mlp-week04

February 9, 2021

1 Machine Learning in Python - Workshop 4

1.1 1. Setup

1.1.1 1.1 Packages

In the cell below we will load the core libraries we will be using for this workshop and setting some sensible defaults for our plot size and resolution.

[1]:
```
# Display plots inline
%matplotlib inline

# Data libraries
import pandas as pd
import numpy as np

# Plotting libraries
import matplotlib.pyplot as plt
import seaborn as sns

# Plotting defaults
plt.rcParams['figure.figsize'] = (8,5)
plt.rcParams['figure.dpi'] = 80

# sklearn modules
import sklearn
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import make_pipeline
```

1.1.2 1.2 Data

To start we will again be using the same data set from Workshop 3, which was generated via a random draw from a Gaussian Process model. It represent an unknown smooth function $f(x)$ that is observed with noise such that $y_i = f(x_i) + \epsilon_i$. The data have been randomly thinned to include only 100 observations.

We can read the data in from `gp.csv` and plot the data to see the overall trend in the data,

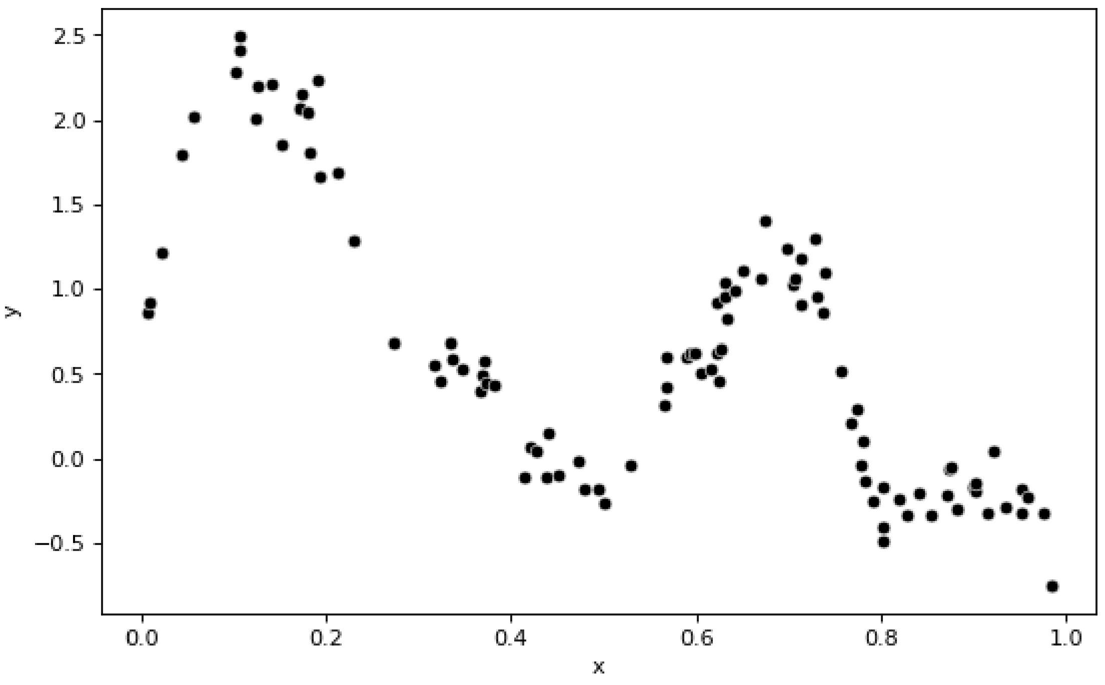
```
[2]: d = pd.read_csv("gp.csv")
     n = d.shape[0] # number of rows

     sns.scatterplot(x='x', y='y', data=d, color="black")

     d
```

```
[2]:            x         y
     0   0.006209  0.863336
     1   0.009349  0.916748
     2   0.022680  1.218414
     3   0.043324  1.797545
     4   0.057116  2.016364
     ..       …         …
     95  0.950810 -0.323806
     96  0.952195 -0.184413
     97  0.958802 -0.232287
     98  0.974495 -0.320891
     99  0.983324 -0.751002

     [100 rows x 2 columns]
```



2 2. Cross validation

In this section we will explore some of the tools that sklearn provides for cross validation for the purpose of model evaluation and selection. The most basic form of CV is to split the data into a testing and training set, this can be achieved using `train_test_split` from the `model_selection` submodule. Here we provide the function with our model matrix X and outcome vector y to obtain a test and train split of both.

[3]:
```
from sklearn.model_selection import train_test_split

X = np.c_[d.x]
y = d.y

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=0)
```

The additional arguments `test_size` determines the proportion of data to include in the test set and `random_state` is the seed used when determining the partition assignments (keeping the seed the same will result in the same partition(s) each time this cell is rerun).

We can check the dimensions of the original and new objects using their shape attributes,

[4]:
```
print("orig sizes :", X.shape, y.shape)
print("train sizes:", X_train.shape, y_train.shape)
print("test sizes :", X_test.shape, y_test.shape)
```

```
orig sizes : (100, 1) (100,)
train sizes: (80, 1) (80,)
test sizes : (20, 1) (20,)
```

With these new objects we can try several polynomial regression models, with different values of `M`, and compare their performance. Our goal is to fit the models using the training data and then evaluating their performance using the test data so that we can avoid potential overfitting.

We will assess the models' performance using root mean squared error,

$$\text{rmse} = \left(\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \right)^{1/2}$$

with sklearn this is calculated using the `mean_squared_error` function with the argument `squared=False`. This metric is entirely equivalent to mean squared error for purposes of ranking / ordering models (as the square root is a monotonic transformation) but the rmse is often prefered as it is more interpretable, as it has the same units as y.

The following code uses a `for` loop to fit 30 polynomial regression models with $M = [1, 2, \ldots, 30]$ and calculates the rmse of the training data and the testing data for each model. Note that we are fitting the model *only* using the `train` split and predicting using both the `train` and the `test` split to get the train and test rmses respectively.

[5]:
```
degree = []
train_rmse = []
test_rmse = []

M = 30

for i in np.arange(1, M+1):
    m = make_pipeline(
        PolynomialFeatures(degree=i),
        LinearRegression(fit_intercept=False)
    ).fit(X_train, y_train)

    degree.append(i)
    train_rmse.append(mean_squared_error(y_train, m.predict(X_train),␣
 ↪squared=False))
    test_rmse.append(mean_squared_error(y_test, m.predict(X_test),␣
 ↪squared=False))

fit = pd.DataFrame(data = {"degree": degree, "train_rmse": train_rmse,␣
 ↪"test_rmse": test_rmse})
fit
```

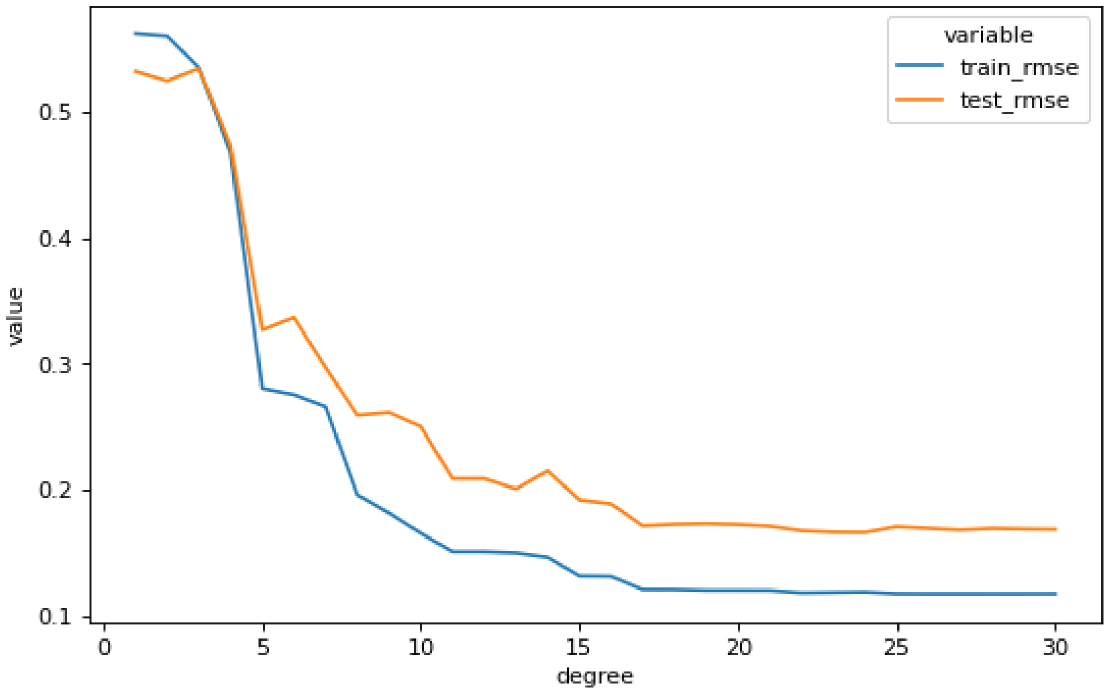
[5]:
```
    degree  train_rmse  test_rmse
0        1    0.562112   0.532058
1        2    0.560099   0.524406
2        3    0.534969   0.534363
3        4    0.467536   0.473563
4        5    0.280606   0.326807
5        6    0.275643   0.336668
6        7    0.266056   0.297164
7        8    0.196140   0.259276
8        9    0.181609   0.261407
9       10    0.165666   0.250260
10      11    0.151061   0.209072
11      12    0.151058   0.209093
12      13    0.150081   0.200831
13      14    0.146542   0.215181
14      15    0.131695   0.192004
15      16    0.131431   0.188881
16      17    0.121015   0.171482
17      18    0.120960   0.172690
18      19    0.120291   0.173085
19      20    0.120278   0.172530
20      21    0.120226   0.171152
21      22    0.118344   0.167641
22      23    0.118652   0.166502
23      24    0.118999   0.166288
```

```
24      25    0.117620   0.170805
25      26    0.117550   0.169481
26      27    0.117566   0.168197
27      28    0.117549   0.169394
28      29    0.117565   0.168931
29      30    0.117601   0.168673
```

We can then plot `degree` vs `rmse` for these splits and observe the resulting pattern.

```
[6]: sns.lineplot(x="degree", y="value", hue="variable", data = pd.
      ↪melt(fit,id_vars=["degree"]))
```

```
[6]: <matplotlib.axes._subplots.AxesSubplot at 0x7febe32496d0>
```



2.0.1 Exercise 1

Based on these results, what value of M produces the best model, justify your answer.

M = 17 provides the lowest train_rmse and is the simplest model with the low rmse. However, one could argue that M=11 is much simpler and only increases the rmse a little

---

2.0.2 Exercise 2

Try adjusting the proportion of the data in the test vs training data, how does this change the Training and Testing rmse curves?

We try: test_size = 0.2 test_Size = 0.1 test_size = 0.5 test_Size = 0.75

test_size = 0.2 is our baseline.

test_size = 0.1 provides similar cuves to the original test size, however the test_rmse is worse in this case being slightly above 0.2.

With test_size = 0.5, the train curve is similar to the original, however the test_rmse is huge (over 50 for degree 17+)

2.1 2.1 k-fold cross validation

The basic implementation of k-fold cross validation is provided by `KFold` in the `model_selection` submodule of `sklearn`. This function / object behaves in a similar way to the other `sklearn` tools we've seen before, we use the function to construct an object with some basic options and the resulting object then can be used to interact with our data / model matrix.

[7]:
```
from sklearn.model_selection import KFold
```

[8]:
```
kf = KFold(n_splits=5)
```

Here we have set up the object `kf` to implement 5-fold cross validation for our data, which we can then apply using the `split` method on our model matrix `X` and response vector `y`.

[9]:
```
kf.split(X, y)
```

[9]:
```
<generator object _BaseKFold.split at 0x7febed4bdc50>
```

This returns a generator which is then used to generate the indexes of the training data and test data for each fold. We can use list comprehension with this generator to view these values.

[10]:
```
[train for train, test in kf.split(X, y)]

# This gives all the training sets exlcuding a certain range of indexes which␣
↪become the test set
```

[10]:
```
[array([20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
        37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53,
        54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
        71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87,
        88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]),
 array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53,
        54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
        71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87,
        88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]),
 array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
        34, 35, 36, 37, 38, 39, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
        71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87,
        88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]),
 array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
        34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
        51, 52, 53, 54, 55, 56, 57, 58, 59, 80, 81, 82, 83, 84, 85, 86, 87,
        88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]),
 array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
        34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
        51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
        68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79])]
```

[11]:
```
[test for train, test in kf.split(X, y)]

# this gives each test set
```

[11]:
```
[array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
         17, 18, 19]),
 array([20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
         37, 38, 39]),
 array([40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56,
         57, 58, 59]),
 array([60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76,
         77, 78, 79]),
 array([80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96,
         97, 98, 99])]
```

2.1.1 Exercise 3

Examine the index values that make up the test and training sets, do you notice anything intesting about how `KFold` has divided up the data?

kFold has divided the data into five uniform folds (0-19,20-39,40-59,60-79,80-99)

2.1.2 Exercise 4

Does the structure in Exercise 3 have any implications for model evaluation? Specifically, are the data in the different folds independent of each other, explain why this is important.

The data are not independent - that is, not selected randomly. Once we build our model on the training set we then validate on the test set. When carrying out validation in this case, the model is always being validated over a local region of the functions (One of [0,0.2],[0.2,0.4],[0.4,0.6][0.6,0.8] or [0.8,1]) and so the complexity is likely to be selected to match the local conditions of the data at that point rather than over the whole curve.

Using a `for` loop with the `KFold` generator it is then possible to create test and train subsets, fit the model of interest and calculate the training and testing metrics but this requires a fair bit of book keeping code to implement, see an example of this here.

However, `sklearn` provides a more convenient way of doing all of this using the `cross_val_score` function from the `model_selection` submodule. This function takes as arguments our model or pipeline (any object implementing `fit`) and then our full model matrix X and response y. The argument `cv` is a cross-validation generator that determines the splitting strategy (e.g. a `KFold` object).

```
[12]: from sklearn.model_selection import cross_val_score

model = make_pipeline(
    PolynomialFeatures(degree=1),
    LinearRegression(fit_intercept=False)
)

# Use shuffle to avoid the issue seen w/ Ex. 3 & 4
# random_state again sets a random seed so we get the same results each time␣
 ↪this cell is run
kf = KFold(n_splits=5, shuffle=True, random_state=0)
cross_val_score(model, X, y, cv=kf, scoring="neg_root_mean_squared_error")

#sorted(sklearn.metrics.SCORERS.keys())
```

```
[12]: array([-0.53205832, -0.49736481, -0.66842376, -0.54219565, -0.570626  ])
```

Here we have used `"neg_root_mean_squared_error"` as our scoring metric which returns the negative of the root mean squared error. As the name implies this returns the negative of the usual fit metric, this is because sklearn expects to always optimize for the maximum of a score and the model with the largest negative rmse will therefore be the "best". To get a list of all available scoring metrics for `cross_val_score` you can run `sorted(sklearn.metrics.SCORERS.keys())`.

To obtain these 5-fold CV estimates of rmse for our models we slightly modify our original code as follows,

[13]:
```
# Original Code
#degree = []
#train_rmse = []
#test_rmse = []

#M = 30

#for i in np.arange(1, M+1):
#    m = make_pipeline(
#        PolynomialFeatures(degree=i),
#        LinearRegression(fit_intercept=False)
#    ).fit(X_train, y_train)
#
#    degree.append(i)
#    train_rmse.append(mean_squared_error(y_train, m.predict(X_train),␣
 ↪squared=False))
#    test_rmse.append(mean_squared_error(y_test, m.predict(X_test),␣
 ↪squared=False))

#fit = pd.DataFrame(data = {"degree": degree, "train_rmse": train_rmse,␣
 ↪"test_rmse": test_rmse})
#fit

degree = []
test_mean_rmse = []
test_rmse = []

M = 30
kf = KFold(n_splits=5, shuffle=True, random_state=0)

for i in np.arange(1,M+1):
    model = make_pipeline(
        PolynomialFeatures(degree=i),
        LinearRegression(fit_intercept=False)
    )
    cv = -1 * cross_val_score(model, X, y, cv=kf,␣
 ↪scoring="neg_root_mean_squared_error")
    degree.append(i)
    test_mean_rmse.append(np.mean(cv))
    test_rmse.append(cv)

cv = pd.DataFrame(
    data = np.c_[degree, test_mean_rmse, test_rmse],
    columns = ["degree", "mean_rmse"] + ["fold" + str(i) for i in range(1,6) ]
)
```

This fits 5×30 polynomial regression models for and stores the results in the data frame `cv`. The mean_rmse column contains the average rmse across all 5 folds.

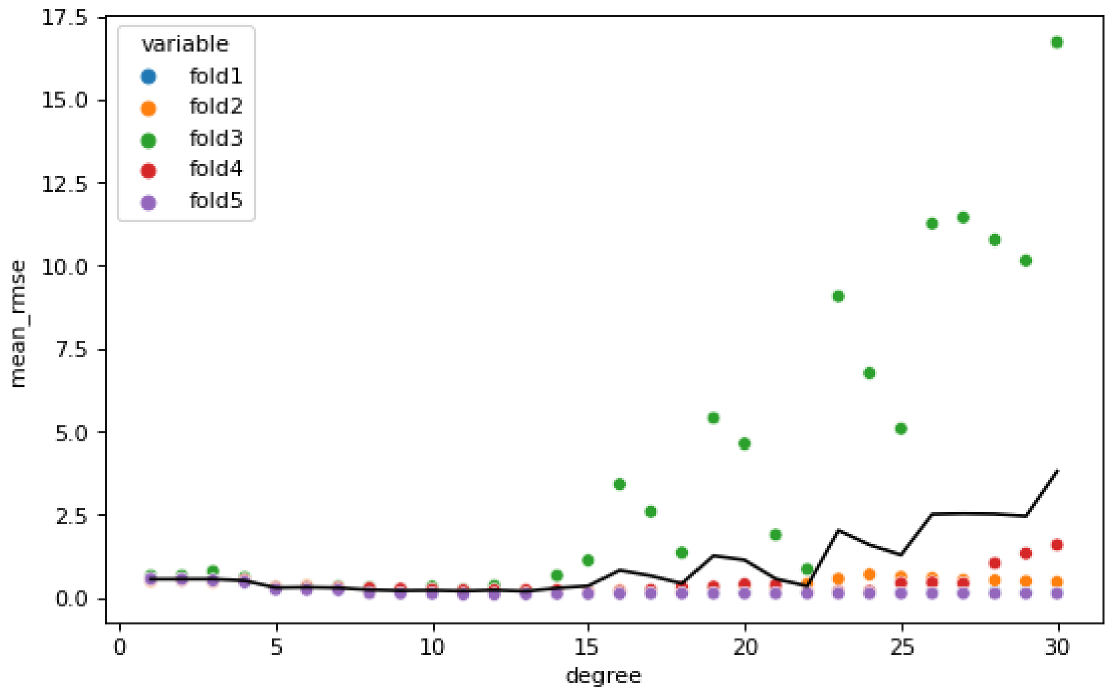
```
[14]: cv.head(n=15)
```

```
[14]:     degree  mean_rmse     fold1     fold2     fold3     fold4     fold5
      0      1.0   0.562134  0.532058  0.497365  0.668424  0.542196  0.570626
      1      2.0   0.563810  0.524406  0.523596  0.670822  0.537015  0.563210
      2      3.0   0.565059  0.534363  0.469241  0.800306  0.499975  0.521408
      3      4.0   0.518660  0.473563  0.502625  0.616382  0.531899  0.468832
      4      5.0   0.303684  0.326807  0.331068  0.312491  0.305007  0.243048
      5      6.0   0.316684  0.336668  0.371119  0.325595  0.310843  0.239193
      6      7.0   0.299734  0.297164  0.320832  0.346016  0.304593  0.230064
      7      8.0   0.239039  0.259276  0.190588  0.323880  0.288796  0.132657
      8      9.0   0.215289  0.261407  0.205739  0.216862  0.270135  0.122302
      9     10.0   0.224875  0.250260  0.170494  0.336424  0.248162  0.119036
      10    11.0   0.204303  0.209072  0.237949  0.253485  0.222947  0.098061
      11    12.0   0.231847  0.209093  0.260421  0.364024  0.227382  0.098314
      12    13.0   0.196166  0.200831  0.233899  0.210455  0.228865  0.106781
      13    14.0   0.290809  0.215181  0.224600  0.670841  0.227332  0.116092
      14    15.0   0.357732  0.192005  0.175511  1.126737  0.179234  0.115174
```

We can now plot these data, to assess the different models and their performance on fitting these data.

```
[15]: sns.lineplot(x="degree", y="mean_rmse", data = cv, color="black")
      sns.scatterplot(x="degree", y="value", hue="variable", data = pd.
       ↪melt(cv,id_vars=["degree", "mean_rmse"]))
```

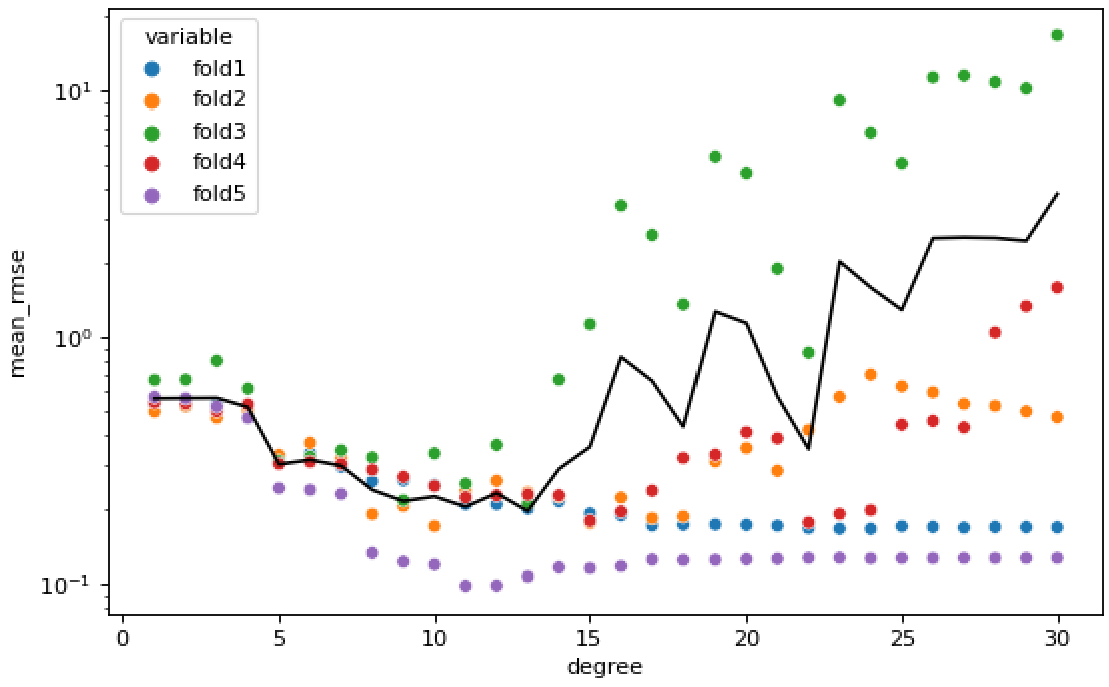
```
[15]: <matplotlib.axes._subplots.AxesSubplot at 0x7febe3274110>
```



The plot above is a bit hard to read and compare, particularly for the lower degree polynomial models. We can address this by using a log scale for the y-axis.

```
[16]: g = sns.lineplot(x="degree", y="mean_rmse", data = cv, color="black")
      g = sns.scatterplot(x="degree", y="value", hue="variable", data = pd.
       ↪melt(cv,id_vars=["degree", "mean_rmse"]))

      g.set_yscale("log")
```



2.1.3 Exercise 5

Do these CV rmse's agree with the results we obtained when using `train_test_split`? How do they differ, is it only a single fold that differs or several?

The CV rmse's in this case are are neither consistently worse or better than those of train_split_test. The third fold appears to have a high rmse in this case even as the degree increases. The first and fifth folds have the lowest rmse.

2.1.4 Exercise 6

Based on these results, what value of M do you think produces the best model, justify your answer.

M = 13 provides the lowest rmse and I would select this

2.2 2.2 CV Grid Search

We can further reduce the amount of code needed if we wish to test over a specific set of parameter values using cross validation. This is accomplished by using the `GridSearchCV` function from the `model_selection` submodule.

This function works similarly to `cross_val_score`, but with the addition of the `param_grid` argument. This argument is a dictionary containing parameters names as keys and lists of parameter settings to try as values. Since we are using a pipeline, out parameter name will be the name of the pipeline step, `polynomialfeatures`, followed by `__`, and then the parameter name, `degree`. So for our pipeline the parameter is named `polynomialfeatures__degree`. If you want to list any models available parameters you can call the `get_params()` method on the model object, e.g. `m.get_params()` here.

[17]:
```
from sklearn.model_selection import GridSearchCV

m = make_pipeline(
        PolynomialFeatures(),
        LinearRegression(fit_intercept=False)
    )

parameters = {
    'polynomialfeatures__degree': np.arange(1,31,1)
}

kf = KFold(n_splits=5, shuffle=True, random_state=23)

grid_search = GridSearchCV(m, parameters, cv=kf,␣
 ↪scoring="neg_root_mean_squared_error").fit(X, y)
```

The above code goes through the process of fitting all 5×30 models as well as storing and ranking the results for the requested scoring metric(s).

Once all of the submodels are fit, we can determine the optimal hyperparameter value by accessing the object's `best_*` attributes,

[18]:
```
print("best index: ", grid_search.best_index_)
print("best param: ", grid_search.best_params_)
print("best score: ", grid_search.best_score_)
```

```
best index:  16
best param:  {'polynomialfeatures__degree': 17}
best score:  -0.17318516961675368
```

Here we see that this proceedure has selected the model with polynomial degree 13 as having the best fit, and this model achieved an average rmse of 0.196.

Additional useful details from the CV process are available in the `cv_results_` attribute, which provides CV and scoring details,

[19]:
```
grid_search.cv_results_["mean_test_score"]
```

[19]:
```
array([-0.57656638, -0.5779595 , -0.56984964, -0.50283095, -0.30669383,
       -0.31328966, -0.29045491, -0.22415395, -0.21455427, -0.1987302 ,
       -0.19236843, -0.19997391, -0.18966052, -0.1874897 , -0.18487839,
       -0.20260268, -0.17318517, -0.20855725, -0.24852888, -0.34336519,
       -0.31273923, -0.30124072, -0.35851221, -0.40414278, -0.32779843,
       -0.34665604, -0.36023223, -0.49118591, -0.51144025, -0.50160664])
```

[20]:
```
grid_search.cv_results_["split0_test_score"]
```

[20]:
```
array([-0.46058144, -0.45226649, -0.41115292, -0.45194596, -0.27593217,
       -0.28173729, -0.27114171, -0.2220431 , -0.20151204, -0.1833081 ,
       -0.14851918, -0.15521502, -0.15188302, -0.1537881 , -0.12647497,
       -0.13751026, -0.13449278, -0.14019395, -0.13767707, -0.13765795,
       -0.1379832 , -0.13699473, -0.13762329, -0.1381811 , -0.13531093,
       -0.13461968, -0.1337368 , -0.14124783, -0.14118608, -0.14111581])
```

Finally, the `best_estimator_` attribute gives direct access to the "best" model (pipeline) object. Which allows for direct inspection of model coefficients, make predictions, etc.

[21]:
```
grid_search.best_estimator_
```

[21]:
```
Pipeline(steps=[('polynomialfeatures', PolynomialFeatures(degree=17)),
                ('linearregression', LinearRegression(fit_intercept=False))])
```

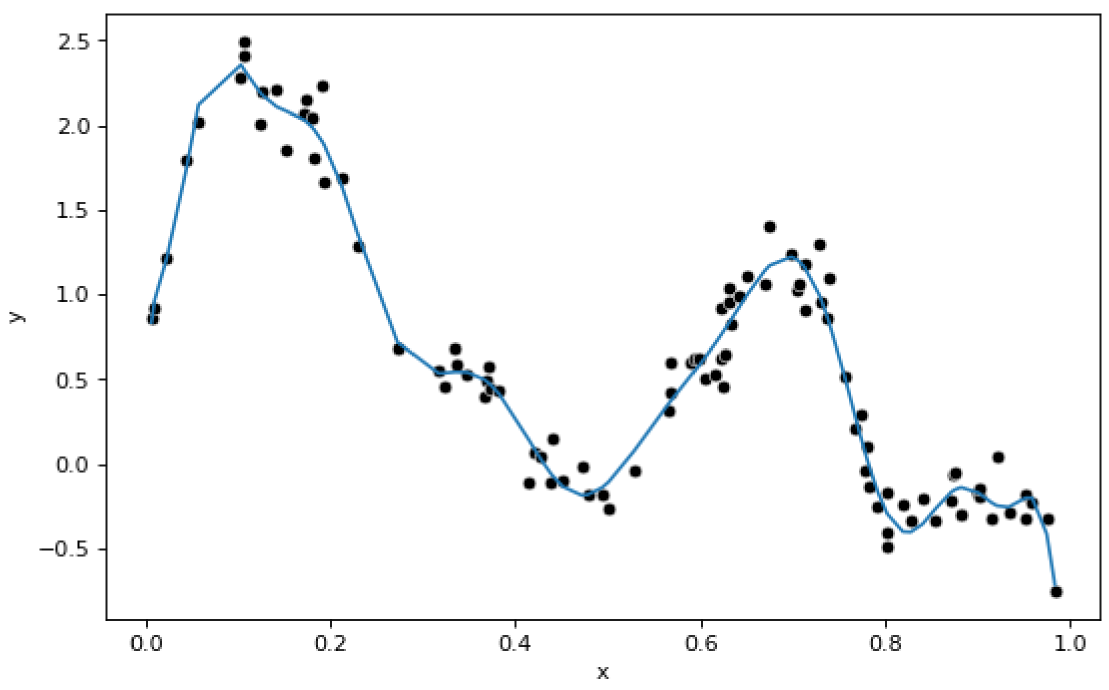
[22]:
```
grid_search.best_estimator_.named_steps['linearregression'].coef_
```

[22]:
```
array([ 3.33371719e-01,  1.19303801e+02, -7.87391325e+03,  2.93765503e+05,
       -5.92239887e+06,  7.22477088e+07, -5.79455806e+08,  3.22409402e+09,
       -1.28885261e+10,  3.78543087e+10, -8.27263671e+10,  1.35086263e+11,
       -1.64094670e+11,  1.46106362e+11, -9.25715905e+10,  3.95028910e+10,
       -1.01747522e+10,  1.19483224e+09])
```

For example if we wanted to plot this "best" model's fit to the data we can use the object's predict method directly.

[23]:
```
sns.scatterplot(x='x', y='y', data=d, color="black")
sns.lineplot(
    x=d.x,
    y=grid_search.best_estimator_.predict(X)
)
```

[23]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7febe317ce90>
```



2.2.1 Exercise 7

Do these results appear to be “robust”? More specifically, do you think that the degree 13 polynomial model is actually the best? If you were to change the `random_state` used for the shuffling of the folds, will this result change?

I beleive that the degree 13 polynomial is indeed the best quantitatively. It may be contested that a lower degree can be selected without substantially decreasing the mean rmse and so one may justify selecting a model of say degree 9 with mean_rmse of 0.215289 which is only 0.02 higher than the degree 14 polynomial.

If we change the random_state, say to 42 then the best model is of degree 11. If we change the random state to

---

3 3. Additional dimensions and CV

Let us now consider an additive regression model of the following form,

$$y = f(x_1) + g(x_2) + h(x_3) + \epsilon$$

where $f()$, $g()$, and $h()$ are polynomials with fix degrees, we will assume linear, quadratic and cubic in this case with the following coefficients:

$$
\begin{aligned}
f(x) &= 1.2x + 1.1 \\
g(x) &= 2.5x^2 - 0.9x - 3.2 \\
h(x) &= 2x^3 + 0.4x^2 - 5.2x + 2.7
\end{aligned}
$$

We generate values for x_1, x_2, x_3, and ϵ and then use these values to calculate observations of y using the following code.

[24]:
```
np.random.seed(1234)
n = 500

f = lambda x: 1.2 * x + 1.1
g = lambda x: 2.5 * x**2 - 0.9 * x - 3.2
h = lambda x: 2 * x**3 + 0.4 * x**2 - 5.2 * x + 2.7

ex2 = pd.DataFrame({
    "x1": np.random.rand(n),
    "x2": np.random.rand(n),
    "x3": np.random.rand(n)
}).assign(
   y = lambda d: f(d.x1) + g(d.x2) + h(d.x3) + 0.25*np.random.randn(n) # epsilon
)

print(ex2)
```

```
           x1        x2        x3         y
0    0.191519  0.883951  0.401106  0.207226
1    0.622109  0.741361  0.930614 -0.966704
2    0.437728  0.515711  0.515336 -0.899870
3    0.785359  0.135252  0.809582 -1.917369
4    0.779976  0.039884  0.881772 -1.179076
..        …         …         …         …
495  0.267568  0.995838  0.202188  1.562021
496  0.932827  0.944205  0.372405  1.223071
497  0.826145  0.131144  0.173656  0.741909
498  0.145443  0.518355  0.345234 -0.633376
499  0.647004  0.477245  0.567738 -0.833677

[500 rows x 4 columns]
```

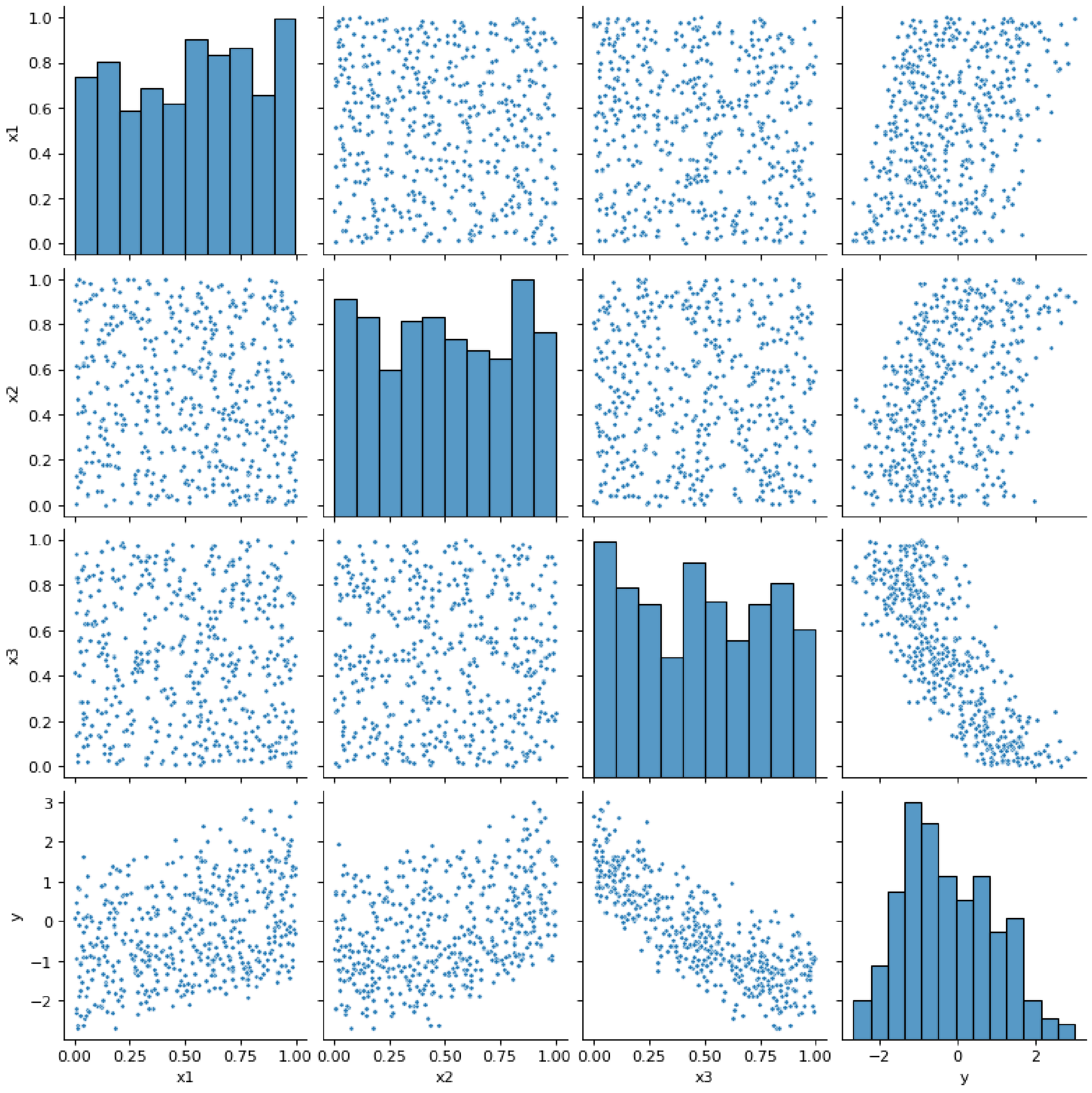
3.0.1 Exercise 8

Create a pairs plot of these data, from this alone is it possible to identify the polynomial relationships between y and the xs?

[25]:
```
sns.pairplot(ex2.dropna(), markers=".")

# It looks like there are linear relationships between y and each of the xs,
# however we cannot determine the relationships precisely
# The ys also look normally distributed
```

[25]:
```
<seaborn.axisgrid.PairGrid at 0x7febe2ecbc90>
```



We will assume that we know that each of the functions $f()$, $g()$, and $h()$ are at most of degree 3 - we can try to fit a polynomial model to these data using the tools we've seen thus far.

[26]:
```
X = ex2.drop(columns=['y']) # Keep as a data frame not a nparray
y = ex2.y

print(X, y)

# The model is trying to predict bi such that the following is correct
# y = b1 x1^3 + b2 x1^2 + b3 x1 + b4 + b5 x2^3 + b6 x2^2 + b7 x2 + b8 + b9 x3^3␣
↪+ b10 x3^2 + b11 x3 + b12 + b13
#      + a bunch of other powers
```

```
           x1        x2        x3
0    0.191519  0.883951  0.401106
1    0.622109  0.741361  0.930614
2    0.437728  0.515711  0.515336
3    0.785359  0.135252  0.809582
4    0.779976  0.039884  0.881772
..        …         …         …
495  0.267568  0.995838  0.202188
496  0.932827  0.944205  0.372405
497  0.826145  0.131144  0.173656
498  0.145443  0.518355  0.345234
499  0.647004  0.477245  0.567738

[500 rows x 3 columns] 0      0.207226
1     -0.966704
2     -0.899870
3     -1.917369
4     -1.179076
          …
495    1.562021
496    1.223071
497    0.741909
498   -0.633376
499   -0.833677
Name: y, Length: 500, dtype: float64
```

[27]:
```
m = make_pipeline(
    PolynomialFeatures(degree=3),
    LinearRegression(fit_intercept=False)
)

fit = m.fit(X, y)
```

The following line prints out the model coefficients for the fitted model. While we may have expected only 10 (or 9) parameters in this model, we instead get 20 coefficients for this model.

[28]:
```
print(fit.named_steps['linearregression'].coef_)
```

```
[ 0.28199407  2.32656568  0.20117018 -4.55915257 -1.10824194 -1.16081456
 -0.98352703  0.83509629 -0.48174047 -0.32581289  0.32341883  0.46312716
  0.55649515  0.53131633  0.12888933  0.34799729  0.66614765  0.52534563
  0.03664093  2.2074557 ]
```

The reason for this becomes more clear if we examine the `powers_` attribute which returns lists of the powers of x_1, x_2, and x_3 for each of the model terms. So for example the coeficient 0.282 belongs to the $x_1^0 x_2^0 x_3^0$ term (i.e. the model intercept).

[30]:
```
print(fit.named_steps['polynomialfeatures'].powers_ )

#a = fit.named_steps['polynomialfeatures'].powers_

#for i in a:
#    count = 0
#    if i[0] == i[1] and i[1] == i[2] and i[1] == 0:
#        print(1, end = '')
#    for j in i:
#        if j != 0:
#            print('x{:d}^{:d}'.format(count,j),end='')
#        count += 1
    #print('x1^{:d} + x2^{:d} + x3^{:d}'.format(i[0],i[1],i[2]), end='')
#    print()

#print('a={:d}, b={:d}'.format(f(x,n),g(x,n)))
```

```
[[0 0 0]
 [1 0 0]
 [0 1 0]
 [0 0 1]
 [2 0 0]
 [1 1 0]
 [1 0 1]
 [0 2 0]
 [0 1 1]
 [0 0 2]
 [3 0 0]
 [2 1 0]
 [2 0 1]
 [1 2 0]
 [1 1 1]
 [1 0 2]
 [0 3 0]
 [0 2 1]
 [0 1 2]
 [0 0 3]]
```

3.0.2 Exercise 10

Compare the fitted coefficients with the "true" values for our data, how close are they?

The true values of our coefficients are:

$$\begin{aligned} f(x) &= 1.2x + 1.1 \\ g(x) &= 2.5x^2 - 0.9x - 3.2 \\ h(x) &= 2x^3 + 0.4x^2 - 5.2x + 2.7 \end{aligned}$$

But the model preditions are:

[31]:
```
print(fit.named_steps['linearregression'].coef_)
```

```
[ 0.28199407  2.32656568  0.20117018 -4.55915257 -1.10824194 -1.16081456
 -0.98352703  0.83509629 -0.48174047 -0.32581289  0.32341883  0.46312716
  0.55649515  0.53131633  0.12888933  0.34799729  0.66614765  0.52534563
  0.03664093  2.2074557 ]
```

[32]:
```
print(fit.named_steps['polynomialfeatures'].powers_ )
```

```
[[0 0 0]
 [1 0 0]
 [0 1 0]
 [0 0 1]
 [2 0 0]
 [1 1 0]
 [1 0 1]
 [0 2 0]
 [0 1 1]
 [0 0 2]
 [3 0 0]
 [2 1 0]
 [2 0 1]
 [1 2 0]
 [1 1 1]
 [1 0 2]
 [0 3 0]
 [0 2 1]
 [0 1 2]
 [0 0 3]]
```

We can examine the coefficients to decide if they are similar. Firstly, our model includes the extra powers which are not represented in the true data. Therefore, we can say that the model is not accurate in this regard. Let's look at the indidual powers, The model predicts the coefficient of x_1 to be 2.33, differing from the true value 1.2. The model predicts the coefficient of x_2^2 to be -1.11, differing from the true value 2.5. The model predicts the coefficient of x_2 to be 0.20, differing from the true value of -0.9. The model predicts the coefficient of x_3^3 to be 0.32, differing from the true value of 2. The model predicts the coefficient of x_3^2 to be -0.33, differing from the true value of 0.4. The model predicts the coefficient of x_3^1 to be -4.56, differing from the true value of -5.2.

```
[ ]:
```

3.0.3 Exercise 11

Calculate the 5-fold cross validation rmse for this model.

```
[33]: kf2 = KFold(n_splits=5, shuffle=True,random_state=0)

      test_mean_rmse = []
      test_rmse = []

      cv = -1 * cross_val_score(m, X, y, cv=kf2,␣
       ↪scoring="neg_root_mean_squared_error")

      print(cv)
      print(cv.mean())
```

```
[0.26798009 0.25624857 0.23095929 0.24516022 0.267326  ]
0.253534831926424
```

3.1 3.2 Column Transformers

For this particular model we do not want these interaction terms between our features, but this is not something that `sklearn` allows us to disable for `PolynomialFeatures`. This is actually a specific example of a more general issue where we do not want to apply a transformer to all of the features of a model at the same time. For this particularly example, we would like to apply an individual polynomial transformation to each of the three x. To do this we will use sklearn's `ColumnTransformer` and the `make_column_transformer` helper function from the `compose` submodule.

```
[34]: from sklearn.compose import ColumnTransformer, make_column_transformer
```

```
[35]: ind_poly = make_column_transformer(
          (PolynomialFeatures(degree=3, include_bias=False), ['x1']),
          (PolynomialFeatures(degree=3, include_bias=False), ['x2']),
          (PolynomialFeatures(degree=3, include_bias=False), ['x3']),
      )
```

The arguments for `make_column_transformer` are tuples of the desired transformer object and the name of the columns that will have that transformer applied. Once constructed the column transfomer works like any other transformer object and can be applied via `fit` and `transform` or `fit_transform` methods.

```
[36]: trans = ind_poly.fit_transform(X, y)
```

```
[37]: pd.DataFrame(trans) # printing as a DataFrame makes the array more readable
```

```
[37]:            0         1         2         3         4         5         6  \
     0    0.191519  0.036680  0.007025  0.883951  0.781370  0.690693  0.401106
     1    0.622109  0.387019  0.240768  0.741361  0.549615  0.407463  0.930614
     2    0.437728  0.191606  0.083871  0.515711  0.265958  0.137157  0.515336
     3    0.785359  0.616788  0.484400  0.135252  0.018293  0.002474  0.809582
     4    0.779976  0.608362  0.474508  0.039884  0.001591  0.000063  0.881772
     ..        …         …         …         …         …         …         …
     495  0.267568  0.071593  0.019156  0.995838  0.991694  0.987567  0.202188
     496  0.932827  0.870166  0.811714  0.944205  0.891522  0.841779  0.372405
     497  0.826145  0.682516  0.563857  0.131144  0.017199  0.002256  0.173656
     498  0.145443  0.021154  0.003077  0.518355  0.268692  0.139278  0.345234
     499  0.647004  0.418614  0.270845  0.477245  0.227763  0.108699  0.567738

                 7         8
     0    0.160886  0.064533
     1    0.866043  0.805952
     2    0.265571  0.136859
     3    0.655423  0.530619
     4    0.777522  0.685598
     ..        …         …
     495  0.040880  0.008265
     496  0.138686  0.051647
     497  0.030156  0.005237
     498  0.119186  0.041147
     499  0.322327  0.182997

     [500 rows x 9 columns]
```

`ColumnTransformers` are like `pipelines` but they include a specific column or columns for the transformer to be applied. By using this transformer we take each feature and apply a single polynomial feature transformer, of degree 3 (excluding the intercept column (bias)), resulting in 9 total features as output (3 for each input feature). We can check these values make sense by examining them along with the original values of the xs. Here we are using `include_bias=False` to avoid creating a rank deficient model matrix, which would result if all three polynomial features transforms included the same intercept column.

```
[38]: pd.concat([X, pd.DataFrame(trans)], axis=1)
```

[38]:
```
           x1        x2        x3         0         1         2         3  \
0    0.191519  0.883951  0.401106  0.191519  0.036680  0.007025  0.883951
1    0.622109  0.741361  0.930614  0.622109  0.387019  0.240768  0.741361
2    0.437728  0.515711  0.515336  0.437728  0.191606  0.083871  0.515711
3    0.785359  0.135252  0.809582  0.785359  0.616788  0.484400  0.135252
4    0.779976  0.039884  0.881772  0.779976  0.608362  0.474508  0.039884
..        …         …         …         …         …         …         …
495  0.267568  0.995838  0.202188  0.267568  0.071593  0.019156  0.995838
496  0.932827  0.944205  0.372405  0.932827  0.870166  0.811714  0.944205
497  0.826145  0.131144  0.173656  0.826145  0.682516  0.563857  0.131144
498  0.145443  0.518355  0.345234  0.145443  0.021154  0.003077  0.518355
499  0.647004  0.477245  0.567738  0.647004  0.418614  0.270845  0.477245

            4         5         6         7         8
0    0.781370  0.690693  0.401106  0.160886  0.064533
1    0.549615  0.407463  0.930614  0.866043  0.805952
2    0.265958  0.137157  0.515336  0.265571  0.136859
3    0.018293  0.002474  0.809582  0.655423  0.530619
4    0.001591  0.000063  0.881772  0.777522  0.685598
..        …         …         …         …         …
495  0.991694  0.987567  0.202188  0.040880  0.008265
496  0.891522  0.841779  0.372405  0.138686  0.051647
497  0.017199  0.002256  0.173656  0.030156  0.005237
498  0.268692  0.139278  0.345234  0.119186  0.041147
499  0.227763  0.108699  0.567738  0.322327  0.182997

[500 rows x 12 columns]
```

A ColumnTransformer is like any other transformer and can therefore be included in a pipeline, this enables us to create a pipeline for fitting our desired polynomial regression model (with no interaction terms). Since the polynomial features no longer include an intercept, we can add this back to the model with fit_intercept=True in the linear regression step.

[39]:
```python
m2 = make_pipeline(
    make_column_transformer(
        (PolynomialFeatures(degree=3, include_bias=False), ['x1']),
        (PolynomialFeatures(degree=3, include_bias=False), ['x2']),
        (PolynomialFeatures(degree=3, include_bias=False), ['x3']),
    ),
    LinearRegression(fit_intercept=True)
)

fit = m2.fit(X, y)
```

We can examine the fitted values of the coefficients by accessing the linearregression step and its coef_ and intercept_ attributes.

[40]:
```python
fit.named_steps['linearregression'].coef_
```

```
[40]: array([ 1.59319047, -0.60142411,  0.30713487, -0.47485212,  1.45500567,
         0.61273406, -4.97657888,  0.05251414,  2.07979504])
```

```
[41]: fit.named_steps['linearregression'].intercept_
```

```
[41]: 0.5121273713472774
```

Instead of directly fitting, we can also use this pipeline with cross validation functions like `cross_val_score` to obtain a more reliable estimate of our model's rmses.

```
[42]: cv = -1 * cross_val_score(m2, X, y, cv=5, scoring="neg_root_mean_squared_error")

      print(cv)
      print(cv.mean())
```

```
[0.26494573 0.27297603 0.23030777 0.22129478 0.2645753 ]
0.25081992382431384
```

3.1.1 Exercise 12

Is this rmse better or worse than the rmse calculated for the original model that included interactions? Explain why you think this is.

The rmse is slightly better than in the original model including interactions.

Since the data is derived from functions f,g and h that do not include any interacting terms, this model that removes the interacting terms is more similar to the functions from which we derived the data. Hence the modified model will more closely match f,g and h. We could likely improve the rmse further if we altered the degrees of each model within our model.

3.2 3.3 Column Transformers & CV Grid Search

Finally we will see if we can come close to recovering the original forms of $f()$, $g()$, and $h()$ using `GridSearchCV`. This builds on our previous use of this function, but now we need to optimize over the degree parameter of all three of the polynomial feature transformers. We can examine the names of these transforms by examining the `named_transformers_` attribute associated with the `columntransformer`,

```
[43]: m2.named_steps['columntransformer'].named_transformers_
```

```
[43]: {'polynomialfeatures-1': PolynomialFeatures(degree=3, include_bias=False),
       'polynomialfeatures-2': PolynomialFeatures(degree=3, include_bias=False),
       'polynomialfeatures-3': PolynomialFeatures(degree=3, include_bias=False)}
```

This gives us the transformer names: `polynomialfeatures-1`, `polynomialfeatures-2`, and `polynomialfeatures-3` which are referenced in the same way by combining the step name with the transformer name and then the parameter name separated by `__`. As such, the degree parameter for the first transformer will be `columntransformer__polynomialfeatures-1__degree`. It is also possible to view all of the parameters for a model or pipeline by looking at the keys returns by the `get_params` method.

[44]:
```
m2.get_params().keys()
```

[44]:
```
dict_keys(['memory', 'steps', 'verbose', 'columntransformer',
'linearregression', 'columntransformer__n_jobs', 'columntransformer__remainder',
'columntransformer__sparse_threshold', 'columntransformer__transformer_weights',
'columntransformer__transformers', 'columntransformer__verbose',
'columntransformer__polynomialfeatures-1',
'columntransformer__polynomialfeatures-2',
'columntransformer__polynomialfeatures-3',
'columntransformer__polynomialfeatures-1__degree',
'columntransformer__polynomialfeatures-1__include_bias',
'columntransformer__polynomialfeatures-1__interaction_only',
'columntransformer__polynomialfeatures-1__order',
'columntransformer__polynomialfeatures-2__degree',
'columntransformer__polynomialfeatures-2__include_bias',
'columntransformer__polynomialfeatures-2__interaction_only',
'columntransformer__polynomialfeatures-2__order',
'columntransformer__polynomialfeatures-3__degree',
'columntransformer__polynomialfeatures-3__include_bias',
'columntransformer__polynomialfeatures-3__interaction_only',
'columntransformer__polynomialfeatures-3__order', 'linearregression__copy_X',
'linearregression__fit_intercept', 'linearregression__n_jobs',
'linearregression__normalize'])
```

To keep the space of parameters being explored reasonable we will restrict the possible value of the degrees parameter to be in $[1, \ldots, 5]$. This may take a little bit as we are now fitting a decently large number of models.

[45]:
```
parameters = {
    'columntransformer__polynomialfeatures-1__degree': np.arange(1,5,1),
    'columntransformer__polynomialfeatures-2__degree': np.arange(1,5,1),
    'columntransformer__polynomialfeatures-3__degree': np.arange(1,5,1),
}

kf = KFold(n_splits=5, shuffle=True, random_state=0)

grid_search = GridSearchCV(m2, parameters, cv=kf,␣
 ↪scoring="neg_root_mean_squared_error").fit(X, y)
```

3.2.1 Exercise 13

How many models have been fit and scored by GridSearchCV?

```
[52]: #grid_search.get_params()

      #grid_search.cv_results_["mean_test_score"]

      #print(parameters)

      grid_search.best_estimator_

      # We are fitting three 5 x 5 models.
```

```
[52]: Pipeline(steps=[('columntransformer',
                       ColumnTransformer(transformers=[('polynomialfeatures-1',
                                                        PolynomialFeatures(degree=1,
include_bias=False),
                                                        ['x1']),
                                                       ('polynomialfeatures-2',
                                                        PolynomialFeatures(degree=3,
include_bias=False),
                                                        ['x2']),
                                                       ('polynomialfeatures-3',
                                                        PolynomialFeatures(degree=3,
include_bias=False),
                                                        ['x3'])])),
                      ('linearregression', LinearRegression())])
```

Once fit, we can determine the optimal parameter value by accessing grid_search's attributes,

```
[47]: print("best index: ", grid_search.best_index_)
      print("best param: ", grid_search.best_params_)
      print("best score: ", grid_search.best_score_)
```

```
best index:  10
best param:  {'columntransformer__polynomialfeatures-1__degree': 1,
'columntransformer__polynomialfeatures-2__degree': 3,
'columntransformer__polynomialfeatures-3__degree': 3}
best score:  -0.24827845229249973
```

3.2.2 Exercise 14

Based on these results have we done a good job of recovering the general structure of the functions $f()$, $g()$, and $h()$? e.g. have we correctly recovered the degrees of these functions.

When we run best estimator above, we can see that the best estimator has a column transformer with degree for x1 of 1, degree for x2 of 3 and degree for x3 of 3. But we know that f has degree 1, g has degree 2 and h has degree 3. So our model has not correctly recovered the degrees of the functions.

We can directly access the properties of the "best" model, according to our scoring method, using the `best_estimator_` attribute. From this we can access the `linearregression` step of the pipeline to recover the model coefficients.

[50]:
```python
grid_search.best_estimator_.named_steps["linearregression"].intercept_
```

[50]:
```
0.547234949864674
```

[56]:
```python
grid_search.best_estimator_.named_steps["linearregression"].coef_
```

[56]:
```
array([ 1.26880491e+00, -4.80356378e-01,  1.47923181e+00,  5.92674399e-01,
       -4.95420711e+00,  2.40266994e-03,  2.11215152e+00])
```

3.2.3 Exercise 15

Compare the coefficient values we obtained via `GridSearchCV` to the true values used to generate the y observations, how well have recovered the truth values of the coefficients?

The true values of our coefficients are:

$$
\begin{aligned}
f(x) &= 1.2x + 1.1 \\
g(x) &= 2.5x^2 - 0.9x - 3.2 \\
h(x) &= 2x^3 + 0.4x^2 - 5.2x + 2.7
\end{aligned}
$$

We can see by comparison that we have not recovered the true values of the parameters very well

---

3.2.4 Bonus Exercise

Repeat the analysis above but use only 200 observations of `ex2` instead of all 500. How does your resulting "best" model change? What about 100 or 50 observations? How dependent are the results on the original sample size?

3.3 4. Competing the worksheet

At this point you have hopefully been able to complete all the preceeding exercises. Now is a good time to check the reproducibility of this document by restarting the notebook's kernel and rerunning all cells in order.

Once that is done and you are happy with everything, you can then run the following cell to generate your PDF and turn it in on gradescope under the `mlp-week04` assignment.

[ ]:
```
#!jupyter nbconvert --to pdf mlp-week04.ipynb
```

[ ]:
```

```