scikit-learn

Lecture 14

Dr. Colin Rundel

scikit-learn

> Scikit-learn is an open source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection, model evaluation, and many other utilities.

- Simple and efficient tools for predictive data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

This is one of several other “scikits” (e.g. scikit-image) which are scientific toolboxes built on top of scipy.

Submodules

The `sklearn` package contains a large number of submodules which are specialized for different tasks / models,

- `sklearn.base` - Base classes and utility functions
- `sklearn.calibration` - Probability Calibration
- `sklearn.cluster` - Clustering
- `sklearn.compose` - Composite Estimators
- `sklearn.covariance` - Covariance Estimators
- `sklearn.cross_decomposition` - Cross decomposition
- `sklearn.datasets` - Datasets
- `sklearn.decomposition` - Matrix Decomposition
- `sklearn.discriminant_analysis` - Discriminant Analysis
- `sklearn.ensemble` - Ensemble Methods
- `sklearn.exceptions` - Exceptions and warnings
- `sklearn.experimental` - Experimental
- `sklearn.feature_extraction` - Feature Extraction
- `sklearn.feature_selection` - Feature Selection
- `sklearn.gaussian_process` - Gaussian Processes
- `sklearn.impute` - Impute
- `sklearn.inspection` - Inspection
- `sklearn.isotonic` - Isotonic regression
- `sklearn.kernel_approximation` - Kernel Approximation
- `sklearn.kernel_ridge` - Kernel Ridge Regression
- `sklearn.linear_model` - Linear Models
- `sklearn.manifold` - Manifold Learning
- `sklearn.metrics` - Metrics
- `sklearn.mixture` - Gaussian Mixture Models
- `sklearn.model_selection` - Model Selection
- `sklearn.multiclass` - Multiclass classification
- `sklearn.multioutput` - Multioutput regression and classification
- `sklearn.naive_bayes` - Naive Bayes
- `sklearn.neighbors` - Nearest Neighbors
- `sklearn.neural_network` - Neural network models
- `sklearn.pipeline` - Pipeline
- `sklearn.preprocessing` - Preprocessing and Normalization
- `sklearn.random_projection` - Random projection
- `sklearn.semi_supervised` - Semi-Supervised Learning
- `sklearn.svm` - Support Vector Machines
- `sklearn.tree` - Decision Trees
- `sklearn.utils` - Utilities

Model Fitting

Sample data

To begin, we will examine a simple data set on the size and weight of a number of books. The goal is to model the weight of a book using some combination of the other features in the data.

The included columns are:

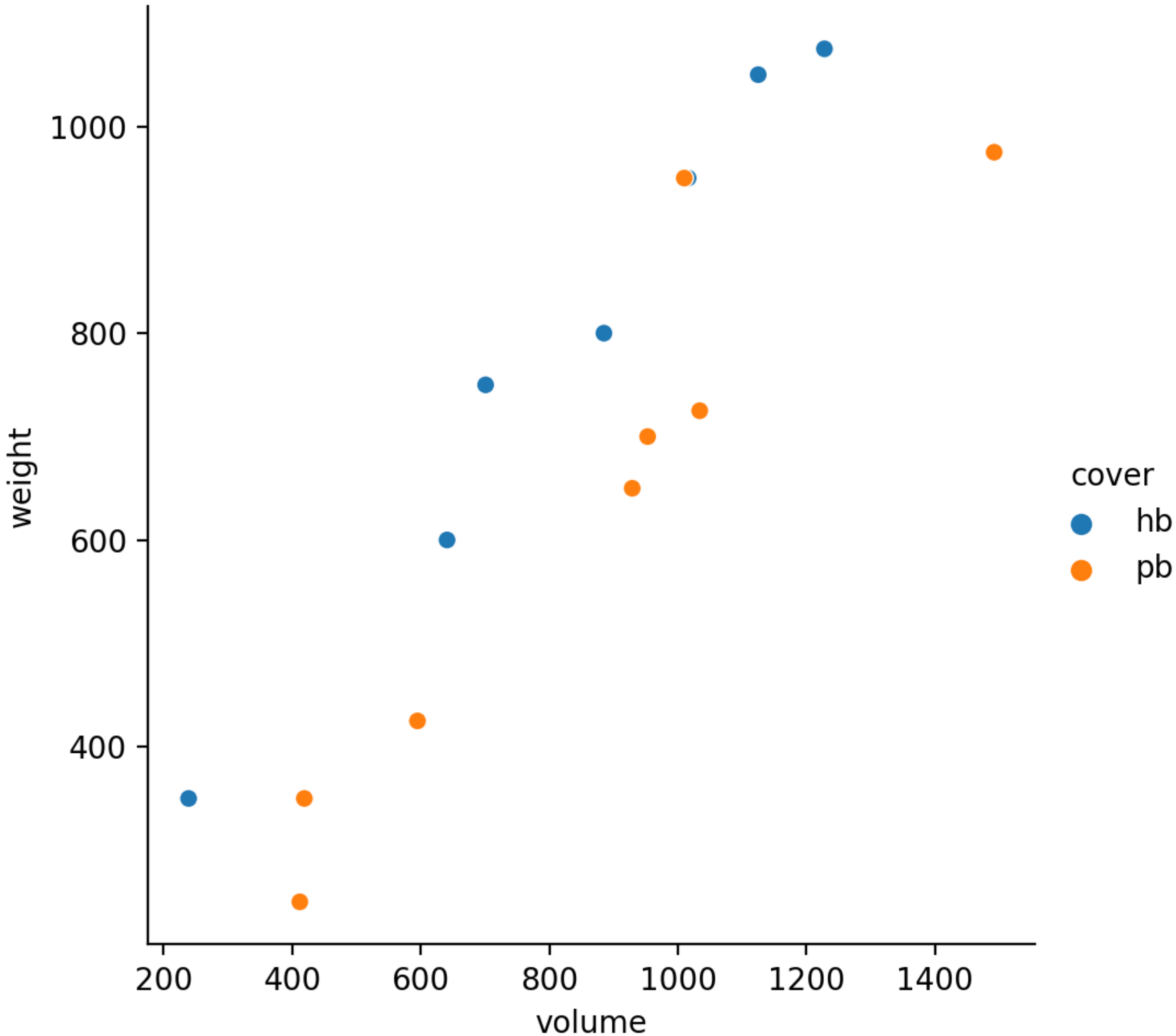
- `volume` - book volumes in cubic centimeters
- `weight` - book weights in grams
- `cover` - a categorical variable with levels `"hb"` hardback, `"pb"` paperback

```
books = pd.read_csv("data/daag_books.csv"); book
```

	volume	weight	cover
0	885	800	hb
1	1016	950	hb
2	1125	1050	hb
3	239	350	hb
4	701	750	hb
5	641	600	hb
6	1228	1075	hb
7	412	250	pb
8	953	700	pb
9	929	650	pb
10	1492	975	pb
11	419	350	pb
12	1010	950	pb
13	595	425	pb
14	1034	725	pb

These data come from the `allbacks` data set from the `DAAG` package in R

```python
sns.relplot(data=books, x="volume", y="weight", hue="cover")
```



Linear regression

scikit-learn uses an object oriented system for implementing the various modeling approaches, the class for `LinearRegression` is part of the `linear_model` submodule.

```
from sklearn.linear_model import LinearRegression
```

Each modeling class needs to be constructed (potentially with options) and then the resulting object will provide attributes and methods.

```
lm = LinearRegression()

m = lm.fit(
  X = books[["volume"]],
  y = books.weight
)

m.coef_
```

```
array([0.70863714])
```

```
m.intercept_
```

```
107.679310613766
```

Note `lm` and `m` are labels for the same object,

```
lm.coef_
```

```
array([0.70863714])
```

```
lm.intercept_
```

```
107.679310613766
```

A couple of considerations

When fitting a model, scikit-learn expects `X` to be a 2d array-like object (e.g. a `np.array` or `pd.DataFrame`) but will not accept a `pd.Series` or 1d `np.array`.

```
lm.fit(
  X = books.volume,
  y = books.weight
)
```

```
Error: ValueError: Expected 2D array, got 1D array i
array=[ 885 1016 1125  239  701  641 1228  412  953
 1034].
Reshape your data either using array.reshape(-1, 1) :
```

```
lm.fit(
  X = np.array(books.volume),
  y = books.weight
)
```

```
Error: ValueError: Expected 2D array, got 1D array i
array=[ 885 1016 1125  239  701  641 1228  412  953
 1034].
Reshape your data either using array.reshape(-1, 1) :
```

```
lm.fit(
  X = np.array(books.volume).reshape(-1,1),
  y = books.weight
)
```

```
▾ LinearRegression
LinearRegression()
```

```
lm.fit(
  X = books.drop(["weight", "cover"], axis=1),
  y = books.weight
)
```

```
▾ LinearRegression
LinearRegression()
```

Model parameters

Depending on the model being used, there will be a number of parameters that can be configured when creating the model object or via the `set_params()` method.

```python
lm.get_params()
```

```
{'copy_X': True, 'fit_intercept': True, 'n_jobs': None, 'positive': False}
```

```python
lm.set_params(fit_intercept = False)
```

```
▾ LinearRegression
LinearRegression(fit_intercept=False)
```

```python
lm = lm.fit(X = books[["volume"]], y = books.weight)
lm.intercept_
```

```
0.0
```

```python
lm.coef_
```

```
array([0.81932487])
```

Model prediction

Once the model coefficients have been fit, it is possible to predict using the model via the `predict()` method, this method requires a matrix-like `X` as input and in the case of `LinearRegression` returns an array of predicted y values.

```
lm.predict(X = books[["volume"]])
```

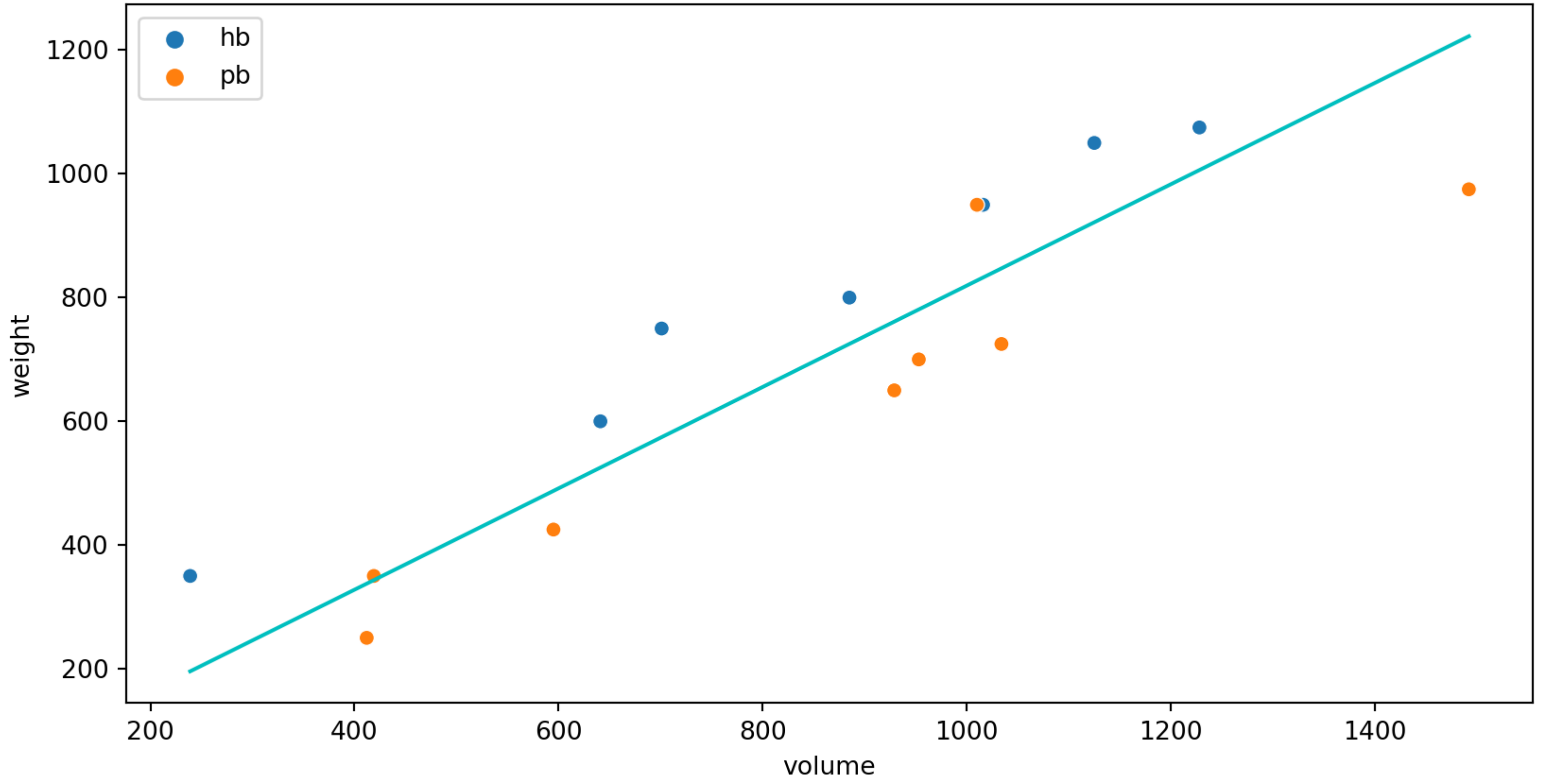
```
array([ 725.10251417,  832.43407276,  921.74048411,  195.81864507,
        574.34673721,  525.18724472, 1006.13094621,  337.5618484 ,
        780.81660565,  761.15280865, 1222.43271315,  343.29712253,
        827.51812351,  487.49830048,  847.1819205 ])
```

```
books["weight_lm_pred"] = lm.predict(X = books[["volume"]])
books
```

	volume	weight	cover	weight_lm_pred
0	885	800	hb	725.102514
1	1016	950	hb	832.434073
2	1125	1050	hb	921.740484
3	239	350	hb	195.818645
4	701	750	hb	574.346737
5	641	600	hb	525.187245
6	1228	1075	hb	1006.130946
7	412	250	pb	337.561848
8	953	700	pb	780.816606
9	929	650	pb	761.152809
10	1492	975	pb	1222.432713

```
11     419     350    pb      343.297123
12    1010     950    pb      827.518124
13     595     425    pb      487.498300
14    1034     725    pb      847.181921
```

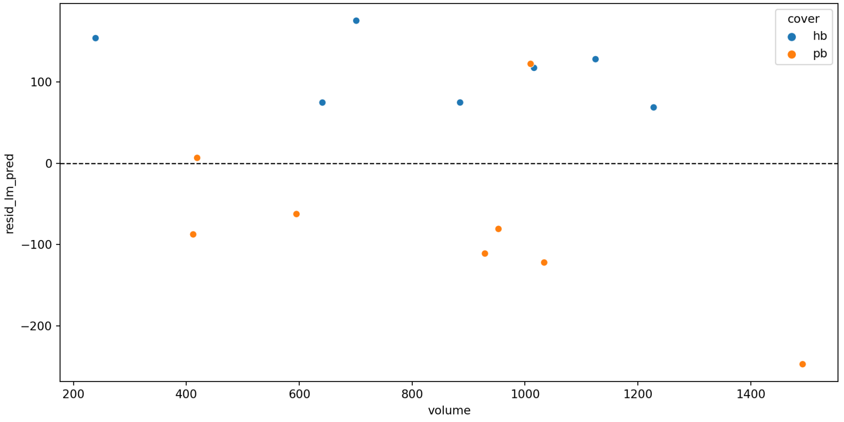
```
plt.figure()
sns.scatterplot(data=books, x="volume", y="weight", hue="cover")
sns.lineplot(data=books, x="volume", y="weight_lm_pred", color="c")
plt.show()
```



Residuals?

There is no built in functionality for calculating residuals, so this needs to be done by hand.

```
books["resid_lm_pred"] = books["weight"] - books["weight_lm_pred"]

plt.figure(layout="constrained")
ax = sns.scatterplot(data=books, x="volume", y="resid_lm_pred", hue="cover")
ax.axhline(c="k", ls="--", lw=1)
plt.show()
```



Categorical variables?

Scikit-learn expects that the model matrix be numeric before fitting,

```
lm = lm.fit(
  X = books[["volume", "cover"]],
  y = books.weight
)
```

```
Error: ValueError: could not convert string to float: 'hb'
```

the solution here is to dummy code the categorical variables - this can be done with pandas via `pd.get_dummies()` or with a scikit-learn preprocessor.

```
pd.get_dummies(books[["volume", "cover"]])
```

```
    volume  cover_hb  cover_pb
0      885         1         0
1     1016         1         0
2     1125         1         0
3      239         1         0
4      701         1         0
5      641         1         0
6     1228         1         0
7      412         0         1
8      953         0         1
9      929         0         1
10    1492         0         1
```

```
11     419          0          1
12    1010          0          1
13     595          0          1
14    1034          0          1
```

What went wrong?

Do the following results look reasonable? What went wrong?

```
lm = LinearRegression().fit(
  X = pd.get_dummies(books[["volume", "cover"]]),
  y = books.weight
)

lm.intercept_
```

```
105.93920788192202
```

```
lm.coef_
```

```
array([  0.71795374,  92.02363569, -92.02363569])
```

Quick comparison with R

```r
d = read.csv('data/daag_books.csv')
d['cover_hb'] = ifelse(d$cover == "hb", 1, 0)
d['cover_pb'] = ifelse(d$cover == "pb", 1, 0)
lm = lm(weight~volume+cover_hb+cover_pb, data=d)
summary(lm)
```

```

Call:
lm(formula = weight ~ volume + cover_hb + cover_pb, data = d)

Residuals:
    Min      1Q  Median      3Q     Max 
-110.10  -32.32  -16.10   28.93  210.95 

Coefficients: (1 not defined because of singularities)
             Estimate Std. Error t value Pr(>|t|)    
(Intercept)  13.91557   59.45408   0.234 0.818887    
volume        0.71795    0.06153  11.669  6.6e-08 ***
cover_hb    184.04727   40.49420   4.545 0.000672 ***
cover_pb           NA         NA      NA       NA    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 78.2 on 12 degrees of freedom
Multiple R-squared:  0.9275,	Adjusted R-squared:  0.9154 
F-statistic: 76.73 on 2 and 12 DF,  p-value: 1.455e-07
```

Avoiding co-linearity

```python
lm = LinearRegression(
  fit_intercept = False
).fit(
  X = pd.get_dummies(books[["volume", "cover"]])
  y = books.weight
)

lm.intercept_
```

```
0.0
```

```python
lm.coef_
```

```
array([  0.71795374, 197.96284357,  13.91557219])
```

```python
lm.feature_names_in_
```

```
array(['volume', 'cover_hb', 'cover_pb'], dtype=obje
```

```python
lm = LinearRegression(
).fit(
  X = pd.get_dummies(books[["volume", "cover"]],
  y = books.weight
)

lm.intercept_
```

```
197.96284357271753
```

```python
lm.coef_
```

```
array([   0.71795374, -184.04727138])
```

```python
lm.feature_names_in_
```

```
array(['volume', 'cover_pb'], dtype=object)
```

Preprocessors

Preprocessors

These are a set of transformer classes present in the `sklearn.preprocessing` submodule that are designed to help with the preparation of raw feature data into quantities more suitable for downstream modeling tools.

Like the modeling classes, they have an object oriented design that shares a common interface (methods and attributes) for bringing in data, transforming it, and returning it.

OneHotEncoder

For dummy coding we can use the `OneHotEncoder` preprocessor, the default is to use one hot encoding but standard dummy coding can be achieved via the `drop` parameter.

```
from sklearn.preprocessing import OneHotEncoder
```

```python
enc = OneHotEncoder(sparse_output=False)
enc.fit(X = books[["cover"]])
```

```
▼ OneHotEncoder
OneHotEncoder(sparse_output=False)
```

```python
enc.transform(X = books[["cover"]])
```

```
array([[1., 0.],
       [1., 0.],
       [1., 0.],
       [1., 0.],
       [1., 0.],
       [1., 0.],
       [1., 0.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.]])
```

```python
enc = OneHotEncoder(sparse_output=False, drop="f
enc.fit_transform(X = books[["cover"]])
```

```
array([[0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.]])
```

Other useful bits

```python
enc.get_feature_names_out()
```

```
array(['cover_hb', 'cover_pb'], dtype=object)
```

```python
f = enc.transform(X = books[["cover"]])
f
```

```
array([[1., 0.],
       [1., 0.],
       [1., 0.],
       [1., 0.],
       [1., 0.],
       [1., 0.],
       [1., 0.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.]])
```

```python
enc.inverse_transform(f)
```

```
array([['hb'],
       ['hb'],
       ['hb'],
       ['hb'],
       ['hb'],
       ['hb'],
       ['hb'],
       ['pb'],
       ['pb'],
       ['pb'],
       ['pb'],
       ['pb'],
       ['pb'],
       ['pb'],
       ['pb']], dtype=object)
```

A cautionary note

Unlike `pd.get_dummies()` it is not safe to use `OneHotEncoder` with both numerical and categorical features, as the former will also be transformed.

```
enc = OneHotEncoder(sparse_output=False)
X = enc.fit_transform(
  X = books[["volume", "cover"]]
)

pd.DataFrame(
  data=X,
  columns = enc.get_feature_names_out()
)
```

```
    volume_239  volume_412  volume_419  ...  volume_1492  cover_hb  cover_pb
0          0.0         0.0         0.0  ...          0.0       1.0       0.0
1          0.0         0.0         0.0  ...          0.0       1.0       0.0
2          0.0         0.0         0.0  ...          0.0       1.0       0.0
3          1.0         0.0         0.0  ...          0.0       1.0       0.0
4          0.0         0.0         0.0  ...          0.0       1.0       0.0
5          0.0         0.0         0.0  ...          0.0       1.0       0.0
6          0.0         0.0         0.0  ...          0.0       1.0       0.0
7          0.0         1.0         0.0  ...          0.0       0.0       1.0
8          0.0         0.0         0.0  ...          0.0       0.0       1.0
9          0.0         0.0         0.0  ...          0.0       0.0       1.0
10         0.0         0.0         0.0  ...          1.0       0.0       1.0
11         0.0         0.0         1.0  ...          0.0       0.0       1.0
```

```
12          0.0          0.0          0.0  ...          0.0          0.0          1.0
13          0.0          0.0          0.0  ...          0.0          0.0          1.0
14          0.0          0.0          0.0  ...          0.0          0.0          1.0

[15 rows x 17 columns]
```

Putting it together

```
cover = OneHotEncoder(
  sparse_output=False
).fit_transform(
  books[["cover"]]
)
X = np.c_[books.volume, cover]

lm2 = LinearRegression(
  fit_intercept=False
).fit(
  X = X,
  y = books.weight
)

lm2.coef_
```

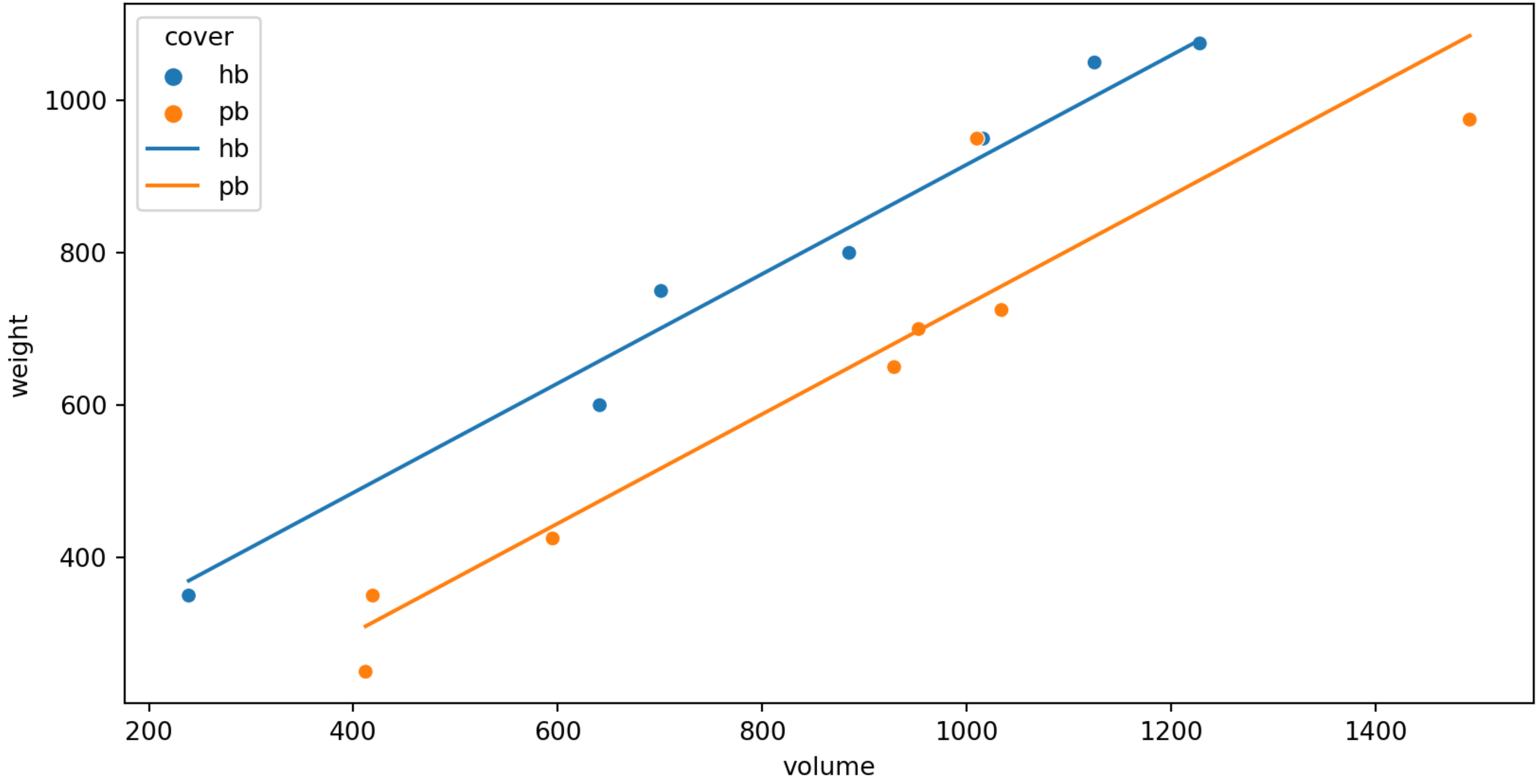
```
array([  0.71795374, 197.96284357,  13.91557219])
```

```
books["weight_lm2_pred"] = lm2.predict(X=X)
books.drop(["weight_lm_pred", "resid_lm_pred"],
```

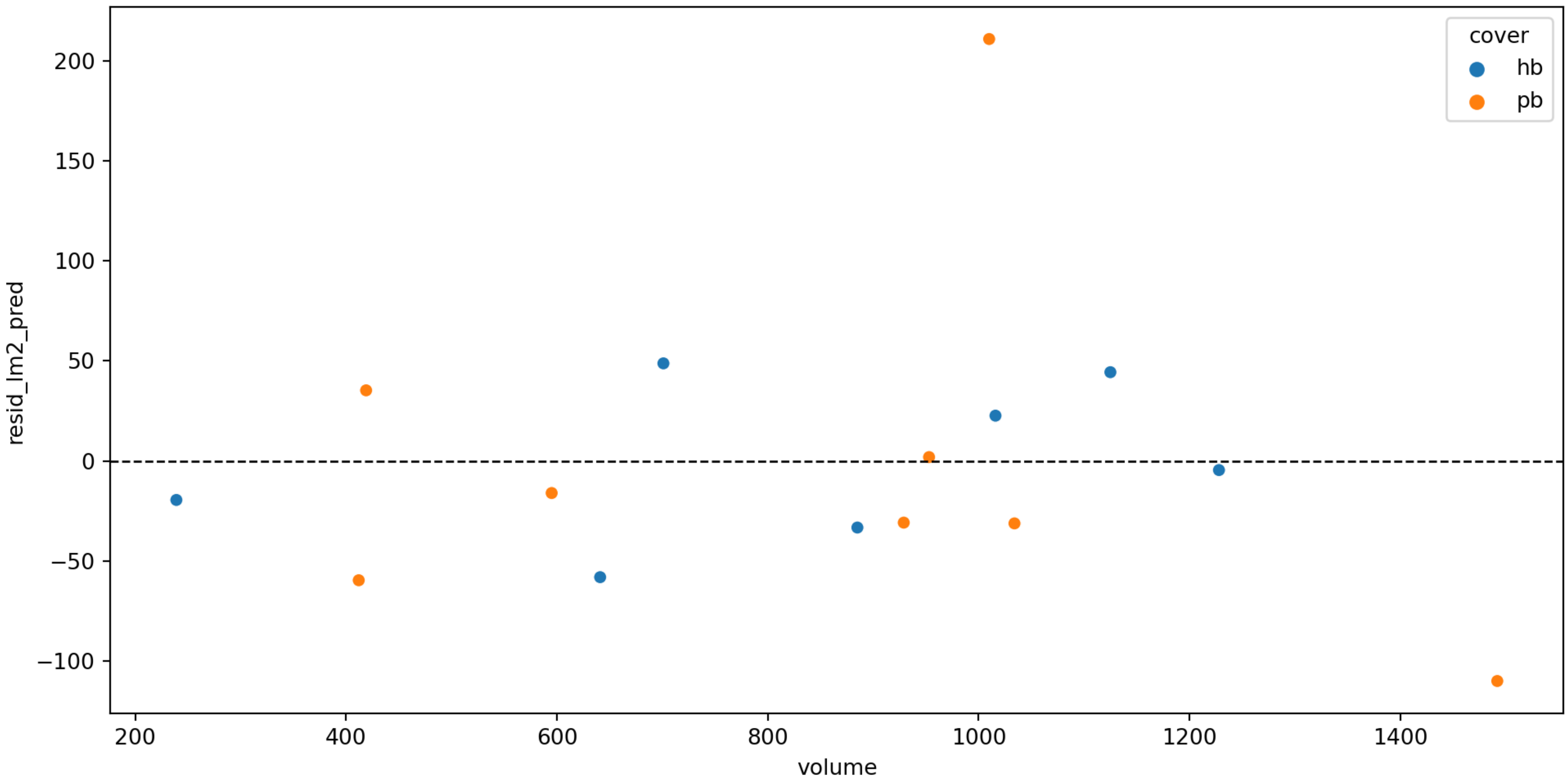
	volume	weight	cover	weight_lm2_pred
0	885	800	hb	833.351907
1	1016	950	hb	927.403847
2	1125	1050	hb	1005.660805
3	239	350	hb	369.553788
4	701	750	hb	701.248418
5	641	600	hb	658.171193
6	1228	1075	hb	1079.610041
7	412	250	pb	309.712515
8	953	700	pb	698.125490
9	929	650	pb	680.894600
10	1492	975	pb	1085.102558
11	419	350	pb	314.738191
12	1010	950	pb	739.048853
13	595	425	pb	441.098050
14	1034	725	pb	756.279743

We'll see a more elegant way of doing this in the near future

Model fit


cover
hb
pb
hb
pb
1000
800
600
400
weight
200
400
600
800
1000
1200
1400
volume

Model residuals



Model performance

Scikit-learn comes with a number of builtin functions for measuring model performance in the `sklearn.metrics` submodule - these are generally just functions that take the vectors `y_true` and `y_pred` and return a scalar score.

```
from sklearn.metrics import mean_squared_error, r2_score
```

```
r2_score(books.weight, books.weight_lm_pred)
```

```
0.7800969547785038
```

```
mean_squared_error(
  books.weight, books.weight_lm_pred
)
```

```
14833.68208377448
```

```
mean_squared_error(
  books.weight, books.weight_lm_pred,
  squared=False
)
```

```
121.79360444528473
```

```
r2_score(books.weight, books.weight_lm2_pred)
```

```
0.9274775756821679
```

```
mean_squared_error(
  books.weight, books.weight_lm2_pred
)
```

```
4892.040422595093
```

```
mean_squared_error(
  books.weight, books.weight_lm2_pred,
  squared=False
)
```

```
69.94312276839727
```

See API Docs for a list of available metrics

Exercise 1

Create and fit a model for the `books` data that includes an interaction effect between `volume` and `cover`.

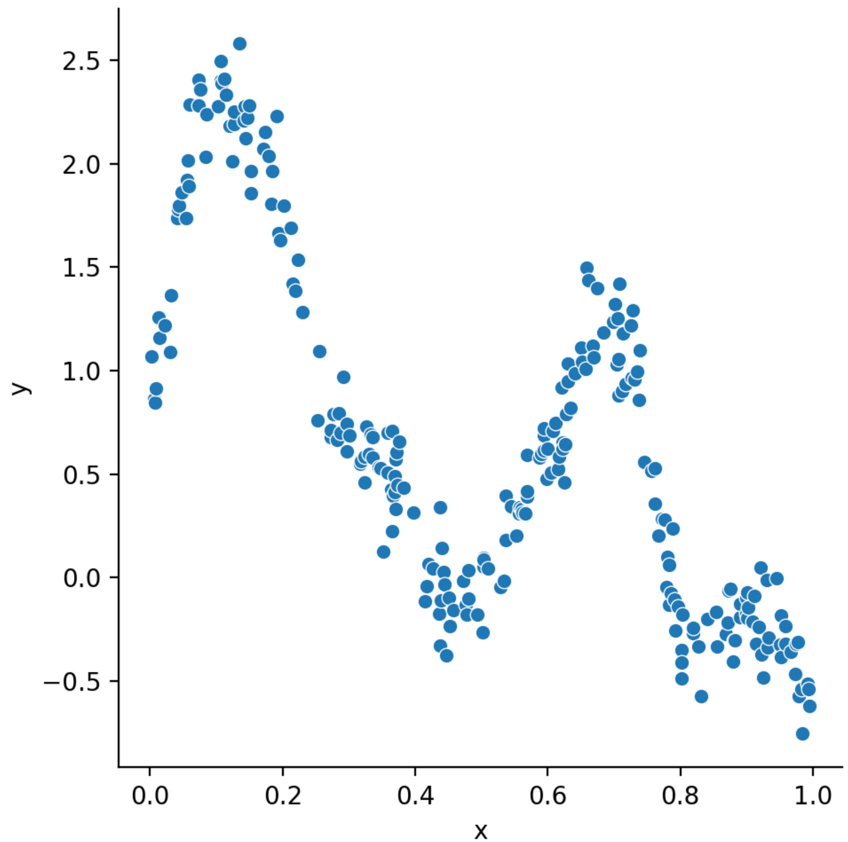
You will need to do this manually with `pd.getdummies()` and some additional data munging.

Other transformers

Polynomial regression

We will now look at another flavor of regression model, that involves preprocessing and a hyperparameter - namely polynomial regression.

```
df = pd.read_csv("data/gp.csv")
sns.relplot(data=df, x="x", y="y")
```



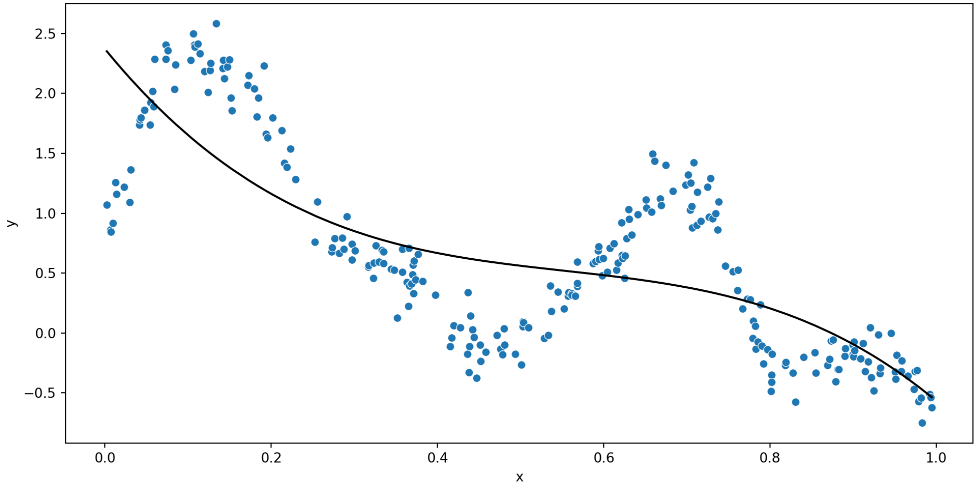
By hand

It is certainly possible to construct the necessary model matrix by hand (or even use a function to automate the process), but this is less then desirable generally - particularly if we want to do anything fancy (e.g. cross validation)

```
X = np.c_[
    np.ones(df.shape[0]),
    df.x,
    df.x**2,
    df.x**3
]

plm = LinearRegression(
  fit_intercept = False
).fit(
  X=X, y=df.y
)

plm.coef_
```

```
array([ 2.36985684, -8.49429068, 13.95066369, -8.392
```

```
df["y_pred"] = plm.predict(X=X)

plt.figure(layout="constrained")
sns.scatterplot(data=df, x="x", y="y")
sns.lineplot(data=df, x="x", y="y_pred", color="
plt.show()
```



PolynomialFeatures

This is another transformer class from `sklearn.preprocessing` that simplifies the process of constructing polynormial features for your model matrix. Usage is similar to that of `OneHotEncoder`.

```python
from sklearn.preprocessing import PolynomialFeatures
X = np.array(range(6)).reshape(-1,1)
```

```python
pf = PolynomialFeatures(degree=3)
pf.fit(X)
```

```
▼ PolynomialFeatures
PolynomialFeatures(degree=3)
```

```python
pf.transform(X)
```

```
array([[  1.,   0.,   0.,   0.],
       [  1.,   1.,   1.,   1.],
       [  1.,   2.,   4.,   8.],
       [  1.,   3.,   9.,  27.],
       [  1.,   4.,  16.,  64.],
       [  1.,   5.,  25., 125.]])
```

```python
pf.get_feature_names_out()
```

```
array(['1', 'x0', 'x0^2', 'x0^3'], dtype=object)
```

```python
pf = PolynomialFeatures(
  degree=2, include_bias=False
)
pf.fit_transform(X)
```

```
array([[ 0.,  0.],
       [ 1.,  1.],
       [ 2.,  4.],
       [ 3.,  9.],
       [ 4., 16.],
       [ 5., 25.]])
```

```python
pf.get_feature_names_out()
```

```
array(['x0', 'x0^2'], dtype=object)
```

Interactions

If the feature matrix `X` has more than one column then `PolynomialFeatures` transformer will include interaction terms with total degree up to `degree`.

```
X.reshape(-1, 2)
```

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

```
pf = PolynomialFeatures(
  degree=3, include_bias=False
)
pf.fit_transform(
  X.reshape(-1, 2)
)
```

```
array([[  0.,   1.,   0.,   0.,   1.,   0.,   0.,
       [  2.,   3.,   4.,   6.,   9.,   8.,  12.,  1
       [  4.,   5.,  16.,  20.,  25.,  64.,  80., 10
```

```
pf.get_feature_names_out()
```

```
array(['x0', 'x1', 'x0^2', 'x0 x1', 'x1^2', 'x0^3',
       'x1^3'], dtype=object)
```

```
X.reshape(-1, 3)
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

```
pf = PolynomialFeatures(
  degree=2, include_bias=False
)
pf.fit_transform(
  X.reshape(-1, 3)
)
```

```
array([[ 0.,  1.,  2.,  0.,  0.,  0.,  1.,  2.,  4.],
       [ 3.,  4.,  5.,  9., 12., 15., 16., 20., 25.]
```

```
pf.get_feature_names_out()
```

```
array(['x0', 'x1', 'x2', 'x0^2', 'x0 x1', 'x0 x2', '
       'x2^2'], dtype=object)
```

Modeling with PolynomialFeatures

```python
def poly_model(X, y, degree):
  X  = PolynomialFeatures(
    degree=degree, include_bias=False
  ).fit_transform(
    X=X
  )
  y_pred = LinearRegression(
  ).fit(
    X=X, y=y
  ).predict(
    X
  )
  return mean_squared_error(y, y_pred, squared=F
```

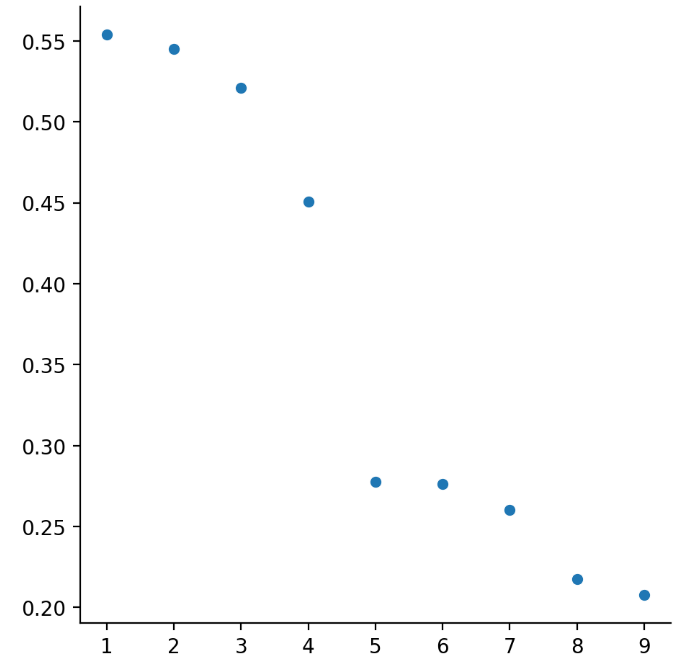
```python
poly_model(X = df[["x"]], y = df.y, degree = 2)
```

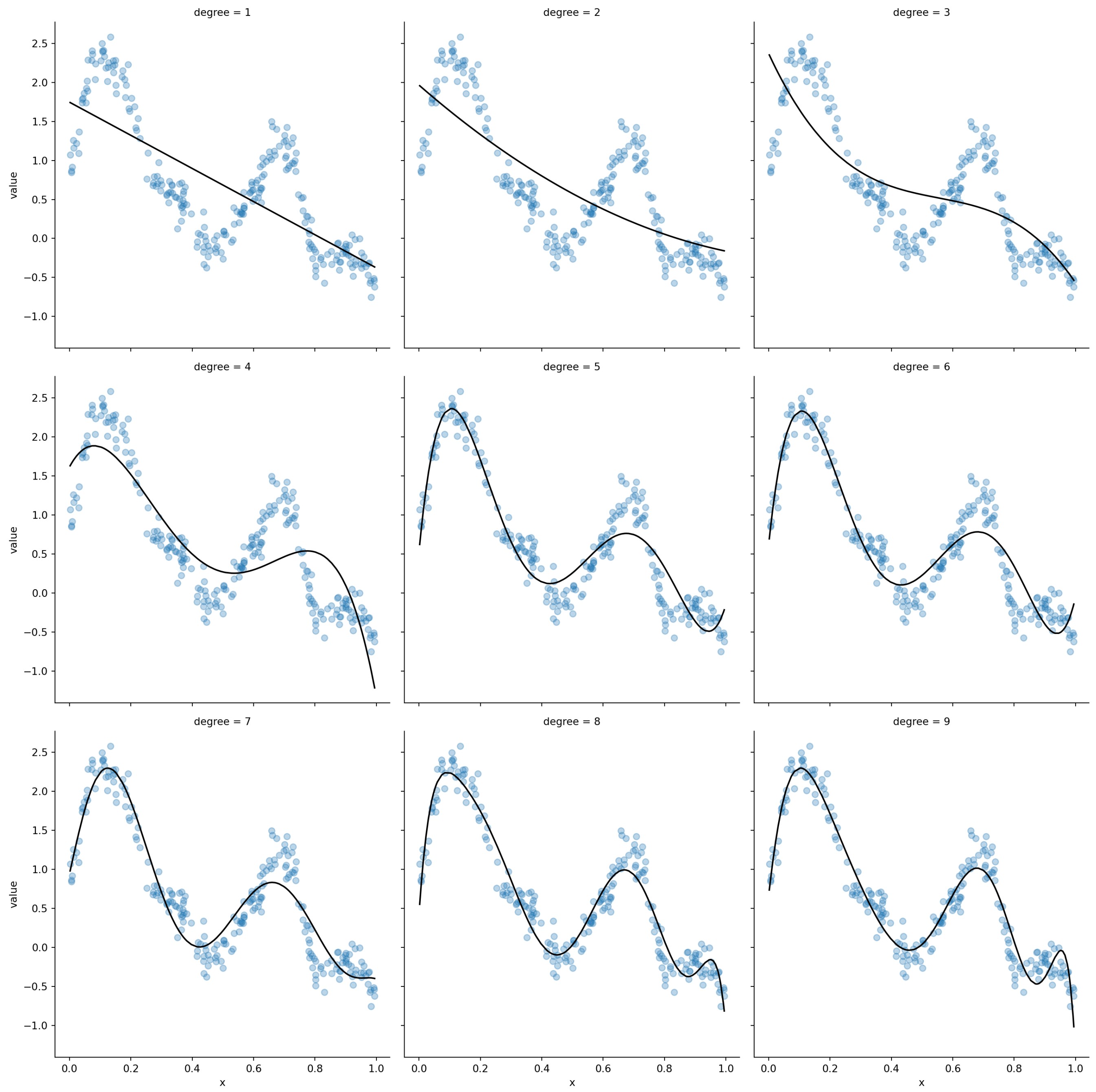
```
0.5449418707295371
```

```python
poly_model(X = df[["x"]], y = df.y, degree = 3)
```

```
0.5208157900621085
```

```python
degrees = range(1,10)
rmses = [
  poly_model(X=df[["x"]], y=df.y, degree=d)
  for d in degrees
]
sns.relplot(x=degrees, y=rmses)
```

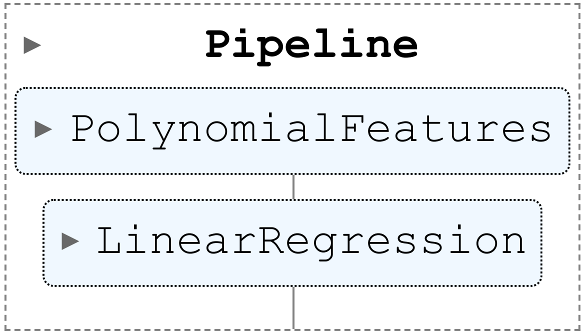



degree = 1
degree = 2
degree = 3
degree = 4
degree = 5
degree = 6
degree = 7
degree = 8
degree = 9
value
x

Pipelines

You may have noticed that `PolynomialFeatures` takes a model matrix as input and returns a new model matrix as output which is then used as the input for `LinearRegression`. This is not an accident, and by structuring the library in this way sklearn is designed to enable the connection of these steps together, into what sklearn calls a *pipeline*.

```
from sklearn.pipeline import make_pipeline

p = make_pipeline(
  PolynomialFeatures(degree=4),
  LinearRegression()
)
p
```



Using Pipelines

Once constructed, this object can be used just like our previous `LinearRegression` model (i.e. fit to our data and then used for prediction)

```python
p = p.fit(X = df[["x"]], y = df.y)
p
```

```
▸ Pipeline
  ▸ PolynomialFeatures
  ▸ LinearRegression
```

```python
p.predict(X = df[["x"]])
```

```
array([ 1.6295693 ,  1.65734929,  1.6610466 ,  1.677
        1.70475286,  1.75280126,  1.78471392,  1.790
        1.82966357,  1.83376043,  1.84494343,  1.860
        1.86619112,  1.86837909,  1.87065283,  1.884
        1.88527174,  1.88577463,  1.88544367,  1.868
        1.86252922,  1.86047349,  1.85377801,  1.849
        1.82623453,  1.82024199,  1.81799793,  1.797
        1.77034143,  1.76574288,  1.75371272,  1.743
        1.73356954,  1.65527727,  1.64812184,  1.618
        1.5960389 ,  1.56080881,  1.55036459,  1.540
        1.45096594,  1.43589836,  1.41886389,  1.394
        1.23072992,  1.21355164,  1.11776117,  1.115
        1.06449719,  1.04672121,  1.03662739,  1.014
        0.98081577,  0.96176797,  0.87491417,  0.871
        0.84171166,  0.82875003,  0.8085086 ,  0.791
        0.78078036,  0.73538157,  0.7181484 ,  0.700
        0.67229069,  0.64782899,  0.64050946,  0.637
        0.62323271,  0.61965166,  0.61705548,  0.614
        0.60347713,  0.5909255 ,  0.566617  ,  0.509
        0.44177711,  0.43291379,  0.40957833,  0.384
        0.38067928,  0.3791518 ,  0.37610476,  0.369
        0.35806518,  0.3475729 ,  0.3466828 ,  0.333
        0.3006981 ,  0.29675876,  0.29337641,  0.293
```

```python
plt.figure(layout="constrained")
sns.scatterplot(data=df, x="x", y="y")
sns.lineplot(x=df.x, y=p.predict(X = df[["x"]]),
plt.show()
```


Model coefficients (or other attributes)

The attributes of steps are not directly accessible, but can be accessed via `steps` or `named_steps` attributes,

```
p.coef_
```

```
Error: AttributeError: 'Pipeline' object has no attribute 'coef_'
```

```
p.named_steps["linearregression"].intercept_
```

```
1.6136636604768615
```

```
p.steps[1][1].coef_
```

```
array([  0.        ,   7.39051417, -57.67175293, 102.72227443,
       -55.38181361])
```

```
p.steps
```

```
[('polynomialfeatures', PolynomialFeatures(degree=4)), ('linearregression', LinearRegression())]
```

```
p.steps[0][1].get_feature_names_out()
```

```
array(['1', 'x', 'x^2', 'x^3', 'x^4'], dtype=object)
```

Anyone notice a problem?

What about step parameters?

By accessing each step we can adjust their parameters (via `set_params()`),

```
p.named_steps[
  "linearregression"
].get_params()
```

```
{'copy_X': True, 'fit_intercept': True, 'n_jobs': No
```

```
p.named_steps[
  "linearregression"
].set_params(
  fit_intercept=False
)
```

```
▾ LinearRegression
LinearRegression(fit_intercept=False)
```

```
p.fit(X = df[["x"]], y = df.y)
```

```
▸ Pipeline
▸ PolynomialFeatures
▸ LinearRegression
```

```
p.named_steps["linearregression"].intercept_
```

```
0.0
```

```
p.named_steps["linearregression"].coef_
```

```
array([  1.61366366,   7.39051417, -57.67175293, 102.
       -55.38181361])
```

Pipeline parameter names

These parameters can also be directly accessed at the pipeline level, note how the names are constructed:

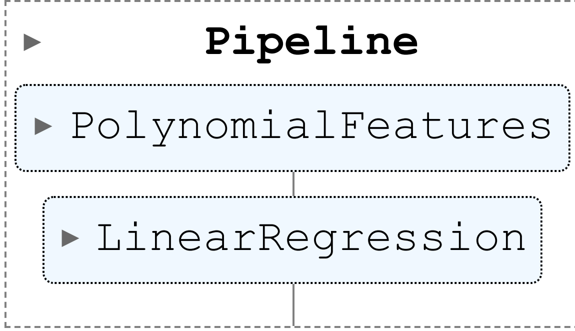
```
p.get_params()
```

```
{'memory': None, 'steps': [('polynomialfeatures', PolynomialFeatures(degree=4)), ('linearregression', Linear
```

```
p.set_params(
  linearregression__fit_intercept=True,
  polynomialfeatures__include_bias=False
)
```

```
▸ Pipeline
  ▸ PolynomialFeatures
  ▸ LinearRegression
```

```
p.fit(X = df[["x"]], y = df.y)
```




```
p.named_steps["linearregression"].intercept_
```

```
1.6136636604768375
```

```
p.named_steps["linearregression"].coef_
```

```
array([  7.39051417, -57.67175293, 102.72227443, -55.38181361])
```

Tuning parameters

We've already seen a manual approach to tuning models over the degree parameter, scikit-learn also has built in tools to aide with this process. Here we will leverage `GridSearchCV` to tune the degree parameter in our pipeline.

```
from sklearn.model_selection import GridSearchCV, KFold

p = make_pipeline(
    PolynomialFeatures(include_bias=True),
    LinearRegression(fit_intercept=False)
)

grid_search = GridSearchCV(
  estimator = p,
  param_grid = {"polynomialfeatures__degree": range(1,10)},
  scoring = "neg_root_mean_squared_error",
  cv = KFold(shuffle=True)
)
```


Much more detail on this next time - including the proper way to do cross-validation

Preview - Performing a grid search

```
grid_search.fit(X = df[["x"]], y = df.y)
```

```
▸ GridSearchCV
▸ estimator: Pipeline
  ▸ PolynomialFeatures
  ▸ LinearRegression
```

```
grid_search.best_index_
```

```
8
```

```
grid_search.best_params_
```

```
{'polynomialfeatures__degree': 9}
```

```
grid_search.best_score_
```

```
-0.21997083726848174
```

cv_results_

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

```
array([-0.55749641, -0.55411766, -0.53337083, -0.46185477, -0.28410974,
       -0.2837936 , -0.26782574, -0.22542718, -0.21997084])
```

```
grid_search.cv_results_["rank_test_score"]
```

```
array([9, 8, 7, 6, 5, 4, 3, 2, 1], dtype=int32)
```

```
grid_search.cv_results_["mean_fit_time"]
```

```
array([0.00097075, 0.00087657, 0.00086861, 0.00086889, 0.00087271,
       0.0008781 , 0.00087924, 0.00088468, 0.00088801])
```

```
grid_search.cv_results_.keys()
```

```
dict_keys(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time', 'param_polynomialfeatures__
```