```
In [1]: %matplotlib inline
        import matplotlib
        import seaborn as sns
        sns.set()
        matplotlib.rcParams['figure.dpi'] = 144
```

Scikit-learn Workflow

- Understand how to write custom classes in Scikit-learn
- Pipelines
- Feature Unions

Writing custom estimators and transformers

The Scikit-learn library has a wealth of functionality available in its classes (http://scikit-learn.org/stable/modules/classes.html). Occasionally you might want to customize the behavior of these classes, for example to add in functionality or for engineering reasons.

Scikit-learn has some detailed documentation on customizing these classes - start here (http://scikit-learn.org/stable/developers/contributing.html#apis-of-scikit-learn-objects) and make sure to get to the "Rolling your own Estimator" section.

This notebook contains a bit more exposition and some examples for implementing custom classes. Think of it as a field guide to writing your own pluggable functionality.

All estimators (e.g. Linear Regression, KMeans, etc ...) support `fit` and `predict` methods. In fact, you can build your own by inheriting from classes in `sklearn.base` by using this template:

```
class Estimator(base.BaseEstimator, base.RegressorMixin):
  def __init__(self, ...):
    # initialization code

  def fit(self, X, y):
    # fit the model ...
    return self

  def predict(self, X):
    return # prediction

  def score(self, X, y):
    return # custom score implementation
```

Conforming to this convention has the benefit that many tools (e.g. cross-validation, grid search) rely on this interface so you can use your new estimators with the existing `sklearn` infrastructure.

For example `model_selection.GridSearchCV` takes an estimator and some hyperparameters as arguments, and returns another estimator. Upon fitting, it fits the best model (based on the inputted hyperparameters) and uses that for prediction.

Of course, we sometimes need to process or transform the data before we can do machine learning on it. `sklearn` has Transformers to help with this. They implement this interface:

```
class Transformer(base.BaseEstimator, base.TransformerMixin):
  def __init__(self, ...):
    # initialization code

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

  def transform(self, X):
    return ... # transformation
```

A comprehensive `.fit_transform` is implemented based on the `.fit` and `.transform` methods in `base.TransformerMixin` (docs (http://scikit-learn.org/stable/modules/generated/sklearn.base.TransformerMixin.html)). However, especially for transformers, `.fit` is often empty and only `.transform` actually does something.

The following is some example code to demonstrate the usage of custom classes.

In [2]:
```
import numpy as np
import scipy as sp
import sklearn as sk
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
import gzip
import json

cali_data = json.load(gzip.open('small_data/cal_house.json.gz'))

cali_data['data'] = np.array(cali_data['data'])
cali_data['target'] = np.array(cali_data['target'])

X_train, X_test, y_train, y_test = train_test_split(cali_data['data'], cali_da
ta['target'], test_size=0.2, random_state=42)
cali_data['data'].shape, X_train.shape, X_test.shape
```

Out[2]:
```
((20640, 8), (16512, 8), (4128, 8))
```

In [3]:
```
class CenteringTransformer(sk.base.BaseEstimator, sk.base.TransformerMixin):
    """
    Centers the features about 0
    """
    def fit(self, X, y=None):
        self.means = X.mean(axis=0)
        return self

    def transform(self, X):
        return X - self.means
```

In [4]:
```
ct = CenteringTransformer()
ct.fit(X_train)
# Note: trained on train data, testing on test data
X_test.mean(), ct.transform(X_test).mean()
```

Out[4]:
```
(172.46247653163707, -0.6164963121787063)
```

Some estimators and transformers take arguments. These can be specified in the `__init__` function.

In [5]:
```
class TruncateTransformer(sk.base.BaseEstimator, sk.base.TransformerMixin):
    """
    Returns the first k columns of a feature array
    """
    def __init__(self, k=None):
        self.k = k

    def fit(self, X, y):
        return self

    def transform(self, X):
        if self.k is None:
            return X
        else:
            return X[:,:self.k]
```

In [6]:
```
truncator = TruncateTransformer(2)
truncator.transform(X_train).shape
```

Out[6]:
```
(16512, 2)
```

Here's a silly estimator that returns a values based on whether there are more positive or negative features for an observation.

In [7]:
```
class PosNegEstimator(sk.base.BaseEstimator, sk.base.RegressorMixin):
    """
    Predict mean values based on whether there are more positive or negative f
eatures.

    Yes, this will do a terrible job.
    """
    def __init__(self):
        self.pos_mean = 0
        self.neg_mean = 0

    def fit(self, X, y):
        pos_rows = (X > 0).sum(axis=1) > X.shape[1]/2
        self.pos_mean = y[pos_rows].mean() if sum(pos_rows) > 0 else 0
        self.neg_mean = y[~pos_rows].mean() if sum(~pos_rows) > 0 else 0
        return self

    def predict(self, X):
        pos_rows = (X > 0).sum(axis=1) > X.shape[1]/2
        y = np.zeros(X.shape[0])
        y[pos_rows] = self.pos_mean
        y[~pos_rows] = self.neg_mean
        return y
```

```
In [8]: pne = PosNegEstimator()
        pne.fit(X_train, y_train)
        pne.predict(X_test)
        pne.score(X_test, y_test)

Out[8]: -0.00021908714592466794
```

The `RegressorMixin` provides a `.score` method that calculates the coefficient of determination. In this case, all rows are positive, so this is just a mean model.

```
In [9]: ct = CenteringTransformer()
        pne = PosNegEstimator()

        ct.fit(X_train, y_train)
        X_train_center = ct.transform(X_train)
        pne.fit(X_train_center, y_train)
        X_test_center = ct.transform(X_test)
        pne.score(X_test_center, y_test)

Out[9]: 0.009968372699187822
```

The fit method returns the `self` object, so that method calls can be chained. The `.fit_transform` method is equivalent to chaining the `.transform` method to the `.fit` method. Together, these can turn the above into a one-liner.

```
In [10]: pne.fit(ct.fit_transform(X_train), y_train).score(ct.transform(X_test), y_test
         )

Out[10]: 0.009968372699187822
```

Connecting transformers to estimators can become tedious. One way around this is to build an estimator that combines them into one object.

In [11]:
```
class ShellEstimator(sk.base.BaseEstimator, sk.base.RegressorMixin):
    """
    A shell estimator that combines a transformer and regressor into a single
 object.
    """
    def __init__(self, transformer, model):
        self.transformer = transformer
        self.model = model

    def fit(self, X, y):
        X_trans = self.transformer.fit(X, y).transform(X)
        self.model.fit(X_trans, y)
        return self

    def score(self, X, y):
        X_test = self.transformer.transform(X)
        return self.model.score(X_test, y)

    def predict(self, X):
        X_test = self.transformer.transform(X)
        return self.model.predict(X_test)
```

In [12]:
```
truncator = TruncateTransformer(2)
X_train_k = truncator.transform(X_train)
X_test_k = truncator.transform(X_test)

linreg = LinearRegression()
k_model = linreg.fit(X_train_k, y_train)
y_pred_k = k_model.predict(X_test_k)
print(k_model.score(X_test_k, y_test))
```

```
0.4940606792889837
```

In [13]:
```
k_shell = ShellEstimator(TruncateTransformer(2), LinearRegression(fit_intercep
t=True))
k_shell.fit(X_train, y_train)
assert((k_shell.predict(X_test) == y_pred_k).all())
print(k_shell.score(X_test, y_test))
```

```
0.4940606792889837
```

Pipelines

It turns out there's a built-in tool to chain together our transformers and estimators into one unit, and it scales much easier than custom estimators. They're called pipelines. The following code would replace all the fitting and scoring code above. That is, the pipeline itself is an estimator (and implements the `.fit` and `.predict` methods). Note that a pipeline can have multiple transformers chained up but at most one (optional) terminal estimator.

In [14]:
```
from sklearn import pipeline

k_pipe = pipeline.Pipeline([
  ('truncate', TruncateTransformer(2)),
  ('linreg', LinearRegression(fit_intercept=True))
  ])
k_pipe.fit(X_train, y_train)
print(k_pipe.score(X_test, y_test))
```

```
0.4940606792889837
```

The hyper-parameters of the component elements are available as parameters in the pipeline with the `element__param_name` format. This means the pipelines can be used in cross-validation routines.

In [15]:
```
k_pipe.get_params()
```

Out[15]:
```
{'memory': None,
 'steps': [('truncate', TruncateTransformer(k=2)),
  ('linreg',
   LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=Fals
e))],
 'truncate': TruncateTransformer(k=2),
 'linreg': LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normal
ize=False),
 'truncate__k': 2,
 'linreg__copy_X': True,
 'linreg__fit_intercept': True,
 'linreg__n_jobs': 1,
 'linreg__normalize': False}
```

Given a Pipeline, `pipe`, defined as

```
pipe = Pipe([
    ('mod_a', model_a),
    ('mod_b', model_b),
    ('mod_c', model_c)
])
```

When `pipe.fit(X, y)` is called, the following chain of operations will take place (note this is pseudocode):

```
Xt = model_a.fit_transform(X, y)
Xt = model_b.fit_transform(Xt, y)
model_c.fit(Xt, y)
```

Thus the fit method of a pipeline will fit the last model in the pipeline with a feature matrix obtained by successive transformations of the original feature matrix by all of the previous models.

When `pipe.predict(X)` is called, the following chain of operations will take place (again in pseudocode, Scikit-learn checks a few things under the hood we will ignore here):

```
Xt = model_a.transform(X)
Xt = model_b.transform(Xt)
model_c.predict(Xt)
```

Thus the predict method will use the predict method of the last model with a feature matrix obtained by successive transformations of the original feature matrix by all of the previous models.

A good place to delve into the details is the source code located here (https://github.com/scikit-learn/scikit-learn/blob/0.19.X/sklearn/pipeline.py#L29).

Feature Unions

Feature unions (http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.FeatureUnion.html) are designed to get around the problem that you might not be able to prepare your desired feature matrix with just one series of transformations. For example, you might have text features, categorical features, and a couple different kinds of numerical features and want to feed them all into the same estimator or pipeline. Each feature type would require a different kind of transformer.

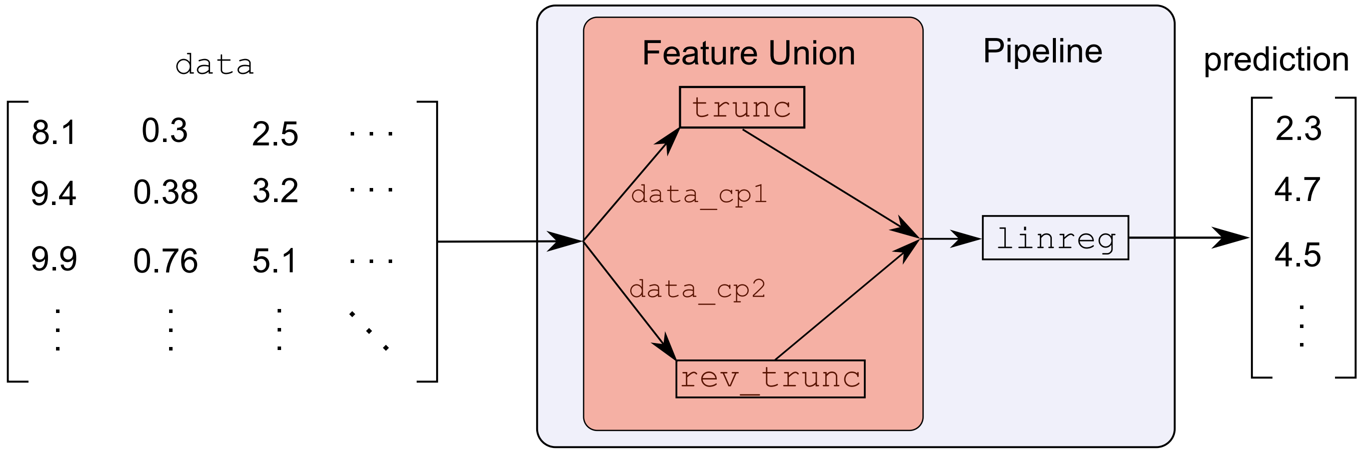
What the feature union does is a kind of *parallel* transformation operation using multiple transformers and consolidating them into one output matrix — which can then be fed into an estimator or pipeline. You can imagine that between the serial behavior of pipelines and the parallel behavior of feature unions you can create complex multi-stage workflows.

This example code applies several different transformations to X before throwing the features into a Linear Regressor.

In [16]:
```
class ReverseTruncateTransformer(sk.base.BaseEstimator, sk.base.TransformerMix
in):
    """
    Returns the last k columns of a feature array
    """
    def __init__(self, k=None):
        self.k = k

    def fit(self, X, y):
        return self

    def transform(self, X):
        if self.k is None:
            return X
        else:
            return X[:,-self.k:]

all_features = pipeline.FeatureUnion([
  ('first two cols', TruncateTransformer(2)),
  ('last two cols', ReverseTruncateTransformer(2))
  ])
k_union = pipeline.Pipeline([
  ("features", all_features),
  ("linreg", LinearRegression(fit_intercept=True))
  ])
k_union.fit(X_train, y_train)
print(k_union.score(X_test, y_test))
```

```
0.5810616616067183
```

By combining feature unions and pipelines, it is possible to build arbitrarily complex flows from raw data to final predictions, all contained within a single Scikit-learn object. For complex combinations, it can be helpful to diagram the architecture.



You can also use feature unions to combine the predictions of multiple estimators. If you rewrite your estimators as transformers and feed them into a feature union which has, for example, a linear regressor as an estimator, you'll be able to automatically weight and combine each individual prediction into an ensemble model.

To do this, you'll need to write custom transformers where the `.transform` method carries out the `.predict` implementation.

Data Types

What types of data can be fed in as the `X` and `y` arguments to transformers and estimators? There are two considerations:

1. If you are writing your own transformers and estimators, you can make them take whatever file data format you like, as long as output and input types agree for sequential elements in a chain. Scikit-learn will happily pass these objects through pipelines, feature union, grid searches, etc.
2. If you want to interact with transforms and estimators provided by Scikit-learn, you need to provide data in a compatible type. Scikit-learn, like much of the Python ecosystem, uses duck typing. This means that instead of testing that the input is a particular type, it tests that the input has particular capabilities. For most elements, Scikit-learn will require that the inputs behave like NumPy arrays. If you're not sure if a particular input will work, just try it!

Validating Your Implementations

Once you've implemented a custom estimator/transformer, you can validate that it correctly works using `sklearn.utils.estimator_checks.check_estimator` docs (http://scikit-learn.org/stable/modules/generated/sklearn.utils.estimator_checks.check_estimator.html). Here's an example, validating our above implementations.

Warning: This is a bit of code inside of sklearn that's still under active development and discussion. It should help you get the basics down for implementing sklearn's interface, but you might run into some weird issues, or this validation might be "too much" for the pipeline you're trying to build or even not general enough for something that might be compatible. You might actually have something that's compatible with `GridSearchCV` for example, and would still fail validation here (e.g. input validation below).

For example, let's try validating the transformer we wrote:

In [17]:
```
from sklearn.utils import estimator_checks as ec
try:
    ec.check_estimator(CenteringTransformer)
except AssertionError as e:
    print(e)
```

```
"argument must be a string or a number" does not match "unsupported operand t
ype(s) for +: 'dict' and 'float'"
```

As you can see, we still have an error around input validation. This exception is expected... http://scikit-learn.org/stable/developers/contributing.html#input-validation (http://scikit-learn.org/stable/developers/contributing.html#input-validation) And is even something worth worrying about during development. Here's the validation Scikit-learn is looking for:

In [18]:
```
from sklearn.utils import check_array

def validate_args(arg):
    check_array(arg)
    return np.asarray(arg)

class ValidatingCenteringTransformer(sk.base.BaseEstimator, sk.base.Transforme
rMixin):
    """
    Centers the features about 0
    """
    def fit(self, X, y=None):
        v_X = validate_args(X)
        v_y = np.asarray(y) if y is not None else None
        self._means = v_X.mean(axis=0)
        return self

    def transform(self, X):
        v_X = validate_args(X)
        return v_X - self._means


ec.check_estimator(ValidatingCenteringTransformer)
```

And this time, we pass!