

scikit-learn

Cross-validation

Lecture 15

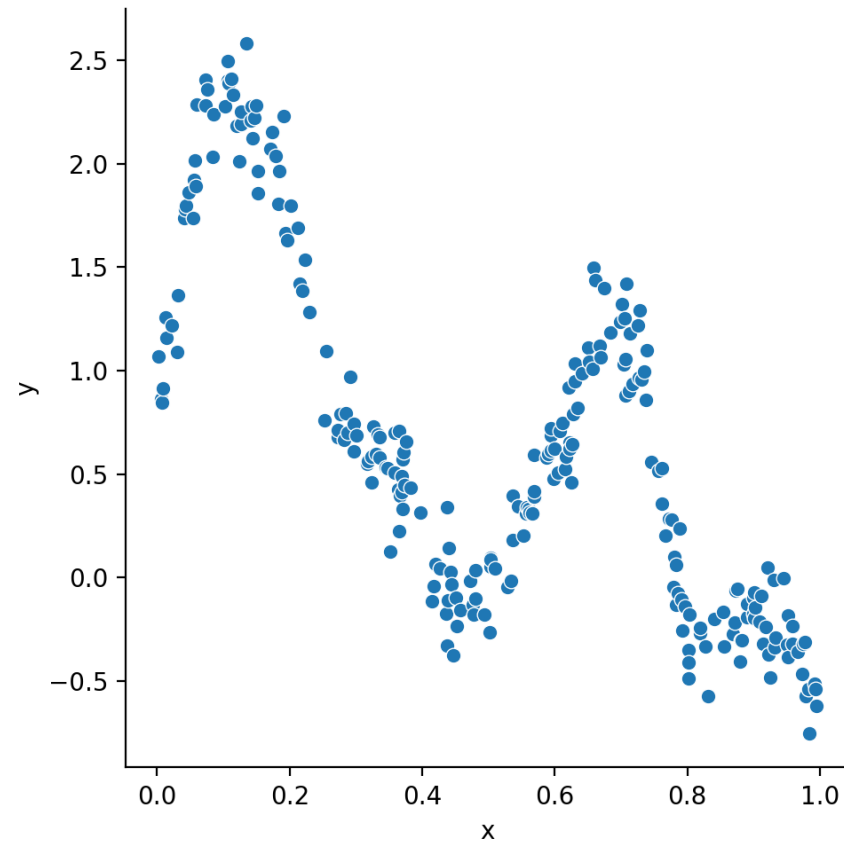
Dr. Colin Rundel

Pipelines

From last time

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

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



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*.

```
1 from sklearn.pipeline import make_pipeline
2
3 p = make_pipeline(
4     PolynomialFeatures(degree=4),
5     LinearRegression()
6 )
7 p
```

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

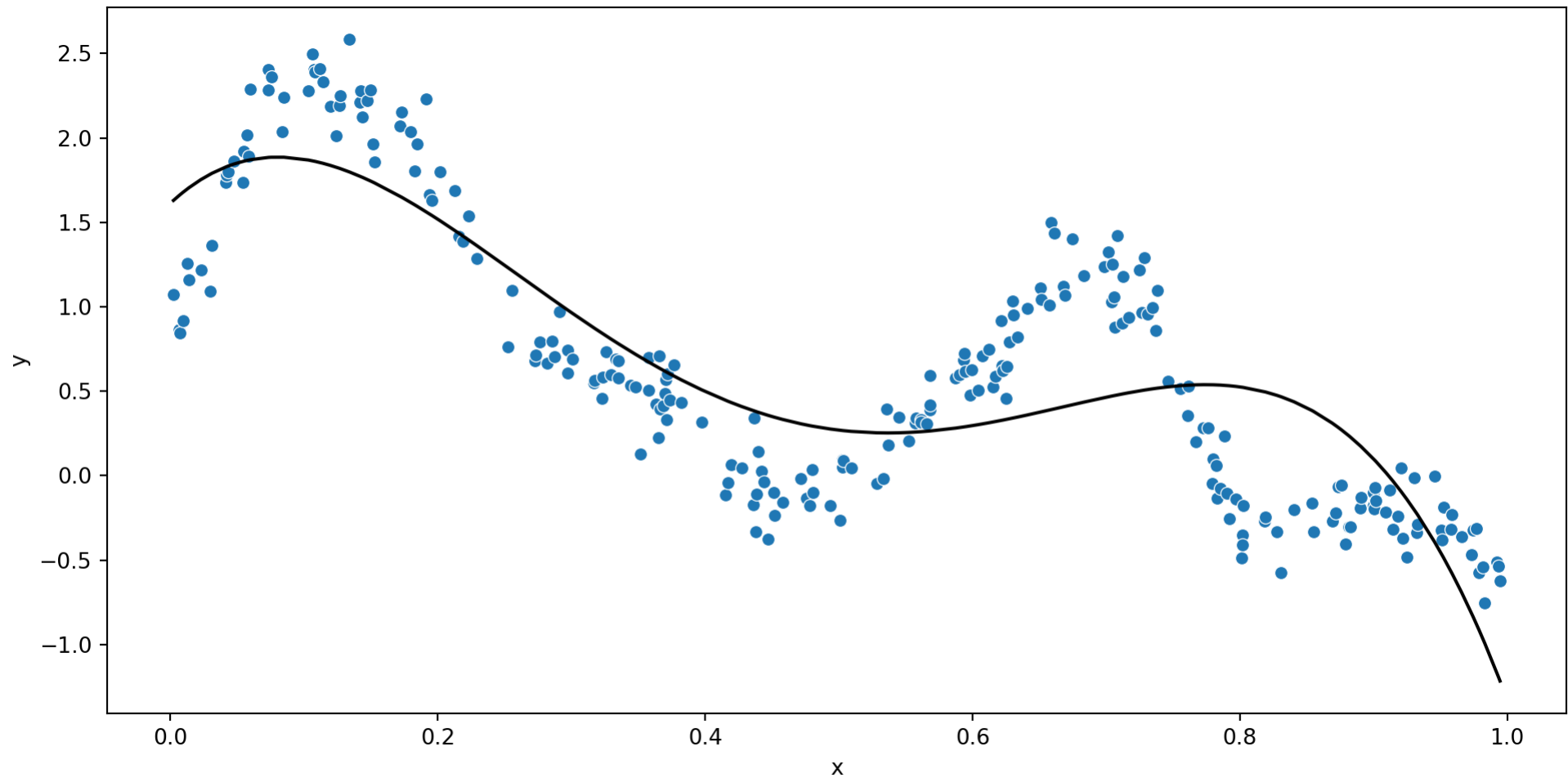
Using Pipelines

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

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

```
array([ 1.62957,  1.65735,  1.66105,  1.6778 ,  1.69667,  1.70475,  1.7528 ,  1.78471,  1.7905 ,  1.8269 ,
        1.8844 ,  1.88527,  1.88577,  1.88544,  1.86891,  1.86365,  1.86253,  1.86047,  1.85378,  1.84938,
        1.7439 ,  1.73804,  1.73357,  1.65528,  1.64812,  1.61868,  1.60413,  1.59604,  1.56081,  1.55036,
        1.11776,  1.11522,  1.09595,  1.0645 ,  1.04672,  1.03663,  1.01407,  0.98209,  0.98082,  0.96177,
        0.73538,  0.71815,  0.70047,  0.67234,  0.67229,  0.64783,  0.64051,  0.63727,  0.63526,  0.62323,
        0.44178,  0.43291,  0.40958,  0.3848 ,  0.38289,  0.38068,  0.37915,  0.3761 ,  0.36933,  0.36493,
        0.27632,  0.26899,  0.26761,  0.26726,  0.26716,  0.26242,  0.25405,  0.25335,  0.25323,  0.25323,
        0.26486,  0.26489,  0.28177,  0.28525,  0.28861,  0.28918,  0.29004,  0.29445,  0.2956 ,  0.30233,
        0.33711,  0.34111,  0.34141,  0.34707,  0.35926,  0.37678,  0.37775,  0.38885,  0.39078,  0.39518,
        0.4751 ,  0.47762,  0.48382,  0.48474,  0.49067,  0.50203,  0.50448,  0.50675,  0.5096 ,  0.51457,
        0.53839,  0.53823,  0.53757,  0.53749,  0.5365 ,  0.53481,  0.53372,  0.53274,  0.52872,  0.52378,
        0.38104,  0.31132,  0.29845,  0.28774,  0.27189,  0.2524 ,  0.23846,  0.22915,  0.17792,  0.17355,
       -0.09117, -0.10696, -0.13889, -0.20218, -0.22105, -0.23335, -0.39046, -0.46281, -0.47156, -0.48247, -
       -1.16341, -1.19337, -1.21549])
```

```
1 plt.figure(layout="constrained")
2 sns.scatterplot(data=df, x="x", y="y")
3 sns.lineplot(x=df.x, y=p.predict(X = df[["x"]]), color="k")
4 plt.show()
```



Model coefficients (or other attributes)

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

```
1 p.coef_
```

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

```
1 p.steps
```

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

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

```
array([ 0.          ,  7.39051, -57.67175, 102.72227, -55.38181])
```

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

```
1.6136636604768615
```

Other useful bits

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

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

```
1 p.steps[1][1].get_params()
```

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

Anyone notice a problem?

```
1 p.steps[1][1].rank_
```

4

```
1 p.steps[1][1].n_features_in_
```

5

What about step parameters?

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

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

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

```
1 p.named_steps["linearregression"].set_params(  
2     fit_intercept=False  
3 )
```

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

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

```
Pipeline(steps=[('polynomialfeatures', PolynomialFeatures(degree=4)),  
                 ('linearregression', LinearRegression(fit_intercept=False))])
```

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

```
0.0
```

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

```
array([ 1.61366,  7.39051, -57.67175, 102.72227, -55.38181])
```

Pipeline parameter names

These parameters can also be directly accessed at the pipeline level, names are constructed as step name + `__` + parameter name:

```
1 p.get_params()
```

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

```
1 p.set_params(  
2     linearregression__fit_intercept=True,  
3     polynomialfeatures__include_bias=False  
4 )
```

```
Pipeline(steps=[('polynomialfeatures',  
                  PolynomialFeatures(degree=4, include_bias=False)),  
                ('linearregression', LinearRegression())])
```

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

```
Pipeline(steps=[('polynomialfeatures',  
                  PolynomialFeatures(degree=4, include_bias=False)),  
                ('linearregression', LinearRegression())])
```

```
1 p.named_steps["polynomialfeatures"].get_feature_names_out()
```

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

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

```
1.6136636604768375
```

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

```
array([ 7.39051, -57.67175, 102.72227, -55.38181])
```

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.

```
1 from sklearn.model_selection import GridSearchCV, KFold
2
3 p = make_pipeline(
4     PolynomialFeatures(include_bias=True),
5     LinearRegression(fit_intercept=False)
6 )
7
8 grid_search = GridSearchCV(
9     estimator = p,
10    param_grid = {"polynomialfeatures__degree": range(1,10)},
11    scoring = "neg_root_mean_squared_error",
12    cv = KFold(shuffle=True)
13 )
```

Preview - Performing a grid search

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

```
GridSearchCV(cv=KFold(n_splits=5, random_state=None, shuffle=True),  
             estimator=Pipeline(steps=[('polynomialfeatures',  
                                       PolynomialFeatures()),  
                                       ('linearregression',  
                                       LinearRegression(fit_intercept=False))]),  
             param_grid={'polynomialfeatures__degree': range(1, 10)},  
             scoring='neg_root_mean_squared_error')
```

```
1 grid_search.best_index_
```

8

```
1 grid_search.best_params_
```

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

```
1 grid_search.best_score_
```

```
-0.22148116153456804
```

Column Transformers

Column Transformers

Are a tool for selectively applying transformer(s) to column(s) of an array or DataFrame, they function in a way that is similar to a pipeline and similarly have a `make` helper function `make_column_transformer()`.

```
1 from sklearn.compose import make_column_transformer
2 from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

```

1 ct = make_column_transformer(
2     (StandardScaler(), ["volume"]),
3     (OneHotEncoder(), ["cover"]),
4 ).fit(
5     books
6 )

```

```
1 ct.get_feature_names_out()
```

```
array(['standardscaler__volume', 'onehotencoder__cover'], dtype=object)
```

```
1 ct.transform(books)
```

```

array([[ 0.12101,  1.      ,  0.      ],
       [ 0.51997,  1.      ,  0.      ],
       [ 0.85192,  1.      ,  0.      ],
       [-1.84637,  1.      ,  0.      ],
       [-0.43936,  1.      ,  0.      ],
       [-0.62209,  1.      ,  0.      ],
       [ 1.16561,  1.      ,  0.      ],
       [-1.31951,  0.      ,  1.      ],
       [ 0.3281 ,  0.      ,  1.      ],
       [ 0.25501,  0.      ,  1.      ],
       [ 1.96962,  0.      ,  1.      ],
       [-1.29819,  0.      ,  1.      ],
       [ 0.50169,  0.      ,  1.      ],
       [-0.76218,  0.      ,  1.      ],
       [ 0.57478,  0.      ,  1.      ]])

```


Keeping or dropping other columns

One addition important argument is `remainder` which determines what happens to unspecified columns. The default is `"drop"` which is why `weight` was removed, the alternative is `"passthrough"` which retains untransformed columns.

```
1 ct = make_column_transformer(  
2     (StandardScaler(), ["volume"]),  
3     (OneHotEncoder(), ["cover"]),  
4     remainder = "passthrough"  
5 ).fit(  
6     books  
7 )
```

```
1 ct.get_feature_names_out()
```

```
array(['standardscaler__volume', 'onehotencoder__cover', 'standardscaler__
```

```
1 ct.transform(books)
```

```
array([[ 0.12101, 1., 0., 800.
        [ 0.51997, 1., 0., 950.
        [ 0.85192, 1., 0., 1050.
        [-1.84637, 1., 0., 350.
        [-0.43936, 1., 0., 750.
        [-0.62209, 1., 0., 600.
        [ 1.16561, 1., 0., 1075.
        [-1.31951, 0., 1., 250.
        [ 0.3281 , 0., 1., 700.
        [ 0.25501, 0., 1., 650.
        [ 1.96962, 0., 1., 975.
        [-1.29819, 0., 1., 350.
        [ 0.50169, 0., 1., 950.
        [-0.76218, 0., 1., 425.
        [ 0.57478, 0., 1., 725.]])
```

Column selection

One lingering issue with the above approach is that we've had to hard code the column names (or indexes). Often we want to select columns based on their dtype (e.g. categorical vs numerical) this can be done via pandas or sklearn,

```
1 from sklearn.compose import make_column_selector
```

```

1 ct = make_column_transformer(
2     ( StandardScaler(),
3       make_column_selector(dtype_include=np.number)
4     ( OneHotEncoder(),
5       make_column_selector(dtype_include=[object,
6     ])
7
8 ct.fit_transform(books)

```

```

array([[ 0.12101,  0.35936,  1.      ,  0.      ],
       [ 0.51997,  0.9369 ,  1.      ,  0.      ],
       [ 0.85192,  1.32193,  1.      ,  0.      ],
       [-1.84637, -1.37326,  1.      ,  0.      ],
       [-0.43936,  0.16685,  1.      ,  0.      ],
       [-0.62209, -0.4107 ,  1.      ,  0.      ],
       [ 1.16561,  1.41818,  1.      ,  0.      ],
       [-1.31951, -1.75829,  0.      ,  1.      ],
       [ 0.3281 , -0.02567,  0.      ,  1.      ],
       [ 0.25501, -0.21818,  0.      ,  1.      ],
       [ 1.96962,  1.03316,  0.      ,  1.      ],
       [-1.29819, -1.37326,  0.      ,  1.      ],
       [ 0.50169,  0.9369 ,  0.      ,  1.      ],
       [-0.76218, -1.08449,  0.      ,  1.      ],
       [ 0.57478,  0.07059,  0.      ,  1.      ]])

```

```

1 ct = make_column_transformer(
2     ( StandardScaler(),
3       books.select_dtypes(include=[ 'number' ]).columns
4     ( OneHotEncoder(),
5       books.select_dtypes(include=[ 'object' ]).columns
6     ])
7
8 ct.fit_transform(books)

```

```

array([[ 0.12101,  0.35936,  1.      ,  0.      ],
       [ 0.51997,  0.9369 ,  1.      ,  0.      ],
       [ 0.85192,  1.32193,  1.      ,  0.      ],
       [-1.84637, -1.37326,  1.      ,  0.      ],
       [-0.43936,  0.16685,  1.      ,  0.      ],
       [-0.62209, -0.4107 ,  1.      ,  0.      ],
       [ 1.16561,  1.41818,  1.      ,  0.      ],
       [-1.31951, -1.75829,  0.      ,  1.      ],
       [ 0.3281 , -0.02567,  0.      ,  1.      ],
       [ 0.25501, -0.21818,  0.      ,  1.      ],
       [ 1.96962,  1.03316,  0.      ,  1.      ],
       [-1.29819, -1.37326,  0.      ,  1.      ],
       [ 0.50169,  0.9369 ,  0.      ,  1.      ],
       [-0.76218, -1.08449,  0.      ,  1.      ],
       [ 0.57478,  0.07059,  0.      ,  1.      ]])

```

```
1 ct.fit_transform(books)
```

```
array([[ 0.12101,  0.35936,  1.      ,  0.      ],
       [ 0.51997,  0.9369 ,  1.      ,  0.      ],
       [ 0.85192,  1.32193,  1.      ,  0.      ],
       [-1.84637, -1.37326,  1.      ,  0.      ],
       [-0.43936,  0.16685,  1.      ,  0.      ],
       [-0.62209, -0.4107 ,  1.      ,  0.      ],
       [ 1.16561,  1.41818,  1.      ,  0.      ],
       [-1.31951, -1.75829,  0.      ,  1.      ],
       [ 0.3281 , -0.02567,  0.      ,  1.      ],
       [ 0.25501, -0.21818,  0.      ,  1.      ],
       [ 1.96962,  1.03316,  0.      ,  1.      ],
       [-1.29819, -1.37326,  0.      ,  1.      ],
       [ 0.50169,  0.9369 ,  0.      ,  1.      ],
       [-0.76218, -1.08449,  0.      ,  1.      ],
       [ 0.57478,  0.07059,  0.      ,  1.      ]])
```

```
1 ct.get_feature_names_out()
```

```
array(['standardscaler__volume', 'standardscaler__we:
```

```
1 ct.fit_transform(books)
```

```
array([[ 0.12101,  0.35936,  1.      ,  0.      ],
       [ 0.51997,  0.9369 ,  1.      ,  0.      ],
       [ 0.85192,  1.32193,  1.      ,  0.      ],
       [-1.84637, -1.37326,  1.      ,  0.      ],
       [-0.43936,  0.16685,  1.      ,  0.      ],
       [-0.62209, -0.4107 ,  1.      ,  0.      ],
       [ 1.16561,  1.41818,  1.      ,  0.      ],
       [-1.31951, -1.75829,  0.      ,  1.      ],
       [ 0.3281 , -0.02567,  0.      ,  1.      ],
       [ 0.25501, -0.21818,  0.      ,  1.      ],
       [ 1.96962,  1.03316,  0.      ,  1.      ],
       [-1.29819, -1.37326,  0.      ,  1.      ],
       [ 0.50169,  0.9369 ,  0.      ,  1.      ],
       [-0.76218, -1.08449,  0.      ,  1.      ],
       [ 0.57478,  0.07059,  0.      ,  1.      ]])
```

```
1 ct.get_feature_names_out()
```

```
array(['standardscaler__volume', 'standardscaler__we:
```

Demo 1 - Putting it together

Interaction model

Cross validation & hyper parameter tuning

Ridge regression

One way to expand on the idea of least squares regression is to modify the loss function. One such approach is known as Ridge regression, which adds a scaled penalty for the sum of the squares of the **\betas** to the least squares loss.

$$\operatorname{argmin}_{\beta} \|y - X\beta\|^2 + \lambda(\beta^T \beta)$$

```
1 d = pd.read_csv("data/ridge.csv")
2 d
```

	y	x1	x2	x3	x4	x5
0	-0.151710	0.353658	1.633932	0.553257	1.415731	A
1	3.579895	1.311354	1.457500	0.072879	0.330330	B
2	0.768329	-0.744034	0.710362	-0.246941	0.008825	B
3	7.788646	0.806624	-0.228695	0.408348	-2.481624	B
4	1.394327	0.837430	-1.091535	-0.860979	-0.810492	A
..
495	-0.204932	-0.385814	-0.130371	-0.046242	0.004914	A
496	0.541988	0.845885	0.045291	0.171596	0.332869	A
497	-1.402627	-1.071672	-1.716487	-0.319496	-1.163740	C
498	-0.043645	1.744800	-0.010161	0.422594	0.772606	A
499	-1.550276	0.910775	-1.675396	1.921238	-0.232189	B

dummy coding

```
1 d = pd.get_dummies(d)
2 d
```

	y	x1	x2	x3	x4	x5_A	x5_B	x5_C	x5_D
0	-0.151710	0.353658	1.633932	0.553257	1.415731	1	0	0	0
1	3.579895	1.311354	1.457500	0.072879	0.330330	0	1	0	0
2	0.768329	-0.744034	0.710362	-0.246941	0.008825	0	1	0	0
3	7.788646	0.806624	-0.228695	0.408348	-2.481624	0	1	0	0
4	1.394327	0.837430	-1.091535	-0.860979	-0.810492	1	0	0	0
..
495	-0.204932	-0.385814	-0.130371	-0.046242	0.004914	1	0	0	0
496	0.541988	0.845885	0.045291	0.171596	0.332869	1	0	0	0
497	-1.402627	-1.071672	-1.716487	-0.319496	-1.163740	0	0	1	0
498	-0.043645	1.744800	-0.010161	0.422594	0.772606	1	0	0	0
499	-1.550276	0.910775	-1.675396	1.921238	-0.232189	0	1	0	0

[500 rows x 9 columns]

Fitting a ridge regression model

The `linear_model` submodule also contains the `Ridge` model which can be used to fit a ridge regression. Usage is identical other than `Ridge()` takes the parameter `alpha` to specify the regularization parameter.

```
1 from sklearn.linear_model import Ridge, LinearRegression
2
3 X, y = d.drop(["y"], axis=1), d.y
4
5 rg = Ridge(fit_intercept=False, alpha=10).fit(X, y)
6 lm = LinearRegression(fit_intercept=False).fit(X, y)
```

```
1 rg.coef_
```

```
array([ 0.97809,  1.96215,  0.00172, -2.94457,  0.451
```

```
1 lm.coef_
```

```
array([ 0.99505,  2.00762,  0.00232, -3.00088,  0.491
```

```
1 mean_squared_error(y, rg.predict(X))
```

```
0.019101431349883385
```

```
1 mean_squared_error(y, lm.predict(X))
```

```
0.009872435924102045
```

Generally for a Ridge (or Lasso) model it is important to scale the features before fitting (i.e. `StandardScaler()`) -

Test-Train split

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.

```
1 from sklearn.model_selection import train_test_split
2
3 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234)
```

```
1 X.shape
```

```
(500, 8)
```

```
1 X_train.shape
```

```
(400, 8)
```

```
1 X_test.shape
```

```
(100, 8)
```

```
1 y.shape
```

```
(500,)
```

```
1 y_train.shape
```

```
(400,)
```

```
1 y_test.shape
```

```
(100,)
```

```
1 x_train
```

```
      x1      x2      x3      x4 x5_A x!
296 -0.261142 -0.887193 -0.441300  0.053902    0
220  0.155596  0.551363  0.749117  0.875181    0
0    0.353658  1.633932  0.553257  1.415731    1
255 -1.206309 -0.073534 -1.920777 -0.554861    1
335 -0.380790 -0.117404 -0.037709  0.202757    0
..      ...      ...      ...      ...    ...
204 -2.646094  1.170804 -0.185098  0.165830    0
53  -0.483511  0.452531  0.223226 -0.753872    0
294 -1.424818 -0.396870 -0.595927 -1.114747    1
211 -1.000845 -0.842665  0.407765  0.375650    0
303  1.037404 -0.961266  0.433180  0.890055    0
```

```
[400 rows x 8 columns]
```

```
1 y_train
```

```
296 -2.462944
220 -1.760134
0    -0.151710
255  0.668016
335 -1.178652
...
204 -0.657622
53   2.831201
294  1.566109
211 -3.711740
303 -3.552971
Name: y, Length: 400, dtype: float64
```

Train vs Test rmse

```
1 alpha = np.logspace(-2,1, 100)
2 train_rmse = []
3 test_rmse = []
4
5 for a in alpha:
6     rg = Ridge(alpha=a).fit(X_train, y_train)
7
8     train_rmse.append(
9         mean_squared_error(
10             y_train, rg.predict(X_train), squared=False
11         )
12     )
13     test_rmse.append(
14         mean_squared_error(
15             y_test, rg.predict(X_test), squared=False
16         )
17     )
18
19 res = pd.DataFrame(
20     data = {"alpha": alpha,
21             "train": train_rmse,
22             "test": test_rmse}
23 )
```

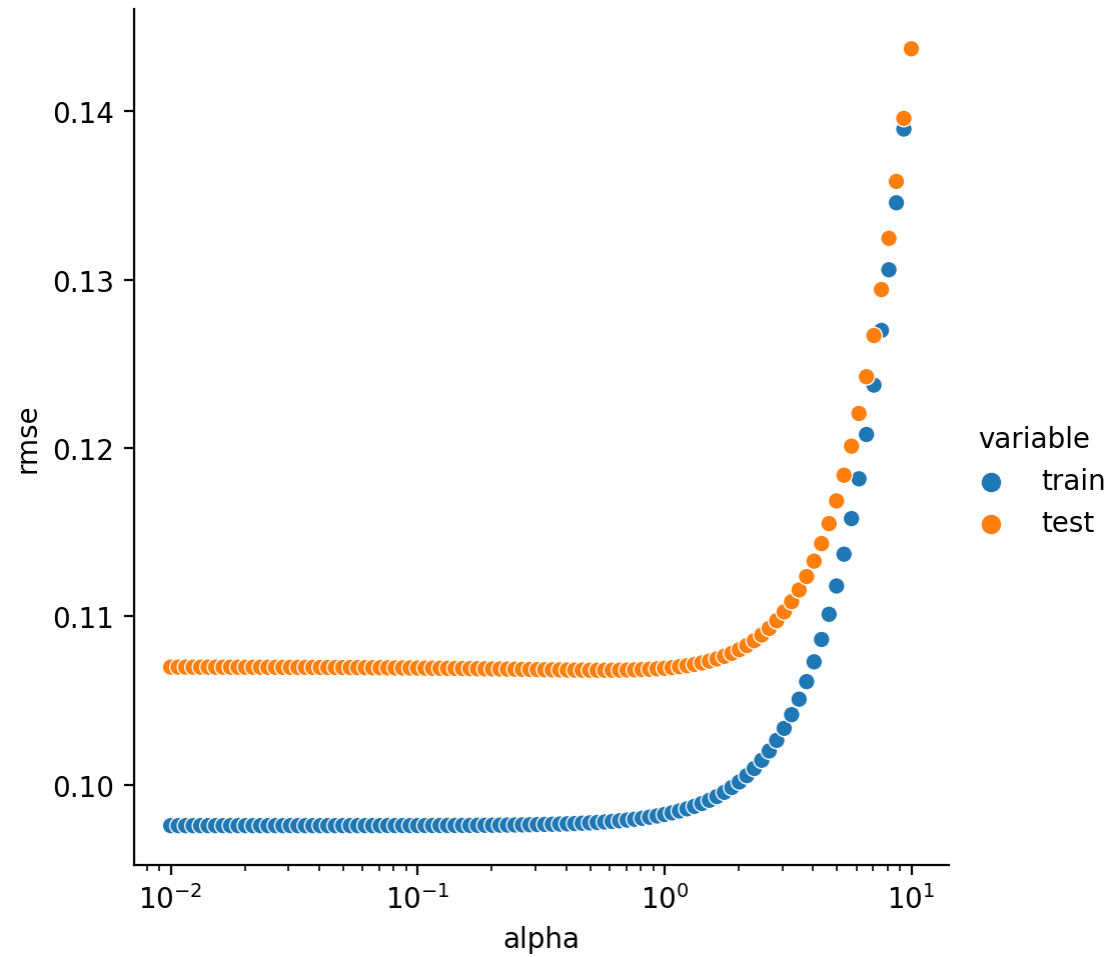
	alpha	train	test
0	0.010000	0.097568	0.106985
1	0.010723	0.097568	0.106984
2	0.011498	0.097568	0.106984
3	0.012328	0.097568	0.106983
4	0.013219	0.097568	0.106983
..
95	7.564633	0.126990	0.129414
96	8.111308	0.130591	0.132458
97	8.697490	0.134568	0.135838
98	9.326033	0.138950	0.139581
99	10.000000	0.143764	0.143715

[100 rows x 3 columns]

```

1 sns.relplot(
2     x="alpha", y="rmse", hue="variable", data = pd.melt(res, id_vars=["alpha"],value_name="rmse")
3 ).set(
4     xscale="log"
5 )

```



Best alpha?

```
1 min_i = np.argmin(res.train)
2 min_i
```

0

```
1 res.iloc[[min_i],:]
```

	alpha	train	test
0	0.01	0.097568	0.106985

```
1 min_i = np.argmin(res.test)
2 min_i
```

58

```
1 res.iloc[[min_i],:]
```

	alpha	train	test
58	0.572237	0.097787	0.1068

k-fold cross validation

The previous approach was relatively straight forward, but it required a fair bit of book keeping to implement and we only examined a single test/train split. If we would like to perform k-fold cross validation we can use `cross_val_score` from the `model_selection` submodule.

```
1 from sklearn.model_selection import cross_val_score
2
3 cross_val_score(
4     Ridge(alpha=0.59, fit_intercept=False),
5     X, y,
6     cv=5,
7     scoring="neg_root_mean_squared_error"
8 )
```

```
array([-0.09364, -0.09995, -0.10474, -0.10273, -0.10597])
```

►►► Note that the default k-fold cross validation used here does not shuffle your data which can be massively

Controlling k-fold behavior

Rather than providing `cv` as an integer, it is better to specify a cross-validation scheme directly (with additional options). Here we will use the `KFold` class from the `model_selection` submodule.

```
1 from sklearn.model_selection import KFold
2
3 cross_val_score(
4     Ridge(alpha=0.59, fit_intercept=False),
5     X, y,
6     cv = KFold(n_splits=5, shuffle=True, random_state=1234),
7     scoring="neg_root_mean_squared_error"
8 )
```

```
array([-0.10658, -0.104    , -0.1037 , -0.10125, -0.09228])
```


KFold object

`KFold()` returns a class object which provides the method `split()` which in turn is a generator that returns a tuple with the indexes of the training and testing selects for each fold given a model matrix `X`,

```
1 ex = pd.DataFrame(data = list(range(10)), columns=["x"])
```

```
1 cv = KFold(5)
2 for train, test in cv.split(ex):
3     print(f'Train: {train} | test: {t
```

```
Train: [2 3 4 5 6 7 8 9] | test: [0 1]
Train: [0 1 4 5 6 7 8 9] | test: [2 3]
Train: [0 1 2 3 6 7 8 9] | test: [4 5]
Train: [0 1 2 3 4 5 8 9] | test: [6 7]
Train: [0 1 2 3 4 5 6 7] | test: [8 9]
```

```
1 cv = KFold(5, shuffle=True, random_
2 for train, test in cv.split(ex):
3     print(f'Train: {train} | test: {t
```

```
Train: [0 1 3 4 5 6 8 9] | test: [2 7]
Train: [0 2 3 4 5 6 7 8] | test: [1 9]
Train: [1 2 3 4 5 6 7 9] | test: [0 8]
Train: [0 1 2 3 6 7 8 9] | test: [4 5]
Train: [0 1 2 4 5 7 8 9] | test: [3 6]
```

Train vs Test rmse (again)

```
1 alpha = np.logspace(-2,1, 30)
2 test_mean_rmse = []
3 test_rmse = []
4 cv = KFold(n_splits=5, shuffle=True, random_state=1234)
5
6 for a in alpha:
7     rg = Ridge(fit_intercept=False, alpha=a).fit(X_train, y_train)
8
9     scores = -1 * cross_val_score(
10         rg, X, y,
11         cv = cv,
12         scoring="neg_root_mean_squared_error"
13     )
14     test_mean_rmse.append(np.mean(scores))
15     test_rmse.append(scores)
16
17 res = pd.DataFrame(
18     data = np.c_[alpha, test_mean_rmse, test_rmse],
19     columns = ["alpha", "mean_rmse"] + ["fold" + str(i) for i in range(1,6) ]
20 )
```

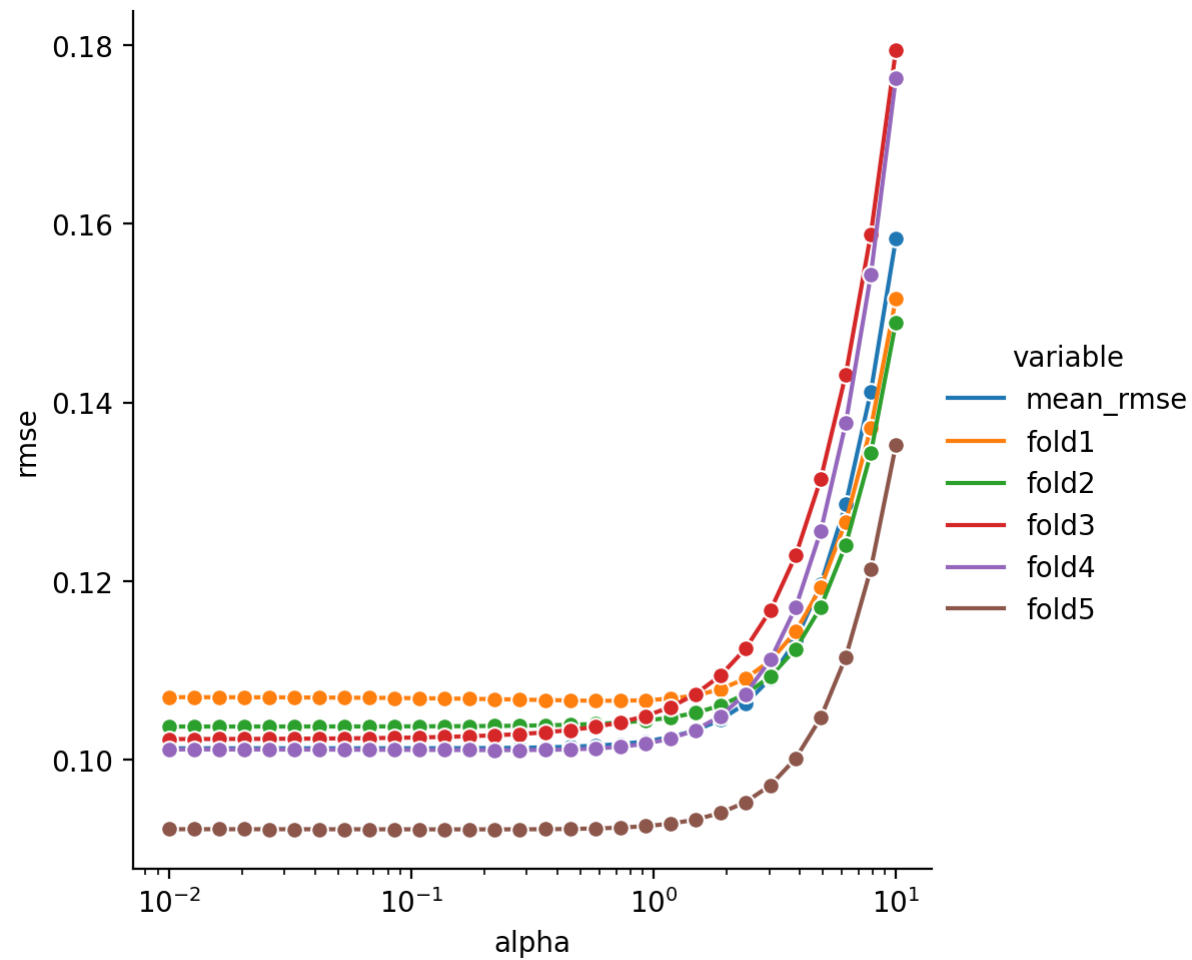
1 res

	alpha	mean_rmse	fold1	fold2	fold3	fold4	fold5
0	0.010000	0.101257	0.106979	0.103691	0.102288	0.101130	0.092195
1	0.012690	0.101257	0.106976	0.103692	0.102292	0.101129	0.092194
2	0.016103	0.101256	0.106971	0.103692	0.102298	0.101126	0.092194
3	0.020434	0.101256	0.106966	0.103693	0.102306	0.101123	0.092193
4	0.025929	0.101256	0.106959	0.103694	0.102316	0.101120	0.092191
5	0.032903	0.101256	0.106951	0.103696	0.102328	0.101116	0.092190
6	0.041753	0.101256	0.106940	0.103698	0.102344	0.101110	0.092188
7	0.052983	0.101256	0.106927	0.103701	0.102365	0.101104	0.092186
8	0.067234	0.101257	0.106911	0.103704	0.102391	0.101096	0.092184
9	0.085317	0.101259	0.106890	0.103709	0.102426	0.101088	0.092181
10	0.108264	0.101262	0.106865	0.103716	0.102471	0.101078	0.092178
11	0.137382	0.101267	0.106835	0.103725	0.102529	0.101069	0.092176
12	0.174333	0.101276	0.106800	0.103739	0.102607	0.101060	0.092174
13	0.221222	0.101291	0.106758	0.103758	0.102710	0.101055	0.092175
14	0.280722	0.101317	0.106712	0.103786	0.102848	0.101059	0.092180
15	0.356225	0.101360	0.106663	0.103828	0.103036	0.101078	0.092193
16	0.452035	0.101430	0.106617	0.103890	0.103293	0.101128	0.092221
17	0.573615	0.101544	0.106584	0.103984	0.103650	0.101229	0.092273
18	0.727895	0.101729	0.106580	0.104128	0.104149	0.101420	0.092367
19	0.923671	0.102026	0.106639	0.104348	0.104856	0.101757	0.092530
20	1.172102	0.102501	0.106809	0.104690	0.105864	0.102334	0.092805
21	1.487352	0.103253	0.107174	0.105220	0.107314	0.103295	0.093263

```

1 sns.relplot(
2     x="alpha", y="rmse", hue="variable", data=res.melt(id_vars=["alpha"], value_name="rmse"),
3     marker="o", kind="line"
4 ).set(
5     xscale="log"
6 )

```



Best alpha? (again)

```
1 i = res.drop(  
2     ["alpha"], axis=1  
3 ).agg(  
4     np.argmin  
5 ).to_numpy()  
6  
7 i = np.sort(np.unique(i))  
8  
9 res.iloc[ i, : ]
```

	alpha	mean_rmse	fold1	fold2	fold3	fold4	fold5
0	0.010000	0.101257	0.106979	0.103691	0.102288	0.101130	0.092195
5	0.032903	0.101256	0.106951	0.103696	0.102328	0.101116	0.092190
12	0.174333	0.101276	0.106800	0.103739	0.102607	0.101060	0.092174
13	0.221222	0.101291	0.106758	0.103758	0.102710	0.101055	0.092175
18	0.727895	0.101729	0.106580	0.104128	0.104149	0.101420	0.092367

Aside - Available metrics

For most of the cross validation functions we pass in a string instead of a scoring function from the metrics submodule - if you are interested in seeing the names of the possible metrics, these are available via the `sklearn.metrics.SCORERS` dictionary,

```
1 np.array( sorted(  
2     sklearn.metrics.SCORERS.keys()  
3 ) )
```

```
array(['accuracy', 'adjusted_mutual_info_score', 'adjusted_rand_score', 'average_precision', 'balanced_accu  
      'f1_samples', 'f1_weighted', 'fowlkes_mallows_score', 'homogeneity_score', 'jaccard', 'jaccard_macro'  
      'mutual_info_score', 'neg_brier_score', 'neg_log_loss', 'neg_mean_absolute_error', 'neg_mean_absolute  
      'neg_mean_squared_error', 'neg_mean_squared_log_error', 'neg_median_absolute_error', 'neg_negative_li  
      'positive_likelihood_ratio', 'precision', 'precision_macro', 'precision_micro', 'precision_samples',  
      'recall_samples', 'recall_weighted', 'roc_auc', 'roc_auc_ovo', 'roc_auc_ovo_weighted', 'roc_auc_ovr',
```

Grid Search

We can further reduce the amount of code needed if there is a specific set of parameter values we would like to explore using cross validation. This is done using the `GridSearchCV` function from the `model_selection` submodule.

```
1 from sklearn.model_selection import GridSearchCV
2
3 gs = GridSearchCV(
4     Ridge(fit_intercept=False),
5     {"alpha": np.logspace(-2, 1, 30)},
6     cv = KFold(5, shuffle=True, random_state=1234),
7     scoring = "neg_root_mean_squared_error"
8 ).fit(
9     X, y
10 )
```

```
1 gs.best_index_
```

5

```
1 gs.best_params_
```

```
{'alpha': 0.03290344562312668}
```

```
1 gs.best_score_
```

```
-0.10125611767453653
```


best_estimator_ attribute

If `refit = True` (the default) with `GridSearchCV()` then the `best_estimator_` attribute will be available which gives direct access to the “best” model or pipeline object. This model is constructed by using the parameter(s) that achieved the maximum score and refitting the model to the complete data set.

```
1 gs.best_estimator_
```

```
Ridge(alpha=0.03290344562312668, fit_intercept=False)
```

```
1 gs.best_estimator_.coef_
```

```
array([ 0.99499,  2.00747,  0.00231, -3.0007 ,  0.49316,  0.10189, -0.29408,  1.00767])
```

```
1 gs.best_estimator_.predict(X)
```

```
array([ -0.12179,  3.34151,  0.76055,  7.89292,  1.56523, -5.33575, -4.37469,  3.13003, -0.16859, -
-1.96548,  2.99039,  0.56796, -5.26672,  5.4966 ,  3.47247, -2.66117,  3.35011,  0.64221, -
 0.76008,  5.49779,  2.6521 , -0.83127,  0.04167, -1.92585, -2.48865,  2.29127,  3.62514, -
-2.78598, -12.55143,  2.79189, -1.89763, -5.1769 ,  1.87484,  2.18345, -6.45358,  0.91006,
 1.04564, -1.54843,  0.76161, -1.65495,  0.22378, -0.68221,  0.12976,  2.58875,  2.54421, -
 0.36935,  0.87397,  9.22348, -1.29078,  1.74347, -1.55169, -0.69398, -1.40445,  0.23072,
 1.70208,  7.15821,  3.96172,  5.75363, -4.50718, -5.81785, -2.47424,  1.19276,  2.57431, -
 2.65413, -0.67486, -3.01324,  0.34118, -3.83856,  0.33096, -3.59485, -1.55578,  0.96765,
-2.65588, -5.77111, -1.20292,  2.66903, -1.11387,  3.05231,  6.34596, -1.42886, -2.29709, -
 1.14603, -3.35087, -5.91052, -1.23355,  2.8308 , -3.21438,  4.09019, -5.95969, -0.98044,
 2.67859,  2.45406, -2.28901,  1.56963, -5.51199,  2.67089,  2.39878,  6.65249,
```

-0.85644,	1.90162,	-1.23686,	3.22403,	5.31725,	0.31415,	0.17128,	-1.53623,	1.73354,	-
-0.67864,	-0.67348,	4.22499,	3.34704,	-1.44927,	-6.3229 ,	4.83881,	-3.71184,	6.32207,	
5.69233,	6.28949,	5.37201,	-0.63177,	2.88795,	4.01781,	7.03453,	1.76797,	5.86793,	
-0.92299,	-4.85603,	4.18714,	-3.60775,	-2.31532,	1.27459,	0.37238,	-1.21 ,	2.44074,	-
0.8058 ,	0.23748,	1.13615,	0.63385,	-0.2395 ,	6.07024,	0.85521,	0.18951,	3.27772,	-
-10.86754,	-9.25489,	7.0615 ,	0.01263,	3.93274,	3.40325,	-1.57858,	-4.94508,	-2.69779,	
-0.03725,	-1.15642,	8.92035,	2.63769,	-1.39664,	1.62241,	-4.87487,	-2.49769,	1.39569,	-
-4.41299,	-4.79775,	-3.79204,	-3.61711,	-2.92489,	7.15104,	-3.24195,	3.03705,	-4.01473,	-
10.77554,	-1.64465,	-2.13624,	-2.16392,	1.92049,	-2.47602,	-4.34462,	-2.09427,	-0.32466,	
-3.28827,	-5.73513,	4.76249,	-1.24714,	0.08253,	-1.71446,	1.3742 ,	1.85738,	-6.37864,	-
-4.48298,	-0.28666,	-4.92509,	2.6523 ,	-4.59622,	3.09283,	3.50353,	-6.1787 ,	-2.08203,	-

cv_results_ attribute

Other useful details about the grid search process are stored in the dictionary `cv_results_` attribute which includes things like average test scores, fold level test scores, test ranks, test runtimes, etc.

```
1 gs.cv_results_.keys()
```

```
dict_keys(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time', 'param_alpha', 'params', 's
```

```
1 gs.cv_results_["param_alpha"]
```

```
masked_array(data=[0.01, 0.01268961003167922, 0.01610262027560939, 0.020433597178569417, 0.02592943797404667,
                  0.08531678524172806, 0.10826367338740546, 0.1373823795883263, 0.17433288221999882, 0.2212,
                  0.5736152510448679, 0.727895384398315, 0.9236708571873861, 1.1721022975334805, 1.48735210,
                  4.893900918477494, 6.2101694189156165, 7.880462815669913, 10.0],
             mask=[False, False, False, False, False, False, False, False, False, False, False, False, False,
                  False, False, False, False, False],
             fill_value='?',
             dtype=object)
```

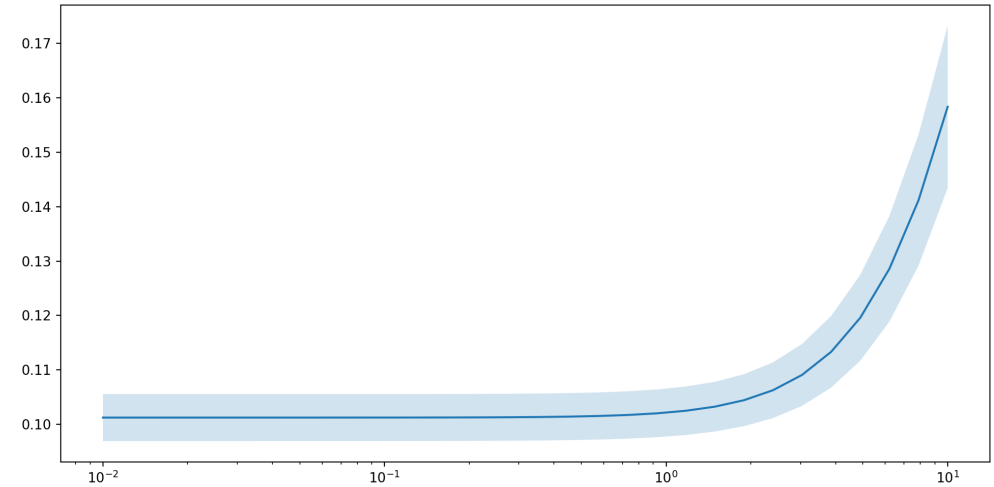
```
1 gs.cv_results_["mean_test_score"]
```

```
array([-0.10126, -0.10126, -0.10126, -0.10126, -0.10126, -0.10126, -0.10126, -0.10126, -0.10126, -0.10126, -
      -0.10203, -0.1025 , -0.10325, -0.10444, -0.10627, -0.10909, -0.11333, -0.11959, -0.12859, -0.14119, -
```

```

1  alpha = np.array(gs.cv_results_["param_alpha"], dtype=float)
2  score = -gs.cv_results_["mean_test_score"]
3  score_std = gs.cv_results_["std_test_score"]
4  n_folds = gs.cv.get_n_splits()
5
6  plt.figure(layout="constrained")
7
8  ax = sns.lineplot(x=alpha, y=score)
9  ax.set_xscale("log")
10
11 plt.fill_between(
12     x = alpha,
13     y1 = score + 1.96*score_std / np.sqrt(n_folds)
14     y2 = score - 1.96*score_std / np.sqrt(n_folds)
15     alpha = 0.2
16 )
17
18 plt.show()

```



Ridge traceplot

```
1 alpha = np.logspace(-1,5, 100)
2 betas = []
3
4 for a in alpha:
5     rg = Ridge(alpha=a).fit(X, y)
6
7     betas.append(rg.coef_)
8
9 res = pd.DataFrame(
10     data = betas, columns = rg.feature_names_in_
11 ).assign(
12     alpha = alpha
13 )
```

```

1 g = sns.relplot(
2     data = res.melt(id_vars="alpha", value_name="coef values", var_name="feature"),
3     x = "alpha", y = "coef values", hue = "feature",
4     kind = "line", aspect=2
5 )
6 g.set(xscale="log")

```

