```
# load dataset
data = datasets.load_boston()

# separate features and target
X = data.data
y = data.target

# build the pipeline model
pipe = make_pipeline(
    PCA(n_components=9),
    RandomForestRegressor()
)

# run the pipeline
kfold = KFold(n_splits=4, shuffle=True)
cv_result = cross_val_score(pipe, X, y, cv=kfold)

# evaluate the model performance
print("Accuracy: %.3f%% (%.3f%%)" % (cv_result.mean()*100.0, cv_result.
std()*100.0))
'Output':
Accuracy: 73.750% (2.489%)
```

Pipelines Using FeatureUnion

Scikit-learn provides a module for merging the output of several transformers called **feature_union**. It does this by fitting each transformer independently to the dataset, and then their respective outputs are combined to form a transformed dataset for training the model.

FeatureUnion works in the same way as a Pipeline, and in many ways can be thought of as a means of building complex pipelines within a Pipeline.

Let's see an example using FeatureUnion. Here, we will combine the output of recursive feature elimination (RFE) and PCA for feature engineering, and then we'll apply the Stochastic Gradient Boosting (SGB) ensemble model for regression to train the model.

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn import datasets
```

```
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.feature_selection import RFE
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline
from sklearn.pipeline import make_pipeline
from sklearn.pipeline import make_union

# load dataset
data = datasets.load_boston()

# separate features and target
X = data.data
y = data.target

# construct pipeline for feature engineering - make_union similar to make_
pipeline
feature_engr = make_union(
    RFE(estimator=RandomForestRegressor(n_estimators=100), n_features_to_
                select=6),
    PCA(n_components=9)
)

# build the pipeline model
pipe = make_pipeline(
    feature_engr,
    GradientBoostingRegressor(n_estimators=100)
)

# run the pipeline
kfold = KFold(n_splits=4, shuffle=True)
cv_result = cross_val_score(pipe, X, y, cv=kfold)

# evaluate the model performance
print("Accuracy: %.3f%% (%.3f%%)" % (cv_result.mean()*100.0, cv_result.
std()*100.0))
'Output':
Accuracy: 88.956% (1.493%)
```

Model Tuning

Each machine learning model has a set of options or configurations that can be tuned to optimize the model when fitting to data. These configurations are called **hyper-parameters**. Hence, for each hyper-parameter, there exist a range of values that can be chosen. Taking into consideration the number of hyper-parameters that an algorithm has, the entire space can become exponentially large and infeasible to explore all of them. Scikit-learn provides two convenient modules for searching through the hyper-parameter space of an algorithm to find the best values for each hyper-parameter that optimizes the model.

These modules are the

- Grid search
- Randomized search

Grid Search

Grid search comprehensively explores all the specified hyper-parameter values for an estimator. It is implemented using the **GridSearchCV** module. Let's see an example using the Random forest for regression. The hyper-parameters we'll search over are

- The number of trees in the forest, **n_estimators**
- The maximum depth of the tree, **max_depth**
- The minimum number of samples required to split an internal node, **min_samples_leaf**

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestRegressor
from sklearn import datasets

# load dataset
data = datasets.load_boston()

# separate features and target
X = data.data
y = data.target
```