scikit-learn Cross-validation & Classification

Lecture 11

Dr. Colin Rundel

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 squared βs to the least squares loss.

$$\underset{\boldsymbol{\beta}}{\text{argmin}} \; \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2 + \lambda(\boldsymbol{\beta}^T\boldsymbol{\beta})$$

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

[500 rows x 6 columns]
```

dummy coding

```
d = pd.get_dummies(d); d
```

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

[500 rows x 9 columns]
```

Fitting a ridge regession 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.

```
from sklearn.linear_model import Ridge, LinearRegression

X, y = d.drop(["y"], axis=1), d.y

lm = LinearRegression(fit_intercept=False).fit(X, y)
rg = Ridge(fit_intercept=False, alpha=10).fit(X, y)
```

```
lm.coef_
```

```
array([ 0.99505,  2.00762,  0.00232, -3.00088,
0.49329,  0.10193, -0.29413,  1.00856])
```

```
root_mean_squared_error(y, lm.predict(X))
```

```
0.09936013246821912
```

```
rg.coef_
```

```
array([ 0.97809,  1.96215,  0.00172, -2.94457,
0.45558,  0.09001, -0.28193,  0.79781])
```

```
root_mean_squared_error(y, rg.predict(X))
```

```
0.13820792795597311
```

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

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.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
  X, y, test_size=0.2, random_state=1234
)
```

```
X.shape
```

```
(500, 8)
```

```
y.shape
```

```
(500,)
```

```
X_train.shape
```

```
(400, 8)
```

```
y_train.shape
```

```
(400,)
```

```
X_test.shape
```

```
(100, 8)
```

```
y_test.shape
```

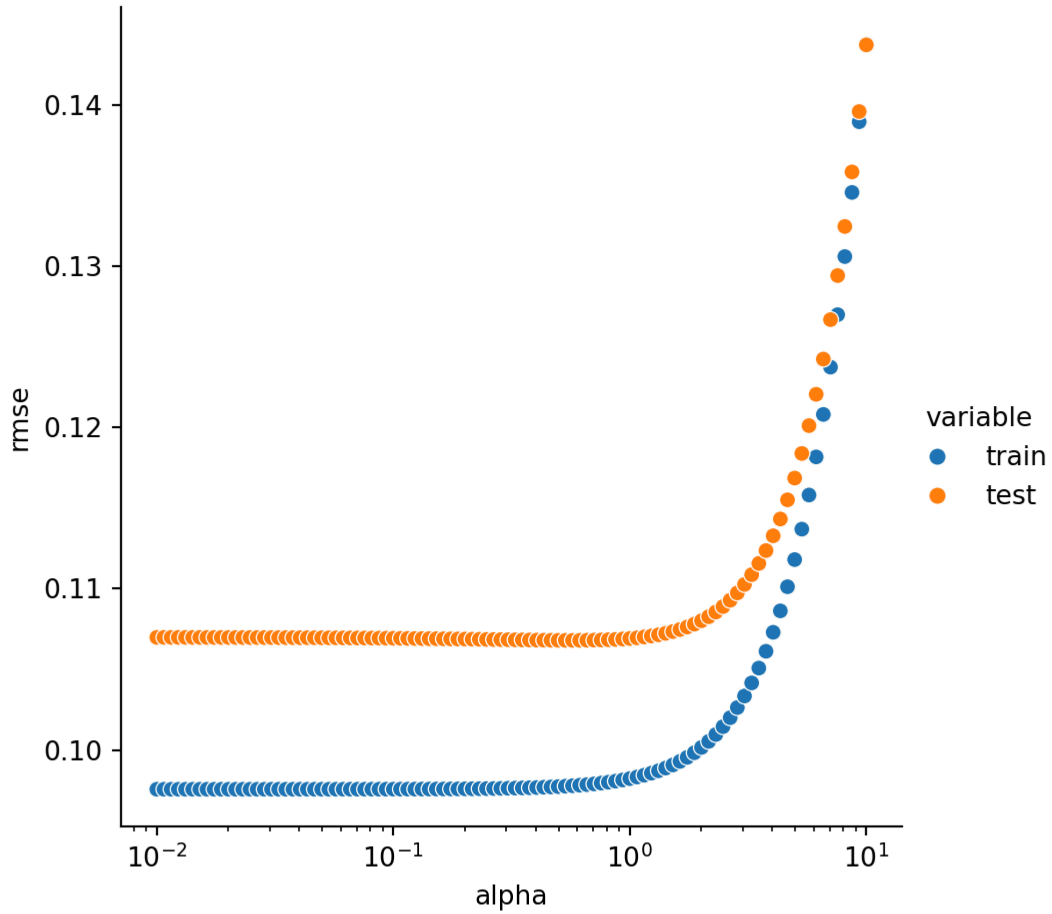
```
(100,)
```

Train vs Test rmse

```
alpha = np.logspace(-2,1, 100)
train_rmse = []
test_rmse = []

for a in alpha:
    rg = Ridge(alpha=a).fit(
      X_train, y_train
    )
    train_rmse.append(
     root_mean_squared_error(
        y_train, rg.predict(X_train)
      )
    )
    test_rmse.append(
      root_mean_squared_error(
        y_test, rg.predict(X_test)
      )
    )

res = pd.DataFrame(
  data = {"alpha": alpha,
          "train": train_rmse,
          "test": test_rmse}
)
```

```
        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]
```

```python
g = sns.relplot(
  x="alpha", y="rmse", hue="variable", data = pd.melt(res, id_vars=["alpha"],value_name="rmse
).set(
  xscale="log"
)
```



Best alpha?

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

```
np.int64(0)
```

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

	alpha	train	test
0	0.01	0.097568	0.106985

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

```
np.int64(58)
```

```
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 bookkeeping 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.

```
from sklearn.model_selection import cross_val_score

cross_val_score(
  Ridge(alpha=0.59, fit_intercept=False),
  X, y,
  cv=5,
  scoring="neg_root_mean_squared_error"
)
```

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

🚩🚩🚩 Note that the default k-fold cross validation used here does not shuffle the 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.

```python
from sklearn.model_selection import KFold

cross_val_score(
  Ridge(alpha=0.59, fit_intercept=False),
  X, y,
  cv = KFold(n_splits=5, shuffle=True, random_state=1234),
  scoring="neg_root_mean_squared_error"
)
```

```
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`,

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

```
cv = KFold(5)
for train, test in cv.split(ex):
  print(f'Train: {train} | test: {test}')
```

```
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]
```

```
cv = KFold(5, shuffle=True, random_state=12
for train, test in cv.split(ex):
  print(f'Train: {train} | test: {test}')
```

```
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]
```

`scoring`

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 `sklearn.metrics.get_scorer_names()`,

```
np.array( sklearn.metrics.get_scorer_names() )
```

```
array(['accuracy', 'adjusted_mutual_info_score', 'adjusted_rand_score',
       'average_precision', 'balanced_accuracy', 'completeness_score',
       'd2_absolute_error_score', 'explained_variance', 'f1', 'f1_macro', 'f1_micro',
       'f1_samples', 'f1_weighted', 'fowlkes_mallows_score', 'homogeneity_score',
       'jaccard', 'jaccard_macro', 'jaccard_micro', 'jaccard_samples',
       'jaccard_weighted', 'matthews_corrcoef', 'mutual_info_score', 'neg_brier_score',
       'neg_log_loss', 'neg_max_error', 'neg_mean_absolute_error',
       'neg_mean_absolute_percentage_error', 'neg_mean_gamma_deviance',
       'neg_mean_poisson_deviance', 'neg_mean_squared_error',
       'neg_mean_squared_log_error', 'neg_median_absolute_error',
       'neg_negative_likelihood_ratio', 'neg_root_mean_squared_error',
       'neg_root_mean_squared_log_error', 'normalized_mutual_info_score',
       'positive_likelihood_ratio', 'precision', 'precision_macro', 'precision_micro',
       'precision_samples', 'precision_weighted', 'r2', 'rand_score', 'recall',
       'recall_macro', 'recall_micro', 'recall_samples', 'recall_weighted', 'roc_auc',
       'roc_auc_ovo', 'roc_auc_ovo_weighted', 'roc_auc_ovr', 'roc_auc_ovr_weighted',
       'top_k_accuracy', 'v_measure_score'], dtype='<U34')
```

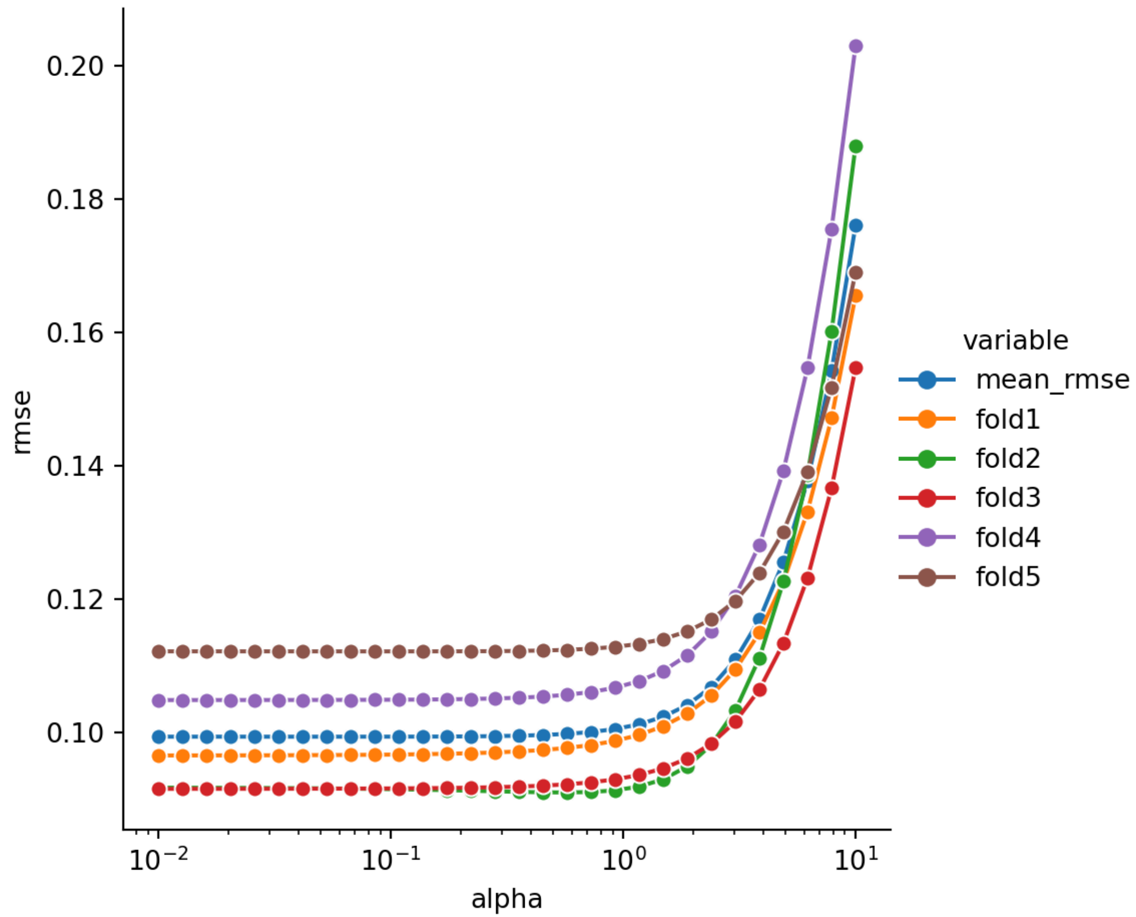
Train vs Test rmse (again)

```
alpha = np.logspace(-2,1, 30)
test_mean_rmse = []
test_rmse = []
cv = KFold(n_splits=5, shuffle=True, random_state=1234)

for a in alpha:
    rg = Ridge(fit_intercept=False, alpha=a)

    scores = -1 * cross_val_score(
      rg, X_train, y_train,
      cv = cv,
      scoring="neg_root_mean_squared_error"
    )
    test_mean_rmse.append(np.mean(scores))
    test_rmse.append(scores)

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

```python
res
```

	alpha	mean_rmse	fold1	fold2	fold3	fold4	fold5
0	0.010000	0.099393	0.096577	0.091750	0.091573	0.104881	0.112186
1	0.012690	0.099393	0.096581	0.091743	0.091575	0.104882	0.112185
2	0.016103	0.099393	0.096585	0.091734	0.091577	0.104884	0.112185
3	0.020434	0.099392	0.096591	0.091722	0.091580	0.104885	0.112184
4	0.025929	0.099392	0.096599	0.091708	0.091583	0.104888	0.112183
5	0.032903	0.099392	0.096608	0.091690	0.091588	0.104891	0.112182
6	0.041753	0.099391	0.096621	0.091667	0.091594	0.104895	0.112181
7	0.052983	0.099392	0.096637	0.091639	0.091602	0.104900	0.112180
8	0.067234	0.099392	0.096657	0.091604	0.091612	0.104908	0.112179
9	0.085317	0.099394	0.096684	0.091561	0.091626	0.104919	0.112179
10	0.108264	0.099398	0.096720	0.091510	0.091644	0.104935	0.112179
11	0.137382	0.099405	0.096767	0.091448	0.091669	0.104958	0.112181
12	0.174333	0.099417	0.096829	0.091376	0.091704	0.104992	0.112186
13	0.221222	0.099439	0.096913	0.091294	0.091751	0.105042	0.112196
14	0.280722	0.099477	0.097028	0.091207	0.091819	0.105117	0.112215
15	0.356225	0.099540	0.097185	0.091121	0.091914	0.105231	0.112249
16	0.452035	0.099644	0.097403	0.091052	0.092052	0.105406	0.112309

```python
g = sns.relplot(
  x="alpha", y="rmse", hue="variable", data=res.melt(id_vars=["alpha"], value_name
  marker="o", kind="line"
).set(
  xscale="log"
)
```



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.

```
from sklearn.model_selection import GridSearchCV

gs = GridSearchCV(
  Ridge(fit_intercept=False),
  {"alpha": np.logspace(-2, 1, 30)},
  cv = KFold(5, shuffle=True, random_state=1234),
  scoring = "neg_root_mean_squared_error"
).fit(
  X, y
)
```

`best_*` attributes

`GridSearchCV()`'s return object contains attributes with details on the "best" model based on the chosen scoring metric.

```
gs.best_index_
```

```
np.int64(5)
```

```
gs.best_params_
```

```
{'alpha': np.float64(0.03290344562312668)}
```

```
gs.best_score_
```

```
np.float64(-0.101256117674536 5)
```

`best_estimator_` attribute

If `refit = True` (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 minimum score and refitting the model to the complete data set.

```
gs.best_estimator_
```

```
Ridge(alpha=np.float64(0.03290344562312668), fit_intercept=False)
```

```
gs.best_estimator_.coef_
```

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

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

```
array([ -0.12179,   3.34151,   0.76055,   7.89292,   1.56523,  -5.33575,  -4.37469,
         3.13003,  -0.16859,  -1.60087,  -1.89073,   1.44596,   3.99773,   4.70003,
        -6.45959,   4.11085,   3.60426,  -1.96548,   2.99039,   0.56796,  -5.26672,
         5.4966 ,   3.47247,  -2.66117,   3.35011,   0.64221,  -1.50238,   2.41562,
         3.11665,   1.11236,  -2.11839,   1.36006,  -0.53666,  -2.78112,   0.76008,
         5.49779,   2.6521 ,  -0.83127,   0.04167,  -1.92585,  -2.48865,   2.29127,
         3.62514,  -2.01226,  -0.69725,  -1.94514,  -0.47559,  -7.36557,  -3.20766,
         2.9218 ,  -0.8213 ,  -2.78598, -12.55143,   2.79189,  -1.89763,  -5.1769 ,
         1.87484,   2.18345,  -6.45358,   0.91006,   0.94792,   2.91799,   6.12323,
        -1.87654,   3.63259,  -0.53797,  -3.23506,  -2.23885,   1.04564,  -1.54843,
         0.76161,  -1.65495,   0.22378,  -0.68221,   0.12976,   2.58875,   2.54421,
        -3.69056,   3.73479,  -0.90278,   1.22394,  -3.22614,   7.16719,  -5.6168 ,
```

```
 3.3433 ,   0.36935,   0.87397,   9.22348,  -1.29078,   1.74347,  -1.55169,
-0.69398,  -1.40445,   0.23072,   1.06277,   2.84797,   2.35596,  -1.93292,
 8.35129,  -2.98221,  -6.35071,  -5.15138,   1.70208,   7.15821,   3.96172,
 5.75363,  -4.50718,  -5.81785,  -2.47424,   1.19276,   2.57431,  -2.57053,
-0.53682,  -1.65955,   1.99839,  -6.19607,  -1.73962,  -2.11993,  -2.29362,
```

`cv_results_` attribute

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

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

```
dict_keys(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time', 'param_alpha',
'params', 'split0_test_score', 'split1_test_score', 'split2_test_score', 'split3_test_score',
'split4_test_score', 'mean_test_score', 'std_test_score', 'rank_test_score'])
```

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

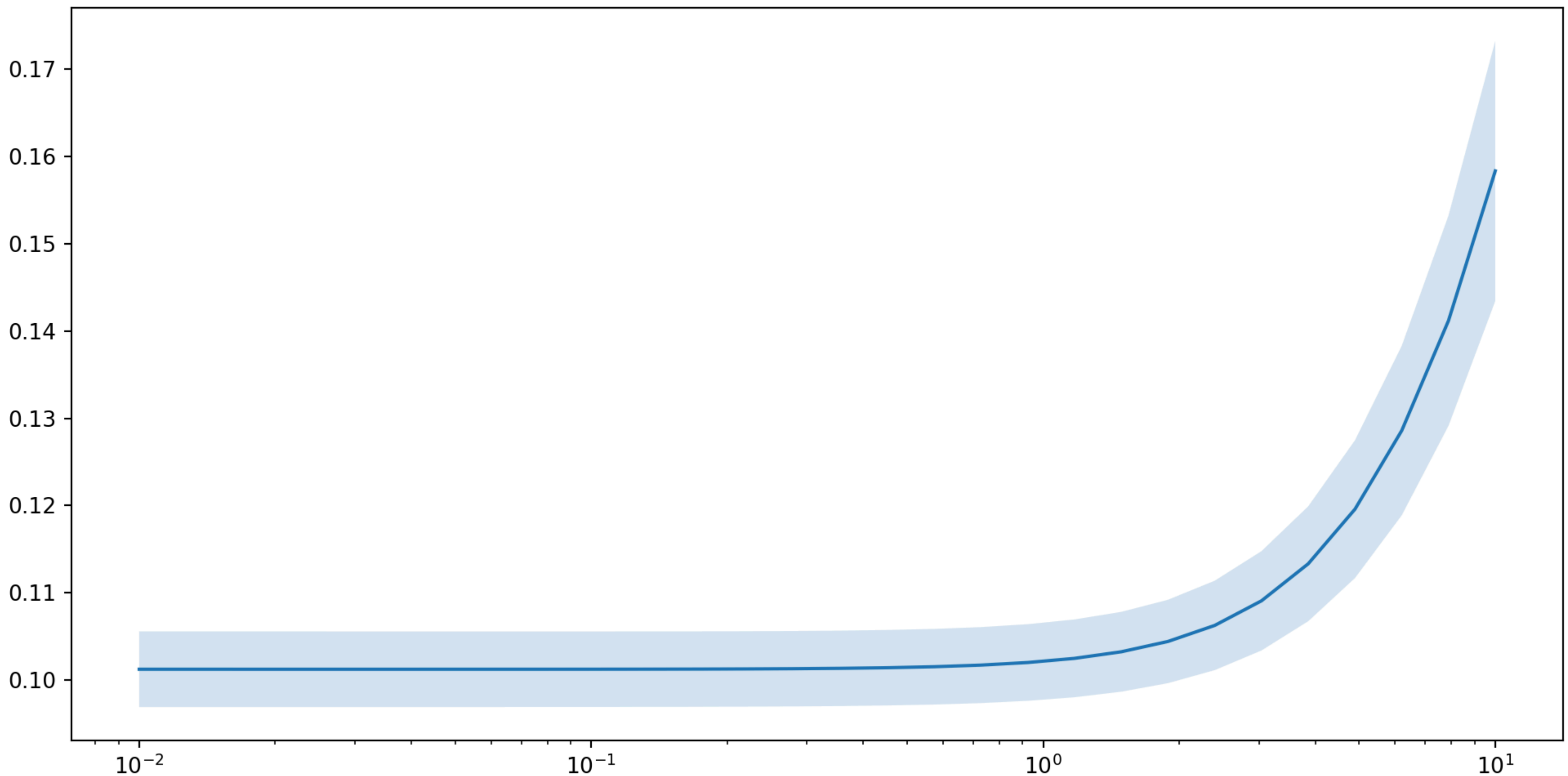
```
array([-0.10126, -0.10126, -0.10126, -0.10126, -0.10126, -0.10126, -0.10126, -0.10126,
       -0.10126, -0.10126, -0.10126, -0.10127, -0.10128, -0.10129, -0.10132, -0.10136,
       -0.10143, -0.10154, -0.10173, -0.10203, -0.1025 , -0.10325, -0.10444, -0.10627,
       -0.10909, -0.11333, -0.11959, -0.12859, -0.14119, -0.15832])
```

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

```
masked_array(data=[0.01, 0.012689610031679222, 0.016102620275609393, 0.020433597178569417,
                   0.02592943797404667, 0.03290344562312668, 0.041753189365604,
                   0.05298316906283707, 0.06723357536499334, 0.08531678524172806,
                   0.10826367338740546, 0.1373823795883263, 0.17433288221999882,
                   0.2212216291070449, 0.2807216203941177, 0.3562247890262442,
                   0.4520353656360243, 0.5736152510448679, 0.727895384398315,
                   0.9236708571873861, 1.1721022975334805, 1.4873521072935119,
                   1.8873918221350976, 2.395026619987486, 3.039195382313198,
```

```
             3.856620421163472, 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, False, False,
             False, False, False, False, False, False, False, False, False, False],
  fill_value=1e+20)
```

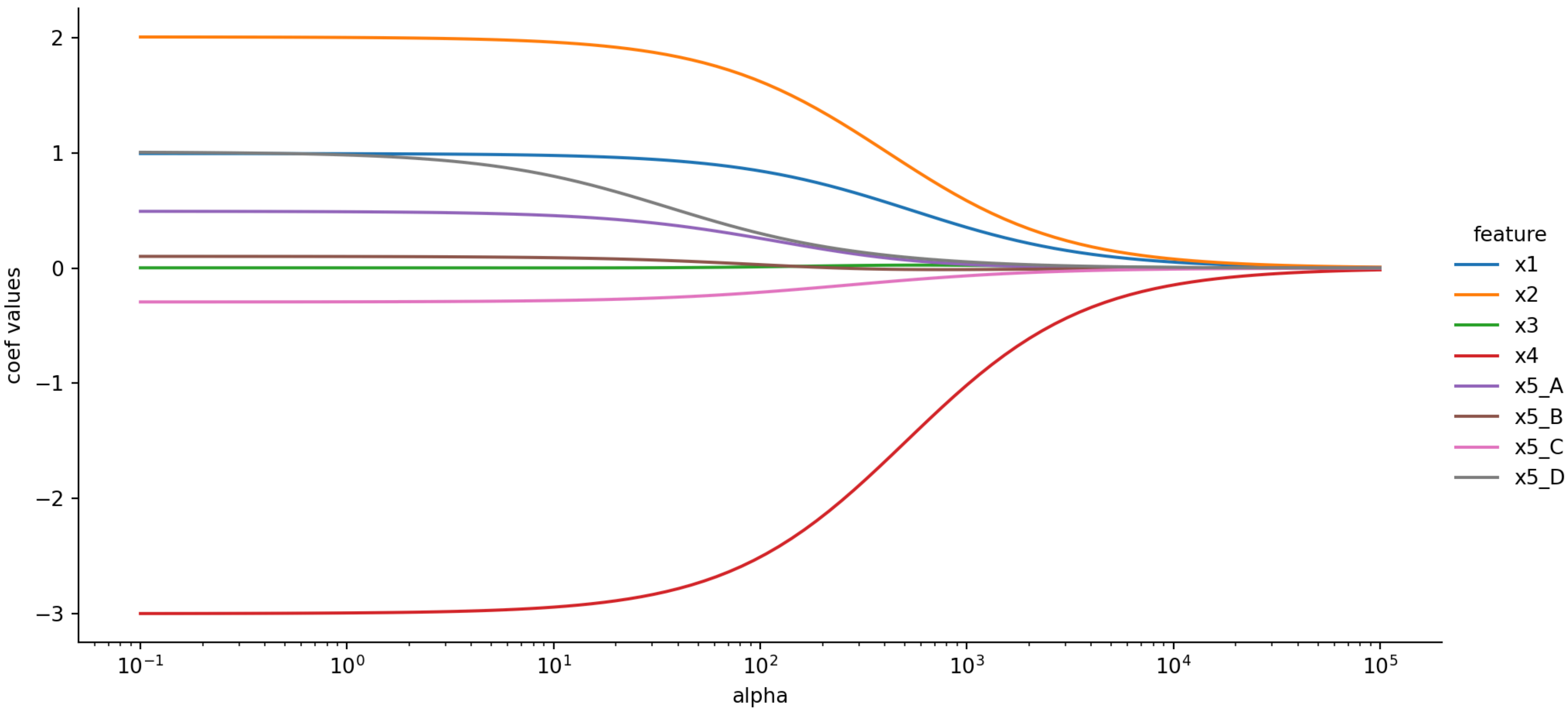
```python
alpha = np.array(gs.cv_results_["param_alpha"], dtype="float64")
score = -gs.cv_results_["mean_test_score"]
score_std = gs.cv_results_["std_test_score"]
n_folds = gs.cv.get_n_splits()

plt.figure(layout="constrained")
ax = sns.lineplot(x=alpha, y=score)
ax.set_xscale("log")
plt.fill_between(
  x = alpha,
  y1 = score + 1.96*score_std / np.sqrt(n_folds),
  y2 = score - 1.96*score_std / np.sqrt(n_folds),
  alpha = 0.2
)
plt.show()
```


0.17
0.16
0.15
0.14
0.13
0.12
0.11
0.10
10^{-2}
10^{-1}
10^{0}
10^{1}

Ridge traceplot

```
alpha = np.logspace(-1,5, 100)
betas = []

for a in alpha:
    rg = Ridge(alpha=a, fit_intercept=False).fit(X, y)
    betas.append(rg.coef_)

res = pd.DataFrame(
  data = betas, columns = rg.feature_names_in_
).assign(
  alpha = alpha
)
```

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



Classification

OpenIntro - Spam

We will start by looking at a data set on spam emails from the OpenIntro project. A full data dictionary can be found here. To keep things simple this week we will restrict our exploration to including only the following columns: `spam`, `exclaim_mess`, `format`, `num_char`, `line_breaks`, and `number`.

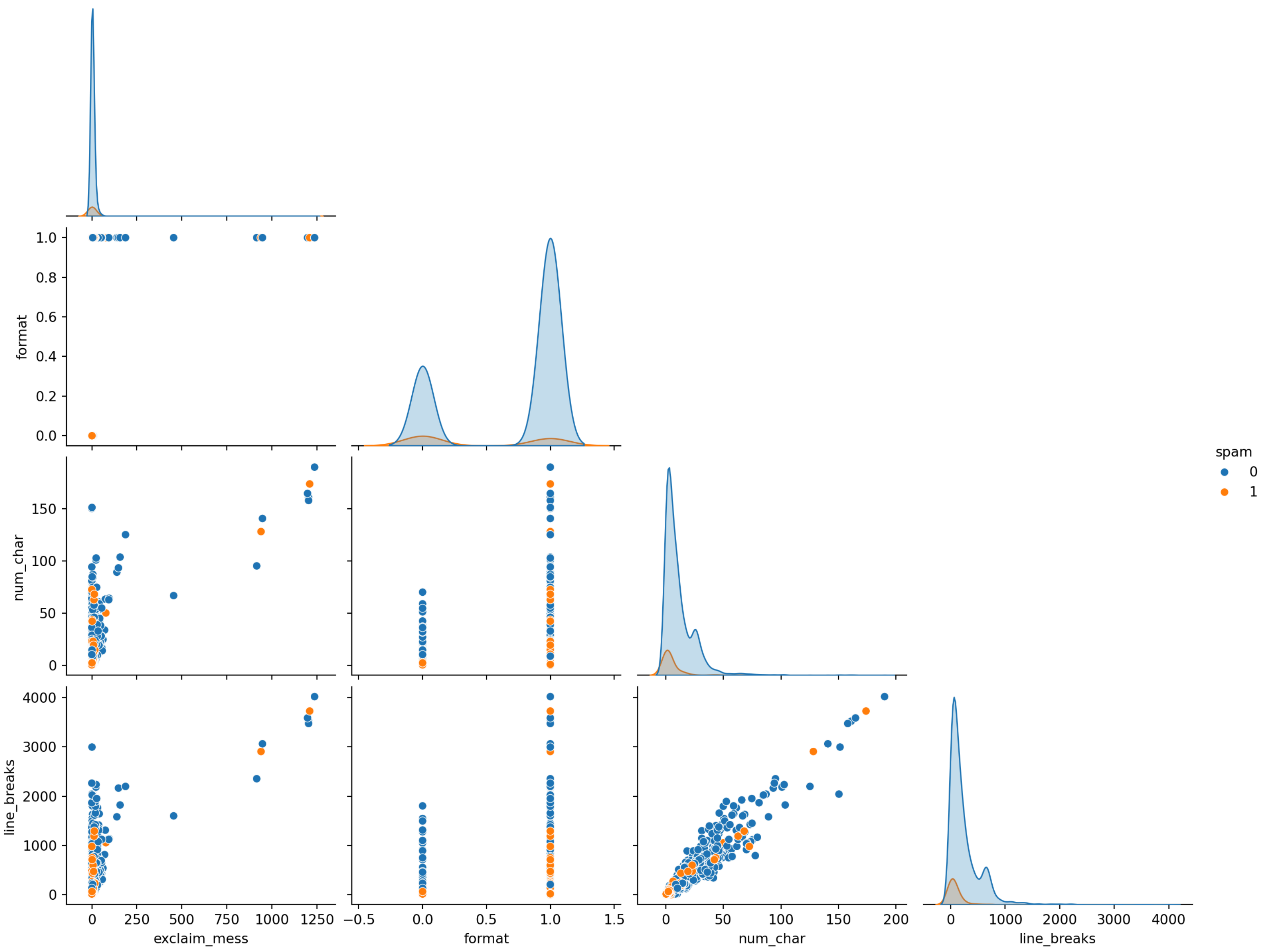
- `spam` - Indicator for whether the email was spam.
- `exclaim_mess` - The number of exclamation points in the email message.
- `format` - Indicates whether the email was written using HTML (e.g. may have included bolding or active links).
- `num_char` - The number of characters in the email, in thousands.
- `line_breaks` - The number of line breaks in the email (does not count text wrapping).
- `number` - Factor variable saying whether there was no number, a small number (under 1 million), or a big number.

As `number` is categorical, we will take care of the necessary dummy coding via `pd.get_dummies()`,

```
email = pd.read_csv('data/email.csv')[
  ['spam', 'exclaim_mess', 'format', 'num_char', 'line_breaks', 'number']
]
email_dc = pd.get_dummies(email)
email_dc
```

```
      spam  exclaim_mess  format  ...  number_big  number_none  number_small
0        0             0       1  ...        True        False         False
1        0             1       1  ...       False        False          True
2        0             6       1  ...       False        False          True
3        0            48       1  ...       False        False          True
4        0             1       0  ...       False         True         False
...    ...           ...     ...  ...         ...          ...           ...
3916     1             0       0  ...       False        False          True
3917     1             0       0  ...       False        False          True
3918     0             5       1  ...       False        False          True
3919     0             0       0  ...       False        False          True
3920     1             1       0  ...       False        False          True

[3921 rows x 8 columns]
```

```python
g = sns.pairplot(email, hue='spam', corner=True, aspect=1.25)
```



Model fitting

```
from sklearn.linear_model import LogisticRegression

y = email_dc.spam
X = email_dc.drop('spam', axis=1)

m = LogisticRegression(fit_intercept = False).fit(X, y)
```

```
m.feature_names_in_
```

```
array(['exclaim_mess', 'format', 'num_char', 'line_breaks', 'number_big', 'number_none',
       'number_small'], dtype=object)
```

```
m.coef_
```

```
array([[ 0.00982, -0.619  ,  0.05449, -0.00556, -1.21233, -0.6932 , -1.92064]])
```

A quick comparison

R output

```
glm(spam~.-1, data=d, family=binomial) |>
  coef()
```

```
exclaim_mess        format      num_char
line_breaks     numberbig   numbernone
 0.009586821 -0.604781649  0.054765496
-0.005480427 -1.264826746 -0.706842516
 numbersmall
-1.950440237
```

sklearn output

```
m.feature_names_in_
```

```
array(['exclaim_mess', 'format', 'num_char',
'line_breaks', 'number_big', 'number_none',
       'number_small'], dtype=object)
```

```
m.coef_
```

```
array([[ 0.00982, -0.619  ,  0.05449, -0.00556,
-1.21233, -0.6932 , -1.92064]])
```

`sklearn.linear_model.LogisticRegression`

From the documentations,

> This class implements regularized logistic regression using the 'liblinear' library, 'newton-cg', 'sag', 'saga' and 'lbfgs' solvers. **Note that regularization is applied by default.** It can handle both dense and sparse input. Use C-ordered arrays or CSR matrices containing 64-bit floats for optimal performance; any other input format will be converted (and copied).

Penalty parameter

🚩🚩🚩

`LogisticRegression()` has a parameter called penalty that applies a `"l1"` (lasso), `"l2"` (ridge), `"elasticnet"` or `None` with `"l2"` being the default. To make matters worse, the regularization is controlled by the parameter `C` which defaults to 1 (not 0) - also `C` is the inverse regularization strength (e.g. different from `alpha` for ridge and lasso models).

🚩🚩🚩

$$\min_{w,c} \frac{1-\rho}{2} w^T w + \rho |w|_1 + C \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1),$$

Another quick comparison

R output

```r
glm(spam~.-1, data = d, family=binomial) |>
  coef()
```

```
exclaim_mess        format      num_char
line_breaks     numberbig    numbernone
 0.009586821 -0.604781649  0.054765496
-0.005480427 -1.264826746 -0.706842516
 numbersmall
-1.950440237
```

sklearn output (penalty None)

```python
m = LogisticRegression(
  fit_intercept = False, penalty=None
).fit(
  X, y
)
m.feature_names_in_
```

```
array(['exclaim_mess', 'format', 'num_char',
'line_breaks', 'number_big', 'number_none',
       'number_small'], dtype=object)
```

```python
m.coef_
```

```
array([[ 0.00958, -0.60663,  0.05478, -0.00548,
-1.26108, -0.70611, -1.94884]])
```

Solver parameter

It is also possible specify the solver to use when fitting a logistic regression model, to complicate matters somewhat the choice of the algorithm depends on the penalty chosen:

- `newton-cg` - [`"l2"`, `None`]
- `lbfgs` - [`"l2"`, `None`]
- `liblinear` - [`"l1"`, `"l2"`]
- `sag` - [`"l2"`, `None`]
- `saga` - [`"elasticnet"`, `"l1"`, `"l2"`, `None`]

Also there can be issues with feature scales for some of these solvers:

> **Note:** ‘sag’ and ‘saga’ fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from sklearn.preprocessing.

Prediction

Classification models have multiple prediction methods depending on what type of output you would like,

```
m.predict(X)
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, ..., 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0], shape=(3921,))
```

Scoring

Classification models also include a `score()` method which returns the model's *accuracy*,

```
m.score(X, y)
```

```
0.9064014282070901
```

Other scoring options are available via the metrics submodule

```
from sklearn.metrics import accuracy_score, roc_auc_score, f1_score, confusion_matrix
```

```
accuracy_score(y, m.predict(X))
```

```
0.9064014282070901
```

```
roc_auc_score(y, m.predict_proba(X)[:,1])
```

```
np.float64(0.7607243785641231)
```

```
f1_score(y, m.predict(X))
```

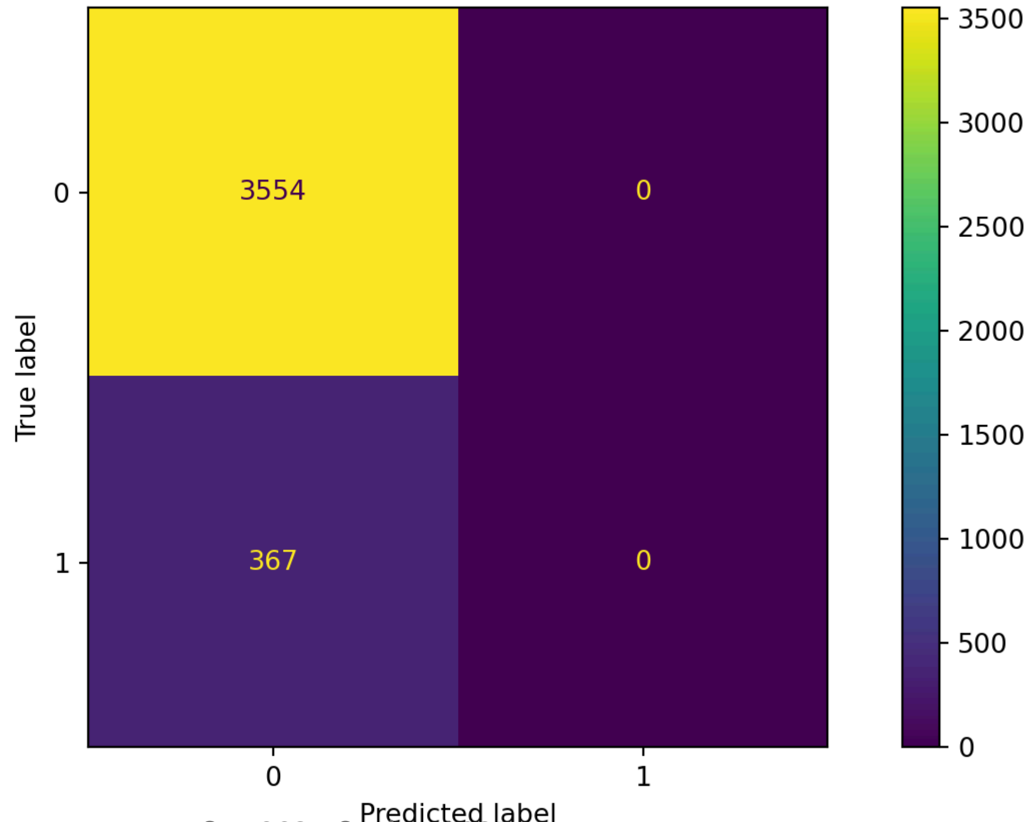
```
0.0
```

```
confusion_matrix(y, m.predict(X), labels=m.c
```

```
array([[3554,    0],
       [ 367,    0]])
```

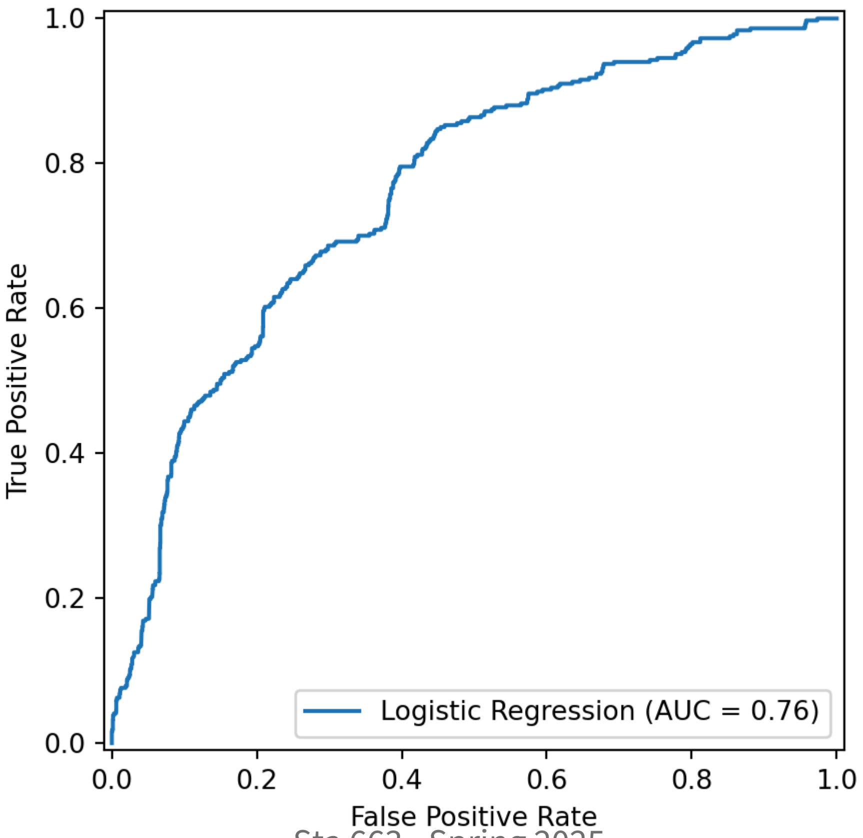
Scoring visualizations - confusion matrix

```
from sklearn.metrics import ConfusionMatrixDisplay
cm = confusion_matrix(y, m.predict(X), labels=m.classes_)

disp = ConfusionMatrixDisplay(cm).plot()
plt.show()
```



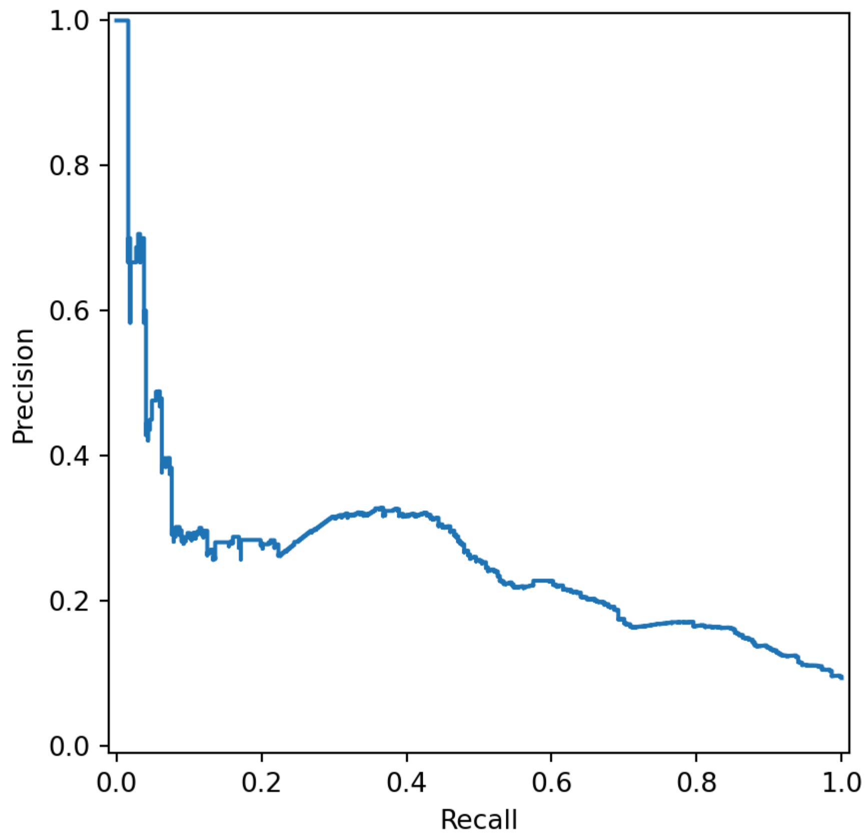
Scoring visualizations - ROC curve

```
from sklearn.metrics import auc, roc_curve, RocCurveDisplay

fpr, tpr, thresholds = roc_curve(y, m.predict_proba(X)[:,1])
roc_auc = auc(fpr, tpr)
disp = RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc,
                       estimator_name='Logistic Regression').plot()
plt.show()
```



Scoring visualizations - Precision Recall

```
from sklearn.metrics import precision_recall_curve, PrecisionRecallDisplay

precision, recall, _ = precision_recall_curve(y, m.predict_proba(X)[:,1])
disp = PrecisionRecallDisplay(precision=precision, recall=recall).plot()
plt.show()
```



MNIST

MNIST handwritten digits

```
from sklearn.datasets import load_digits
digits = load_digits(as_frame=True)
```

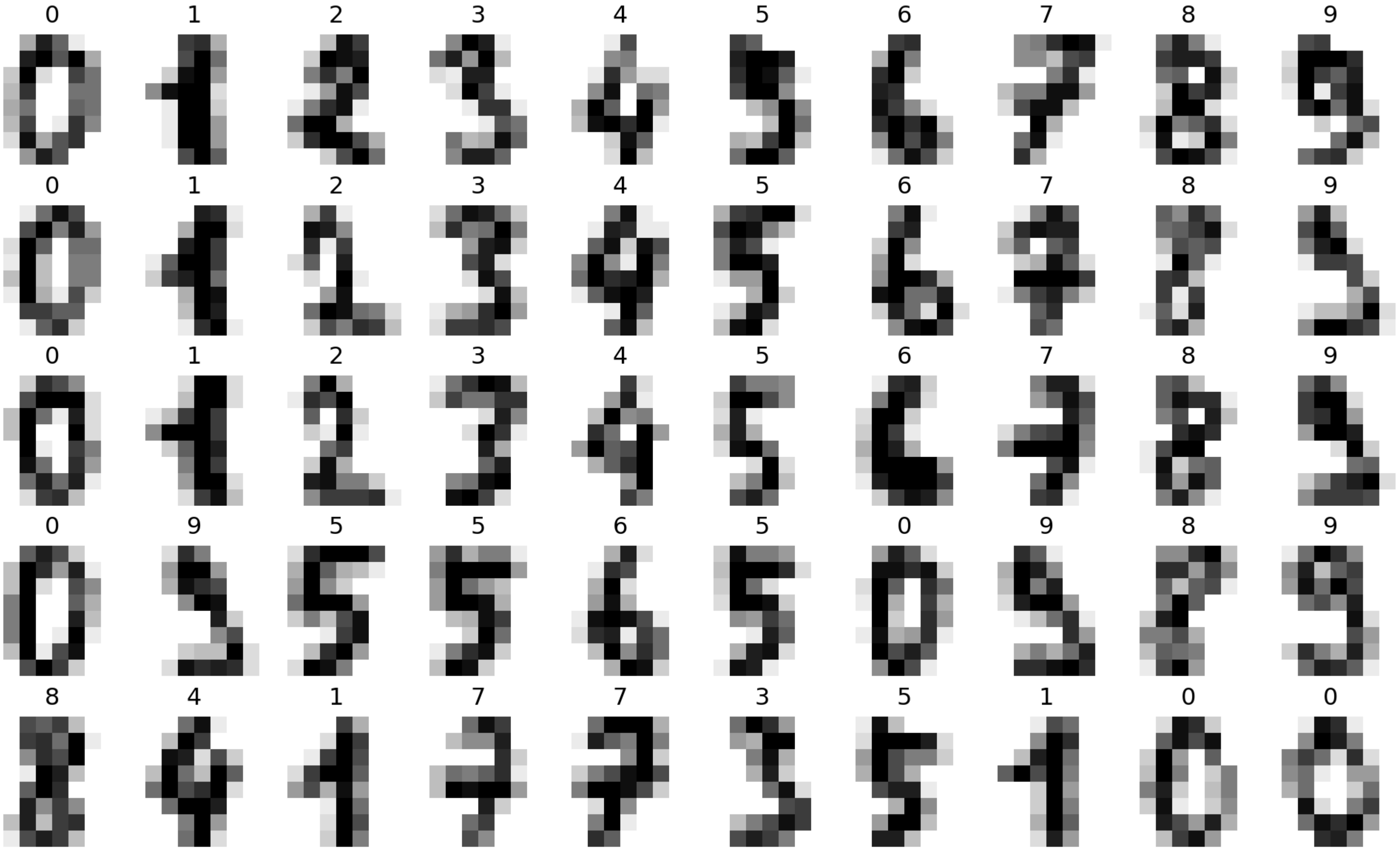
```
X = digits.data
X
```

```
      pixel_0_0  pixel_0_1  pixel_0_2  ...  pixel_7_5  pixel_7_6
pixel_7_7
0           0.0        0.0        5.0  ...        0.0        0.0
0.0
1           0.0        0.0        0.0  ...       10.0        0.0
0.0
2           0.0        0.0        0.0  ...       16.0        9.0
0.0
3           0.0        0.0        7.0  ...        9.0        0.0
0.0
4           0.0        0.0        0.0  ...        4.0        0.0
0.0
...         ...        ...        ...  ...        ...        ...
...
1792        0.0        0.0        4.0  ...        9.0        0.0
0.0
1793        0.0        0.0        6.0  ...        6.0        0.0
```

```
y = digits.target
y
```

```
0       0
1       1
2       2
3       3
4       4
       ..
1792    9
1793    0
1794    8
1795    9
1796    8
Name: target, Length: 1797,
dtype: int64
```

Example digits



Doing things properly - train/test split

To properly assess our modeling we will create a training and testing set of these data, only the training data will be used to learn model coefficients or hyperparameters, test data will only be used for final model scoring.

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, shuffle=True, random_state=1234
)
```

Multiclass logistic regression

Fitting a multiclass logistic regression model will involve selecting a value for the `multi_class` parameter, which can be either `multinomial` for multinomial regression or `ovr` for one-vs-rest where `k` binary models are fit.

```
mc_log_cv = GridSearchCV(
  LogisticRegression(penalty=None, max_iter = 5000),
  param_grid = {"multi_class": ["multinomial", "ovr"]},
  cv = KFold(10, shuffle=True, random_state=12345)
).fit(
  X_train, y_train
)
```

```
mc_log_cv.best_estimator_
```

```
LogisticRegression(max_iter=5000, multi_class='multinomial', penalty=None)
```

```
mc_log_cv.best_score_
```

```
np.float64(0.9468044077134987)
```

```
for param, score in  zip(mc_log_cv.cv_results_["params"], mc_log_cv.cv_results_["mean_test_sc
  f"{param=}, {score=}"
```

```
"param={'multi_class': 'multinomial'}, score=np.float64(0.9468044077134987)"
"param={'multi_class': 'ovr'}, score=np.float64(0.8927548209366393)"
```

Model coefficients

```
pd.DataFrame(
  mc_log_cv.best_estimator_.coef_
)
```

```
     0         1         2         3   ...        60        61        62        63
0  0.0 -0.075305 -0.460840  0.501212  ... -0.223561 -0.907504 -0.428323 -0.104403
1  0.0 -0.103594 -0.700507  0.761253  ...  0.200720  1.452495  0.731313  1.301708
2  0.0  0.068725  0.332420  0.435007  ...  0.458393  1.469109  1.391141  0.424594
3  0.0  0.137573 -0.165962  0.251926  ...  0.264964  0.595178  0.292151 -0.565132
4  0.0 -0.062672 -0.674331 -1.194883  ... -0.920782 -0.395735 -0.377680 -0.056259
5  0.0  0.383520  2.342286 -0.380072  ... -0.021154 -0.803677 -1.149027 -0.117984
6  0.0 -0.059272 -0.831077 -0.734510  ...  0.154089  1.384478  0.555634 -0.345914
7  0.0  0.048905  0.776396  0.665069  ... -1.829921 -1.503090 -0.408205 -0.057764
8  0.0 -0.193920 -0.196370 -1.052127  ...  0.977735 -1.241261 -0.868792 -0.366412
9  0.0 -0.143959 -0.422015  0.747124  ...  0.939517 -0.049994  0.261787 -0.112433

[10 rows x 64 columns]
```

```
mc_log_cv.best_estimator_.coef_.shape
```

```
(10, 64)
```

```
mc_log_cv.best_estimator_.intercept_
```

```
array([ 0.00855, -0.06081, -0.00306,  0.04737,  0.05523, -0.01002, -0.00606,  0.02771,
       -0.00762, -0.0513 ])
```

Confusion Matrix

Within sample

```
accuracy_score(
  y_train,
  mc_log_cv.best_estimator_.predict(X_train)
)
```

```
1.0
```

```
confusion_matrix(
  y_train,
  mc_log_cv.best_estimator_.predict(X_train)
)
```

```
array([[125,   0,   0,   0,   0,   0,   0,   0,   0,   0],
       [  0, 118,   0,   0,   0,   0,   0,   0,   0,   0],
       [  0,   0, 119,   0,   0,   0,   0,   0,   0,   0],
       [  0,   0,   0, 123,   0,   0,   0,   0,   0,   0],
       [  0,   0,   0,   0, 110,   0,   0,   0,   0,   0],
       [  0,   0,   0,   0,   0, 114,   0,   0,   0,   0],
       [  0,   0,   0,   0,   0,   0, 124,   0,   0,   0],
       [  0,   0,   0,   0,   0,   0,   0, 124,   0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0, 119,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0, 127]])
```

Out of sample

```
accuracy_score(
  y_test,
  mc_log_cv.best_estimator_.predict(X_test)
)
```

```
0.9494949494949495
```

```
confusion_matrix(
  y_test,
  mc_log_cv.best_estimator_.predict(X_test),
  labels = digits.target_names
)
```

```
array([[53,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0, 62,  0,  0,  1,  0,  0,  0,  1,  0],
       [ 0,  2, 56,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  1, 58,  0,  1,  0,  0,  0,  0],
       [ 1,  0,  0,  0, 69,  0,  0,  0,  1,  0],
       [ 0,  0,  0,  1,  1, 64,  1,  0,  0,  1],
       [ 1,  1,  0,  0,  0,  0, 55,  0,  0,  0],
       [ 0,  0,  0,  0,  2,  0,  0, 53,  0,  0],
       [ 0,  4,  2,  3,  1,  0,  0,  0, 43,  2],
       [ 0,  0,  0,  0,  0,  1,  0,  0,  1, 51]])
```

Report

```
print( classification_report(
  y_test,
  mc_log_cv.best_estimator_.predict(X_test)
) )
```

```
              precision    recall  f1-score   support

           0       0.96      1.00      0.98        53
           1       0.90      0.97      0.93        64
           2       0.95      0.97      0.96        58
           3       0.94      0.97      0.95        60
           4       0.93      0.97      0.95        71
           5       0.97      0.94      0.96        68
           6       0.98      0.96      0.97        57
           7       1.00      0.96      0.98        55
           8       0.93      0.78      0.85        55
           9       0.94      0.96      0.95        53

    accuracy                           0.95       594
   macro avg       0.95      0.95      0.95       594
weighted avg       0.95      0.95      0.95       594
```

Prediction

```
mc_log_cv.best_estimator_.predict(X_test)
```

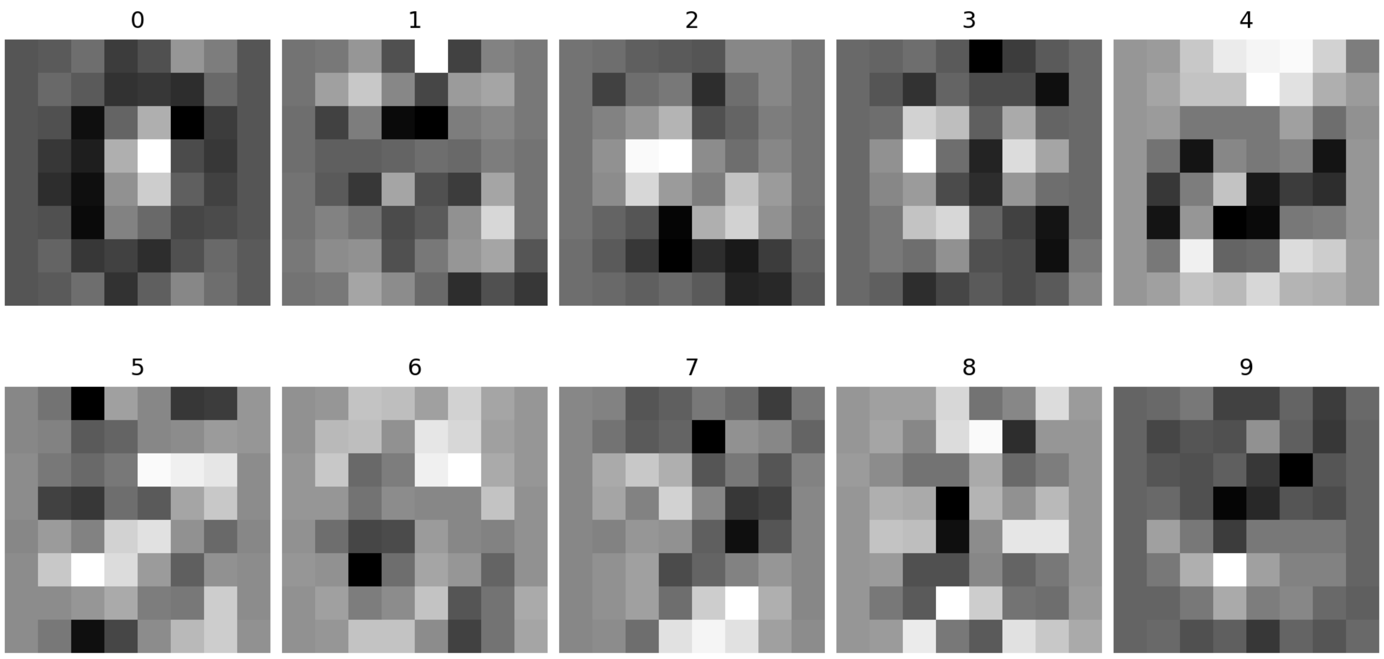
```
array([7, 1, 7, 6, 0, 2, 4, 3, 6, 3, 7, 8, 7, 9, 4, 3, 3,
       7, 8, 4, 0, 3, 9, 1, 3, 6, 6, 0, 5, 4, 1, 2, 1, 2,
       3, 2, 7, 6, 4, 8, 6, 4, 4, 0, 9, 1, 9, 5, 4, 4, 4,
       1, 7, 6, 9, 2, 9, 9, 9, 0, 4, 3, 1, 8, 8, 1, 3, 9,
       1, 3, 9, 6, 9, 5, 2, 1, 9, 2, 1, 3, 8, 7, 3, 3, 2,
       7, 7, 5, 8, 2, 6, 8, 9, 1, 6, 4, 5, 2, 2, 4, 5, 4,
       4, 6, 5, 9, 2, 4, 1, 0, 7, 6, 1, 2, 9, 5, 2, 5, 0,
       3, 2, 7, 6, 4, 8, 2, 1, 1, 6, 4, 6, 2, 3, 4, 7, 5,
       0, 9, 1, 0, 5, 6, 7, 6, 3, 8, 3, 2, 0, 4, 0, 1, 5,
       4, 6, 1, 1, 1, 6, 1, 7, 9, 0, 7, 9, 5, 4, 1, 3, 8,
       6, 4, 7, 1, 5, 7, 4, 7, 4, 5, 2, 2, 1, 1, 4, 4, 3,
       5, 5, 9, 4, 5, 5, 9, 3, 9, 3, 1, 2, 0, 8, 2, 8, 9,
       2, 4, 6, 8, 3, 9, 1, 0, 8, 1, 8, 5, 6, 8, 7, 1, 8,
       2, 4, 9, 7, 0, 5, 5, 6, 1, 3, 0, 5, 8, 2, 0, 9, 8,
       6, 7, 8, 4, 1, 0, 5, 2, 5, 1, 6, 4, 7, 1, 2, 6, 4,
       4, 6, 3, 2, 3, 2, 6, 5, 2, 9, 4, 7, 0, 1, 0, 4, 3,
       1, 2, 7, 9, 8, 5, 9, 5, 7, 0, 4, 8, 4, 9, 4, 0, 7,
       7, 2, 5, 3, 5, 3, 9, 7, 5, 5, 2, 7, 0, 8, 9, 1, 7,
       9, 8, 5, 0, 2, 0, 8, 7, 0, 9, 5, 5, 9, 6, 1, 2, 3,
       9, 1, 3, 2, 9, 3, 4, 3, 4, 1, 0, 1, 8, 5, 0, 9, 2,
       7, 2, 3, 5, 2, 6, 3, 4, 1, 5, 0, 5, 4, 6, 3, 2, 5,
       0, 4, 3, 6, 0, 8, 6, 0, 0, 2, 2, 0, 1, 4, 6, 5, 0,
       9, 5, 6, 8, 4, 4, 2, 8, 2, 9, 4, 7, 3, 8, 6, 3, 8,
```

```
mc_log_cv.best_estimator_.predict_proba(X_test),
```

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

Examining the coefs

```
coef_img = mc_log_cv.best_estimator_.coef_.reshape(10,8,8)

fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(10, 5), layout="constrained")
axes2 = [ax for row in axes for ax in row]

for ax, image, label in zip(axes2, coef_img, range(10)):
    ax.set_axis_off()
    img = ax.imshow(image, cmap=plt.cm.gray_r, interpolation="nearest")
    txt = ax.set_title(f"{label}")

plt.show()
```



Example 1 - DecisionTreeClassifier

Using these data we will now fit a `DecisionTreeClassifier` to these data, we will employ `GridSearchCV` to tune some of the parameters (`max_depth` at a minimum) - see the full list here.

```python
from sklearn.datasets import load_digits
digits = load_digits(as_frame=True)


X, y = digits.data, digits.target
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, shuffle=True, random_state=1234
)
```

Example 2 - GridSearchCV w/ Multiple models (Trees vs Forests)