=sentinel),
    ).mark('predictor', 'ignore_predict')
    sampler.steps.add(
        'down',
        make_outlier_remover(bad_value=sentinel),
    )
    return sampler
```

7.4 Benefits

- Users can use old code with new pipeline: usual `__init__`, `set_params`, `get_params`, `fit`, `transform` and `predict` are the only requirements of subestimators.
- Users can use new pipeline with their old code: pipeline is stil usual estimator, that supports usual set of methods.
- We finally can transform `y` in a pipeline.

7.5 Drawbacks

Well, it's a lot of code to write and support…

CHAPTER 8

SLEP003: Consistent inspection for transformers

. topic:: **Summary**

> Inspect transformers' output shape and dependence on input features consistently with `get_feature_dependence() -> boolean (n_outputs, n_inputs)`

Table of contents

8.1 Goals

- Make it easy to describe features output from transformation pipelines in terms of input features.
- Enable describing (e.g. with feature names) only a subset of output features.
- Enable inspecting which input features are used or important in the model, such that models can be compressed.

8.2 Design

Transformers will provide a method of the following description.

```
def get_feature_dependence(self):
    """Identify dependencies between input and output features

    Returns
    =======
    D : array or CSR matrix, shape [n_output_features, n_input_features]
        If D[i, j] == 0 then input feature j does not contribute to output
        feature i. Otherwise, feature i may depend on feature j.

        This should not be mutated by the user.
    """
```

Thus:

- it is easy to inspect a transformer to see how many output features it produces.
- by calling this method on the consitutent transformers of a `FeatureUnion` can determine from the shapes which features derive from which transformer.
- it is easy to inspect a transformer to see how many input features it requires, making it easy to invent default feature names for transformation.
- models can be compressed by first identifying features unused among the most informative features for a downstream classifier
- determining and storing all feature names/descriptions within a pipeline may be expensive. By tracing dependencies, feature descriptions can be calculated for sparsely many features, perhaps via a method `describe_features(sparse_input_feature_names, indices)`

8.3 Examples

The following are rough implementations for some existing transformers:

```
class StandardScaler:
    def get_feature_dependence(self):
        # A diagonal matrix
        n_features = len(self.scale_)
        # csr_matrix constructed with (data, indices, indptr)
        return csr_matrix((np.ones(n_features), np.arange(n_features),
                           np.arange(n_features + 1)))

class PCA:
    def get_feature_dependence(self):
        return self.components_

class SelectorMixin:
    def get_feature_dependence(self):
        # One nonzero element per row
        mask = self._get_support_mask()
        idx = np.flatnonzero(mask)
        return csr_matrix((np.ones(idx.shape), idx, np.arange(len(idx) + 1)),
                          shape=(len(mask), len(idx)))

class Pipeline:
    def get_feature_dependence(self):
        # Dot product of constituent dependency matrices
```

(continues on next page)

(continued from previous page)

```
        D = None
        for name, trans in self.steps:
            if trans is not None:
                if D is None:
                    D = trans.get_feature_dependence()
                else:
                    D = safe_sparse_dot(trans.get_feature_dependence(), D)
        return D

class FeatureUnion:
    def get_feature_dependence(self):
        # Concatenation of constituent dependency matrices
        Ds = [trans.get_feature_dependence()
              for _, trans in self.transformer_list
              if trans is not None]
        if any(issparse(D) for D in Ds):
            Ds = [csr_matrix(D) for D in Ds]
            return sp.hstack(Ds)
        return np.hstack(Ds)
```

8.4 Problem cases

8.4.1 1d input / output

The input to a transformer may be a 1-dimensional array-like. This is often the case for feature extractors, which may take a list of dicts, a list of strings or a list of files, for instance. In this case, `get_feature_dependence` should spoof the existence of a single input feature, returning a matrix of shape `(1, n_output_features)`.

While not included in scikit-learn repository, transformers may translate one 1-d array (or Series) into another 1-d array. It would be appropriate in this context for `get_feature_dependence` to return `array([[1]])`.

8.4.2 Pandas DataFrame input

The input features should correspond to columns in the case that a transformer is designed to take a Pandas DataFrame as input.

8.4.3 Constituent transformers lack this feature

Where `Pipeline` or `FeatureUnion` has a constituent transformer that lacks this method, calling `hasattr(pipeline, 'get_feature_dependence')` should similarly return False. This can be implemented using a `property`.

8.5 Alternatives

8.5.1 An attribute

An attribute `feature_dependence_` could be used instead of a method, but for the following issues:

1. `feature_dependence_` cannot be constructed statically in `Pipeline` and `FeatureUnion` in case some constituent transformers. These could be implemented dynamically with a `property` and raise an error when .

2. Often one would want to calculate the feature dependence matrix for all steps of a `Pipeline` excluding the last. This entails a dynamic approach to calculating the dependencies.

3. An attribute will in some cases be redundant relative to existing attributes, such as `RFE.support_`

The main advantage of an attribute is that it may encourage the information to be stored at fit time avoiding recalculation. However this can be done when necessary with a method. An attribute may or may not have greater visibility to users.

8.5.2 Allow other sparse formats

I have suggested consistently using CSR so that it is efficient to perform matrix products as well as to look up active input features given selected output feature (a standard model inspection task).

DIA format may be more efficient in some cases, taking half the memory and allowing for more efficient matrix products and lookup relative to CSR. However the API assurances of a single format seem to outweigh DIA's benefits.

8.5.3 Require binary values

Not binarising the output has the benefit of not copying in some cases. It does, howeve, risk numerical over/underflow in matrix multiplication.

8.5.4 Transposition

Return shape could be `(n_input, n_output)`, which some users may find more intuitive. The current proposal has the following advantages:

- consistency with notion of dependence: matrix maps first axis to dependencies in second.
- consistency with `PCA.components_`
- main purpose is model inspection, hence lookup by row is common in the current proposal.
- transposing the shape would imply using CSC for the same efficiencies, which is less commonly used than CSR throghout scikit-learn.

CHAPTER 9

SLEP004: Data information

This is a specification to introduce data information (as `sample_weights`) during the computation of an estimator methods (`fit`, `score`, `...`) based on the different discussion proposes on issues and PR :

- Consistent API for attaching properties to samples #4497
- Acceptance of sample_weights in pipeline.score #7723
- Establish global error state like np.seterr #4660
- Should cross-validation scoring take sample-weights into account? #4632
- Sample properties #4696

Probably related PR: - Add feature_extraction.ColumnTransformer #3886 - Categorical split for decision tree #3346

Google doc of the sample_prop discussion done during the sklearn day in paris the 7th June 2017: https://docs.google.com/document/d/1k8d4vyw87gWODiyAyQTz91Z1KOnYr6runx-N074qIBY/edit

Table of contents

9.1 1. Requirement

These requirements are defined from the different issues and PR discussions:

- User can attach information to samples.

- Must be a DataFrame like object.
- Can be given to `fit`, `score`, `split` and every time user give X.
- Must work with every meta-estimator (`Pipeline, GridSearchCV, cross_val_score`).
- Can specify what sample property is used by each part of the meta-estimator.
- Must raise an error if not necessary extra information are given to an estimator. In the case of meta-estimator these errors are not raised.

Requirement proposed but not used by this specification: - User can attach feature properties to samples.

9.2 2. Definition

Some estimator in sklearn can change their behavior when an attribute `sample_props` is provided. `sample_props` is a dictionary (`pandas.DataFrame` compatible) defining sample properties. The example bellow explain how a `sample_props` can be provided to LogisticRegression to weighted the samples:

```
import numpy as np
from sklearn import datasets
from sklearn.linear_model import LogisticRegression

digits = datasets.load_digits()
X = digits.data
y = digits.target

# Define weights used by sample_props
weights_fit = np.random.rand(X.shape[0])
weights_fit /= np.sum(weights_fit)
weights_score = np.random.rand(X.shape[0])
weights_score /= np.sum(weights_score)

logreg = LogisticRegression()

# Fit and score a LogisticRegression without sample weights
logreg = logreg.fit(X, y)
score = logreg.score(X, y)
print("Score obtained without applying weights: %f" % score)

# Fit LogisticRegression without sample weights and score with sample weights
logreg = logreg.fit(X, y)
score = logreg.score(X, y, sample_props={'weight': weights_score})
print("Score obtained by applying weights only to score: %f" % score)

# Fit and score a LogisticRegression with sample weights
log_reg = logreg.fit(X, y, sample_props={'weight': weights_fit})
score = logreg.score(X, y, sample_props={'weight': weights_score})
print("Score obtained by applying weights to both"
      " score and fit: %f" % score)
```

When an estimator expects a mandatory `sample_props`, an error is raised for each property not provided. Moreover if an unintended properties is given through `sample_props`, a warning will be launched to prevent that the result may be different from the one expected. For example, the following code :

```
import numpy as np
from sklearn import datasets
```

(continues on next page)

(continued from previous page)

```
from sklearn.cluster import KMeans
from sklearn.pipeline import Pipeline

digits = datasets.load_digits()
X = digits.data
y = digits.target
weights = np.random.rand(X.shape[0])

logreg = LogisticRegression()

# This instruction will raise the warning
logreg = logreg.fit(X, y, sample_props={'bad_property': weights})
```

will **raise the warning message**: “sample_props[‘bad_property’] is not used by `LogisticRegression.fit`. The results obtained may be different from the one expected.”

We provide the function `sklearn.seterr` in the case you want to change the behavior of theses messages. Even if there are considered as warnings by default, we recommend to change the behavior to raise as errors. You can do it by adding the following code:

```
sklearn.seterr(sample_props="raise")
```

Please refer to the documentation of `np.seterr` for more information.

9.3 3. Behavior of `sample_props` for meta-estimator

9.3.1 3.1 Common routing scheme

Meta-estimators can also change their behavior when an attribute `sample_props` is provided. On that case, `sample_props` will be sent to any internal estimator and function supporting the `sample_props` attribute. In other terms all the property defined by `sample_props` will be transmitted to each internal functions or classes supporting `sample_props`. For example in the following example, the property `weights` is sent through `sample_props` to `pca.fit_transform` and `logistic.fit`:

```
import numpy as np
from sklearn import decomposition, datasets, linear_model
from sklearn.pipeline import Pipeline

digits = datasets.load_digits()
X = digits.data
y = digits.target

logistic = linear_model.LogisticRegression()
pca = decomposition.PCA()
pipe = Pipeline(steps=[('pca', pca), ('logistic', logistic),])

# Define weights
weights = np.random.rand(X.shape[0])
weights /= np.sum(weights)

# weights is send to pca.fit_transform and logistic.fit
pipe.fit(X, sample_props={"weights": weights})
```

By contrast with the estimator, no warning will be raised by a meta-estimator if an extra property is sent through sample_props. Anyway, errors are still raised if a mandatory property is not provided.

9.3.2 3.2 Override common routing scheme

You can override the common routing scheme of sample_props of nested objects by defining sample properties of the form <component>__<property>.

You can override the common routing scheme of sample_props by defining your own routes through the routing attribute of a meta-estimator.

A route defines a way to override the value of a key of sample_props by the value of another key in the same sample_props. This modification is done every time a method compatible with sample_prop is called.

To illustrate how it works, if you want to send weights only to pca, you can define a sample_prop with a property pca__weights:

```python
import numpy as np
from sklearn import decomposition, datasets, linear_model
from sklearn.pipeline import Pipeline

digits = datasets.load_digits()
X = digits.data
y = digits.target

logistic = linear_model.LogisticRegression()
pca = decomposition.PCA()

# Create a route using routing
pipe = Pipeline(steps=[('pca', pca), ('logistic', logistic),])

# Define weights
weights = np.random.rand(X.shape[0])
weights /= np.sum(pca_weights)
pca_weights = np.random.rand(X.shape[0])
pca_weights /= np.sum(pca_weights)

# Only pca will receive pca_weights as weights
pipe.fit(X, sample_props={'pca__weights': pca_weights})

# pca will receive pca_weights and logistic will receive weights as weights
pipe.fit(X, sample_props={'pca__weights': pca_weights,
                          'weights': weights})
```

By defining pca__weights, we have overridden the property weights for pca. On all cases, the property pca__weights will be send to pca and logistic.

Overriding the routing scheme can be subtle and you must remember the priority of application of each route types:

1. Routes applied specifically to a function/estimator: {'pca__weights': weights}}
2. Routes defined globally: {'weights': weights}

Let's consider the following code to familiarized yourself with the different routes definitions :

```python
import numpy as np
from sklearn import datasets
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import cross_val_score, GridSearchCV, LeaveOneLabelOut
```

(continues on next page)

(continued from previous page)

```
digits = datasets.load_digits()
X = digits.data
y = digits.target

# Define the groups used by cross_val_score
cv_groups = np.random.randint(3, size=y.shape)

# Define the groups used by GridSearchCV
gs_groups = np.random.randint(3, size=y.shape)

# Define weights used by cross_val_score
weights = np.random.rand(X.shape[0])
weights /= np.sum(weights)

# We define the GridSearchCV used by cross_val_score
grid = GridSearchCV(SGDClassifier(), params, cv=LeaveOneLabelOut())

# When cross_val_score is called, we send all parameters for internal values
cross_val_score(grid, X, y, cv=LeaveOneLabelOut(),
                sample_props={'cv__groups': groups,
                              'split__groups': gs_groups,
                              'weights': weights})
```

With this code, the `sample_props` sent to each function of `GridSearchCV` and `cross_val_score` will be:

function	sample_props
grid.fit	{'weights': weights, 'cv__groups': cv_groups, split_groups ': gs_groups}
grid.score	{'weights': weights, 'cv__groups': cv_groups, split_groups ': gs_groups}
grid.split	{'weights': weights, 'groups': gs_groups, 'cv__groups': cv_groups, split_groups': gs_groups}
cross_val_score	{'weights': weights, 'groups': groups, 'cv__groups': cv_groups, split_groups': gs_groups}

Thus, these functions receive as `weights` and `groups` properties :

function	weights	groups
grid.fit	weights	None
grid.score	weights	None
grid.split	weights	gs_groups
cross_val_score	weights	cv_groups

9.4 4. Alternative propositions for sample_props (06.17.17)

The meta-estimator says which columns of sample_props they wanted to use.

```
p = make_pipeline(
     PCA(n_components=10),
     SVC(C=10).with(<method>_<thing_the_method_knows>=<column_name>)
```

(continues on next page)

(continued from previous page)

```
)
p.fit(X, y, sample_props={column_name=value})
```

For example :

```
p = make_pipeline(
    PCA(n_components=10),
    SVC(C=10).with(fit_weights='weights', score_weights='weights')
)
p.fit(X, y, sample_props={"weights": w})
```

Other proposals: - Olivier suggests to modify `.with(...)` by `.sample_props_mapping(...)`. - Gael suggests to change the `.with(...)` by a property `with_props=...` like :

```
p = make_pipeline(
    PCA(n_components=10),
    SVC(C=10),
    with_props={
        'svc':(<method>_<thing_the_method_knows>=<column_name>)}
)
```

9.4.1 4.1 GridSearch + Pipeline case

Let's consider the case of a `GridSearch` working with a `Pipeline`. How we definer the `sample_props` on that case ?

Alternative 1

Pass through everything in `GridSearchCV`:

```
pipe = make_pipeline(
    PCA(), SVC(),
    with_props={pca__fit_weight: 'my_weights'}})
GridSearchCV(
    pipe, cv=my_cv,
    with_props={'cv__groups': "my_groups", '*':'*')
```

A more complex example with this solution:

```
pipe = make_pipeline(
    make_union(
        CountVectorizer(analyzer='word').with(fit_weight='my_weight'),
        CountVectorizer(analyzer='char').with(fit_weight='my_weight')),
    SVC())

GridSearchCV(
    pipe,
    cv=my_cv.with(groups='my_groups'), score_weight='my_weight')
```

Alternative 2

Grid search manage the `sample_props` of all internal variable.

```
pipe = make_pipeline(PCA(), SVC())
GridSearchCV(
    pipe, cv=my_cv,
    with_props={
        'cv__groups': "my_groups",
        'estimator__pca__fit_weight': "my_weights"),
        })
```

A more complex example with this solution:

```
pipe = make_pipeline(
    make_union(
        CountVectorizer(analyzer='word'),
        CountVectorizer(analyzer='char')),
    SVC())
GridSearchCV(
    pipe, cv=my_cv,
    with_props={
        'cv__groups': "my_groups",
        'estimator__featureunion__countvectorizer-1__fit_weight': "my_weights",
        'estimator__featureunion__countvectorizer-2__fit_weight': "my_weights",
        'score_weight': "my_weights",
    }
)
```

CHAPTER 10

SLEP006: Routing sample-aligned meta-data

Author Joel Nothman

Status Draft

Type Standards Track

Created 2019-03-07

Scikit-learn has limited support for information pertaining to each sample (henceforth “sample properties”) to be passed through an estimation pipeline. The user can, for instance, pass fit parameters to all members of a FeatureUnion, or to a specified member of a Pipeline using dunder (__) prefixing:

```
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.linear_model import LogisticRegression
>>> pipe = Pipeline([('clf', LogisticRegression())])
>>> pipe.fit([[1, 2], [3, 4]], [5, 6],
...          clf__sample_weight=[.5, .7])
```

Several other meta-estimators, such as GridSearchCV, support forwarding these fit parameters to their base estimator when fitting.

Desirable features we do not currently support include:

- passing sample properties (e.g. `sample_weight`) to a scorer used in cross-validation
- passing sample properties (e.g. `groups`) to a CV splitter in nested cross validation
- (maybe in scope) passing sample properties (e.g. `sample_weight`) to some scorers and not others in a multi-metric cross-validation setup
- (likely out of scope) passing sample properties to non-fit methods, for instance to index grouped samples that are to be treated as a single sequence in prediction.

10.1 Definitions

consumer An estimator, scorer, splitter, etc., that receives and can make use of one or more passed props.

key A label passed along with sample prop data to indicate how it should be interpreted (e.g. "weight").

router An estimator or function that passes props on to some other router or consumer, potentially selecting which props to pass to which destination, and by what key.

10.2 History

This version was drafted after a discussion of the issue and potential solutions at the February 2019 development sprint in Paris.

Supersedes SLEP004 with greater depth of desiderata and options.

Primary related issues and pull requests include:

- #4497: Overarching issue, "Consistent API for attaching properties to samples" by @GaelVaroquaux
- #4696 A first implementation by @amueller
- Discussion towards SLEP004 initiated by @tguillemot
- #9566 Another implementation (solution 3 from this SLEP) by @jnothman
- #16079 Another implementation (solution 4 from this SLEP) by @adrinjalali

Other related issues include: #1574, #2630, #3524, #4632, #4652, #4660, #4696, #6322, #7112, #7646, #7723, #8127, #8158, #8710, #8950, #11429, #12052, #15282, **:issues:‘15370‘**, #15425, #18028.

10.3 Desiderata

We will consider the following aspects to develop and compare solutions:

Usability Can the use cases be achieved in succinct, readable code? Can common use cases be achieved with a simple recipe copy-pasted from a QA forum?

Brittleness If a property is being routed through a Pipeline, does changing the structure of the pipeline (e.g. adding a layer of nesting) require rewriting other code?

Error handling If the user mistypes the name of a sample property, or misspecifies how it should be routed to a consumer, will an appropriate exception be raised?

Impact on meta-estimator design How much meta-estimator code needs to change? How hard will it be to maintain?

Impact on estimator design How much will the proposal affect estimator developers?

Backwards compatibility Can existing behavior be maintained?

Forwards compatibility Is the solution going to make users' code more brittle with future changes? (For example, will a user's pipeline change behaviour radically when sample_weight is implemented on some estimator)

Introspection If sensible to do so (e.g. for improved efficiency), can a meta-estimator identify whether its base estimator (recursively) would handle some particular sample property (e.g. so a meta-estimator can choose between weighting and resampling, or for automated invariance testing)?

10.4 Keyword arguments vs. a single argument

Currently, sample properties are provided as keyword arguments to a `fit` method. In redeveloping sample properties, we can instead accept a single parameter (named `props` or `sample_props`, for example) which maps string keys to arrays of the same length (a "DataFrame-like").

Keyword arguments:

```
>>> gs.fit(X, y, groups=groups, sample_weight=sample_weight)
```

Single argument:

```
>>> gs.fit(X, y, props={'groups': groups, 'weight': weight})
```

While drafting this document, we will assume the latter notation for clarity.

Advantages of multiple keyword arguments:

- succinct
- possible to maintain backwards compatible support for sample_weight, etc.
- we do not need to handle cases for whether or not some estimator expects a `props` argument.

Advantages of a single argument:

- we are able to consider kwargs to `fit` that are not sample-aligned, so that we can add further functionality (some that have been proposed: `with_warm_start`, `feature_names_in`, `feature_meta`).
- we are able to redefine the default routing of weights etc. without being concerned by backwards compatibility.
- we can consider the use of keys that are not limited to strings or valid identifiers (and hence are not limited to using _ as a delimiter).

10.5 Test case setup

10.5.1 Case A

Cross-validate a `LogisticRegressionCV(cv=GroupKFold(), scoring='accuracy')` with weighted scoring and weighted fitting.

Error handling: what would happen if the user misspelled `sample_weight` as `sample_eight`?

10.5.2 Case B

Cross-validate a `LogisticRegressionCV(cv=GroupKFold(), scoring='accuracy')` with weighted scoring and unweighted fitting.

10.5.3 Case C

Extend Case A to apply an unweighted univariate feature selector in a `Pipeline`.

10.5.4 Case D

Different weights for scoring and for fitting in Case A.

TODO: case involving props passed at test time, e.g. to pipe.transform (???). TODO: case involving score() method, e.g. not specifying scoring in cross_val_score when wrapping an estimator with weighted score func …

Solution sketches will import these definitions:

```
import numpy as np
from sklearn.feature_selection import SelectKBest
from sklearn.linear_model import LogisticRegressionCV
from sklearn.metrics import accuracy
from sklearn.metrics import make_scorer
from sklearn.model_selection import GroupKFold, cross_validate
from sklearn.pipeline import make_pipeline

N, M = 100, 4
X = np.random.rand(N, M)
y = np.random.randint(0, 1, size=N)
my_groups = np.random.randint(0, 10, size=N)
my_weights = np.random.rand(N)
my_other_weights = np.random.rand(N)
```

10.6 Status quo solution 0a: additional feature

Without changing scikit-learn, the following hack can be used:

Additional numeric features representing sample props can be appended to the data and passed around, being handled specially in each consumer of features or sample props.

```
import numpy as np

from defs import (accuracy, group_cv, get_scorer, SelectKBest,
                  LogisticRegressionCV, cross_validate,
                  make_pipeline, X, y, my_groups, my_weights,
                  my_other_weights)

# %%
# Case A: weighted scoring and fitting


GROUPS_IDX = -1
WEIGHT_IDX = -2


def unwrap_X(X):
    return X[:, -2:]


class WrappedGroupCV:
    def __init__(self, base_cv, groups_idx=GROUPS_IDX):
        self.base_cv = base_cv
        self.groups_idx = groups_idx

    def split(self, X, y, groups=None):
```

(continues on next page)

(continued from previous page)

```
        groups = X[:, self.groups_idx]
        return self.base_cv.split(unwrap_X(X), y, groups=groups)

    def get_n_splits(self, X, y, groups=None):
        groups = X[:, self.groups_idx]
        return self.base_cv.get_n_splits(unwrap_X(X), y, groups=groups)


wrapped_group_cv = WrappedGroupCV(group_cv)


class WrappedLogisticRegressionCV(LogisticRegressionCV):
    def fit(self, X, y):
        return super().fit(unwrap_X(X), y, sample_weight=X[:, WEIGHT_IDX])


acc_scorer = get_scorer('accuracy')


def wrapped_weighted_acc(est, X, y, sample_weight=None):
    return acc_scorer(est, unwrap_X(X), y, sample_weight=X[:, WEIGHT_IDX])


lr = WrappedLogisticRegressionCV(
    cv=wrapped_group_cv,
    scoring=wrapped_weighted_acc,
).set_props_request(['sample_weight'])
cross_validate(lr, np.hstack([X, my_weights, my_groups]), y,
               cv=wrapped_group_cv,
               scoring=wrapped_weighted_acc)

# %%
# Case B: weighted scoring and unweighted fitting

class UnweightedWrappedLogisticRegressionCV(LogisticRegressionCV):
    def fit(self, X, y):
        return super().fit(unwrap_X(X), y)


lr = UnweightedWrappedLogisticRegressionCV(
    cv=wrapped_group_cv,
    scoring=wrapped_weighted_acc,
).set_props_request(['sample_weight'])
cross_validate(lr, np.hstack([X, my_weights, my_groups]), y,
               cv=wrapped_group_cv,
               scoring=wrapped_weighted_acc)


# %%
# Case C: unweighted feature selection

class UnweightedWrappedSelectKBest(SelectKBest):
    def fit(self, X, y):
        return super().fit(unwrap_X(X), y)


lr = WrappedLogisticRegressionCV(
```

(continues on next page)

(continued from previous page)

```
    cv=wrapped_group_cv,
    scoring=wrapped_weighted_acc,
).set_props_request(['sample_weight'])
sel = UnweightedWrappedSelectKBest()
pipe = make_pipeline(sel, lr)
cross_validate(pipe, np.hstack([X, my_weights, my_groups]), y,
               cv=wrapped_group_cv,
               scoring=wrapped_weighted_acc)

# %%
# Case D: different scoring and fitting weights

SCORING_WEIGHT_IDX = -3

# TODO: proceed from here. Note that this change implies the need to add
# a parameter to unwrap_X, since we will now append an additional column to X.
```

10.7 Status quo solution 0b: Pandas Index and global resources

Without changing scikit-learn, the following hack can be used:

If y is represented with a Pandas datatype, then its index can be used to access required elements from props stored in a global namespace (or otherwise made available to the estimator before fitting). This is possible everywhere that a ground-truth y is passed, including fit, split, score, and metrics. A similar solution with X is also possible (except for metrics), if all Pipeline components retain the original Pandas Index.

Issues:

- use of global data source
- requires Pandas data types and indices to be maintained

```
import pandas as pd
from defs import (accuracy, group_cv, get_scorer, SelectKBest,
                  LogisticRegressionCV, cross_validate,
                  make_pipeline, X, y, my_groups, my_weights,
                  my_other_weights)

X = pd.DataFrame(X)
MY_GROUPS = pd.Series(my_groups)
MY_WEIGHTS = pd.Series(my_weights)
MY_OTHER_WEIGHTS = pd.Series(my_other_weights)

# %%
# Case A: weighted scoring and fitting


class WrappedGroupCV:
    def __init__(self, base_cv):
        self.base_cv = base_cv

    def split(self, X, y, groups=None):
        return self.base_cv.split(X, y, groups=MY_GROUPS.loc[X.index])

    def get_n_splits(self, X, y, groups=None):
```

(continues on next page)

(continued from previous page)

```
        return self.base_cv.get_n_splits(X, y, groups=MY_GROUPS.loc[X.index])


wrapped_group_cv = WrappedGroupCV(group_cv)


class WeightedLogisticRegressionCV(LogisticRegressionCV):
    def fit(self, X, y):
        return super().fit(X, y, sample_weight=MY_WEIGHTS.loc[X.index])


acc_scorer = get_scorer('accuracy')


def wrapped_weighted_acc(est, X, y, sample_weight=None):
    return acc_scorer(est, X, y, sample_weight=MY_WEIGHTS.loc[X.index])


lr = WeightedLogisticRegressionCV(
    cv=wrapped_group_cv,
    scoring=wrapped_weighted_acc,
).set_props_request(['sample_weight'])
cross_validate(lr, X, y,
               cv=wrapped_group_cv,
               scoring=wrapped_weighted_acc)

# %%
# Case B: weighted scoring and unweighted fitting

lr = LogisticRegressionCV(
    cv=wrapped_group_cv,
    scoring=wrapped_weighted_acc,
).set_props_request(['sample_weight'])
cross_validate(lr, X, y,
               cv=wrapped_group_cv,
               scoring=wrapped_weighted_acc)


# %%
# Case C: unweighted feature selection

lr = WeightedLogisticRegressionCV(
    cv=wrapped_group_cv,
    scoring=wrapped_weighted_acc,
).set_props_request(['sample_weight'])
sel = SelectKBest()
pipe = make_pipeline(sel, lr)
cross_validate(pipe, X, y,
               cv=wrapped_group_cv,
               scoring=wrapped_weighted_acc)

# %%
# Case D: different scoring and fitting weights


def other_weighted_acc(est, X, y, sample_weight=None):
    return acc_scorer(est, X, y, sample_weight=MY_OTHER_WEIGHTS.loc[X.index])
```

(continues on next page)

(continued from previous page)

```
lr = WeightedLogisticRegressionCV(
    cv=wrapped_group_cv,
    scoring=other_weighted_acc,
).set_props_request(['sample_weight'])
sel = SelectKBest()
pipe = make_pipeline(sel, lr)
cross_validate(pipe, X, y,
               cv=wrapped_group_cv,
               scoring=other_weighted_acc)
```

10.8 Solution 1: Pass everything

This proposal passes all props to all consumers (estimators, splitters, scorers, etc). The consumer would optionally use props it is familiar with by name and disregard other props.

We may consider providing syntax for the user to control the interpretation of incoming props:

- to require that some prop is provided (for an estimator where that prop is otherwise optional)
- to disregard some provided prop
- to treat a particular prop key as having a certain meaning (e.g. locally interpreting 'scoring_sample_weight' as 'sample_weight').

These constraints would be checked by calling a helper at the consumer.

Issues:

- Error handling: if a key is optional in a consumer, no error will be raised for misspelling. An introspection API might change this, allowing a user or meta-estimator to check if all keys passed are to be used in at least one consumer.
- Forwards compatibility: newly supporting a prop key in a consumer will change behaviour. Other than a ChangedBehaviorWarning, I don't see any way around this.
- Introspection: not inherently supported. Would need an API like `get_prop_support(names: List[str]) -> Dict[str, Literal["supported", "required", "ignored"]]`.

In short, this is a simple solution, but prone to risk.

```
from defs import (accuracy, group_cv, make_scorer, SelectKBest,
                  LogisticRegressionCV, cross_validate, make_pipeline, X, y,
                  my_groups, my_weights, my_other_weights)

# %%
# Case A: weighted scoring and fitting

lr = LogisticRegressionCV(
    cv=group_cv,
    scoring='accuracy',
)
cross_validate(lr, X, y, cv=group_cv,
               props={'sample_weight': my_weights, 'groups': my_groups},
               scoring='accuracy')
```

(continues on next page)

(continued from previous page)

```
# Error handling: if props={'sample_eight': my_weights, ...} was passed
# instead, the estimator would fit and score without weight, silently failing.

# %%
# Case B: weighted scoring and unweighted fitting


class MyLogisticRegressionCV(LogisticRegressionCV):
    def fit(self, X, y, props=None):
        props = props.copy()
        props.pop('sample_weight', None)
        super().fit(X, y, props=props)


# %%
# Case C: unweighted feature selection

# Currently feature selection does not handle sample_weight, and as long as
# that remains the case, it will simply ignore the prop passed to it. Hence:

lr = LogisticRegressionCV(
    cv=group_cv,
    scoring='accuracy',
)
sel = SelectKBest()
pipe = make_pipeline(sel, lr)
cross_validate(pipe, X, y, cv=group_cv,
               props={'sample_weight': my_weights, 'groups': my_groups},
               scoring='accuracy')

# %%
# Case D: different scoring and fitting weights

weighted_acc = make_scorer(accuracy)


def specially_weighted_acc(est, X, y, props):
    props = props.copy()
    props['sample_weight'] = 'scoring_weight'
    return weighted_acc(est, X, y, props)


lr = LogisticRegressionCV(
    cv=group_cv,
    scoring=specially_weighted_acc,
)
cross_validate(lr, X, y, cv=group_cv,
               props={
                    'scoring_weight': my_weights,
                    'sample_weight': my_other_weights,
                    'groups': my_groups,
               },
               scoring=specially_weighted_acc)
```

10.9 Solution 2: Specify routes at call

Similar to the legacy behavior of fit parameters in `sklearn.pipeline.Pipeline`, this requires the user to specify the path for each "prop" to follow when calling `fit`. For example, to pass a prop named 'weights' to a step named 'spam' in a Pipeline, you might use `my_pipe.fit(X, y, props={'spam__weights': my_weights})`.

SLEP004's syntax to override the common routing scheme falls under this solution.

Advantages:

- Very explicit and robust to misspellings.

Issues:

- The user needs to know the nested internal structure, or it is easy to fail to pass a prop to a specific estimator.
- A corollary is that prop keys need changing when the developer modifies their estimator structure (see case C).
- This gets especially tricky or impossible where the available routes change mid-fit, such as where a grid search considers estimators with different structures.
- We would need to find a different solution for #2630 where a Pipeline could not be the base estimator of AdaBoost because AdaBoost expects the base estimator to accept a fit param keyed 'sample_weight'.
- This may not work if a meta-estimator were to have the role of changing a prop, e.g. a meta-estimator that passes `sample_weight` corresponding to balanced classes onto its base estimator. The meta-estimator would need a list of destinations to pass modified props to, or a list of keys to modify.
- We would need to develop naming conventions for different routes, which may be more complicated than the current conventions; while a GridSearchCV wrapping a Pipeline currently takes parameters with keys like `{step_name}__{prop_name}`, this explicit routing, and conflict with GridSearchCV routing destinations, implies keys like `estimator__{step_name}__{prop_name}`.

```python
from defs import (group_cv, SelectKBest, LogisticRegressionCV,
                  cross_validate, make_pipeline, X, y, my_groups,
                  my_weights, my_other_weights)

# %%
# Case A: weighted scoring and fitting

lr = LogisticRegressionCV(
    cv=group_cv,
    scoring='accuracy',
)
props = {'cv__groups': my_groups,
         'estimator__cv__groups': my_groups,
         'estimator__sample_weight': my_weights,
         'scoring__sample_weight': my_weights,
         'estimator__scoring__sample_weight': my_weights}
cross_validate(lr, X, y, cv=group_cv,
               props=props,
               scoring='accuracy')

# error handling: if props={'estimator__sample_eight': my_weights, ...} was
# passed instead, the estimator would raise an error.

# %%
# Case B: weighted scoring and unweighted fitting

```

(continues on next page)

(continued from previous page)

```
lr = LogisticRegressionCV(
    cv=group_cv,
    scoring='accuracy',
)
props = {'cv__groups': my_groups,
         'estimator__cv__groups': my_groups,
         'scoring__sample_weight': my_weights,
         'estimator__scoring__sample_weight': my_weights}
cross_validate(lr, X, y, cv=group_cv,
               props=props,
               scoring='accuracy')

# %%
# Case C: unweighted feature selection

lr = LogisticRegressionCV(
    cv=group_cv,
    scoring='accuracy',
)
pipe = make_pipeline(SelectKBest(), lr)
props = {'cv__groups': my_groups,
         'estimator__logisticregressioncv__cv__groups': my_groups,
         'estimator__logisticregressioncv__sample_weight': my_weights,
         'scoring__sample_weight': my_weights,
         'estimator__scoring__sample_weight': my_weights}
cross_validate(pipe, X, y, cv=group_cv,
               props=props,
               scoring='accuracy')

# %%
# Case D: different scoring and fitting weights

lr = LogisticRegressionCV(
    cv=group_cv,
    scoring='accuracy',
)
props = {'cv__groups': my_groups,
         'estimator__cv__groups': my_groups,
         'estimator__sample_weight': my_other_weights,
         'scoring__sample_weight': my_weights,
         'estimator__scoring__sample_weight': my_weights}
cross_validate(lr, X, y, cv=group_cv,
               props=props,
               scoring='accuracy')
```

10.10 Solution 3: Specify routes on metaestimators

Each meta-estimator is given a routing specification which it must follow in passing only the required parameters to each of its children. In this context, a GridSearchCV has children including `estimator`, `cv` and (each element of) `scoring`.

Pull request #9566 and its extension in #15425 are partial implementations of this approach.

A major benefit of this approach is that it may allow only prop routing meta-estimators to be modified, not prop consumers.

All consumers would be required to check that

Issues:

- Routing may be hard to get one's head around, especially since the prop support belongs to the child estimator but the parent is responsible for the routing.
- Need to design an API for specifying routings.
- As in Solution 2, each local destination for routing props needs to be given a name.
- Every router along the route will need consistent instructions to pass a specific prop to a consumer. If the prop is optional in the consumer, routing failures may be hard to identify and debug.
- For estimators to be cloned, this routing information needs to be cloned with it. This implies one of: the routing information be stored as a constructor parameter; or `clone` is extended to explicitly copy routing information.

Possible public syntax:

Each meta-estimator has a `prop_routing` parameter to encode local routing rules, and a set of named children which it routes to. In #9566, the `prop_routing` entry for each child may be a white list or black list of named keys passed to the meta-estimator.

```
from defs import (accuracy, make_scorer, SelectKBest, LogisticRegressionCV,
                  group_cv, cross_validate, make_pipeline, X, y, my_groups,
                  my_weights, my_other_weights)

# %%
# Case A: weighted scoring and fitting

lr = LogisticRegressionCV(
    cv=group_cv,
    scoring='accuracy',
    prop_routing={'cv': ['groups'],
                  'scoring': ['sample_weight'],
                  }
    # one question here is whether we need to explicitly route sample_weight
    # to LogisticRegressionCV's fitting...
)

# Alternative syntax, which assumes cv receives 'groups' by default, and that a
# method-based API is provided on meta-estimators:
#   lr = LogisticRegressionCV(
#       cv=group_cv,
#       scoring='accuracy',
#   ).add_prop_route(scoring='sample_weight')

cross_validate(lr, X, y, cv=group_cv,
               props={'sample_weight': my_weights, 'groups': my_groups},
               scoring='accuracy',
               prop_routing={'estimator': '*',  # pass all props
                             'cv': ['groups'],
                             'scoring': ['sample_weight'],
                             })

# Error handling: if props={'sample_eight': my_weights, ...} was passed
# instead, LogisticRegressionCV would have to identify that a key was passed
# that could not be routed nor used, in order to raise an error.

# %%
```

(continues on next page)

(continued from previous page)

```
# Case B: weighted scoring and unweighted fitting

# Here we rename the sample_weight prop so that we can specify that it only
# applies to scoring.
lr = LogisticRegressionCV(
    cv=group_cv,
    scoring='accuracy',
    prop_routing={'cv': ['groups'],
                  # read the following as "scoring should consume
                  # 'scoring_weight' as if it were 'sample_weight'."
                  'scoring': {'sample_weight': 'scoring_weight'},
                  },
)
cross_validate(lr, X, y, cv=group_cv,
               props={'scoring_weight': my_weights, 'groups': my_groups},
               scoring='accuracy',
               prop_routing={'estimator': '*',
                             'cv': ['groups'],
                             'scoring': {'sample_weight': 'scoring_weight'},
                             })

# %%
# Case C: unweighted feature selection

lr = LogisticRegressionCV(
    cv=group_cv,
    scoring='accuracy',
    prop_routing={'cv': ['groups'],
                  'scoring': ['sample_weight'],
                  })
pipe = make_pipeline(SelectKBest(), lr,
                     prop_routing={'logisticregressioncv': ['sample_weight',
                                                            'groups']})
cross_validate(lr, X, y, cv=group_cv,
               props={'sample_weight': my_weights, 'groups': my_groups},
               scoring='accuracy',
               prop_routing={'estimator': '*',
                             'cv': ['groups'],
                             'scoring': ['sample_weight'],
                             })

# %%
# Case D: different scoring and fitting weights
lr = LogisticRegressionCV(
    cv=group_cv,
    scoring='accuracy',
    prop_routing={'cv': ['groups'],
                  # read the following as "scoring should consume
                  # 'scoring_weight' as if it were 'sample_weight'."
                  'scoring': {'sample_weight': 'scoring_weight'},
                  },
)
cross_validate(lr, X, y, cv=group_cv,
               props={'scoring_weight': my_weights, 'groups': my_groups,
                      'fitting_weight': my_other_weights},
               scoring='accuracy',
               prop_routing={'estimator': {'sample_weight': 'fitting_weight',
```

(continues on next page)

(continued from previous page)

```
                                            'scoring_weight': 'scoring_weight',
                                            'groups': 'groups'},
                              'cv': ['groups'],
                              'scoring': {'sample_weight': 'scoring_weight'},
                              })
```

10.11 Solution 4: Each child requests

Here the meta-estimator provides only what each of its children requests. The meta-estimator would also need to request, on behalf of its children, any prop that descendant consumers require.

Each object that could receive props would have a method like `get_prop_request()` which would return a list of prop names (or perhaps a mapping for more sophisticated use-cases). Group* CV splitters would default to returning `['groups']`, for example. Estimators supporting weighted fitting may return `[]` by default, but may have a parameter `request_props` which may be set to `['weight']` if weight is sought, or perhaps just boolean parameter `request_weight`. `make_scorer` would have a similar mechanism for enabling weighted scoring.

Advantages:

- This will not need to affect legacy estimators, since no props will be passed when a props request is not available.
- This does not require defining a new syntax for routing.
- The implementation changes in meta-estimators may be easy to provide via a helper or two (perhaps even `call_with_props(method, target, props)`).
- Easy to reconfigure what props an estimator gets in a grid search.
- Could make use of existing `**fit_params` syntax rather than introducing new `props` argument to `fit`.

Disadvantages:

- This will require modifying every estimator that may want props, as well as all meta-estimators. We could provide a mixin or similar to add prop-request support to a legacy estimator; or `BaseEstimator` could have a `set_props_request` method (instead of the `request_props` constructor parameter approach) such that all legacy base estimators are automatically equipped.
- Aliasing is a bit confusing in this design, in that the consumer still accepts the fit param by its original name (e.g. `sample_weight`) even if it has a request that specifies a different key given to the router (e.g. `fit_sample_weight`). This design has the advantage that the handling of props within a consumer is simple and unchanged; the complexity is in how it is forwarded the data by the router, but it may be conceptually difficult for users to understand. (This may be acceptable, as an advanced feature.)
- For estimators to be cloned, this request information needs to be cloned with it. This implies one of: the request information be stored as a constructor parameter; or `clone` is extended to explicitly copy request information.

Possible public syntax:

- `BaseEstimator` will have methods `set_props_request` and `get_props_request`
- `make_scorer` will have a `request_props` parameter to set props required by the scorer.
- `get_props_request` will return a dict. It maps the key that the user passes to the key that the estimator expects.
- `set_props_request` will accept either such a dict or a sequence `s` to be interpreted as the identity mapping for all elements in `s` (`{x:  x for x in s}`). It will return `self` to enable chaining.

- `Group*` CV splitters will by default request the 'groups' prop, but its mapping can be changed with their `set_props_request` method.

Test cases:

```python
from defs import (accuracy, group_cv, make_scorer, SelectKBest,
                  LogisticRegressionCV, cross_validate,
                  make_pipeline, X, y, my_groups, my_weights,
                  my_other_weights)

# %%
# Case A: weighted scoring and fitting

# Here we presume that GroupKFold requests `groups` by default.
# We need to explicitly request weights in make_scorer and for
# LogisticRegressionCV. Both of these consumers understand the meaning
# of the key "sample_weight".

weighted_acc = make_scorer(accuracy, request_props=['sample_weight'])
lr = LogisticRegressionCV(
    cv=group_cv,
    scoring=weighted_acc,
).set_props_request(['sample_weight'])
cross_validate(lr, X, y, cv=group_cv,
               props={'sample_weight': my_weights, 'groups': my_groups},
               scoring=weighted_acc)

# Error handling: if props={'sample_eight': my_weights, ...} was passed,
# cross_validate would raise an error, since 'sample_eight' was not requested
# by any of its children.

# %%
# Case B: weighted scoring and unweighted fitting

# Since LogisticRegressionCV requires that weights explicitly be requested,
# removing that request means the fitting is unweighted.

weighted_acc = make_scorer(accuracy, request_props=['sample_weight'])
lr = LogisticRegressionCV(
    cv=group_cv,
    scoring=weighted_acc,
)
cross_validate(lr, X, y, cv=group_cv,
               props={'sample_weight': my_weights, 'groups': my_groups},
               scoring=weighted_acc)

# %%
# Case C: unweighted feature selection

# Like LogisticRegressionCV, SelectKBest needs to request weights explicitly.
# Here it does not request them.

weighted_acc = make_scorer(accuracy, request_props=['sample_weight'])
lr = LogisticRegressionCV(
    cv=group_cv,
    scoring=weighted_acc,
).set_props_request(['sample_weight'])
sel = SelectKBest()
```

(continues on next page)

(continued from previous page)

```
pipe = make_pipeline(sel, lr)
cross_validate(pipe, X, y, cv=group_cv,
               props={'sample_weight': my_weights, 'groups': my_groups},
               scoring=weighted_acc)

# %%
# Case D: different scoring and fitting weights

# Despite make_scorer and LogisticRegressionCV both expecting a key
# sample_weight, we can use aliases to pass different weights to different
# consumers.

weighted_acc = make_scorer(accuracy,
                           request_props={'scoring_weight': 'sample_weight'})
lr = LogisticRegressionCV(
    cv=group_cv,
    scoring=weighted_acc,
).set_props_request({'fitting_weight': "sample_weight"})
cross_validate(lr, X, y, cv=group_cv,
               props={
                    'scoring_weight': my_weights,
                    'fitting_weight': my_other_weights,
                    'groups': my_groups,
               },
               scoring=weighted_acc)
```

Extensions and alternatives to the syntax considered while working on #16079:

- `set_prop_request` and `get_props_request` have lists of props requested **for each method** i.e. fit, score, transform, predict and perhaps others.
- `set_props_request` could be replaced by a method (or parameter) representing the routing of each prop that it consumes. For example, an estimator that consumes `sample_weight` would have a `request_sample_weight` method. One of the difficulties of this approach is automatically introducing `request_sample_weight` into classes inheriting from BaseEstimator without too much magic (e.g. metaclasses, which might be the simplest solution).

These are demonstrated together in the following:

```
from defs import (accuracy, group_cv, make_scorer, SelectKBest,
                  LogisticRegressionCV, cross_validate,
                  make_pipeline, X, y, my_groups, my_weights,
                  my_other_weights)

# %%
# Case A: weighted scoring and fitting

# Here we presume that GroupKFold requests `groups` by default.
# We need to explicitly request weights in make_scorer and for
# LogisticRegressionCV. Both of these consumers understand the meaning
# of the key "sample_weight".

weighted_acc = make_scorer(accuracy, request_props=['sample_weight'])
lr = LogisticRegressionCV(
    cv=group_cv,
    scoring=weighted_acc,
).request_sample_weight(fit=['sample_weight'])
```

(continues on next page)

(continued from previous page)

```
cross_validate(lr, X, y, cv=group_cv,
               props={'sample_weight': my_weights, 'groups': my_groups},
               scoring=weighted_acc)

# Error handling: if props={'sample_eight': my_weights, ...} was passed,
# cross_validate would raise an error, since 'sample_eight' was not requested
# by any of its children.

# %%
# Case B: weighted scoring and unweighted fitting

# Since LogisticRegressionCV requires that weights explicitly be requested,
# removing that request means the fitting is unweighted.

weighted_acc = make_scorer(accuracy, request_props=['sample_weight'])
lr = LogisticRegressionCV(
    cv=group_cv,
    scoring=weighted_acc,
)
cross_validate(lr, X, y, cv=group_cv,
               props={'sample_weight': my_weights, 'groups': my_groups},
               scoring=weighted_acc)

# %%
# Case C: unweighted feature selection

# Like LogisticRegressionCV, SelectKBest needs to request weights explicitly.
# Here it does not request them.

weighted_acc = make_scorer(accuracy, request_props=['sample_weight'])
lr = LogisticRegressionCV(
    cv=group_cv,
    scoring=weighted_acc,
).request_sample_weight(fit=['sample_weight'])
sel = SelectKBest()
pipe = make_pipeline(sel, lr)
cross_validate(pipe, X, y, cv=group_cv,
               props={'sample_weight': my_weights, 'groups': my_groups},
               scoring=weighted_acc)

# %%
# Case D: different scoring and fitting weights

# Despite make_scorer and LogisticRegressionCV both expecting a key
# sample_weight, we can use aliases to pass different weights to different
# consumers.

weighted_acc = make_scorer(accuracy,
                           request_props={'scoring_weight': 'sample_weight'})
lr = LogisticRegressionCV(
    cv=group_cv,
    scoring=weighted_acc,
).request_sample_weight(fit='fitting_weight')
cross_validate(lr, X, y, cv=group_cv,
               props={
                    'scoring_weight': my_weights,
                    'fitting_weight': my_other_weights,
```

(continues on next page)

(continued from previous page)

```
                    'groups': my_groups,
                },
                scoring=weighted_acc)
```

10.12 Naming

"Sample props" has become a name understood internally to the Scikit-learn development team. For ongoing usage we have several choices for naming:

- Sample meta
- Sample properties
- Sample props
- Sample extra

10.13 Proposal

Having considered the above solutions, we propose:

- Solution 4 per #16079 which will be used to resolve further, specific details of the solution.
- Props will be known simply as Metadata.
- `**kw` syntax will be used to pass props by key.

TODO:

- if an estimator requests a prop, must it be not-null? Must it be provided or explicitly passed as None?

10.14 Backward compatibility

Under this proposal, consumer behaviour will be backwards compatible, but meta-estimators will change their routing behaviour.

By default, `sample_weight` will not be requested by estimators that support it. This ensures that addition of `sample_weight` support to an estimator will not change its behaviour.

During a deprecation period, fit_params will be handled dually: Keys that are requested will be passed through the new request mechanism, while keys that are not known will be routed using legacy mechanisms. At completion of the deprecation period, the legacy handling will cease.

Similarly, during a deprecation period, `fit_params` in GridSearchCV and related utilities will be routed to the estimator's `fit` by default, per incumbent behaviour. After the deprecation period, an error will be raised for any params not explicitly requested.

Grouped cross validation splitters will request `groups` since they were previously unusable in a nested cross validation context, so this should not often create backwards incompatibilities, except perhaps where a fit param named `groups` served another purpose.

10.15 Discussion

One benefit of the explicitness in Solution 4 is that even if it makes use of `**kw` arguments, it does not preclude keywords arguments serving other purposes in addition. That is, in addition to requesting sample props, a future proposal could allow estimators to request feature metadata or other keys.

10.16 References and Footnotes

10.17 Copyright

This document has been placed in the public domain.[1]

[1] Each SLEP must either be explicitly labeled as placed in the public domain (see this SLEP as an example) or licensed under the Open Publication License.

CHAPTER 11

Rejected SLEPs

Nothing here

CHAPTER 12

SLEP Template and Instructions

Author <list of authors' real names and optionally, email addresses>

Status <Draft | Active | Accepted | Deferred | Rejected | Withdrawn | Final | Superseded>

Type <Standards Track | Process>

Created <date created on, in yyyy-mm-dd format>

Resolution <url> (required for Accepted | Rejected | Withdrawn)

12.1 Abstract

The abstract should be a short description of what the SLEP will achieve.

12.2 Detailed description

This section describes the need for the SLEP. It should describe the existing problem that it is trying to solve and why this SLEP makes the situation better. It should include examples of how the new functionality would be used and perhaps some use cases.

12.3 Implementation

This section lists the major steps required to implement the SLEP. Where possible, it should be noted where one step is dependent on another, and which steps may be optionally omitted. Where it makes sense, each step should include a link related pull requests as the implementation progresses.

Any pull requests or developmt branches containing work on this SLEP should be linked to from here. (A SLEP does not need to be implemented in a single pull request if it makes sense to implement it in discrete phases).

12.4 Backward compatibility

This section describes the ways in which the SLEP breaks backward compatibility.

12.5 Alternatives

If there were any alternative solutions to solving the same problem, they should be discussed here, along with a justification for the chosen approach.

12.6 Discussion

This section may just be a bullet list including links to any discussions regarding the SLEP:

- This includes links to mailing list threads or relevant GitHub issues.

12.7 References and Footnotes

12.8 Copyright

This document has been placed in the public domain.[1]

[1] Each SLEP must either be explicitly labeled as placed in the public domain (see this SLEP as an example) or licensed under the Open Publication License.