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

Scikit-Learn API

Classes vs Objects

In Scikit-Learn, machine learning algorithms are implemented as classes.

For example, the class `LinearRegression`:

- Is a recipe for creating linear models, but is not a model itself
- Cannot learn or store model parameters
- Cannot be used to make predictions

By contrast, the object `LinearRegression()`:

- Is an instance of the `LinearRegression` class and represents a concrete model
- Can learn and store parameters (e.g. regression coefficients)
- Can be applied to test data to make predictions

Note that hyperparameters are often specified as arguments when we create an instance of a class. For example, if we want to specify the regularization parameter used in a Ridge model, we can do so as follows:

In [2]:
```
from sklearn.linear_model import Ridge

ridge_model = Ridge(alpha=0.001)
```

Scikit-Learn has two main types of classes: estimators and transformers.

Estimators

Estimators are objects that can make predictions. All estimators share the following core methods:

- **`.fit(X,y)`** learns and stores model parameters based on a training set (X, y) and then returns the trained model.
- **`.predict(X)`** returns a list of predicted labels for a test set X using parameters learned from previous training.

This common syntax is what we're referring to when we use the phrase Scikit-Learn API. It makes it very easy to pick up and use new models. For example, we can very easily fit a linear model to some toy data and make predictions:

In [3]:
```
from sklearn.linear_model import LinearRegression
import numpy as np

# Training Data
X_ = np.linspace(1,10,10)
X = X_.reshape(-1,1)
y = 2*X_ + 0.3*np.random.randn(10)

# Test Data
X_test = np.linspace(11,15,5).reshape(-1,1)

# Model and Prediction
lin_est = LinearRegression() # Create an instance of the LinearRegression clas
s
lin_est.fit(X,y)             # Learn model parameters by fitting to the test d
ata
lin_est.predict(X_test)      # Make a prediction using the trained model
```

Out[3]:
```
array([22.09914701, 24.10359893, 26.10805086, 28.11250278, 30.1169547 ])
```

And we can implement a more complicated model just as easily (although in this case we'll get worse results).

In [4]:
```
from sklearn.ensemble import RandomForestRegressor

forest_est = RandomForestRegressor()
forest_est.fit(X,y)
forest_est.predict(X_test)
```

```
/opt/conda/envs/data3/lib/python3.6/site-packages/sklearn/ensemble/weight_boo
sting.py:29: DeprecationWarning: numpy.core.umath_tests is an internal NumPy
module and should not be imported. It will be removed in a future NumPy relea
se.
  from numpy.core.umath_tests import inner1d
```

Out[4]:
```
array([19.16922459, 19.16922459, 19.16922459, 19.16922459, 19.16922459])
```

Note that Scikit-Learn is a bit fussy about how our data is formatted. X needs to be 2-dimensional and y needs to be 1-dimensional.

Supervised and probabilistic models have additional methods in common:

- **`.score(X,y)`** returns either the R^2 score or accuracy of the estimator on a test set (X, y)
- **`.predict_proba(X)`** returns an array with the raw probabilities predicted by the model for a test set X. The output will have one column for each class.

In [5]:
```
y_test = [22, 24, 26, 28, 30]

lin_est.score(X_test,y_test), forest_est.score(X_test,y_test)
```

Out[5]: (0.998535671687071, -5.832436581224522)

Individual estimators may have additional methods and attributes, many of which are model specific. In general, Scikit-Learn's documentation is quite good. For example, take a look at the page for `LinearRegression` (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html).

In [6]:
```
lin_est.coef_, lin_est.intercept_
```

Out[6]: (array([2.00445192]), 0.050175856419611975)

Transformers

Transformers are used to do preprocessing on data before feeding it into a model. All transformers share the following core methods:

- **`.fit(X)`** learns and stores parameters based on the set X and then returns the transformer
- **`.transform(X)`** transforms the set X, relying on stored parameters if appropriate, and then returns the result
- **`.fit_transform(X)`** applies fit and transform in sequence

For example, the `.transform` method of Scikit-Learn's `StandardScaler` (http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html) rescales data by applying the formula

$$x \mapsto \frac{x - \mu}{\sigma}$$

to each feature. The `.fit` method calculates and stores the mean μ and the standard deviation σ of whatever data set it is given. This means that `.fit_transform` rescales data to have mean 0 and standard deviation 1.

In general, we do not want to apply `.fit` every time we apply `.transform`. In machine learning setups, we want to transform the training and test sets the same way, so we usually call the `.fit` method only once (on the training set).

In [7]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_trans      = scaler.fit_transform(X)         # We should fit our transformer
when applying it to the training set
X_test_trans = scaler.transform(X_test)        # But not when applying it to t
he test set

# mean and standard deviation
print(X.mean(), X.std())                       # of X
print(X_trans.mean(), X_trans.std())           # of X_trans
print(X_test_trans.mean(), X_test_trans.std()) # of X_test_trans
```

```
5.5 2.8722813232690143
-6.661338147750939e-17 1.0
2.6111648393354674 0.492365963917331
```

Pipelines

Pipelines are a way to combine a sequence of transformers (and optionally an estimator) into a single object. This saves us the work of computing and storing each transformer output separately and can help us avoid errors. Pipelines act either as transformers or estimators, depending on whether they contain an estimator as a component.

- **`.fit(X,y)`** calls the `.fit_transform` method of each transformer component and then calls the `.fit` method of the terminal estimator if one is present
- **`.predict(X)`** calls the `.transform` method of each transformer component and then calls the `.predict` method of the terminal estimator
- **`.transform(X)`** calls the `.transform` method of each component (if none of the components is an estimator)

Syntactically, a pipeline in Scikit-Learn takes a list of tuples as arguments. The first element of each tuple is a string that acts as a label for the corresponding step, and the second element is the transformer or estimator that we want to include.

In [8]:
```
from sklearn.pipeline import Pipeline

pipe = Pipeline([("feature scaling", scaler),
                 ("linear regression", LinearRegression())])

pipe.fit(X,y)
pipe.predict(X_test)
```

Out[8]:
```
array([22.09914701, 24.10359893, 26.10805086, 28.11250278, 30.1169547 ])
```

The individual components of a pipeline can be accessed using the `.named_steps` attribute, which returns a dictionary of steps. The label strings act as keys and the components themselves are returned as values.

In [9]:
```
pipe = Pipeline([("feature scaling", scaler),
                 ("ridge", ridge_model)])

print(type(pipe.named_steps))
print(pipe.named_steps['ridge'])
print(pipe.named_steps['ridge'].alpha)
```

```
<class 'sklearn.utils.Bunch'>
Ridge(alpha=0.001, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001)
0.001
```