Scikit-Learn The Best Parts

Isaac Lemon Laughlin

Lead Instructor, Principal Data Scientist

Galvanize Inc. (www.galvanize.com)

@lemonlaug (www.twitter.com/lemonlaug)

Agenda/Objectives

1. What are Pipelines and FeatureUnions?
2. Why should I care?
3. Basic example of how they work.
4. Best practices for writing custom Transformers.
5. Note some of the weaknesses and forthcoming features.

What are Pipelines and FeatureUnions?

- Method for chaining multiple estimators into a single one.
- Estimators might include models, transformations etc.
- `FeatureUnion` takes calls estimators which returns columns in parallel and `np.hstacks` the results together.
- `Pipeline` Applies a sequence of transforms in series.
- They can be used together.

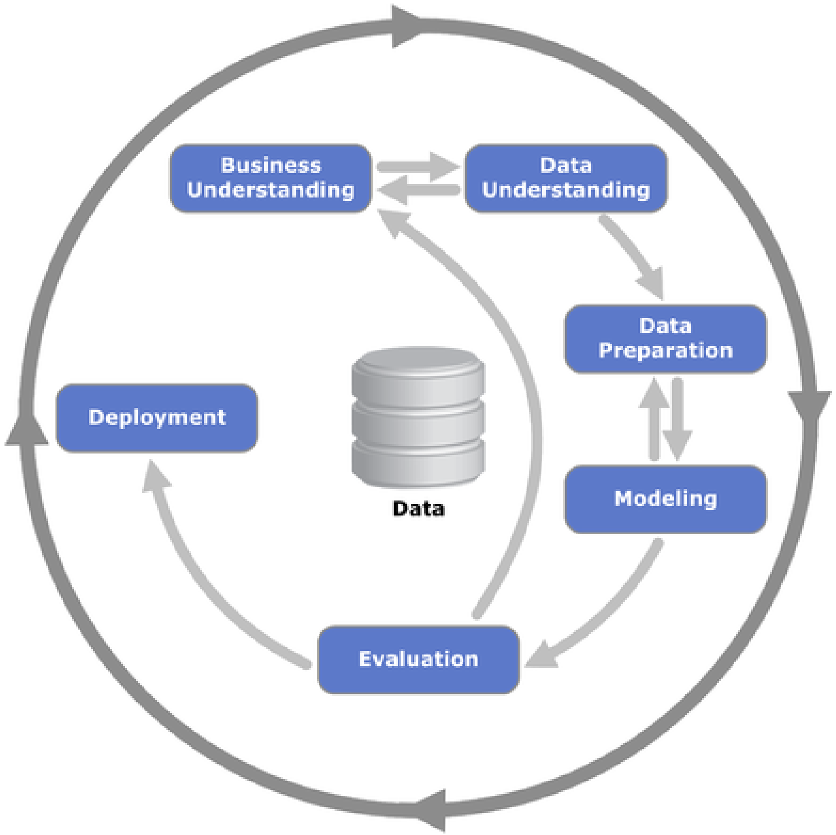
The Best Part of `sklearn`

- Sub-best parts of `sklearn`
  - Supervised learning
  - Unsupervised learning
  - Model selection and evaluation

Why are Pipelines and FeatureUnions so great?

- Encourage good habits like:
  - separation of concerns
    - cross-validation, development/computation
  - avoiding target-leakage by not accepting information about y in `transform`.
  - object orientedness
- Promotes modeling choices to parameters
- Readability
  - Separates implementation details from general approach.
- Efficiency

Why are Pipelines and FeatureUnions so great?



How do they work?

- Initialize with a list of (name, estimator) tuples.
- All but the last of these estimators must implement `transform` method.
- From the docs:

> *Calling fit on the pipeline is the same as calling fit on each estimator in turn, transform the input and pass it on to the next step. The pipeline has all the methods that the last estimator in the pipeline has, i.e. if the last estimator is a classifier, the Pipeline can be used as a classifier. If the last estimator is a transformer, again, so is the pipeline.*

In [6]:
```
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC
from sklearn.decomposition import PCA
#Pipelines are initialized with a list of (name, estimator) tuples.
estimators = [('reduce_dim', PCA()), ('svm', SVC())]
clf = Pipeline(estimators)
clf
```

Out[6]:
```
Pipeline(steps=[('reduce_dim', PCA(copy=True, n_components=None, whiten=False)),
 ('svm', SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False))])
```

In [237]:
```
[x for x in dir(clf) if not x.startswith('_')]
```

Out[237]:
```
['classes_',
 'decision_function',
 'fit',
 'fit_predict',
 'fit_transform',
 'get_params',
 'inverse_transform',
 'named_steps',
 'predict',
 'predict_log_proba',
 'predict_proba',
 'score',
 'set_params',
 'steps',
 'transform']
```

In [260]:

```
from sklearn.grid_search import GridSearchCV
#Grid searching!
params = {'reduce_dim__n_components':[1,5,10,12,15,20],
         'svm__kernel':['linear', 'rbf']}
gs = GridSearchCV(clf, param_grid=params)
#Random data
from sklearn.datasets import make_classification
gs.fit(*make_classification())
gs.best_params_
```

Out[260]:

```
{'reduce_dim__n_components': 12, 'svm__kernel': 'linear'}
```

4. Writing custom transformers

`sklearn` implements lots of good transformers, there are infinitely many more we may want to have so we'll often want to write our own.

```python
from sklearn.base import TransformerMixin, BaseEstimator
class MyTransformer(TransformerMixin, BaseEstimator):
    """Recommended signature for a custom transformer.

    Inheriting from TransformerMixin gives you fit_transform

    Inheriting from BaseEstimator gives you grid-searchable params.
    """
    def __init__(self):
        """If you need to parameterize your transformer,
        set the args here.

        Inheriting from BaseEstimator introduces the constraint
        that the args all be named keyword args, no positional
        args or **kwargs.
        """
        pass
    ...
```

```python
...
    def fit(self, X, y):
        """Recommended signature for custom transformer's
        fit method.

        Set state here with whatever information
        is needed to transform later.

        In some cases fit may do nothing. For example transforming
        degrees Fahrenheit to Kelvin, requires no state.

        You can use y here, but won't have access to it in transform.
        """
        #You have to return self, so we can chain!
        return self
...
```

```
...
    def transform(self, X):
        """Recommended signature for custom transformer's
        transform method.

        Use state (if any) to transform some X data. This X
        may be the same X passed to fit, but it may also be new data,
        as in the case of a CV dataset. Both are treated the same.
        """
        #Do transforms.
        #transformed = foo(X)
        return transformed
```

Practice:

Re-implement StandardScaler using the above stub.

Standardize features by removing the mean and scaling to unit variance:

$$\frac{X - E(X)}{\sigma(X)}$$

Practical hint:

Call your transformer `MyScaler` and save it in `scaler.py` then you can run unittests in tests/test_scaler.py.

Write your transformer from the notebook by:

```
%%writefile scaler.py

class MyScaler:
    ...
```

Then to run the tests from the notebook:

```
!python -m unittest tests.test_scaler
```

In [138]:
```
#Try running some unittests to see if it's working correctly.
!python -m unittest tests.test_scaler
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

In [ ]:

```python
# %load scaler.py
from sklearn.base import TransformerMixin, BaseEstimator
import numpy as np

class MyScaler(TransformerMixin, BaseEstimator):
    """Scale to zero mean and unit variance.
    """
    def fit(self, X, y):
        """Recommended signature for custom transformer's
        fit method.

        Set state in your transformer with whatever information
        is needed to transform later.
        """
        #You have to return self, so we can chain!
        self.mean = np.mean(X, axis=0)
        self.scale = np.std(X, axis=0)
        return self

    def transform(self, X):
        """Recommended signature for custom transformer's
        transform method.

        Use state (if any) to transform some X data. This X
        may be the same X passed to fit, but it may also be new data,
        as in the case of a CV dataset. Both are treated the same.
        """
        #Do transforms.
        Xt = X.copy()
        Xt -= self.mean
        Xt /= self.scale
        return Xt
```

Feature Union

Calls `fit` and `transform` in parallel and `np.hstack`s the output together.

`transformer_weights` can scale the terms in the feature union. Useful for grid searching in regularized settings.

`n_jobs` arg can be used to get parallel computation.

For some complex transformers, alignment may be tricky! Pandas is good at this, but not helpful here because `np.hstack` is called, which ignores indexes.

Writing a generalizable transformer often means you will expect the correct column to be selected from your X matrix, oftentimes this means writing a selector, which is too bad.

In [243]:

```
corpus = ["What is your name?",
          "What is your favorite color?",
          "What is the airspeed velocity of an unladen swallow?"]

fu = FeatureUnion([('tfidf', TfidfVectorizer()),
                  ('counter', WordCounter())])

#Pretty display of the output.
pd.DataFrame(fu.fit_transform(corpus).todense())
```

Out[243]:

	0	1	2	3	4	5	6	7	8
0	0.00000	0.00000	0.00000	0.00000	0.391484	0.66284	0.00000	0.00000	0.00000
1	0.00000	0.00000	0.55249	0.55249	0.326310	0.00000	0.00000	0.00000	0.00000
2	0.36043	0.36043	0.00000	0.00000	0.212876	0.00000	0.36043	0.36043	0.36043

Notes/Direction

Efficiency

Some grid search steps may duplicate a lot of work by fitting/transforming the same data repeatedly. Caching may be forthcoming.

Inverse Transforms

If implemented, can be used.

Post-processing/transformations of y.

Not currently available.

Practice

Putting it all together: creating a matrix of heterogeneous data types.

In [249]:

```
#Data from: https://www.kaggle.com/c/bluebook-for-bulldozers
df = pd.read_csv('data/Train.zip')
df.head(2)
```

Out[249]:

	SalesID	SalePrice	MachineID	ModelID	datasource	auctioneerID	YearMade	Mach
0	1139246	66000	999089	3157	121	3.0	2004	68.0
1	1139248	57000	117657	77	121	3.0	1996	464C

2 rows × 53 columns

Practice

Here are some suggested transformers to use in building your pipeline:

Column	Transformer	Notes
`UsageBand`	`sklearn.preprocessing.OneHotEncoder`	
`YearMade`	`sklearn.preprocessing.Imputer`	May want to also add a dum column noting which rows a affected
`fiProductClassDesc`	`sklearn.preprocessing.CountVectorizer`	You may ultimately want to reduce dimensionality of th using NMF or PCA.
`State`	`sklearn.preprocessing.OneHotEncoder`	
`YearMade`, `SaleDate`	Create a custom transformer to compute the age at sale.	
`SalePrice`	Create a custom transformer that takes the K most recent sales within a ModelID	Beware alignment/target leakage. Use `GroupBy.transform(la x: x.ffill)`. This is delic so feel free to ask for a hint.