CrossValidation

April 3, 2023

Cross Validation

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

[1]:
```
# Surpress warnings:
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn

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
```

[2]:
```
import requests

# Note we are loading a slightly different ("cleaned") pickle file
URL = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/
 ↪IBM-ML240EN-SkillsNetwork/labs/data/boston_housing_clean.pickle'

response = requests.get(URL)
boston = pickle.loads(response.content)
```

[3]:
```
boston.keys()
```

[3]:
```
dict_keys(['dataframe', 'description'])
```

[4]:
```
boston_data = boston['dataframe']
boston_description = boston['description']

boston_data.head()
```

[4]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	\
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	

	PTRATIO	B	LSTAT	MEDV
0	15.3	396.90	4.98	24.0
1	17.8	396.90	9.14	21.6
2	17.8	392.83	4.03	34.7
3	18.7	394.63	2.94	33.4
4	18.7	396.90	5.33	36.2

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?

Answer:

Split the data into three folds: Fold 1, Fold 2, and Fold 3.

For each fold:

Train the model on the other two folds.

Evaluate the model on the current fold and record the mean squared error.

Calculate the average mean squared error across all three folds to get an estimate of how well the model will perform on data outside the dataset.

Repeat the process with different train-test splits to get a more reliable estimate.

It's worth noting that scikit-learn has a cross_val_score function that automates this process, making it much simpler to perform cross-validation. Or KFold can be used to achieve the goal.

Coding this up:

The `KFold` object in SciKit Learn tells the cross validation object (see below) how to split up the data:

[5]:
```
X = boston_data.drop('MEDV', axis=1)
y = boston_data.MEDV

kf = KFold(shuffle=True, random_state=72018, n_splits=3)
```

```
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: [ 1  3  4  5  7  8 10 11 12 13] 337
Test index: [ 0  2  6  9 15 17 19 23 25 26] 169

Train index: [ 0  2  6  9 10 11 12 13 15 17] 337
Test index: [ 1  3  4  5  7  8 14 16 22 27] 169

Train index: [0 1 2 3 4 5 6 7 8 9] 338
Test index: [10 11 12 13 18 20 21 24 28 31] 168
```

[6]:
```
#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
```

[6]:
```
[0.6719348798472737, 0.7485020059212378, 0.6976807323597766]
```

→ **A bit cumbersome, but do-able.** Discussion:

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?

Coding this up:

[7]:
```
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)

scores
```

[7]:
```
[0.6719348798472715, 0.748502005921238, 0.6976807323597745]
```

→ (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.

[8]:
```
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 specify parameters at that step.

[9]:
```
estimator = Pipeline([("scaler", s),
                      ("regression", lr)])
```

cross_val_predict

cross_val_predict is a function that does K-fold cross validation for us, appropriately fitting and transforming at every step of the way.

[10]:
```
kf
```

[10]:
```
KFold(n_splits=3, random_state=72018, shuffle=True)
```

[11]:
```
predictions = cross_val_predict(estimator, X, y, cv=kf)

r2_score(y, predictions)
```

[11]:
```
0.7063531064161561
```

[12]:
```
np.mean(scores) # almost identical!
```

[12]:
```
0.7060392060427613
```

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` 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:

[13]:
```
alphas = np.geomspace(1e-9, 1e0, num=10)
alphas
```

[13]:
```
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.

[14]:
```
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)


list(zip(alphas,scores))
```

[14]:
```
[(1e-09, 0.7063531064981925),
 (1e-08, 0.7063531072356071),
 (1e-07, 0.7063531145602442),
 (1e-06, 0.7063531882052063),
 (1e-05, 0.7063539165191507),
 (0.0001, 0.706361268093463),
 (0.001, 0.706433467041546),
 (0.01, 0.7070865958083233),
 (0.1, 0.705838151167185),
 (1.0, 0.6512724532884888)]
```

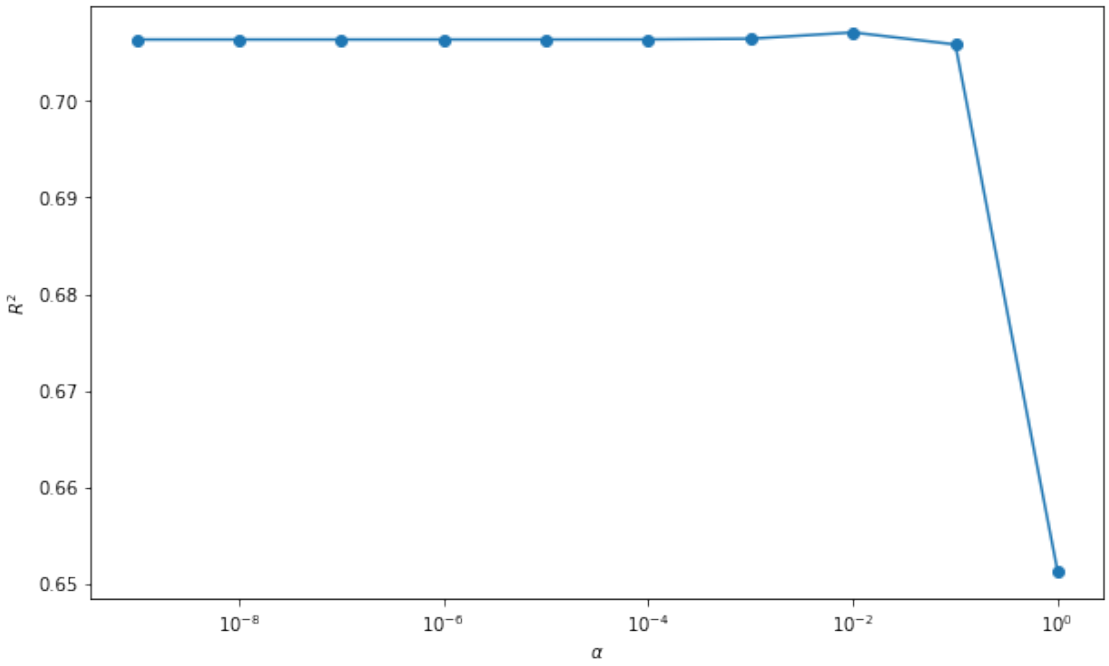
[15]:
```
Lasso(alpha=1e-6).fit(X, y).coef_
```

[15]:
```
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])
```

[16]:
```
Lasso(alpha=1.0).fit(X, y).coef_
```

[16]:
```
array([-0.06342255,  0.04916867, -0.        ,  0.        , -0.        ,
        0.94678567,  0.02092737, -0.66900864,  0.26417501, -0.01520915,
       -0.72319901,  0.00829117, -0.76143296])
```

[17]:
```
plt.figure(figsize=(10,6))
plt.semilogx(alphas, scores, '-o')
plt.xlabel('$\\alpha$')
plt.ylabel('$R^2$');
plt.show()
```



Exercise

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

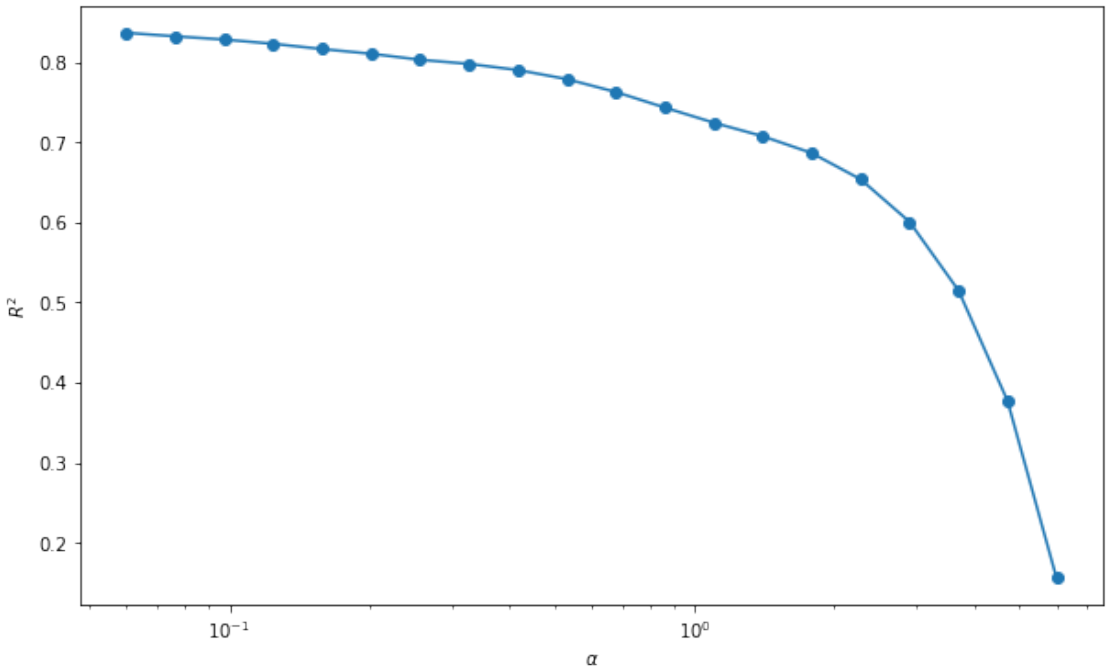
Hint #1: pipelines process input from first to last. Think about the order that it would make sense to add Polynomial Features to the data in sequence and add them in the appropriate place in the pipeline. Hint #2: you should see a significant increase in cross validation accuracy from doing this

given code snippet:

```
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)
```

[18]:
```
pf = PolynomialFeatures(degree=3)

scores_lasso = []
alphas = np.geomspace(0.06, 6.0, 20)

for alpha in alphas:
    las = Lasso(alpha=alpha, max_iter=100000)

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

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

    score = r2_score(y, predictions)

    scores_lasso.append(score)

plt.figure(figsize=(10,6))
plt.semilogx(alphas, scores_lasso, '-o')
plt.xlabel('$\\alpha$')
plt.ylabel('$R^2$');
```



[19]:
```
# 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_lasso = Pipeline([
                    ("scaler", s),
                    ("make_higher_degree", PolynomialFeatures(degree=2)),
                    ("lasso_regression", Lasso(alpha=0.03))])

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

[19]:
```
0.9134777735196521
```

[20]:
```
best_estimator_lasso.named_steps["lasso_regression"].coef_
```

[20]:
```
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,
```

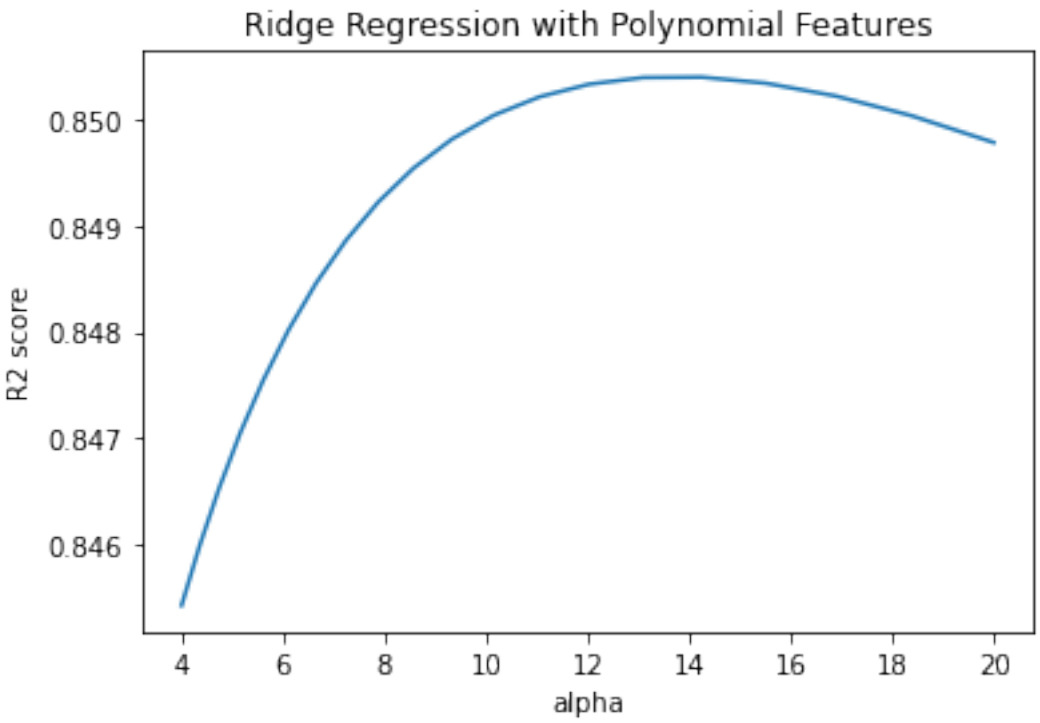
```
 0.00000000e+00,  9.23666274e-01,  3.48873365e-01,  7.29463442e-02,
 0.00000000e+00,  0.00000000e+00,  7.68485586e-02, -7.21083596e-01,
 0.00000000e+00, -5.98542558e-01,  4.18420677e-01, -7.98165728e-01,
-7.25062683e-01,  2.34818861e-01, -0.00000000e+00, -0.00000000e+00,
 0.00000000e+00, -1.68164447e-02,  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?

[21]:
```
pf = PolynomialFeatures(degree=2)
alphas = np.geomspace(4, 20, 20)
scores_ridge = []

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_ridge.append(score)

plt.plot(alphas, scores_ridge)
plt.xlabel('alpha')
plt.ylabel('R2 score')
plt.title('Ridge Regression with Polynomial Features')
plt.show()
```



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:

  `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.values))`

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

[22]:
```
# Extract features and target variable
X = boston_data.drop('MEDV', axis=1)
y = boston_data.MEDV

# Standardize the data
s = StandardScaler()
X_std = s.fit_transform(X)

# Fit and predict on the entire dataset
pf = PolynomialFeatures(degree=2)
X_poly = pf.fit_transform(X_std)
lasso = Lasso(alpha=0.01, max_iter=100000)
lasso.fit(X_poly, y)
y_pred = lasso.predict(X_poly)

# Get the largest coefficients
coef_dict = dict(zip(pf.get_feature_names(), lasso.coef_))
print("Largest coefficients:")
for feature, coef in sorted(coef_dict.items(), key=lambda x: abs(x[1]),␣
 ↪reverse=True)[:10]:
    print(f"{feature}: {coef}")

# Map the features in the PolynomialFeatures DataFrame to the original DataFrame
original_feature_names = dict(zip(list(range(len(X.columns.values))), X.columns.
 ↪values))
coef_dict_mapped = {}
for feature, coef in coef_dict.items():
    feature_num = [int(s) for s in feature if s.isdigit()][0]
    original_feature_name = original_feature_names[feature_num]
    coef_dict_mapped[original_feature_name] = coef
print()
print("Largest coefficients mapped to original features:")
for feature, coef in sorted(coef_dict_mapped.items(), key=lambda x: abs(x[1]),␣
 ↪reverse=True)[:10]:
    print(f"{feature}: {coef}")
```

```
Largest coefficients:
x8^2: -4.991080527810556
x0 x3: 4.603071935785574
x8 x9: 4.240135996998103
x5: 3.4310177264232076
x12: -3.3062917492013835
x7: -2.8095906306308196
x2 x4: 2.3613798041858605
x6 x8: 2.025348594855447
x9 x10: 1.8804440396423487
```

```
x7 x8: -1.461145090718384

Largest coefficients mapped to original features:
RAD: -1.4313647323025422
AGE: -1.2512836094769035
TAX: -1.1151009056641341
RM: -0.9133769723234367
CRIM: 0.8848308972022697
ZN: 0.821859317290605
DIS: 0.8069487079514248
INDUS: -0.604969283887453
NOX: 0.5918337042974662
CHAS: -0.43820943823461894
```

If you want to have a dataframe az a result, let's call it `output_df`, then:

[23]:
```
# Standardize the data
s = StandardScaler()
X_std = s.fit_transform(X)

# Fit and predict on the entire dataset
pf = PolynomialFeatures(degree=2)
X_poly = pf.fit_transform(X_std)
ridge = Ridge(alpha=10, max_iter=100000)
ridge.fit(X_poly, y)
predictions = ridge.predict(X_poly)

# Get feature names from PolynomialFeatures
poly_names = pf.get_feature_names(X.columns)

# Create a dictionary mapping polynomial features to original features
feature_map = dict(zip(list(range(len(poly_names))), poly_names))

# Get coefficients and sort them in descending order
coef_dict = dict(zip(poly_names, ridge.coef_))
sorted_coefs = sorted(coef_dict.items(), key=lambda x: abs(x[1]), reverse=True)

# Create output DataFrame
output_df = pd.DataFrame(columns=['Original Feature', 'Coefficient'])

# Fill output DataFrame with largest coefficients and their corresponding␣
 ↪original features
for name, coef in sorted_coefs:
    if abs(coef) > 0:
        output_df = output_df.append({
            'Original Feature': feature_map[poly_names.index(name)],
            'Coefficient': coef
        }, ignore_index=True)
```

```
# Sort output DataFrame by coefficient
output_df = output_df.sort_values(by='Coefficient', key=lambda x: abs(x))
```

[24]:
```
output_df
```

[24]:
```
    Original Feature  Coefficient
103         CHAS RAD    -0.001890
102       ZN PTRATIO     0.010431
101    PTRATIO LSTAT     0.026827
100               ZN    -0.028282
99             ZN RM     0.037275
..               …          …
4        TAX PTRATIO     1.491081
3          CRIM CHAS     2.193068
2                DIS    -2.223542
1              LSTAT    -3.101785
0                 RM     3.310039

[104 rows x 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`

[25]:
```
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)
```

[26]:
```
grid.fit(X, y)
```

[26]:
```
GridSearchCV(cv=KFold(n_splits=3, random_state=72018, 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.        ,
4.22826702,  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.        ])})
```

[27]:
```
grid.best_score_, grid.best_params_
```

[27]:
```
(0.8504982950750941,
 {'polynomial_features__degree': 2,
  'ridge_regression__alpha': 15.153673507519274})
```

[28]:
```
y_predict = grid.predict(X)
```

[29]:
```
# This includes both in-sample and out-of-sample
r2_score(y, y_predict)
```

[29]:
```
0.9149145594213685
```

[30]:
```
# 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_
```

[30]:
```
array([ 0.00000000e+00, -1.27346408e-01, -6.16205046e-03,  2.36135244e-02,
        1.00398027e-01, -9.74110586e-01,  3.26236441e+00, -9.65057238e-01,
       -1.96344725e+00,  8.56769182e-01, -1.01488960e+00, -7.06985966e-01,
        5.52029222e-01, -3.03254502e+00,  7.74127927e-02,  7.24276605e-02,
        6.82776638e-02,  1.72849044e+00, -4.80758341e-01,  5.76219972e-01,
        1.28132069e-01,  2.22931335e-01, -7.45243542e-01,  1.66582495e-01,
       -8.00025634e-02, -8.54571642e-02,  5.07490801e-01,  2.14820391e-01,
       -1.48833274e-01,  1.42098626e-01,  1.93770221e-01,  5.02304885e-02,
       -1.12667821e-01, -2.77559685e-01, -1.32870713e-01,  7.32239658e-01,
        5.26857333e-02,  8.89966580e-02, -2.72228558e-01,  5.84383917e-01,
        1.06306947e-01,  9.62971619e-01,  5.76845132e-01,  5.33378179e-01,
        7.07913980e-01, -6.21760626e-02,  7.57641545e-02, -4.28157866e-01,
        2.40651011e-01, -6.82201736e-01,  3.40931549e-01, -9.62217889e-01,
```

```
       -8.14997204e-01,  2.81353294e-01,  5.50023518e-02,  8.65917517e-02,
        6.28285056e-01, -1.40764851e-01, -1.03645734e-01, -3.81965497e-01,
       -4.48817407e-01, -4.46562934e-01, -4.97293983e-01,  7.52862844e-01,
       -8.00745322e-01,  7.86779267e-02, -5.78298566e-01, -4.98398516e-02,
        5.37001246e-01,  2.24913740e-01, -7.11059542e-01,  5.70498060e-02,
       -7.85214394e-01, -9.18516132e-01, -1.02907666e+00, -1.58937491e-01,
       -7.77699453e-01,  1.42895792e-01,  7.72299871e-02,  1.08239035e+00,
        3.98859145e-02, -7.26596891e-02, -9.64695031e-01, -1.12682105e+00,
        1.01829108e+00, -6.12786851e-01, -4.22714073e-01, -1.41672983e-01,
       -2.68672373e-01,  8.23071041e-01, -8.66106901e-01,  8.83695240e-01,
        3.63975663e-01, -1.13200717e-01, -1.12043738e+00,  2.19170412e-03,
        1.30087563e+00, -3.65505003e-01, -1.08425883e+00, -1.16852284e-01,
        8.62081670e-02,  1.40937541e-03, -3.62535906e-01, -4.04519520e-01,
        8.07960994e-01])
```

[31]:
```
#uncomment the below to see the results:
#grid.cv_results_
```

[32]:
```
pd.DataFrame(grid.cv_results_)
```

[32]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_polynomial_features__degree	param_ridge_regression__alpha	params	split0_test_score	\
0	0.011029	0.003403	0.004717	0.002882	1	4.0			
1	0.018386	0.006377	0.009937	0.005541	1	4.228267			
2	0.010855	0.003746	0.000000	0.000000	1	4.46956			
3	0.007255	0.000626	0.000000	0.000000	1	4.724624			
4	0.019450	0.016028	0.000000	0.000000	1	4.994243			
..	…	…	…	…	…	…			
85	0.021551	0.003443	0.008580	0.001046	3	16.018444			
86	0.025060	0.000631	0.002396	0.002712	3	16.932565			
87	0.020997	0.003408	0.008572	0.002008	3	17.898852			
88	0.023123	0.002735	0.008417	0.000534	3	18.920281			
89	0.025741	0.004662	0.006690	0.000441	3	20.0			

```
0   {'polynomial_features__degree': 1, 'ridge_regr…          0.672111
1   {'polynomial_features__degree': 1, 'ridge_regr…          0.672103
2   {'polynomial_features__degree': 1, 'ridge_regr…          0.672093
3   {'polynomial_features__degree': 1, 'ridge_regr…          0.672081
4   {'polynomial_features__degree': 1, 'ridge_regr…          0.672067
..                                                  …              …
85  {'polynomial_features__degree': 3, 'ridge_regr…          0.595188
86  {'polynomial_features__degree': 3, 'ridge_regr…          0.599857
87  {'polynomial_features__degree': 3, 'ridge_regr…          0.604457
88  {'polynomial_features__degree': 3, 'ridge_regr…          0.608986
89  {'polynomial_features__degree': 3, 'ridge_regr…          0.613441

    split1_test_score  split2_test_score  mean_test_score  std_test_score  \
0            0.748235           0.701801         0.707382        0.031327
1            0.748207           0.701986         0.707432        0.031307
2            0.748175           0.702178         0.707482        0.031286
3            0.748141           0.702375         0.707533        0.031265
4            0.748104           0.702579         0.707583        0.031243
..                …                  …                …               …
85           0.802959           0.588724         0.662290        0.099503
86           0.806434           0.599156         0.668482        0.097547
87           0.809709           0.609271         0.674479        0.095643
88           0.812795           0.619075         0.680286        0.093789
89           0.815700           0.628578         0.685907        0.091986

    rank_test_score
0                60
1                59
2                58
3                57
4                56
..              …
85               65
86               64
87               63
88               62
89               61

[90 rows x 13 columns]
```

Summary/review

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`.

[ ]: