Week 5

Sklearn API for Classification

There are broadly two types of APIs based on their functionality:

Generic	Specific
• SGD classifier	• Logistic regression • Perceptron • Ridge classifier (for LSC) • K-nearest neighbours (KNNs) • Support vector machines (SVMs) • Naive Bayes
Uses gradient descent for opt	Specialized solvers for opt
Need to specify loss function	

Common Methods

Model training

`fit(X, y[, coef_init, intercept_init, …])`

Prediction

`predict(X)` predicts class label for samples

`decision_function(X)` predicts confidence score for samples.

Evaluation

`score(X, y[, sample_weight])`

Return the mean accuracy on the given test data and labels.

Miscellaneous Methods

`get_params([deep])` gets parameter for this estimator.

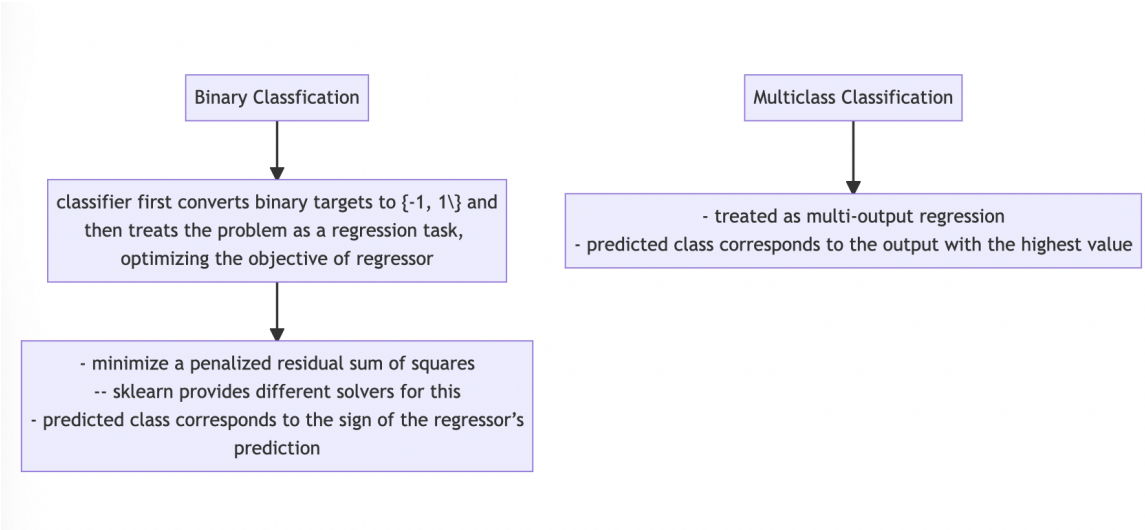
`set_params(**params)` sets the parameters of this estimator.

`densify()` converts coefficient matrix to dense array format.

`sparsify()` converts coefficient matrix to sparse format.

Ridge Classifier

RidgeClassifier is a classifier variant of the Ridge regressor.



Training a least square classifier

Step 1: Instantiate a classification estimator without passing any arguments to it. This creates a ridge classifier object.

```
from sklearn.linear_model import RidgeClassifier
ridge_classifier = RidgeClassifier()
```

Step 2: Call fit method on ridge classifier object with training feature matrix and label vector as arguments.
Note: The model is fitted using X_train and y_train.

```
# Model training with feature matrix X_train and
# label vector or matrix y_train
ridge_classifier.fit(X_train, y_train)
```

Setting regularisation rate

💡 - Set alpha to float value.
- Default is 0.1

```
from sklearn.linear_model import RidgeClassifier
ridge_classifier = RidgeClassifier(alpha=0.001)
```

- alpha should be positive.
- Larger alpha values specify stronger regularization.

Solving optimization problem

Using one of the following solvers

svd	uses a Singular Value Decomposition of the feature matrix to compute the Ridge coefficients.
cholesky	uses **scipy.linalg.solve** function to obtain the closed-form solution
sparse_cg	uses the conjugate gradient solver of **scipy.sparse.linalg.cg** .
lsqr	uses the dedicated regularized least-squares routine **scipy.sparse.linalg.lsqr** and it is fastest.
sag , **saga**	uses a Stochastic Average Gradient descent iterative procedure 'saga' is unbiased and more flexible version of 'sag'
lbfgs	uses L-BFGS-B algorithm implemented in **scipy.optimize.minimize** . can be used only when coefficients are forced to be positive.

Uses of solver

- **sparse_cg:** for large scale data

- **‘sag’ or ‘saga’:** When both n_samples and n_features are large

Automatic selection of solver

```
ridge_classifier = RidgeClassifier(solver=auto)
```

auto chooses the solver automatically based on the type of data.

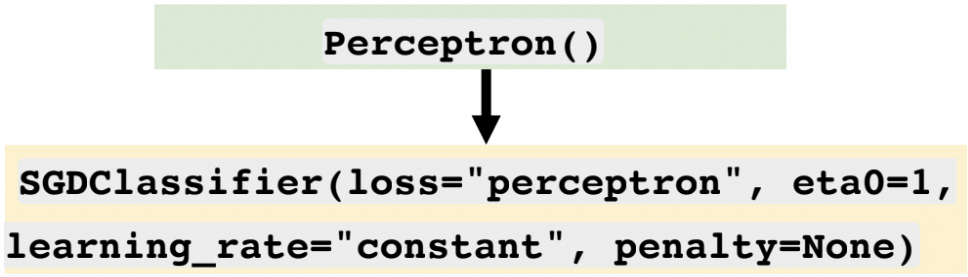
Default choice for solver is auto.

Intercept Estimation

If data is already centered, set fit_intercept as false, so that no intercept will be used in calculations.

Perceptron Classification

- It is a simple classification algorithm suitable for large-scale learning.
- Shares the same underlying implementation with SGDClassifier.
- Perceptron uses SGD for training.



Implementation

Step 1: Instantiate a Perceptron estimator without passing any arguments to it to create a classifier object.

```
from sklearn.linear_model import Perceptron
perceptron_classifier = Perceptron()
```

Step 2: Call fit method on perceptron estimator object with training feature matrix and label vector as arguments.

```
# Model training with feature matrix X_train and
# label vector or matrix y_train
perceptron_classifier.fit(X_train, y_train)
```

- Perceptron classifier can be trained in an iterative manner with **partial_fit** method
- Perceptron classifier can be initialized to the weights ofthe previous run by specifying **warm_start** = **True** in the constructor.

Parameters

`penalty` (default = 'l2')	`l1_ratio` (default = 0.15)
`alpha` (default = 0.0001)	`early_stopping` (default = False)
`fit_intercept` (default = True)	`max_iter` (default = 1000)
`n_iter_no_change` (default = 5)	`tol` (default = 1e-3)
`eta0` (default = 1)	`validation_fraction` (default = 0.1)

Logistic Regression

- also known as logit regression, maximum entropy classifier (maxent) and log-linear classifier.
- This implementation can fit
  - binary classification
  - one-vs-rest (OVR)
  - multinomial logistic regression
- Provision for l1, l2 or elastic-net regularization

Implementation

Step 1: Instantiate a classifier estimator without passing any arguments to it to create a classifier object.

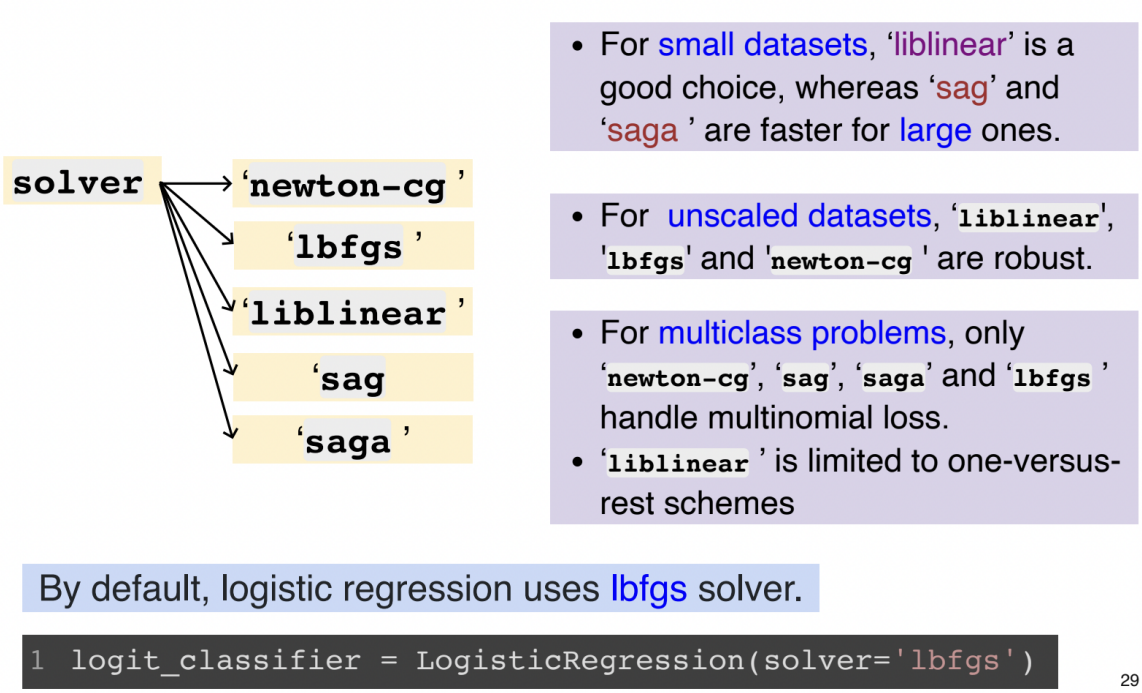
```
from sklearn.linear_model import LogisticRegression
logit_classifier = LogisticRegression()
```

Step 2: Call fit method on logistic regression classifier object with training feature matrix and label vector as arguments.

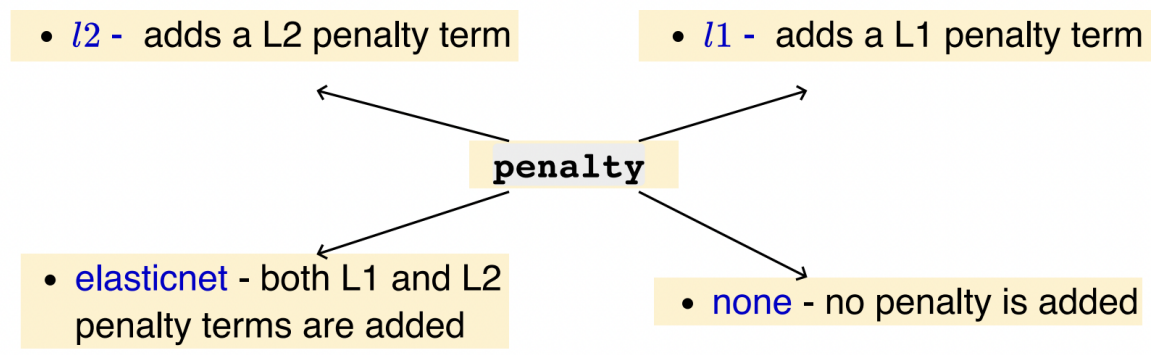
```
# Model training with feature matrix X_train and
# label vector or matrix y_train
logit_classifier.fit(X_train, y_train)
```

Selection of Solvers

- Logistic regression uses specific algorithms for solving the optimization problem in training. These algorithms are known as **solvers**.
- The choice of the solver depends on the classification problem set up such as size of the dataset, number of features and labels.



Regularisation



- Regularization is applied by default because it improves numerical stability.
- By default, it uses L2 penalty.
- Not all the solvers supports all the penalties.
- Select appropriate solver for the desired penalty:

Solver	Penalty
'`newton-cg`'	['l2', 'none']
'`lbfgs`'	['l2', 'none']
'`liblinear`'	['l1', 'l2']
'`sag`'	['l2', 'none']
'`saga`'	['elasticnet', 'l1', 'l2', 'none']

- sklearn implementation uses parameter C, which is inverse of regularization rate to control regularization.

$$\arg\min_{w,C} \text{ regularization penalty} + C \text{ cross entropy loss}$$

- C is specified in the constructor and must be positive
  - Smaller value leads to stronger regularization.
  - Larger value leads to weaker regularization.

- LogisticRegression classifier has a class_weight parameter in its constructor.
  - Handles class imbalanace with differential class weights.

- Mistakes in a class are penalized by the class weight. Higher value here would mean higher emphasis on the class.

SGD Classifier

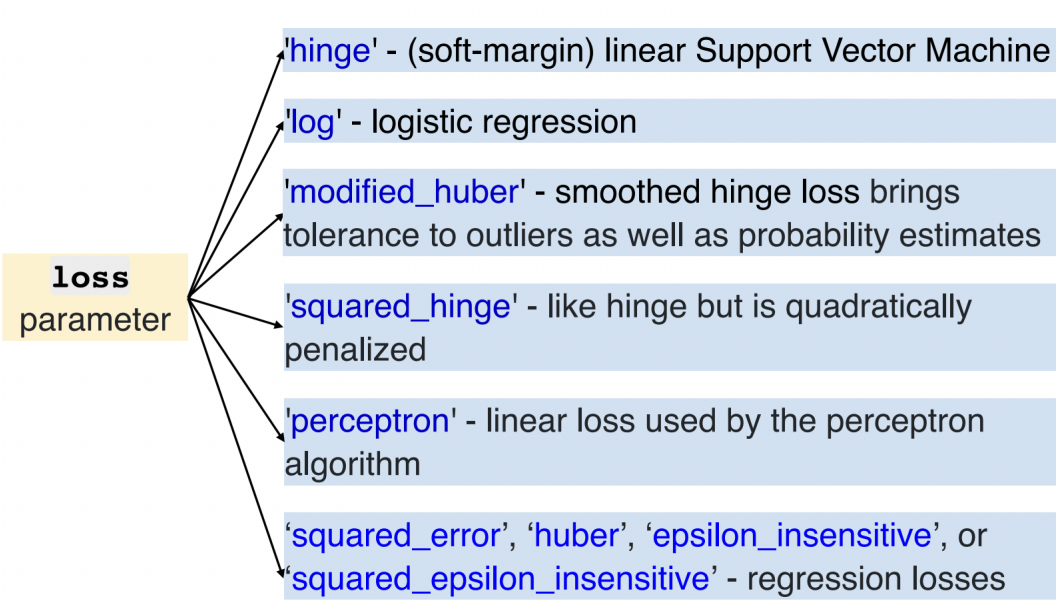
- SGD is a simple yet very efficient approach to fitting linear classifiers under **convex loss functions.**
- It **supports multi-class classification** by combining multiple binary classifiers in a "**one versus all" (OVA) scheme.**
- **Easily scales up to large scale problems** with more than 10^5 training examples and features. It also works with sparse machine learning problems, text classification and natural language processing.

```
SGDClassifier(loss='log')
```

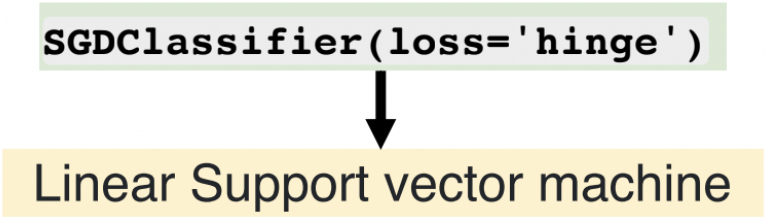
↓

```
LogisticRegression(solver='sgd')
```

Loss parameter



💡 By default SGDClassifier uses hinge loss and hence, trains linear support vector machine classifier.



Working

- SGDClassifier implements a **plain stochastic gradient descent learning routine**.
- the gradient of the loss is estimated with one sample at a time and the model is updated along the way with a **decreasing learning rate** (or strength) schedule.
- **Advantages:**
  - Efficiency
  - Ease of implementation
- **Disadvantages:**
  - Requires a number of hyperparameters.
  - Sensitive to feature scaling.
- It is important to
  - permute (shuffle) the training data before fitting the model
  - standardize the features.

Implementation

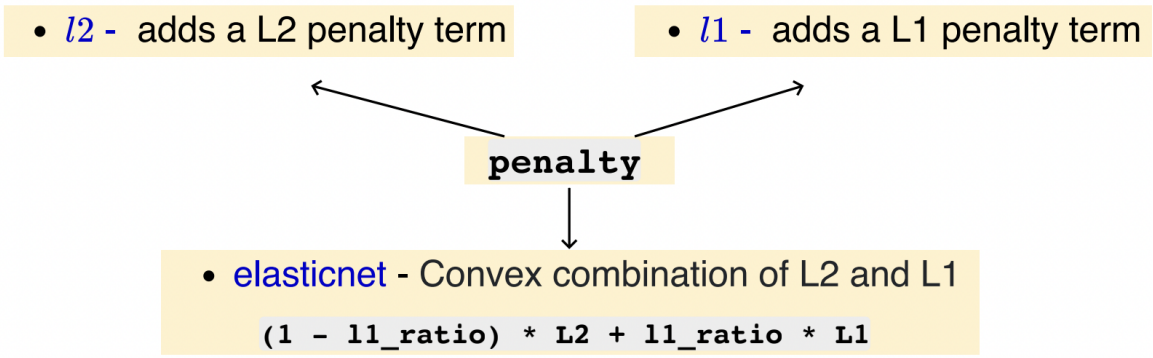
Step 1: Instantiate a SGDClassifer estimator by setting appropriate loss parameter to define classifier of interest. By default it uses hinge loss, which is used for training linear support vector machine. Here we have used `log` loss that defines a logistic regression classifier.

```
from sklearn.linear_model import SGDClassifier
SGD_classifier = SGDClassifier(loss='log')
```

Step 2: Call fit method on SGD classifier object with training feature matrix and label vector as arguments.

```
# Model training with feature matrix X_train and
# label vector or matrix y_train
SGD_classifier.fit(X_train, y_train)
```

Regularisation

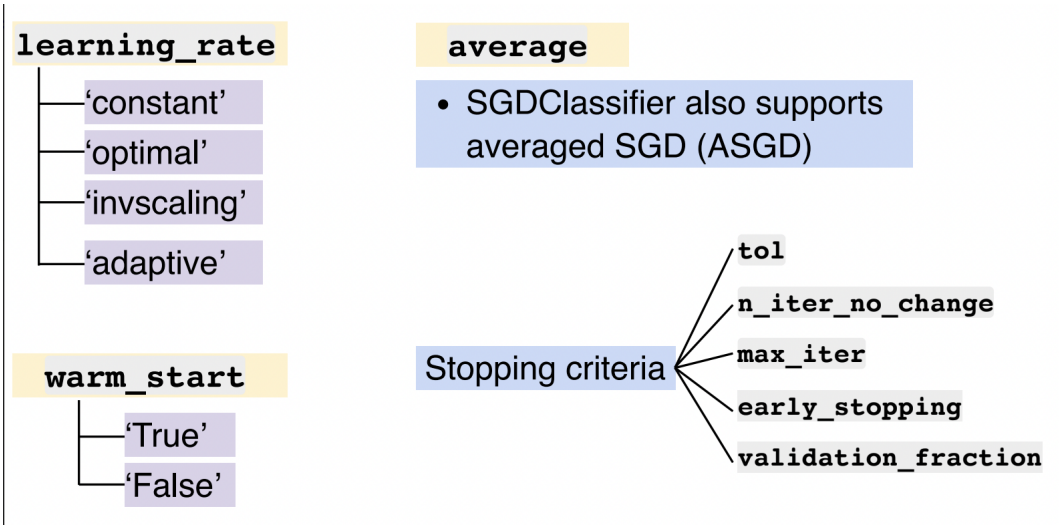



```
#default
SGD_classifier = SGDClassifier(penalty='l2')
```

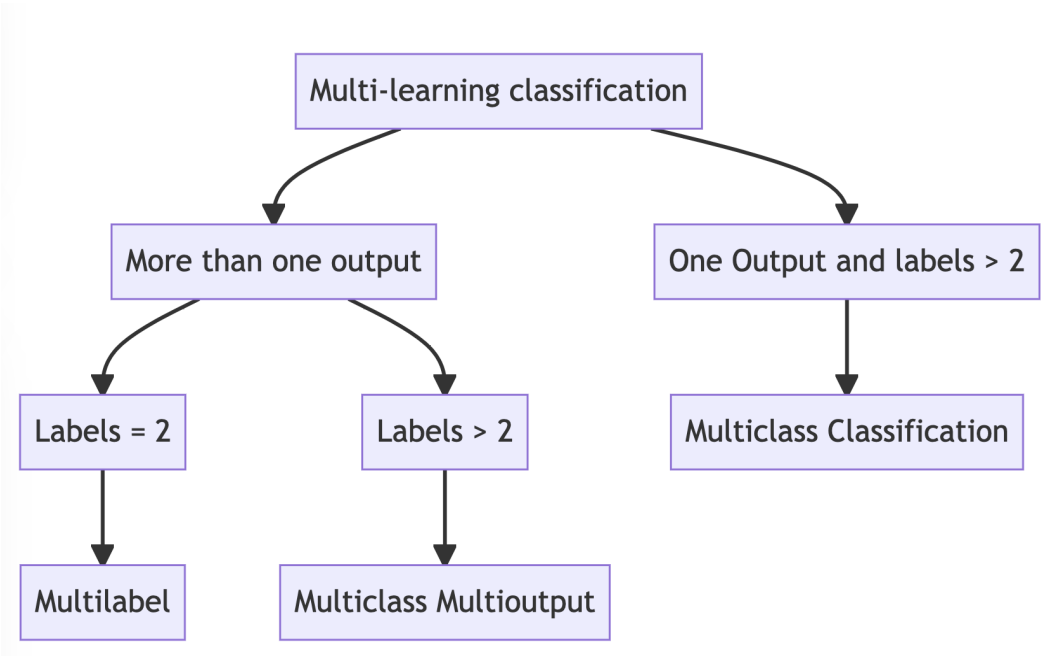
alpha

- Constant that multiplies the regularization term.
- Has float values and default = 0.0001

Common parameters between SGDClassifier and SGDRegressor



Multi-learning Classification



- Use type_of_target to determine the type of the label.

```
from sklearn.utils.multiclass import type_of_target
type_of_target(y)
```

target_type	y
'multiclass'	• contains more than two discrete values • not a sequence of sequences • 1d or a column vector
'multiclass-multioutput'	• 2d array that contains more than two discrete values • not a sequence of sequences • dimensions are of size > 1
'multilabel-indicator'	• label indicator matrix • an array of two dimensions with at least two columns, and at most 2 unique values.
'unknown'	• array-like but none of the above, such as a 3d array, • sequence of sequences, or an array of non-sequence objects.

multiclass

```
>>> type_of_target([1, 0, 2])
'multiclass'
>>> type_of_target([1.0, 0.0, 3.0])
'multiclass'
>>> type_of_target(['a', 'b', 'c'])
'multiclass'
```

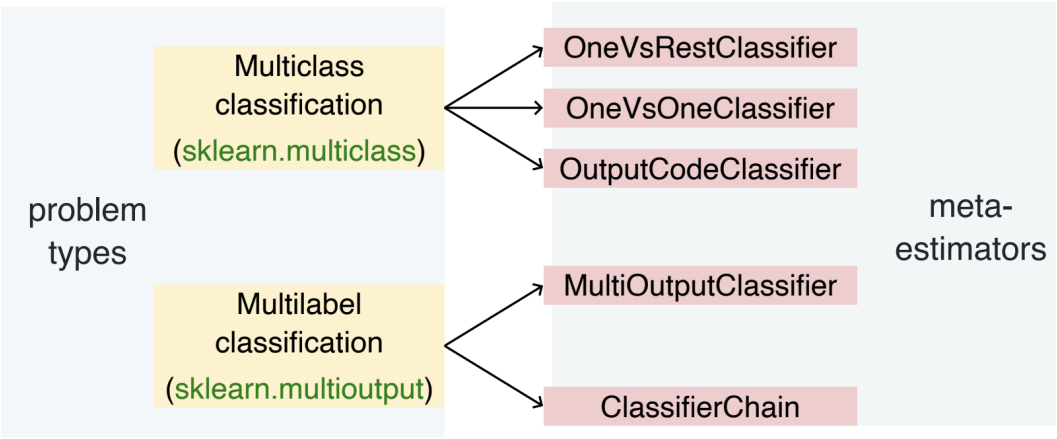
multiclass-multioutput

```
>>> type_of_target(np.array([[1, 2], [3, 1]]))
'multiclass-multioutput'
```

multilabel-indicator

```
type_of_target(np.array([[0, 1], [1, 1]]))
'multilabel-indicator'
>>> type_of_target([[1, 2]])
'multilabel-indicator'
```

- sklearn provides a bunch of meta-estimators, which extend the functionality of base estimators to support multi-learning problems.
- The meta-estimators transform the multi-learning problem into a set of simpler problems and fit one estimator per problem.



- Many sklearn estimators have built-in support for multi-learning problems.
- Meta-estimators are not needed for such estimators, however meta-estimators can be used in case we want to use these base estimators with strategies beyond the built-in ones.
    - **Inherently Multiclass**
        - LogisticRegression (multi_class = 'multinomial')
        - LogisticRegressionCV (multi_class = 'multinomial')
        - RidgeClassifier
        - RidgeClassifierCV
    - **Multiclass vs OVO**
    - **Multiclass vs OVR**
        - LogisticRegression (multi_class = 'ovr')
        - LogisticRegressionCV (multi_class = 'ovr')
        - SGDClassifier
        - Perceptron
    - **Multilabel**
        - RidgeClassifier
        - RidgeClassifierCV

Multiclass classification

💡 - Classification task with **more than two classes**.
- Each example is labeled with exactly one class
- Example - iris flower dataset, MNIST

Representing class labels in multi-class setup

- Each example is marked with a single label out of k labels. The shape of label vector is (n, 1).
- Use LabelBinarizer transformation to convert the class label to multi-class format.

```
from sklearn.preprocessing import LabelBinarizer
y = np.array(['apple', 'pear', 'apple', 'orange'])
y_dense = LabelBinarizer().fit_transform(y)
```

- The resulting label vector has shape of (n, k).

```
[[1 0 0]
 [0 0 1]
 [1 0 0]
 [0 1 0]]
```

Multi-class classification strategies

- **One-vs-all or one-vs-rest (OVR)**
  - OVR is implemented by OneVsRestClassifier API.
  - Fits one classifier per class c - c or not c.
  - This approach is computationally efficient and requires only classifiers.

```
from sklearn.multiclass import OneVsRestClassifier
OneVsRestClassifier(LinearSVC(random_state=0)).fit(X, y)
```

  - We need to supply estimator as an argument in the constructor.
  - OneVsRest classifier also supports multilabel classification. We need to supply labels as indicator matrix of shape (n, k).
- **One-vs-One (OVA)**

- OVR is implemented by OneVsRestClassifier API.
- Fits one classifier per pair of classes. Total classifiers = $\binom{k}{2}$.
- Predicts class that receives maximum votes. The tie among classes is broken by selecting the class with the highest aggregate classification confidence.

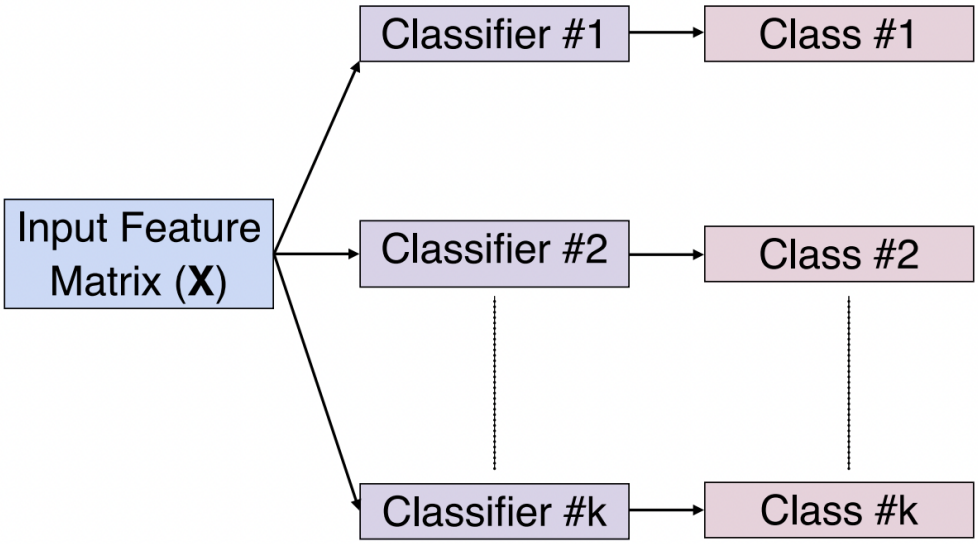
```
from sklearn.multiclass import OneVsOneClassifier
OneVsOneClassifier(LinearSVC(random_state=0)).fit(X, y)
```

- OneVsOne classifier processes subset of data at a time and is useful in cases where the classifier does not scale with the data.

OneVsRestClassifier	OneVsOneClassifier
• Fits one classifier per class. • For each classifier, the class is fitted against all the other classes.	• Fits one classifier per pair of classes. • At prediction time, the class which received the most votes is selected.

MultiOutputClassifier

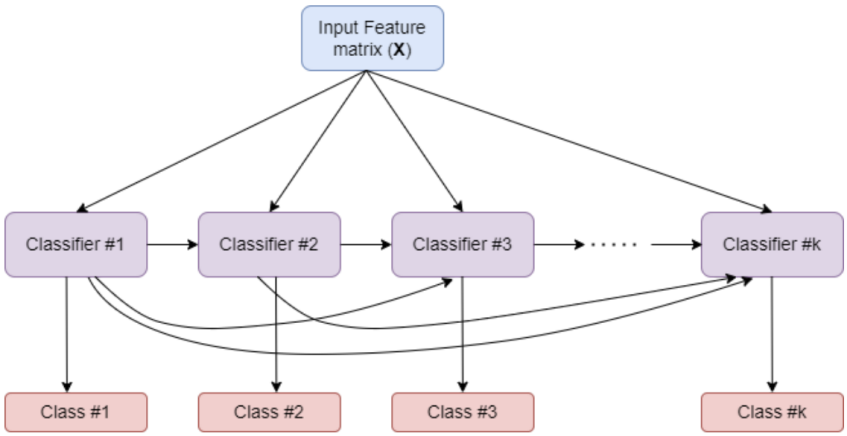
Strategy consists of fitting one classifier per target



ClassifierChain

is

- A multi-label model that arranges binary classifiers into a chain
- a way of combining a number of binary classifiers into a single multi-label model.



MultiOutputClassifier vs Classifier Chain

MultiOutputClassifier	ClassifierChain
• Able to estimate a series of target functions that are trained on a single predictor matrix to predict a series of responses.	• Capable of exploiting correlations among targets.
• Allows multiple target variable classifications.	• For a multi-label classification problem with k classes, k binary classifiers are assigned an integer between 0 and $k-1$. • These integers define the order of models in the chain.

Evaluating Classifiers

Stratified cross validation iterators

sklearn.model_selection module provides three stratified APIs **to create folds such that the overall class distribution is replicated in individual folds**.

- StratifiedKFold
- RepeatedStratifiedKFold
- StratifiedShuffleSplit (Folds obtained may not be completely different)

LogisticRegressionCV

- Support in-build cross validation for optimizing hyperparameters
- The following are key parameters for HPT and cross validation

cv specifies cross validation iterator	scoring specifies scoring function to use for HPT	cs specifies regularization strengths to experiment with.

- Choosing the best hyper-parameters
  - **refit = True:** Scores averaged across folds, values corresponding to the best score are selected and final refit with these parameters

- **refit = False:** the coefs, intercepts and C that correspond to the best scores across folds are averaged.

Classification metrics

- accuracy_score
- balanced_accuracy_score
- top_k_accuracy_score
- roc_auc_score
- precision_score
- recall_score
- f1_score

Confusion Matrix

💡 confusion_matrix evaluates classification accuracy by computing the confusion matrix with each row corresponding to the true class.

```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_true, y_predicted)
```

Example:

```
array([[2, 0, 0],
       [0, 0, 1],
       [1, 0, 2]])
```

Entry i, j in a confusion matrix

number of observations actually in group i, but predicted to be in group j.

Confusion matrix can be displayed with ConfusionMatrixDisplay API in sklearn.metrics.

- **Confusion matrix**

```
ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=clf.classes_)
```

- **From estimators**

```
ConfusionMatrixDisplay.from_estimator(clf, X_test, y_test)
```

- **From predictions**

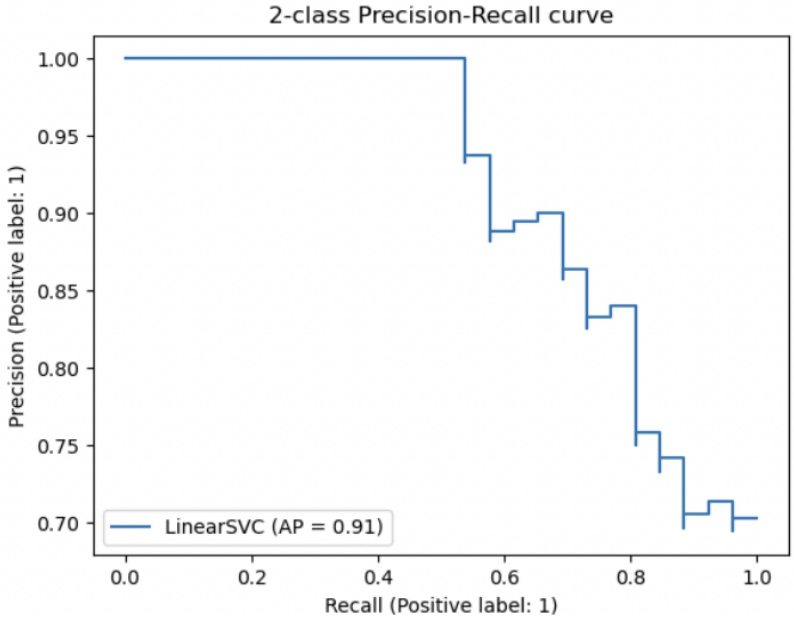
```
ConfusionMatrixDisplay.from_predictions(y_test, y_pred)
```

The classification_report function builds a text report showing the main classification metrics.

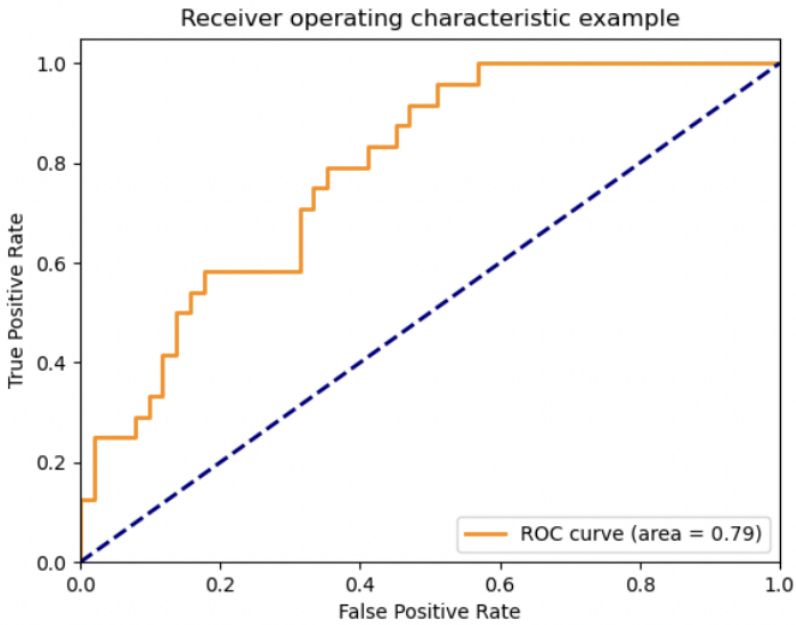
```
from sklearn.metrics import classification_report
print(classification_report(y_true, y_predicted))
```

Classifier Performance across probability thresholds

```
from sklearn.metrics import precision_recall_curve
precision, recall, thresholds = precision_recall_curve(y_true, y_predicted)
```




```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = metrics.roc_curve(y_true, y_scores, pos_label=2)
```



How to extend binary metric to multiclass or multilabel problems?

- Treat data as a collection of binary problems, one for each class.
- Then, average binary metric calculations across the set of classes.

- Can be done using average parameter.

`macro`	calculates the mean of the binary metrics
`weighted`	computes the average of binary metrics in which each class's score is weighted by its presence in the true data sample.
`micro`	gives each sample-class pair an equal contribution to the overall metric
`samples`	calculates the metric over the true and predicted classes for each sample in the evaluation data, and returns their average
`None`	returns an array with the score for each class