Putting it all together

[Tetyana Kutsenko]

August 18, 2016

Objectives

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

- Use Pipeline for 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

Before

We usually start modeling from simple steps like:

- Reading and splitting prepared data into train/test subsets
- Extracting target
- Extracting features
- In cycle: 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 Pipeline

Pipeline can be used to 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.).

scikit-learn Pipeline

```
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

```
99
100 make_pipeline(Binarizer(), MultinomialNB())
101
102 >>> Pipeline(steps=[('binarizer', Binarizer(copy=True, threshold=0.0)),
103                    ('multinomialnb', MultinomialNB(alpha=1.0,
104                                                    class_prior=None,
105                                                    fit_prior=True))])
106
```

Figure 2:Make pipeline

Custom 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:Simple pipeline

Custom Transformers

A transformer is 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.

Custom Transformers

```
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:Simple pipeline

Using FunctionTransformer

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

```
71
72  def all_but_first_column(X):
73      return X[:, 1:]
74
75
76  def drop_first_component(X, y):
77      """ Create a pipeline with PCA and the column selector and use it to
78      transform the dataset.
79      """
80      pipeline = make_pipeline(
81          PCA(), FunctionTransformer(all_but_first_column),
82      )
83      X_train, X_test, y_train, y_test = train_test_split(X, y)
84      pipeline.fit(X_train, y_train)
85      return pipeline.transform(X_test), y_test
86
```

Figure 5:Simple 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

```
41
42  class MajorityClassifier(BaseEstimator, ClassifierMixin):
43      """Predicts the majority class of its training data."""
44      def __init__(self):
45          pass
46
47      def fit(self, X, y):
48          self.classes_, indices = np.unique(["foo", "bar", "foo"],
49                                             return_inverse=True)
50          self.majority_ = np.argmax(np.bincount(indices))
51          return self
52
53      def predict(self, X):
54          return np.repeat(self.classes_[self.majority_], len(X))
55
56
```

Figure 6: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:

```
55
56  def get_params(self, deep=True):
57    # suppose this estimator has parameters "alpha" and "recursive"
58    return {"alpha": self.alpha, "recursive": self.recursive}
59
60  def set_params(self, **parameters):
61    for parameter, value in parameters.items():
62      self.setattr(parameter, value)
63    return self
64
```

Figure 7: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:

```
65
66  from sklearn.utils.estimator_checks import check_estimator
67  from sklearn.svm import LinearSVC
68  check_estimator(LinearSVC)  # passes
69
```

Figure 8:Custom 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.

FeatureUnion

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

```
109
110  pca = PCA(n_components=2)
111  selection = SelectKBest(k=1)
112
113  combined_features = FeatureUnion([("pca", pca), ("univ_select", selection)])
114
115  svm = SVC(kernel="linear")
116  pipeline = Pipeline([("features", combined_features), ("svm", svm)])
117
118
```

Figure 9: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
```

GridSearchCV

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.

GridSearchCV

The following param_grid specifies that two grids should be explored: one with a linear kernel and C values in [1, 10, 100, 1000], and the second one 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
```

Pipeline with GridSearchCV

```
132
133 # define a pipeline combining a text feature extractor with a simple
134 # classifier
135 pipeline = Pipeline([
136     ('vect', CountVectorizer()),
137     ('tfidf', TfidfTransformer()),
138     ('clf', SGDClassifier()),
139 ])
140
141 # uncommenting more parameters will give better exploring power but will
142 # increase processing time in a combinatorial way
143 parameters = {
144     'vect__max_df': (0.5, 0.75, 1.0),
145     #'vect__max_features': (None, 5000, 10000, 50000),
146     'vect__ngram_range': ((1, 1), (1, 2)),  # unigrams or bigrams
147     #'tfidf__use_idf': (True, False),
148     #'tfidf__norm': ('l1', 'l2'),
149     'clf__alpha': (0.00001, 0.000001),
150     'clf__penalty': ('l2', 'elasticnet'),
151     #'clf__n_iter': (10, 50, 80),
152 }
153
154 # find the best parameters for both the feature extraction and the
155 # classifier
156 grid_search = GridSearchCV(pipeline, parameters, n_jobs=-1, verbose=1)
157
158 grid_search.fit(data.data, data.target)
159 print("Best score: %0.3f" % grid_search.best_score_)
160
161 best_parameters = grid_search.best_estimator_.get_params()
162 for param_name in sorted(parameters.keys()):
163     print("\t%s: %r" % (param_name, best_parameters[param_name]))
164
165
```

Figure 10:GridSearch Example