

# Putting it all together

Tanya Kutsenko

August 19, 2016

# Objectives

At the end of the day, you'll be able to:

- Use Pipeline to develop more compact and comprehensive code
- Build your own transformer and estimator
- Use FeatureUnion for running parallel processes for features extraction
- Use GridSearchCV for model tuning

We usually start modeling with simple steps like:

- Reading and splitting prepared data into train/test subsets
- Extracting target
- Extracting features
- Scaling & standardizing features
- handling outliers or missing values
- aggregating data to a suitable grain
- Iterate: choose model, fit, predict, print the scores

Then we want more features, and more models, and more tuning. . .

Example: **Classification of text documents using sparse features**

# Why Pipeline

Why use Pipeline instead of keeping the steps separate?

- It makes code more readable
- It keeps data during intermediate steps
- It makes it easy to change the order or add/remove steps
- You only have to call fit and predict once
- Joint parameter selection: grid search over parameters of all estimators at once

Scikit learn provides the 'Pipeline' class to manage data science pipelines:

- Chain multiple estimators into one
- All estimators in a pipeline, except the last one, must be transformers
- The last estimator may be any type (transformer, classifier, etc.)

## Pipeline Example:

```
9
10 # define a pipeline combining a text feature extractor with a simple classifier
11 pipeline = Pipeline([
12     ('vect', CountVectorizer()),
13     ('tfidf', TfidfTransformer()),
14     ('classifier', MultinomialNB()),
15 ])
16
17 kf = KFold(4, n_folds=2)
18
19 for train_index, test_index in kf:
20     pipeline.fit(X[train_index], Y[train_index])
21     scores.append(pipeline.score(X[test_index], Y[test_index]))
22
```

Figure 1: Simple pipeline

## make\_pipeline

The utility function `make_pipeline` is a shorthand for constructing pipelines:

[illegible]

### Figure 2: Make pipeline

# Custom Steps

You can build your own steps:

```
24
25 pipeline = Pipeline([('selector', ItemSelector(key='body')),
26                       ('stats', TextStats()), # returns a list of dicts
27                       ('vect', DictVectorizer())])
28
29 class ItemSelector(BaseEstimator, TransformerMixin):
30     """For data grouped by feature, select subset of data at a provided key."""
31     def __init__(self, key):
32         self.key = key
33
34     def fit(self, x, y=None):
35         return self
36
37     def transform(self, data_dict):
38         return data_dict[self.key]
39
```

Figure 3: Custom pipeline



All estimators in a pipeline, except the last one, must be transformers:

- It's just an object that responds to `fit`, `transform`, and `fit_transform`
- Inheriting from `TransformerMixin` is not required, but helps to communicate intent, and gets you `fit_transform` for free
- The pipeline treats these objects like any of the built-in transformers and fits them during the training phase, and transforms the data using each one when predicting

# Custom Transformers

Transformer example:

```
40 |  
41 | class TextStats(BaseEstimator, TransformerMixin):  
42 |     """Extract features from each document for DictVectorizer"""  
43 |  
44 |     def fit(self, x, y=None):  
45 |         return self  
46 |  
47 |     def transform(self, posts):  
48 |         return [{'length': len(text), 'num_sentences': text.count('.')}  
49 |             for text in posts]  
50 |
```

Figure 4: Custom transformer

# Using FunctionTransformer

You can convert an existing Python function into a transformer with FunctionTransformer:

```
82
83 def all_but_first_column(X):
84     return X[:, 1:]
85
86
87 def drop_first_component(X, y):
88     """ Create a pipeline with PCA and the column selector and use it to
89     transform the dataset.
90     """
91     pipeline = make_pipeline(PCA(), FunctionTransformer(all_but_first_column))
92
93     X_train, X_test, y_train, y_test = train_test_split(X, y)
94     pipeline.fit(X_train, y_train)
95     return pipeline.transform(X_test), y_test
96
```

Figure 5:FunctionTransformer

FeatureUnion combines several transformer objects into a new transformer that combines their output:

- each of transformer objects is fit to the data independently
- transformers are applied in parallel
- the sample vectors they output are concatenated end-to-end into larger vectors

FeatureUnion and Pipeline can be combined to create complex models.

# FeatureUnion

## FeatureUnion example:

```
107
108 pca = PCA(n_components=2)
109 selection = SelectKBest(k=1)
110 combined_features = FeatureUnion([("pca", pca), ("univ_select", selection)])
111 svm = SVC(kernel="linear")
112 pipeline = Pipeline([("features", combined_features), ("svm", svm)])
113
114 param_grid = dict(features__pca__n_components=[1, 2, 3],
115                    features__univ_select__k=[1, 2],
116                    svm__C=[0.1, 1, 10])
117
```

Figure 6:Feature Union

# Pipeline parameters

The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters.

It enables setting parameters of the various steps using their names and the parameter name separated by a “\_\_” (<estimator>\_\_<parameter>).

```
116
117  pca = PCA(n_components=2)
118  selection = SelectKBest(k=1)
119  pipeline = Pipeline([("features", combined_features), ("svm", svm)])
120
121  param_grid = dict(features__pca__n_components=[1, 2, 3],
122                  features__univ_select__k=[1, 2],
123                  svm__C=[0.1, 1, 10])
124
```

Parameters that are not directly learnt within estimators can be set by searching a parameter space for the best Cross-validation.

The grid search provided by GridSearchCV exhaustively generates candidates from a grid of parameter values specified with the `param_grid` parameter.

The following `param_grid` specifies that two grids should be explored:

- with a linear kernel and C values in [1, 10, 100, 1000],
- with an RBF kernel, and the cross-product of C values ranging in [1, 10, 100, 1000] and gamma values in [0.001, 0.0001].

```
125
126 param_grid = [{'C': [1, 10, 100, 1000], 'kernel': ['linear']},
127               {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001],
128               'kernel': ['rbf']}
129               ]
130
131
```

Figure 7:GridSearchCV



# Pipeline with GridSearchCV

All together:

```
127 |  
128 | # define a pipeline combining a text feature extractor with a simple classifier  
129 | pipeline = Pipeline([  
130 |     ('vect', CountVectorizer()),  
131 |     ('tfidf', TfidfTransformer()),  
132 |     ('clf', SGDClassifier()),  
133 | ])  
134 |  
135 | parameters = {  
136 |     'vect__max_df': (0.5, 0.75, 1.0),  
137 |     'vect__max_features': (None, 5000, 10000, 50000),  
138 |     'vect__ngram_range': ((1, 1), (1, 2)), # unigrams or bigrams  
139 |     'tfidf__use_idf': (True, False),  
140 |     'tfidf__norm': ('l1', 'l2'),  
141 |     'clf__alpha': (0.00001, 0.000001),  
142 |     'clf__penalty': ('l2', 'elasticnet'),  
143 |     'clf__n_iter': (10, 50, 80),  
144 | }  
145 |  
146 | # find the best parameters for both the feature extraction and the classifier  
147 | grid_search = GridSearchCV(pipeline, parameters, n_jobs=-1, verbose=1)  
148 |  
149 | grid_search.fit(data.data, data.target)  
150 | print("Best score: %0.3f" % grid_search.best_score_)  
151 |  
152 | best_parameters = grid_search.best_estimator_.get_params()  
153 | for param_name in sorted(parameters.keys()):  
154 |     print("\t%s: %r" % (param_name, best_parameters[param_name]))  
155 |
```

# Advanced Pipeline

# Custom Estimators

A estimator is an object that fits a model based on some training data and is capable of inferring some properties on new data. All estimators implement the `fit` method.

To create custom estimator, you need to implement the following interface:

- `get_params([deep])` Get parameters for this estimator
- `set_params(**params)` Set the parameters of this estimator

You can inherit from `BaseEstimator` and optionally the mixin classes in `sklearn.base`.

# Custom Estimators

Estimator example:

```

51 |
52 | class MajorityClassifier(BaseEstimator, ClassifierMixin):
53 |     """Predicts the majority class of its training data."""
54 |     def __init__(self):
55 |         pass
56 |
57 |     def fit(self, X, y):
58 |         self.classes_, indices = np.unique(["foo", "bar", "foo"],
59 |             return_inverse=True)
60 |         self.majority_ = np.argmax(np.bincount(indices))
61 |         return self
62 |
63 |     def predict(self, X):
64 |         return np.repeat(self.classes_[self.majority_], len(X))
65 |

```

Figure 9: Custom Estimator

# Custom Estimators

If you do not want to make your code dependent on scikit-learn, the easiest way to implement the interface is:

```
65 |  
66 | def get_params(self, deep=True):  
67 |     # suppose this estimator has parameters "alpha" and "recursive"  
68 |     return {"alpha": self.alpha, "recursive": self.recursive}  
69 |  
70 | def set_params(self, **parameters):  
71 |     for parameter, value in parameters.items():  
72 |         self.setattr(parameter, value)  
73 |     return self  
74 |
```

Figure 10: Custom Estimator

# Custom Estimators

You can check whether your estimator adheres to the scikit-learn interface and standards by running `utils.estimator_checks.check_estimator` on the class:

```
75 |  
76 | from sklearn.utils.estimator_checks import check_estimator  
77 | from sklearn.svm import LinearSVC  
78 | check_estimator(LinearSVC) # passes  
79 |
```

Figure 11: Check Estimator

# Estimator types

Some common functionality depends on the kind of estimator passed. This distinction is implemented using the `_estimator_type` attribute:

- “classifier” for classifiers
- “regressor” for regressors
- “clusterer” for clustering methods

Inheriting from `ClassifierMixin`, `RegressorMixin` or `ClusterMixin` will set the attribute automatically.

# Check for Mastery

- What tools can we use to develop more compact and comprehensive code?
- How can we do joint parameter selection?



# References

- sklearn Pipeline
- sklearn.FeatureUnion
- sklearn.GridSearchCV
- Putting it all together
- Concatenating multiple feature extraction methods
- Pipeline and FeatureUnion: combining estimators
- Grid Search: Searching for estimator parameters
- Sample pipeline for text feature extraction and evaluation
- Using FunctionTransformer to select columns
- Feature Union with Heterogeneous Data Sources
- Workflows in Python: Using Pipeline and GridSearchCV for More Compact and Comprehensive Code
- Using scikit-learn Pipelines and FeatureUnions