Quick tutorial on Sklearn's Pipeline constructor for machine learning

Pipeline-guide.md

If You've Never Used Sklearn's Pipeline Constructor...You're Doing It Wrong

How To Use sklearn Pipelines, FeatureUnions, and GridSearchCV With Your Own Transformers

By Emily Gill and Amber Rivera

What's a Pipeline and Why Use One?

The `Pipeline` constructor from sklearn allows you to chain transformers and estimators together into a sequence that functions as one cohesive unit. For example, if your model involves feature selection, standardization, and then regression, those three steps, each as it's own class, could be encapsulated together via `Pipeline`.

Benefits: readability, reusability and easier experimentation.

- Ensures that each transformation of the data is being performed in the correct order, protects from inadvertent data leakage during cross-validation.
- You can call `.fit` and `.predict` only once despite having a whole sequence of estimators. Convenient.
- You can grid-search once over all parameters of all your transformers and estimators!
- While writing code to search for the best estimator, you're also writing your final pipeline for training, *and* maybe even for predicting on new data once in production!

A Simple Pipeline Example:

All estimators in a pipeline, except for the last one, must be transformers (i.e. they take X, do something to X, and then spit out a transformed X). The final estimator can be another transformer, classifer, regressor, etc.

The following code shows implementation of a pipeline that uses two transformers (`CountVectorizer()` and `TfidfVectorizer`) and one classifier (`LinearSVC`). For this example, assume `X` is a corpus of text from emails and the target (`y`) indicates whether the email was spam (1) or not (0). The data are split into training and test sets. The `.fit` method is called to fit the pipeline on the training data. To predict from the pipeline, one can call `.predict` on the pipeline with the test set or on any new data, X, as long as it has the same features as the original X_train that the model was trained on.

```
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from sklearn.model_selection import train_test_split
from sklearn.svm import LinearSVC

pipe = Pipeline([
  ('vectorize', CountVectorizer()),
  ('tfidf', TfidfVectorizer()),
  ('classify', LinearSVC())
  ])

X_train, X_test, y_train, y_test = train_test_split(X, y)
pipe.fit(X_train, y_train)

y_pred = pipe.predict(X_test)
print(classification_report(y_test, y_pred))
```

Custom Transformers

Often during preprocessing and feature selection, we write our own functions that transform the data (e.g. drop columns, multiply two columns together, etc.). To incorporate those actions into your pipeline, you'll likely need to write your own transformer class.

Follow this template for your own transformer:

```
class MyTransformer(TransformerMixin, BaseEstimator):
    '''A template for a custom transformer.'''

    def __init__(self):
        pass

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        # transform X via code or additional methods
        return X
```

Important things to note in the above template:

- `TransformerMixin` gives your transformer the very useful `.fit_transform` method.
- `BaseEstimator` gives your transformer grid-searchable parameters. This becomes very important later.
- `fit` ALWAYS returns self. Sometimes it can set state variables if you will need those to transform test data later on. Otherwise it just does nothing. Either way, it returns self.
- Even though your `fit` method isn't doing anything with `y`, it still needs to be a parameter. Just set it to `None`. On the flip side, the `transform` method ONLY takes `X` and won't work if you include a `y`.
- `transform` is where most of the transformations happen! In this method, X is transformed somehow (perhaps through other methods within the transformer), and then the transformed X is returned.

For more, see the documentation on `sklearn.preprocessing.FunctionTransformer`, which is basically a wrapper that takes a function and turns it into a class that can then be used within your pipeline.

Using FeatureUnion on Heterogeneous Data

In the above spam example, our `X` was homogeneous in that the columns were all text data. What if we also had numerical or categorical data about the emails that we wanted to include as features, as is often the case? For instance, maybe we also know the domain name (i.e. @domain1.com, @domain2.com, or @domain3.com) and we have an inclination that spam comes from domain3.

So, we write a custom transformer named `MyBinarizer()` that feature engineers a new feature based on whether the email came from domain3 or not. Here's the pseudocode:

- `.fit` returns self.
- `.transform` takes the domain column, creates a new column that is 1 if `X['domain'] == '@domain3'` and 0 if `X['domain'] != '@domain3'`, drops the original domain column, and then returns the transformed X.

The problem is, this feature in either its categorical or binary form cannot be fed through CountVectorizer. **Therefore, it needs to be transformed in parallel with the processing of the text data.** For this, we would make a simple custom transformer that selects the columns that correspond to each parallel pipeline (`MySelector()`), and then use a `FeatureUnion` to apply the appropriate transforms to each type of data, in parallel. After doing the transforms, `FeatureUnion` hstacks the columns back together, before passing X_train (or X_test, or new X data) through the final classifier.

Note that you must select all columns in some way, even if you don't do any transforms on them. Note also that after FeatureUnion, your data will be returned as a NumPy array.

```
from sklearn.pipeline import Pipeline, FeatureUnion

bin_pipe = Pipeline([
  ('select_bin', MySelector(cols=bin_cols)),
  ('binarize', MyBinarizer())
```

```
  ])

text_pipe = Pipeline([
  ('select_text', MySelector(cols=text_cols)),
  ('vectorize', CountVectorizer()),
  ('tfidf', TfidfVectorizer())
  ])

full_pipeline = Pipeline([
    ('feat_union', FeatureUnion(transformer_list=[
          ('text_pipeline', text_pipe),
          ('bin_pipeline', bin_pipe)
          ])),
    ('classify', LinearSVC())
    ])
```

Use GridSearchCV To Identify The Best Estimator And Optimize Over The Entire Pipeline

You've probably used `GridSearchCV` to tune the hyperparameters of your final algorithm. You can do the same thing when using the `Pipeline` constructor - just pass your final pipeline object into GridSearchCV.

By combining `GridSearchCV` with `Pipeline` you can *also* cross-validate and optimize any upstream transforms. For example, this could come in handy if you were doing dimensionality reduction before classifying, and wanted to compare techniques.

Using the spam filtering example from earlier, let's put it all together to find the best of two decomposition techniques, and the best of two classifiers:

```
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from sklearn.decomposition import PCA, NMF
from sklearn.svm import LinearSVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split, GridSearchCV

bin_pipe = Pipeline([
  ('select_bin', MySelector(cols=bin_cols)),
  ('binarize', MyBinarizer())
  ])

text_pipe = Pipeline([
  ('select_text', MySelector(cols=text_cols)),
  ('vectorize', CountVectorizer()),
  ('tfidf', TfidfVectorizer())
  ])

full_pipeline = Pipeline([
    ('feat_union', FeatureUnion(transformer_list=[
          ('text_pipeline', text_pipe),
          ('bin_pipeline', bin_pipe)
          ])),
    ('reduce_dim', PCA())
    ('classify', LinearSVC())
    ])

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

full_pipeline.fit(X_train, y_train)


pg = [
    {
        'reduce_dim': [PCA(iterated_power=7), NMF()],
        'reduce_dim__n_components': [2, 4, 8],
        'classify__C': [1, 10, 100, 1000]
    },
    {
        'classify': [LinearSVC()],
        'classify__penalty': ['l1', 'l2']
    },
    {
```

```
        'classify': [DecisionTreeClassifier()],
        'classify__min_samples_split': [2, 10, 20]
    },
]

grid_search = GridSearchCV(full_pipeline, param_grid=pg, cv=3)

y_pred = full_pipeline.predict(X_test)
print(classification_report(y_test, y_pred))
```

Take a second look at that parameter grid. Two things:

- See how you can try out different methods of the same transform by listing them next to their `Pipeline` step name?
    - Note that different techniques can only share a dictionary within the param_grid when they share hyperparameters.
- The parameters syntax is tricky:
    - Open string
    - *Official name* that you gave the transform step in your Pipeline
    - two underscores
    - Sklearn's name for the parameter (consult the docs for each individual estimator to get all possibilities)
    - close string
    - :
    - List of values to try for the hyperparameter

When you ask for predictions from the `GridSearchCV` object, it automatically returns the predictions from the best model that it tried. If you want to know what the best model and best predictions are, you can explicitly ask for them using methods associated with `GridSearchCV`:

```
print("Best estimator found:")
print(grid_search.best_estimator_)

print("Best score:")
print(grid_search.best_score_)

print("Best parameters found:")
print(grid_search.best_params_)
```

Citations / Resources

Want more? This gist was inspired by these excellent resources:

- Isaac Laughlin and his excellent Pipeline how-to YouTube video
- Pages 67-69 in *"Hands On Machine Learning with Scikit-Learn and TensorFlow"* by Aurélien Géron (O'Reilly). Copyright 2017.
- Sci-Kit Learn's Pipeline and FeatureUnion
- Sci-Kit Learn's Feature Union with Heterogeneous Data Sources
- Michelle Fullwood's github blog on Using Pipelines and FeatureUnions in scikit-learn
- Katie M.'s "Workflows in Python: Using Pipeline and GridSearchCV for More Compact and Comprehensive Code"

sw-yx commented Feb 4, 2019

very useful, thank you!

maikefischer commented Apr 18, 2019

Hey, very very nice example. I am wondering how you would GridSearch over your CustomTransformer (MyBinarizer). You didnt implemnet BaseEstimator yet right?