scikit-learn classification

Lecture 16

Dr. Colin Rundel

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.

```python
email = pd.read_csv('data/email.csv')[
  ['spam', 'exclaim_mess', 'format', 'num_char',
]
email
```

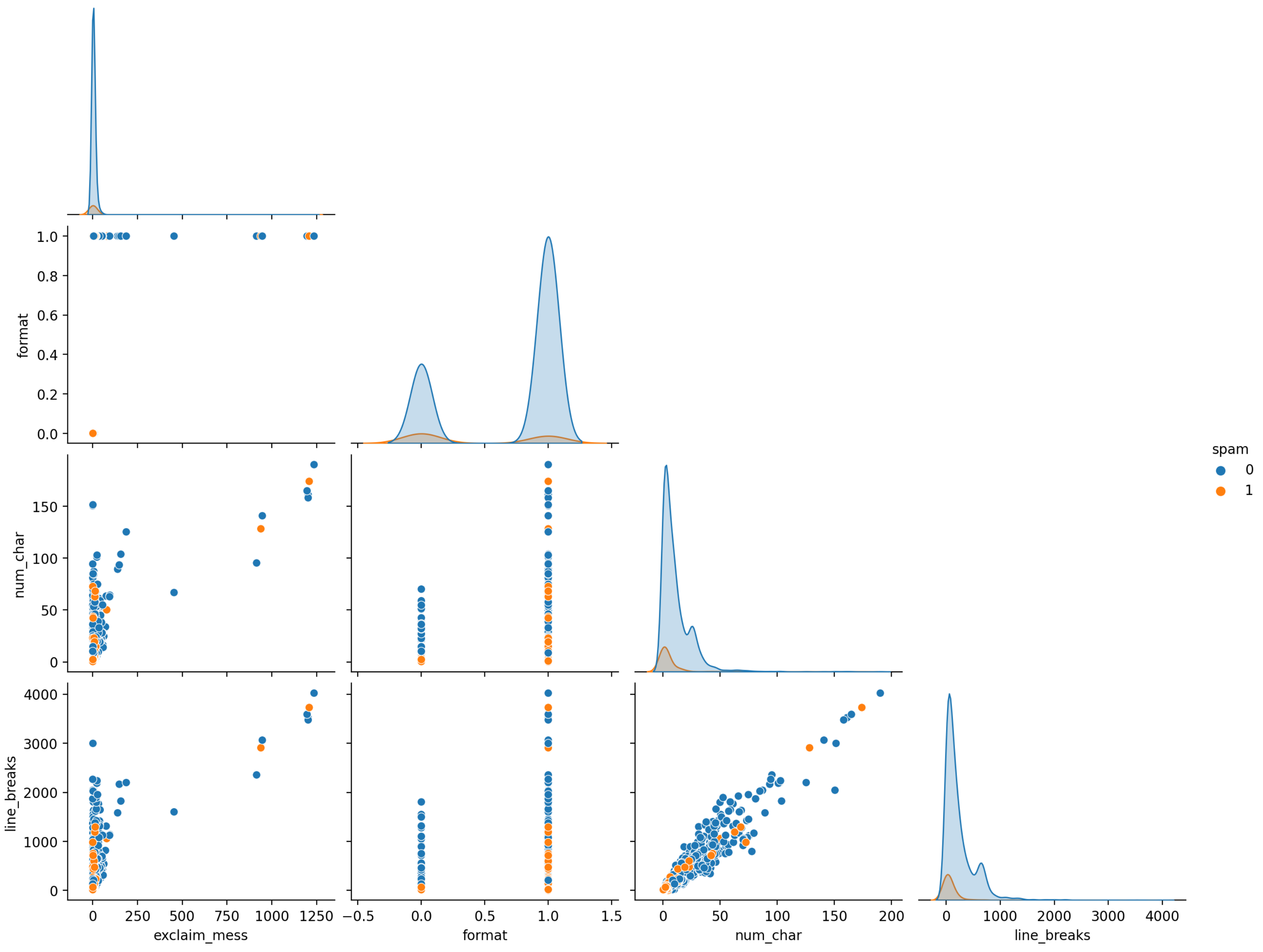
```
      spam  exclaim_mess  format  num_char  line_bre
0        0             0       1    11.370
1        0             1       1    10.504
2        0             6       1     7.773
3        0            48       1    13.256
4        0             1       0     1.231
...    ...           ...     ...       ...
3916     1             0       0     0.332
3917     1             0       0     0.323
3918     0             5       1     8.656
3919     0             0       0    10.185
3920     1             1       0     2.225

[3921 rows x 6 columns]
```

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

```python
email_dc = pd.get_dummies(email)
email_dc
```

```
      spam  exclaim_mess  format  num_char  line_bre
0        0             0       1    11.370
1        0             1       1    10.504
2        0             6       1     7.773
3        0            48       1    13.256
4        0             1       0     1.231
...    ...           ...     ...       ...
3916     1             0       0     0.332
3917     1             0       0     0.323
3918     0             5       1     8.656
3919     0             0       0    10.185
3920     1             1       0     2.225

[3921 rows x 8 columns]
```

```
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([[ 0.0098, -0.619 ,  0.0544, -0.0056, -1.2121,
        -0.6934, -1.9208]])
```

A quick comparison

R output

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

```
Call:  glm(formula = spam ~ . - 1, family = binomial

Coefficients:
exclaim_mess        format      num_char  
    0.009587     -0.604782      0.054765  
 line_breaks     numberbig    numbernone  
   -0.005480     -1.264827     -0.706843  
 numbersmall  
   -1.950440  

Degrees of Freedom: 3921 Total (i.e. Null);  3914 Re
Null Deviance:      5436 
Residual Deviance: 2144     AIC: 2158
```

sklearn output

```python
m.feature_names_in_
```

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

```python
m.coef_
```

```
array([[ 0.0098, -0.619 ,  0.0544, -0.0056, -1.2121,
        -0.6934, -1.9208]])
```

`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)
```

```
Call:  glm(formula = spam ~ . - 1, family = binomial

Coefficients:
exclaim_mess        format       num_char
    0.009587     -0.604782       0.054765
 line_breaks     numberbig     numbernone
   -0.005480     -1.264827      -0.706843
 numbersmall
   -1.950440

Degrees of Freedom: 3921 Total (i.e. Null);  3914 Re:
Null Deviance:       5436
Residual Deviance: 2144      AIC: 2158
```

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([[ 0.0096, -0.6049,  0.0548, -0.0055, -1.2646,
        -0.7068, -1.9505]])
```

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 the 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])
```

```python
m.predict_proba(X)
```

```
array([[0.9132, 0.0868],
       [0.956 , 0.044 ],
       [0.9579, 0.0421],
       [0.9408, 0.0592],
       [0.6876, 0.3124],
       [0.6845, 0.3155],
       [0.9342, 0.0658],
       [0.9636, 0.0364],
       [0.8958, 0.1042],
       [0.9418, 0.0582],
       [0.9325, 0.0675],
       [0.896 , 0.104 ],
       [0.9124, 0.0876],
       [0.9727, 0.0273],
       [0.9283, 0.0717],
       [0.9835, 0.0165],
       [0.9633, 0.0367],
       [0.9538, 0.0462],
       [0.8889, 0.1111],
       [0.8042, 0.1958],
       [0.899 , 0.101 ],
       [0.9564, 0.0436],
       [0.9908, 0.0092],
```

```python
m.predict_log_proba(X)
```

```
array([[-0.0908, -2.4446],
       [-0.045 , -3.1226],
       [-0.043 , -3.1674],
       [-0.061 , -2.8277],
       [-0.3746, -1.1634],
       [-0.3791, -1.1536],
       [-0.0681, -2.7209],
       [-0.0371, -3.3124],
       [-0.11  , -2.2619],
       [-0.06  , -2.8433],
       [-0.0699, -2.6955],
       [-0.1098, -2.2635],
       [-0.0917, -2.4351],
       [-0.0277, -3.6016],
       [-0.0744, -2.6356],
       [-0.0166, -4.1056],
       [-0.0374, -3.304 ],
       [-0.0473, -3.075 ],
       [-0.1178, -2.1973],
       [-0.2179, -1.6306],
       [-0.1064, -2.293 ],
       [-0.0445, -3.1338],
       [-0.0092, -4.6932],
```

Scoring

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

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

```
0.90640142820709
```

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))
```

```
0.90640142820709
```

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

```
0.7606952445645924
```

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

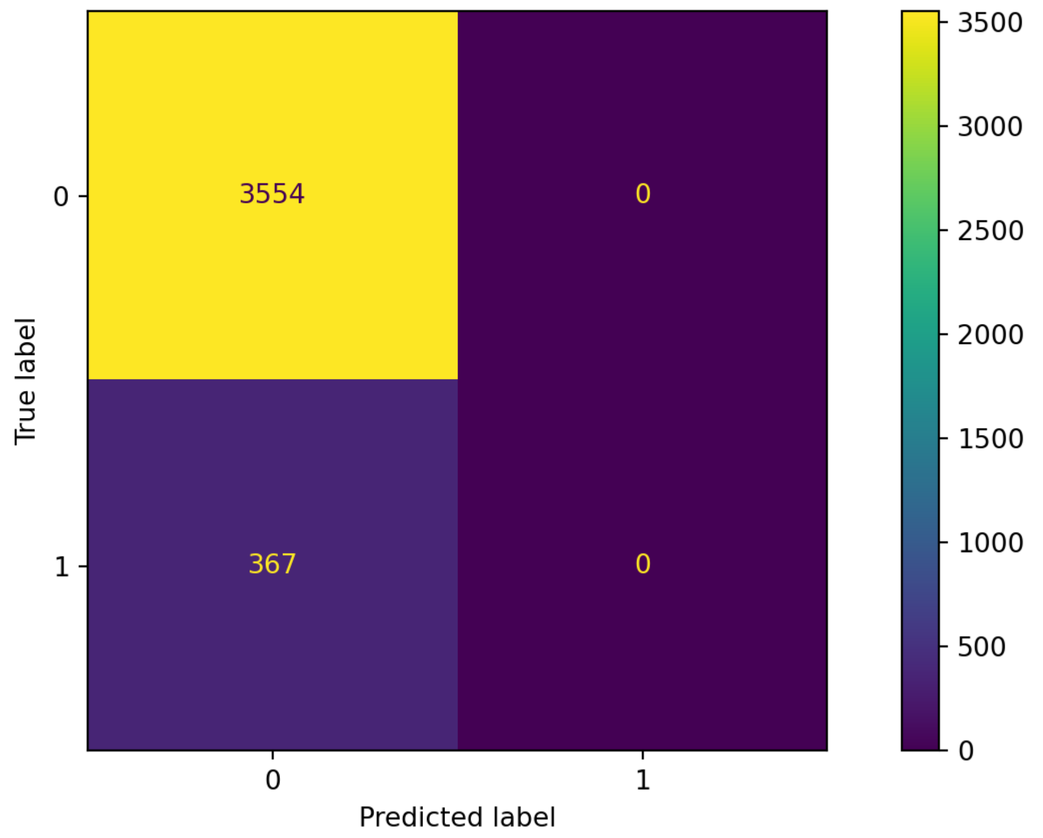
```
0.0
```

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

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

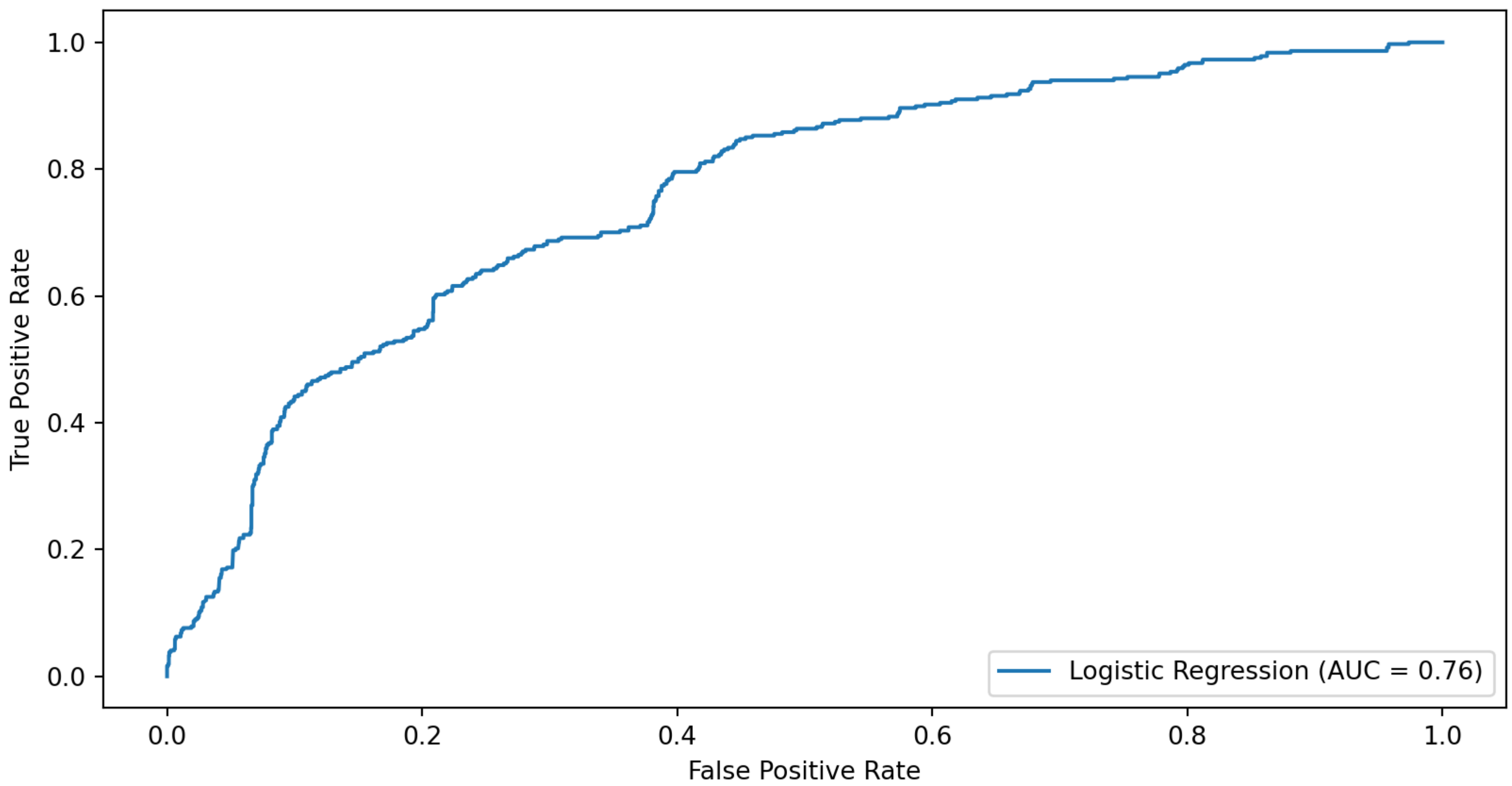
Scoring visualizations - confusion matrix

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

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



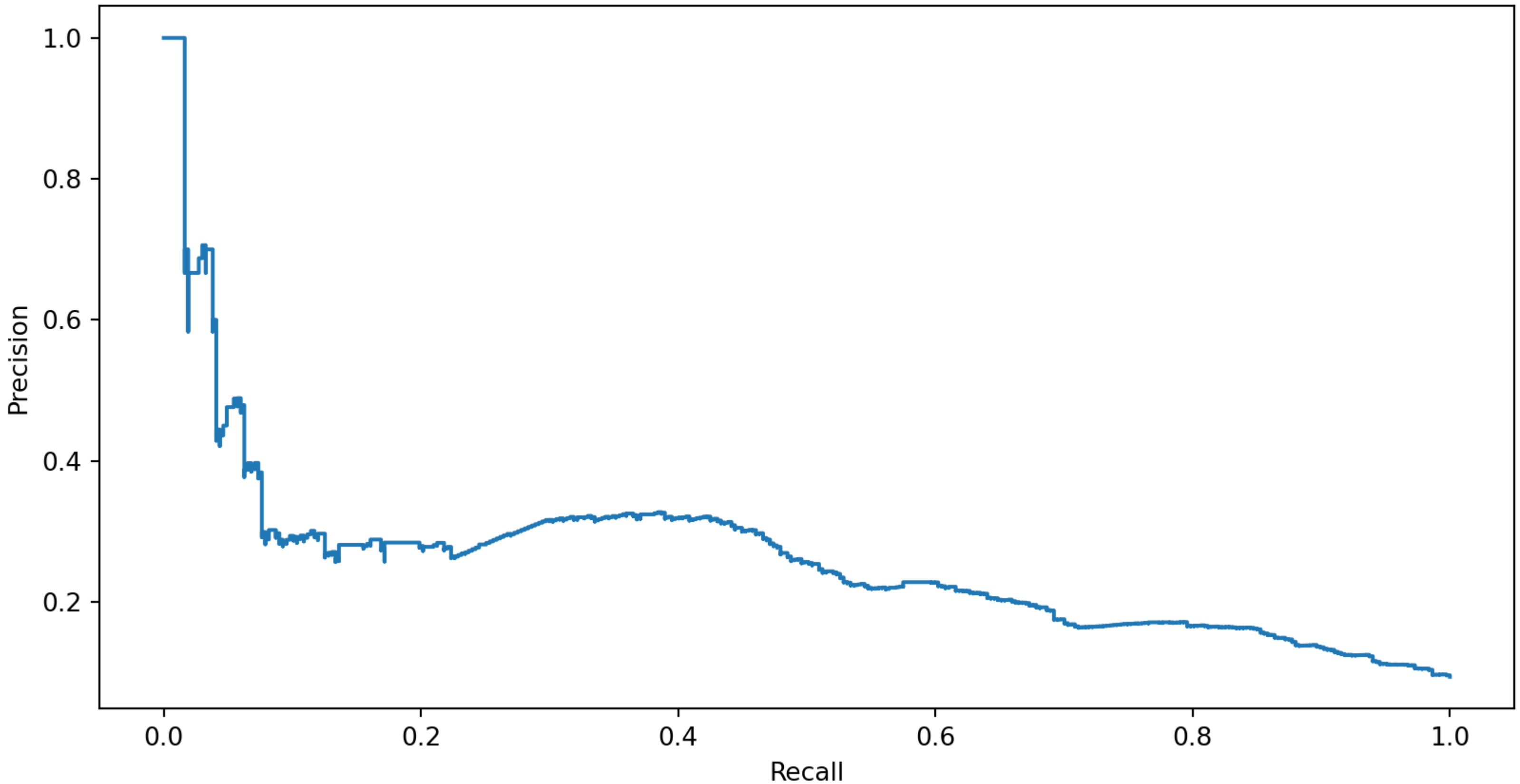
Scoring visualizations - ROC curve

```python
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

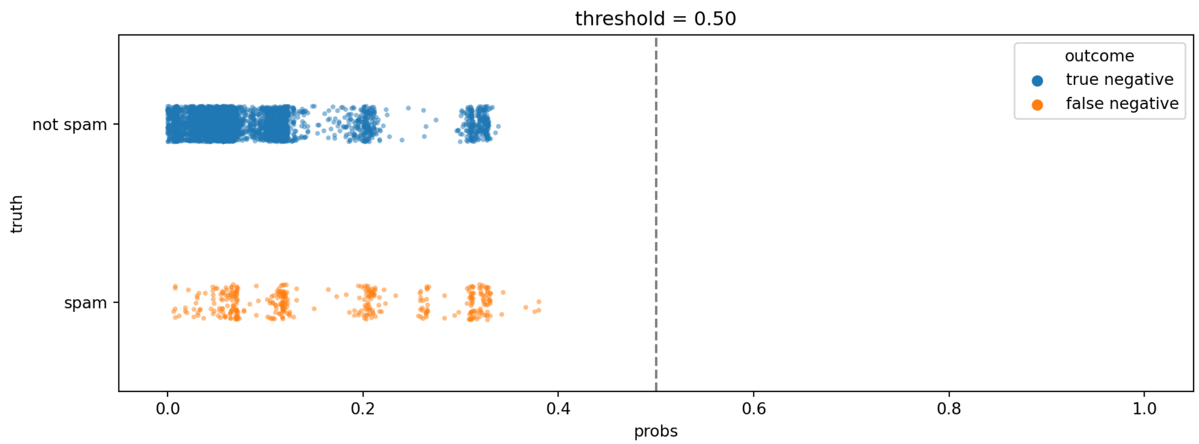
```
from sklearn.metrics import precision_recall_curve, PrecisionRecallDisplay

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



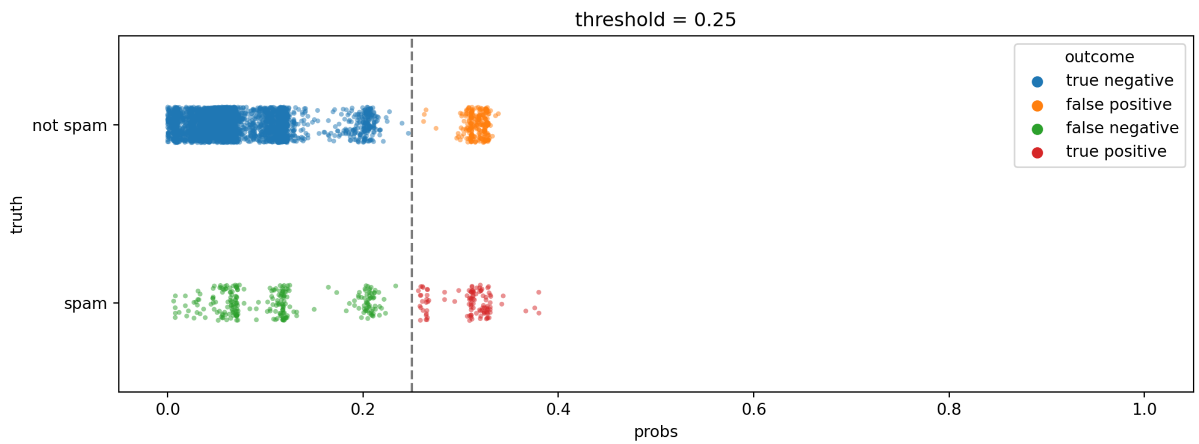
Another visualization

```
def confusion_plot(truth, probs, threshold=0.5):

    d = pd.DataFrame(
        data = {'spam': y, 'truth': truth, 'probs': probs}
    )

    # Create a column called outcome that contains the labeling outcome for the given threshold
    d['outcome'] = 'other'
    d.loc[(d.spam == 1) & (d.probs >= threshold), 'outcome'] = 'true positive'
    d.loc[(d.spam == 0) & (d.probs >= threshold), 'outcome'] = 'false positive'
    d.loc[(d.spam == 1) & (d.probs <  threshold), 'outcome'] = 'false negative'
    d.loc[(d.spam == 0) & (d.probs <  threshold), 'outcome'] = 'true negative'

    # Create plot and color according to outcome
    plt.figure(figsize=(12,4))
    plt.xlim((-0.05,1.05))
    sns.stripplot(y='truth', x='probs', hue='outcome', data=d, size=3, alpha=0.5)
    plt.axvline(x=threshold, linestyle='dashed', color='black', alpha=0.5)
    plt.title("threshold = %.2f" % threshold)
    plt.show()
```

```python
truth = pd.Categorical.from_codes(y, categories = ('not spam','spam'))
probs = m.predict_proba(X)[:,1]
confusion_plot(truth, probs, 0.5)
```




```python
confusion_plot(truth, probs, 0.25)
```



Example 1 - DecisionTreeClassifier

Example 2 - SVC

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_0_3  pi:
0           0.0        0.0        5.0       13.0
1           0.0        0.0        0.0       12.0
2           0.0        0.0        0.0        4.0
3           0.0        0.0        7.0       15.0
4           0.0        0.0        0.0        1.0
...         ...        ...        ...        ...
1792        0.0        0.0        4.0       10.0
1793        0.0        0.0        6.0       16.0
1794        0.0        0.0        1.0       11.0
1795        0.0        0.0        2.0       10.0
1796        0.0        0.0       10.0       14.0

[1797 rows x 64 columns]
```

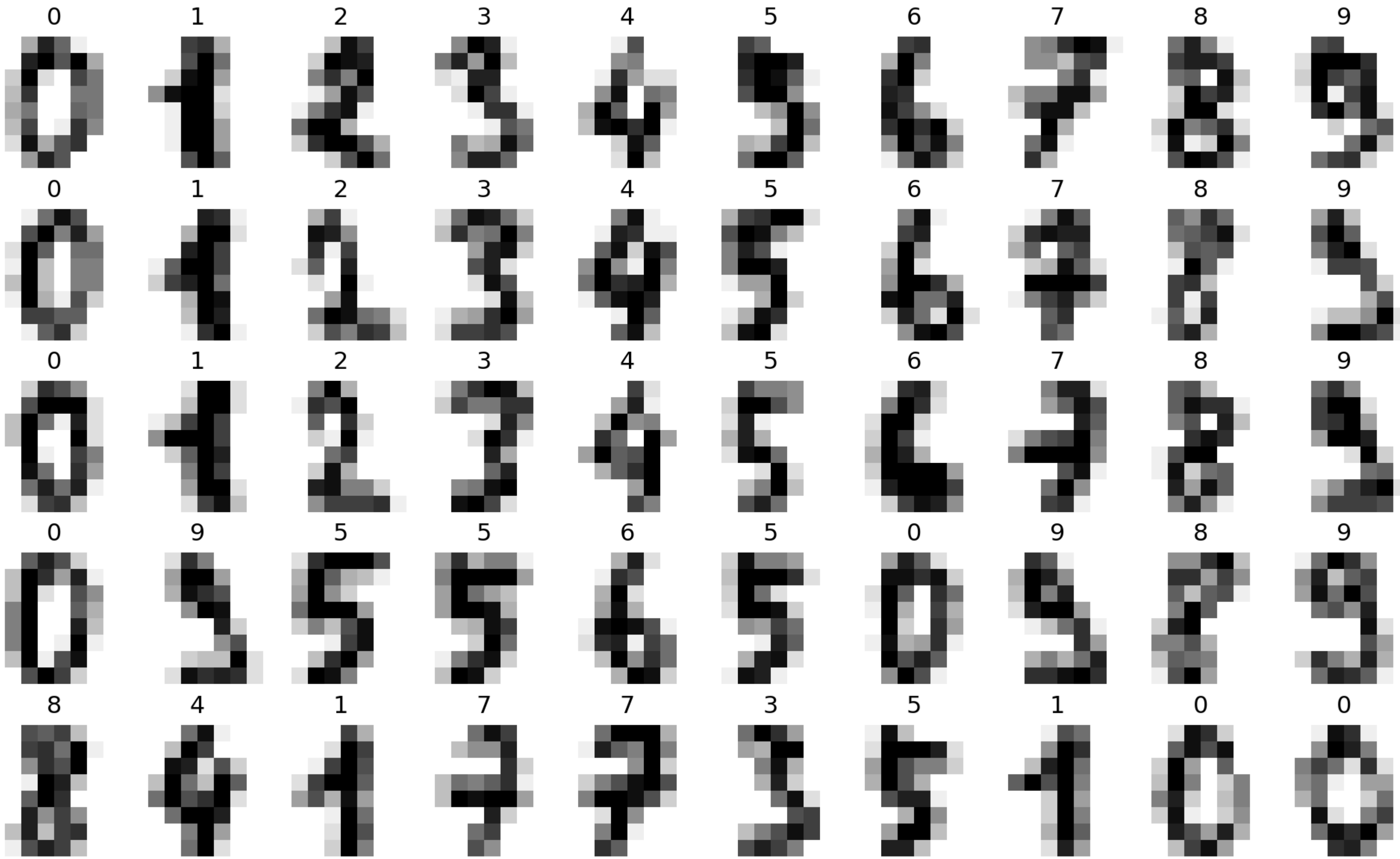
```
y = digits.target
y
```

```
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
```

digit description

```
.. _digits_dataset:

Optical recognition of handwritten digits dataset
--------------------------------------------------

**Data Set Characteristics:**

    :Number of Instances: 1797
    :Number of Attributes: 64
    :Attribute Information: 8x8 image of integer pixels in the range 0..16.
    :Missing Attribute Values: None
    :Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)
    :Date: July; 1998

This is a copy of the test set of the UCI ML hand-written digits datasets
https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

The data set contains images of hand-written digits: 10 classes where
each class refers to a digit.

Preprocessing programs made available by NIST were used to extract
normalized bitmaps of handwritten digits from a preprinted form. From a
total of 43 people, 30 contributed to the training set and different 13
```

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_
```

```
0.943477961432507
```

```
for p, s in  zip(mc_log_cv.cv_results_["params"], mc_log_cv.cv_results_["mean_test_score"]):
  print(p,"Score:",s)
```

```
{'multi_class': 'multinomial'} Score: 0.943477961432507
{'multi_class': 'ovr'} Score: 0.8927617079889807
```

Model coefficients

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

```
     0         1         2         3         4  ...        59        60        61        62        63
0  0.0 -0.133584 -0.823611  0.904385  0.163397  ...  1.211092 -0.444343 -1.660396 -0.750159 -0.184264
1  0.0 -0.184931 -1.259550  1.453983 -5.091361  ... -0.792356  0.384498  2.617778  1.265903  2.338324
2  0.0  0.118104  0.569190  0.798171  0.943558  ...  0.281622  0.829968  2.602947  2.481998  0.788003
3  0.0  0.239612 -0.381815  0.393986  3.886781  ...  1.231868  0.439466  1.070662  0.583209 -1.027194
4  0.0 -0.109904 -1.160712 -2.175923 -2.580281  ... -0.937843 -1.710608 -0.651175 -0.656791 -0.097263
5  0.0  0.701265  4.241974 -0.738130  0.057049  ...  2.045636 -0.001139 -1.412535 -2.097753 -0.210256
6  0.0 -0.103487 -1.454058 -1.310946 -0.400937  ... -1.407609  0.249136  2.466801  1.005207 -0.624921
7  0.0  0.088562  1.386086  1.198007  0.467463  ... -2.710461 -3.176521 -2.635078 -0.710317 -0.099948
8  0.0 -0.347408 -0.306168 -1.933009  1.074249  ...  0.872821  1.722070 -2.302814 -1.602654 -0.679128
9  0.0 -0.268228 -0.811336  1.409475  1.480082  ...  0.205230  1.707472 -0.096190  0.481356 -0.203353

[10 rows x 64 columns]
```

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

```
(10, 64)
```

```
mc_log_cv.best_estimator_.intercept_
```

```
array([ 0.0161, -0.1147, -0.0053,  0.0856,  0.1044,
       -0.0181, -0.0095,  0.0504, -0.0136, -0.0953])
```

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, 118,   0,   0,   0,   0,   0,   0,   0,
       [  0,   0, 119,   0,   0,   0,   0,   0,   0,
       [  0,   0,   0, 123,   0,   0,   0,   0,   0,
       [  0,   0,   0,   0, 110,   0,   0,   0,   0,
       [  0,   0,   0,   0,   0, 114,   0,   0,   0,
       [  0,   0,   0,   0,   0,   0, 124,   0,   0,
       [  0,   0,   0,   0,   0,   0,   0, 124,   0,
       [  0,   0,   0,   0,   0,   0,   0,   0, 119,
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,
```

Out of sample

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

```
0.9579124579124579
```

```
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, 64,  0,  0,  0,  0,  0,  0,  0,  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,  2,  0,  0,  0],
       [ 1,  1,  0,  0,  0,  0, 55,  0,  0,  0],
       [ 0,  0,  0,  0,  2,  0,  0, 53,  0,  0],
       [ 0,  5,  2,  0,  0,  0,  0,  0, 46,  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       0.89      1.00      0.94        64
           2       0.95      0.97      0.96        58
           3       0.98      0.97      0.97        60
           4       0.96      0.97      0.97        71
           5       0.97      0.94      0.96        68
           6       0.96      0.96      0.96        57
           7       1.00      0.96      0.98        55
           8       0.96      0.84      0.89        55
           9       0.96      0.96      0.96        53

    accuracy                           0.96       594
   macro avg       0.96      0.96      0.96       594
weighted avg       0.96      0.96      0.96       594
```

ROC & AUC?

These metrics are slightly awkward to use in the case of multiclass problems since they depend on the probability predictions to calculate.

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

```
Error: ValueError: multi_class must be in ('ovo', 'ovr')
```

```
roc_auc_score(
  y_test, mc_log_cv.best_estimator_.predict_prob
  multi_class = "ovr"
)
```

```
0.9979624274858663
```

```
roc_auc_score(
  y_test, mc_log_cv.best_estimator_.predict_prob
  multi_class = "ovr", average = "weighted"
)
```

```
0.9979869175119241
```

```
roc_auc_score(
  y_test, mc_log_cv.best_estimator_.predict_prob
  multi_class = "ovo"
)
```

```
0.9979645359400721
```

```
roc_auc_score(
  y_test, mc_log_cv.best_estimator_.predict_prob
  multi_class = "ovo", average = "weighted"
)
```

```
0.9979743498851119
```

Prediction

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

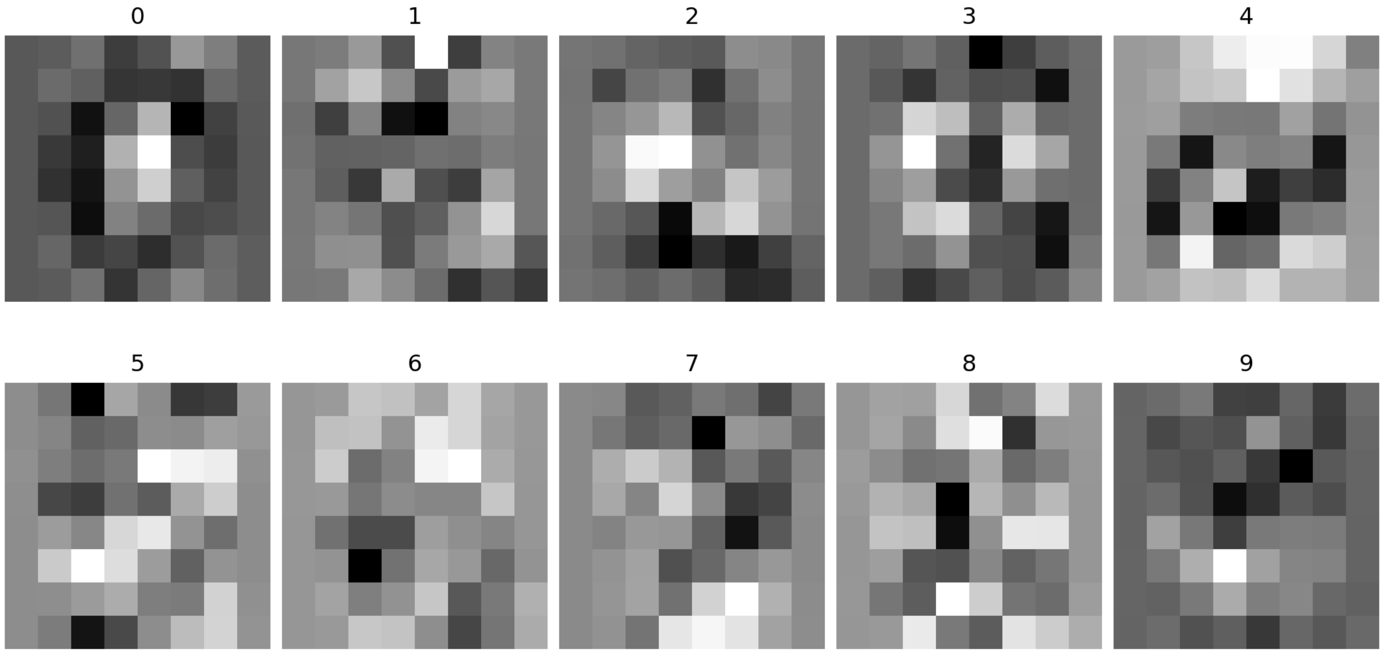
```
array([7, 1, 7, 6, 0, 2, 4, 3, 6, 3, 7, 8, 7, 9, 4,
       7, 8, 4, 0, 3, 9, 1, 3, 6, 6, 0, 5, 4, 1, 2,
       3, 2, 7, 6, 4, 8, 6, 4, 4, 0, 9, 1, 9, 5, 4,
       1, 7, 6, 9, 2, 9, 9, 9, 0, 8, 3, 1, 8, 8, 1,
       1, 3, 9, 6, 9, 5, 2, 1, 9, 2, 1, 3, 8, 7, 3,
       7, 7, 5, 8, 2, 6, 1, 9, 1, 6,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

```python
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.        , 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()
```


0
1
2
3
4
5
6
7
8
9

Example 3 - 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.

```
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 4 - GridSearchCV w/ Multiple models (Trees vs Forests)