Machine Learning Foundation

Cross Validation

Learning objectives

By the end of this lesson, you will be able to:

- Chain multiple data processing steps together using `Pipeline`
- Use the `KFolds` object to split data into multiple folds.
- Perform cross validation using SciKit Learn with `cross_val_predict` and `GridSearchCV`

In [2]:
```
import numpy as np
import pickle
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.model_selection import KFold, cross_val_predict
from sklearn.linear_model import LinearRegression, Lasso, Ridge
from sklearn.metrics import r2_score
from sklearn.pipeline import Pipeline
```

In [4]:
```
# Note we are loading a slightly different ("cleaned") pickle file
boston = pd.read_pickle('boston_housing_clean.pickle')
```

In [5]:
```
boston.keys()
```

Out[5]:
```
dict_keys(['dataframe', 'description'])
```

In [6]:
```
boston_data = boston['dataframe']
boston_description = boston['description']
```

In [7]:
```
boston_data.head()
```

Out[7]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTA
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.9
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.1
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.0
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.9
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.3

Discussion:

Suppose we want to do Linear Regression on our dataset to get an estimate, based on mean squared error, of how well our model will perform on data outside our dataset.

Suppose also that our data is split into three folds: Fold 1, Fold 2, and Fold 3.

What would the steps be, in English, to do this?

Your response below

Coding this up

The `KFold` (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html) object in SciKit Learn tells the cross validation object (see below) how to split up the data:

In [8]:
```
X = boston_data.drop('MEDV', axis=1)
y = boston_data.MEDV
```

In [9]:
```
kf = KFold(shuffle=True, random_state=0, n_splits=3)
```

In [10]:
```
for train_index, test_index in kf.split(X):
    print("Train index:", train_index[:10], len(train_index))
    print("Test index:",test_index[:10], len(test_index))
    print('')
```

```
Train index: [ 0  2  3  9 11 13 16 17 18 19] 337
Test index: [ 1  4  5  6  7  8 10 12 14 15] 169

Train index: [ 0  1  4  5  6  7  8  9 10 11] 337
Test index: [ 2  3 13 16 17 18 19 24 27 29] 169

Train index: [ 1  2  3  4  5  6  7  8 10 12] 338
Test index: [ 0  9 11 23 25 28 31 32 36 38] 168
```

In [11]:
```
#from sklearn.metrics import r2_score, mean_squared_error

scores = []
lr = LinearRegression()

for train_index, test_index in kf.split(X):
    X_train, X_test, y_train, y_test = (X.iloc[train_index, :],
                                        X.iloc[test_index, :],
                                        y[train_index],
                                        y[test_index])

    lr.fit(X_train, y_train)

    y_pred = lr.predict(X_test)

    score = r2_score(y_test.values, y_pred)

    scores.append(score)

scores
```

Out[11]:
```
[0.6695091694290213, 0.716425442308432, 0.757686994640314]
```

A bit cumbersome, but do-able.

Discussion (Part 2):

Now suppose we want to do the same, but appropriately scaling our data as we go through the folds.

What would the steps be *now*?

Your response below

Coding this up

```
In [12]: scores = []

         lr = LinearRegression()
         s = StandardScaler()

         for train_index, test_index in kf.split(X):
             X_train, X_test, y_train, y_test = (X.iloc[train_index, :],
                                                 X.iloc[test_index, :],
                                                 y[train_index],
                                                 y[test_index])

             X_train_s = s.fit_transform(X_train)

             lr.fit(X_train_s, y_train)

             X_test_s = s.transform(X_test)

             y_pred = lr.predict(X_test_s)

             score = r2_score(y_test.values, y_pred)

             scores.append(score)
```

```
In [13]: scores
```

```
Out[13]: [0.6695091694290187, 0.7164254423084311, 0.7576869946403136]
```

(same scores, because for vanilla linear regression with no regularization, scaling actually doesn't matter for performance)

This is getting quite cumbersome!

Very luckily, SciKit Learn has some wonderful functions that handle a lot of this for us.

`Pipeline` and `cross_val_predict`

`Pipeline` lets you chain together multiple operators on your data that both have a `fit` method.

```
In [14]: s = StandardScaler()
         lr = LinearRegression()
```

Combine multiple processing steps into a `Pipeline`

A pipeline contains a series of steps, where a step is ("name of step", actual_model). The "name of step" string is only used to help you identify which step you are on, and to allow you to

In [15]:
```
estimator = Pipeline([("scaler", s),
                      ("regression", lr)])
```

`cross_val_predict`

`cross_val_predict` [(http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_predict.html)](http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_predict.html) is a function that does K-fold cross validation for us, appropriately fitting and transforming at every step of the way.

In [16]:
```
kf
```
Out[16]:
```
KFold(n_splits=3, random_state=0, shuffle=True)
```

In [17]:
```
predictions = cross_val_predict(estimator, X, y, cv=kf)
```

In [18]:
```
r2_score(y, predictions)
```
Out[18]:
```
0.7183471466703697
```

In [19]:
```
np.mean(scores) # almost identical!
```
Out[19]:
```
0.7145405354592546
```

Note that `cross_val_predict` doesn't use the same model for all steps; the predictions for each row are made when that row is in the validation set. We really have the collected results of 3 (i.e. `kf.num_splits`) different models.

When we are done, `estimator` is still not fitted. If we want to predict on *new* data, we still have to train our `estimator` .

Hyperparameter tuning

Definition

Hyperparameter tuning involves using cross validation (or train-test split) to determine which hyperparameters are most likely to generate a model that *generalizes* well outside of your sample.

Mechanics

We can generate an exponentially spaces range of values using the numpy `geomspace` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.geomspace.html#numpy.geomspac function.

```
np.geomspace(1, 1000, num=4)
```

produces:

```
array([    1.,    10.,   100.,  1000.])
```

Use this function to generate a list of length 10 called `alphas` for hyperparameter tuning:

In [20]:
```
alphas = np.geomspace(1e-9, 1e0, num=10)
alphas
```

Out[20]:
```
array([1.e-09, 1.e-08, 1.e-07, 1.e-06, 1.e-05, 1.e-04, 1.e-03, 1.e-02,
       1.e-01, 1.e+00])
```

The code below tunes the `alpha` hyperparameter for Lasso regression.

In [21]:
```
scores = []
coefs = []
for alpha in alphas:
    las = Lasso(alpha=alpha, max_iter=100000)

    estimator = Pipeline([
        ("scaler", s),
        ("lasso_regression", las)])

    predictions = cross_val_predict(estimator, X, y, cv = kf)

    score = r2_score(y, predictions)

    scores.append(score)
```

In [22]:
```
list(zip(alphas,scores))
```

Out[22]:
```
[(1e-09, 0.7183471466860298),
 (1e-08, 0.7183471468302703),
 (1e-07, 0.7183471482341961),
 (1e-06, 0.7183471625570611),
 (1e-05, 0.7183473026004683),
 (0.0001, 0.7183487264657782),
 (0.001, 0.7183617509751856),
 (0.01, 0.7184793043126607),
 (0.1, 0.7134635068883326),
 (1.0, 0.6505747260408858)]
```

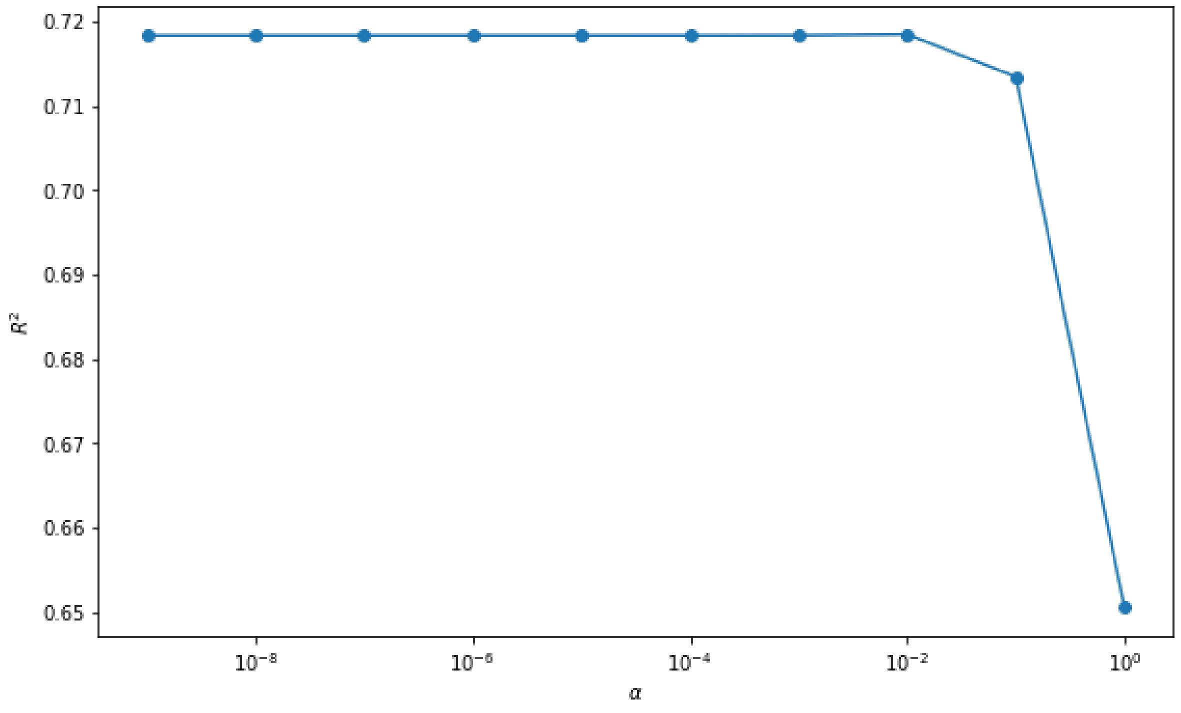
In [23]:
```
Lasso(alpha=1e-6).fit(X, y).coef_
```

Out[23]:
```
array([-1.07170372e-01,  4.63952623e-02,  2.08588308e-02,  2.68854318e+00,
       -1.77954207e+01,  3.80475296e+00,  7.50802707e-04, -1.47575348e+00,
        3.05654279e-01, -1.23293755e-02, -9.53459908e-01,  9.39253013e-03,
       -5.25467196e-01])
```

In [24]:
```
Lasso(alpha=1.0).fit(X, y).coef_
```

Out[24]:
```
array([-0.06342255,  0.04916867, -0.        ,  0.        , -0.        ,
        0.94678567,  0.02092737, -0.66900864,  0.26417501, -0.01520915,
       -0.72319901,  0.00829117, -0.76143296])
```

In [25]:
```
plt.figure(figsize=(10,6))
plt.semilogx(alphas, scores, '-o')
plt.xlabel('$\\alpha$')
plt.ylabel('$R^2$');
```



Exercise

Add `PolynomialFeatures` to this `Pipeline`, and re-run the cross validation with the `PolynomialFeatures` added.

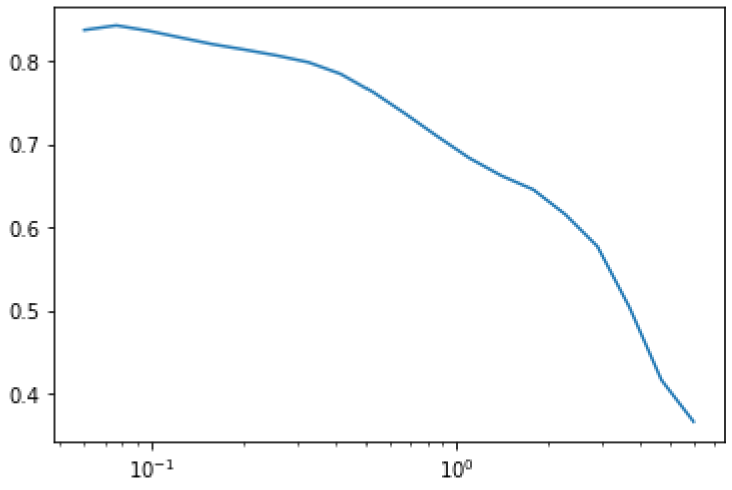
Hint #1: pipelines process input from first to last. Think about the order that it would make sense to add Polynomial Features to the data in sequence and add them in the appropriate place in the pipeline.

Hint #2: you should see a significant increase in cross validation accuracy from doing this

In [26]:
```
pf = PolynomialFeatures(degree=3)

scores = []
alphas = np.geomspace(0.06, 6.0, 20)
for alpha in alphas:
    las = Lasso(alpha=alpha, max_iter=100000)

    estimator = Pipeline([
        ("scaler", s),
        ("make_higher_degree", pf),
        ("lasso_regression", las)])

    predictions = cross_val_predict(estimator, X, y, cv = kf)

    score = r2_score(y, predictions)

    scores.append(score)
```

If you store the results in a list called `scores` , the following will work:

In [27]:
```
plt.semilogx(alphas, scores);
```



In [28]:
```
# Once we have found the hyperparameter (alpha~1e-2=0.01)
# make the model and train it on ALL the data
# Then release it into the wild .....
best_estimator = Pipeline([
                    ("scaler", s),
                    ("make_higher_degree", PolynomialFeatures(degree=2)),
                    ("lasso_regression", Lasso(alpha=0.03))])

best_estimator.fit(X, y)
best_estimator.score(X, y)
```

Out[28]:
```
0.9134777735196521
```

In [29]:
```
best_estimator.named_steps["lasso_regression"].coef_
```

Out[29]:
```
array([ 0.00000000e+00, -0.00000000e+00, -0.00000000e+00, -0.00000000e+00,
        0.00000000e+00, -1.00309168e+00,  3.32679107e+00, -1.01840878e+00,
       -2.56161421e+00,  1.12778302e+00, -1.72266155e+00, -5.37088506e-01,
        4.39555878e-01, -3.39542586e+00,  7.22387712e-02,  0.00000000e+00,
        0.00000000e+00,  3.53653554e+00, -0.00000000e+00,  3.72285440e-01,
        0.00000000e+00,  0.00000000e+00, -5.49528703e-01, -0.00000000e+00,
       -0.00000000e+00, -4.05522485e-02,  2.25864611e-01,  1.78508858e-01,
        0.00000000e+00,  0.00000000e+00,  0.00000000e+00,  6.50874606e-02,
       -0.00000000e+00, -2.07295802e-01, -0.00000000e+00,  3.71781995e-01,
        0.00000000e+00, -0.00000000e+00, -5.89531100e-02,  3.47180625e-01,
        0.00000000e+00,  9.23666274e-01,  3.48873365e-01,  7.29463442e-02,
        0.00000000e+00,  0.00000000e+00,  7.68485586e-02, -7.21083596e-01,
        0.00000000e+00, -5.98542558e-01,  4.18420677e-01, -7.98165728e-01,
       -7.25062683e-01,  2.34818861e-01, -0.00000000e+00, -0.00000000e+00,
        0.00000000e+00, -1.68164447e-02,  0.00000000e+00, -4.04477826e-01,
       -4.22989874e-01, -4.06983988e-01, -3.75443720e-01,  4.17684564e-01,
       -8.91841193e-01,  0.00000000e+00, -2.69309481e-01,  0.00000000e+00,
        1.02286785e-01,  2.02570379e-01, -6.88345376e-01, -0.00000000e+00,
       -1.08598703e+00, -3.98751731e-01, -9.37684760e-01, -1.17343147e-01,
       -7.37427594e-01,  0.00000000e+00,  0.00000000e+00,  1.36340670e+00,
       -0.00000000e+00, -2.94691228e-03, -8.98125013e-01, -8.68198373e-01,
        8.03396788e-01, -1.91683803e-01, -1.14706070e-01,  0.00000000e+00,
       -0.00000000e+00,  5.83161589e-01, -0.00000000e+00,  5.81365491e-02,
        0.00000000e+00, -2.32896159e-01, -1.12440837e+00,  0.00000000e+00,
        1.96286997e+00, -0.00000000e+00, -1.00915801e+00, -7.04656486e-02,
       -1.06456357e-02, -4.78389591e-02, -3.97645601e-01, -3.84121840e-01,
        9.97402419e-01])
```

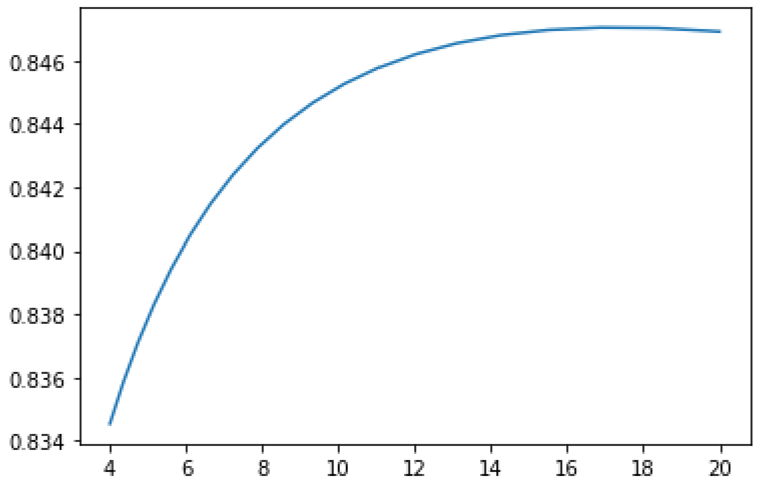
Exercise

Do the same, but with `Ridge` regression

Which model, `Ridge` or `Lasso`, performs best with its optimal hyperparameters on the Boston dataset?

In [30]:
```
pf = PolynomialFeatures(degree=2)
alphas = np.geomspace(4, 20, 20)
scores=[]
for alpha in alphas:
    ridge = Ridge(alpha=alpha, max_iter=100000)

    estimator = Pipeline([
        ("scaler", s),
        ("polynomial_features", pf),
        ("ridge_regression", ridge)])

    predictions = cross_val_predict(estimator, X, y, cv = kf)
    score = r2_score(y, predictions)
    scores.append(score)

plt.plot(alphas, scores)
```

Out[30]:
```
[<matplotlib.lines.Line2D at 0x1e2cb734ca0>]
```



Conclusion: Both Lasso and Ridge with proper hyperparameter tuning give better results than plain ol' Linear Regression!

Exercise:

Now, for whatever your best overall hyperparameter was:

- Standardize the data
- Fit and predict on the entire dataset
- See what the largest coefficients were
  - Hint: use

    ```
    dict(zip(model.coef_, pf.get_feature_names()))
    ```

    for your model `model` to get the feature names from `PolynomialFeatures` .

    Then, use

```
dict(zip(list(range(len(X.columns.values))), X.columns.value
s))
```

to see which features in the `PolynomialFeatures` DataFrame correspond to which columns in the original DataFrame

In [31]:
```
# Once we have found the hyperparameter (alpha~1e-2=0.01)
# make the model and train it on ALL the data
# Then release it into the wild .....
best_estimator = Pipeline([
                    ("scaler", s),
                    ("make_higher_degree", PolynomialFeatures(degree=2)),
                    ("lasso_regression", Lasso(alpha=0.03))])

best_estimator.fit(X, y)
best_estimator.score(X, y)
```

Out[31]:
```
0.9134777735196521
```

In [32]:
```
df_importances = pd.DataFrame(zip(best_estimator.named_steps["make_higher_degr
                 best_estimator.named_steps["lasso_regression"].coef_,
))
```

```
C:\Users\fresh\.ipython\lib\site-packages\sklearn\utils\deprecation.py:87: Fu
tureWarning: Function get_feature_names is deprecated; get_feature_names is d
eprecated in 1.0 and will be removed in 1.2. Please use get_feature_names_out
instead.
  warnings.warn(msg, category=FutureWarning)
```

In [33]:
```
col_names_dict = dict(zip(list(range(len(X.columns.values))), X.columns.values
```

In [34]:
```
col_names_dict
```

Out[34]:
```
{0: 'CRIM',
 1: 'ZN',
 2: 'INDUS',
 3: 'CHAS',
 4: 'NOX',
 5: 'RM',
 6: 'AGE',
 7: 'DIS',
 8: 'RAD',
 9: 'TAX',
 10: 'PTRATIO',
 11: 'B',
 12: 'LSTAT'}
```

In [35]:
```
df_importances.sort_values(by=1)
```

Out[35]:

	0	1
13	x12	-3.395426
8	x7	-2.561614
10	x9	-1.722662
94	x8 x12	-1.124408
72	x5 x8	-1.085987
...	...	...
9	x8	1.127783
79	x6 x8	1.363407
96	x9 x10	1.962870
6	x5	3.326791
17	x0 x3	3.536536

105 rows × 2 columns

Grid Search CV

To do cross-validation, we used two techniques:

- use `KFolds` and manually create a loop to do cross-validation
- use `cross_val_predict` and `score` to get a cross-valiated score in a couple of lines.

To do hyper-parameter tuning, we see a general pattern:

- use `cross_val_predict` and `score` in a manually written loop over hyperparemeters, then select the best one.

Perhaps not surprisingly, there is a function that does this for us -- `GridSearchCV`

In [36]:
```
from sklearn.model_selection import GridSearchCV

# Same estimator as before
estimator = Pipeline([("scaler", StandardScaler()),
        ("polynomial_features", PolynomialFeatures()),
        ("ridge_regression", Ridge())])

params = {
    'polynomial_features__degree': [1, 2, 3],
    'ridge_regression__alpha': np.geomspace(4, 20, 30)
}

grid = GridSearchCV(estimator, params, cv=kf)
```

In [37]:
```
grid.fit(X, y)
```

Out[37]:
```
GridSearchCV(cv=KFold(n_splits=3, random_state=0, shuffle=True),
             estimator=Pipeline(steps=[('scaler', StandardScaler()),
                                       ('polynomial_features',
                                        PolynomialFeatures()),
                                       ('ridge_regression', Ridge())]),
             param_grid={'polynomial_features__degree': [1, 2, 3],
                         'ridge_regression__alpha': array([ 4.        ,  4.22
826702,  4.46956049,  4.7246238 ,  4.99424274,
        5.27924796,  5.58051751,  5.89897953,  6.23561514,  6.59146146,
        6.96761476,  7.36523392,  7.78554391,  8.22983963,  8.69948987,
        9.19594151,  9.72072404, 10.27545421, 10.86184103, 11.48169104,
       12.13691388, 12.82952815, 13.56166768, 14.33558803, 15.15367351,
       16.01844446, 16.93256509, 17.89885162, 18.92028098, 20.        ])})
```

In [38]:
```
grid.best_score_, grid.best_params_
```

Out[38]:
```
(0.8440355370010505,
 {'polynomial_features__degree': 2,
  'ridge_regression__alpha': 17.898851619528912})
```

In [39]:
```
y_predict = grid.predict(X)
```

In [40]:
```
# This includes both in-sample and out-of-sample
r2_score(y, y_predict)
```

Out[40]:
```
0.9133201649751171
```

In [41]:
```
# Notice that "grid" is a fit object!
# We can use grid.predict(X_test) to get brand new predictions!
grid.best_estimator_.named_steps['ridge_regression'].coef_
```

Out[41]:
```
array([ 0.00000000e+00, -1.36995040e-01,  1.48120344e-03, -2.59640942e-02,
        9.82450498e-02, -9.23822591e-01,  3.23846438e+00, -9.11308293e-01,
       -1.84973837e+00,  7.93515177e-01, -9.73033037e-01, -7.25208114e-01,
        5.25945136e-01, -2.99813007e+00,  7.42896889e-02,  7.22139400e-02,
        6.32439949e-02,  1.56367307e+00, -4.53124204e-01,  5.46170910e-01,
        1.38200377e-01,  2.42087300e-01, -7.02784220e-01,  1.50951069e-01,
       -6.44769753e-02, -7.97208467e-02,  4.44675181e-01,  2.18993842e-01,
       -1.36932568e-01,  1.55572301e-01,  2.02020325e-01,  5.76220122e-02,
       -1.00239072e-01, -2.51066703e-01, -1.19775873e-01,  6.42902392e-01,
        6.71870989e-02,  8.24223060e-02, -2.24146135e-01,  5.57286164e-01,
        1.14389782e-01,  8.65941718e-01,  5.18441279e-01,  5.28384958e-01,
        6.44472041e-01, -4.02916380e-02,  6.61966211e-02, -4.14677934e-01,
        2.07347928e-01, -6.83413254e-01,  3.33620470e-01, -9.51849594e-01,
       -7.98805286e-01,  2.84586880e-01,  2.39179119e-02,  1.16152012e-01,
        6.21769359e-01, -1.23067171e-01, -1.16893645e-01, -3.81180298e-01,
       -4.43716890e-01, -4.80963281e-01, -4.45000767e-01,  7.06453835e-01,
       -7.83111601e-01,  4.84151226e-02, -5.32570928e-01, -4.86347169e-02,
        4.98882955e-01,  2.37540863e-01, -6.74847861e-01,  4.85207164e-02,
       -7.89562157e-01, -8.56242562e-01, -1.02823439e+00, -1.53176130e-01,
       -7.58134135e-01,  1.40880036e-01,  6.23345898e-02,  1.00263024e+00,
        1.11326251e-01, -5.88406946e-02, -9.03212579e-01, -1.09758541e+00,
        9.54106785e-01, -5.28338600e-01, -3.62672288e-01, -1.09970042e-01,
       -2.56391200e-01,  8.10507274e-01, -7.71444624e-01,  8.24891601e-01,
        3.88828740e-01, -1.08060361e-01, -1.07653612e+00,  4.22306322e-02,
        1.22114643e+00, -3.45664962e-01, -1.05915567e+00, -1.29522118e-01,
        7.19130984e-02, -7.43072088e-03, -3.64001332e-01, -4.17182150e-01,
        8.07844379e-01])
```

In [42]:
```
grid.cv_results_
```

```
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_ridge_regression__alpha': masked_array(data=[4.0, 4.22826701576941
6, 4.4695604891609,
                    4.724623797826311, 4.994242741567055,
                    5.279247963228449, 5.58051750774668, 5.898979527232258,
                    6.235615140423803, 6.591461455321584,
                    6.9676147643129305, 7.365233921633089,
```

Summary

1. We can manually generate folds by using `KFolds`
2. We can get a score using `cross_val_predict(X, y, cv=KFoldObject_or_integer)` . This will produce the out-of-bag prediction for each row.
3. When doing hyperparameter selection, we should be optimizing on out-of-bag scores. This means either using `cross_val_predict` in a loop, or ....
4. .... use `GridSearchCV` . GridSearchCV takes a model (or pipeline) and a dictionary of parameters to scan over. It finds the hyperparameter set that has the best out-of-sample score on all the parameters, and calls that it's "best estimator". It then retrains on all data with the "best" hyper-parameters.

Extensions

Here are some additional items to keep in mind:

- There is a `RandomSearchCV` that tries random combination of model parameters. This can be helpful if you have a prohibitive number of combinations to test them all exhaustively.
- KFolds will randomly select rows to be in the training and test folds. There are other methods (such as `StratifiedKFolds` and `GroupKFold` , which are useful when you need more control over how the data is split (e.g. to prevent data leakage). You can create these specialized objects and pass them to the `cv` argument of `GridSearchCV` .