# Classification functions in sci-kit learn

Dr. Ashish Tendulkar

IIT Madras

Machine Learning Practice

- In this week, we will study sklearn functionality for implementing classification algorithms.

- We will cover sklearn APIs for
  - Specific classification algorithms for least square classification, perceptron, and logistic regression classifier.
    - with regularization
    - multiclass, multilabel and multi-output setting
  - Various classification metrics.

- Cross validation and hyper parameter search for classification works exactly like how it works in regression setting.
  - However there are a couple of CV strategies that are specific to classification

# Part I: sklearn API for classification

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

**Generic**

- SGD classifier

Uses gradient descent for opt

Need to specify loss function

**Specific**

- Logistic regression
- Perceptron
- Ridge classifier (for LSC)
- K-nearest neighbours (KNNs)
- Support vector machines (SVMs)
- Naive Bayes

Specialized solvers for opt

All sklearn estimators for classification implement a few common methods for model training, prediction and evaluation.

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.

There a few common miscellaneous methods as follows:

| | |
|---|---|
| `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. |

Now let's study how to implement different classifiers with sklearn APIs.

Let's start with implementation of least square classification (LSC) with RidgeClassifier API.

# Ridge classifier

- RidgeClassifier is a classifier variant of the Ridge regressor.

## Binary classification:

- classifier first converts binary targets to {-1, 1} and then treats the problem as a regression task, optimizing the objective of regressor:
  - minimize a penalized residual sum of squares
    - $\min_{\mathbf{w}} ||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2 + \alpha||\mathbf{w}||_2^2$
      - sklearn provides different solvers for this optimization
      - sklearn uses $\alpha$ to denote regularization rate
    - predicted class corresponds to the sign of the regressor's prediction

## Multiclass classification:

- treated as multi-output regression
- predicted class corresponds to the output with the highest value

# How to train a least square classifier?

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

```
1 from sklearn.linear_model import RidgeClassifier
2 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.

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

# How to set regularization rate in RidgeClassifier?

Set `alpha` to float value. The default value is 0.1.

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

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

# How to solve optimization problem in RidgeClassifier?

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<br>'saga' is unbiased and more flexible version of 'sag' |
| `lbfgs` | uses L-BFGS-B algorithm implemented in `scipy.optimize.minimize` .<br>can be used only when coefficients are forced to be positive. |

# Uses of solver in RidgeClassifier

- For large scale data, use '`sparse_cg` ' solver.

- When both `n_samples` and `n_features` are large, use ‘`sag` ’ or ‘`saga`’ solvers.
  - Note that fast convergence is only guaranteed on features with approximately the same scale.

# How to make RidgeClassifier select the solver automatically?

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

**`auto`**

chooses the solver automatically based on the type of data

```
1   if solver == 'auto':
2       if return_intercept:
3           # only sag supports fitting intercept directly
4           solver = "sag"
5       elif not sparse.issparse(X):
6           solver = "cholesky"
7       else:
8           solver = "sparse_cg"
```

Default choice for solver is **`auto`** .

# Is intercept estimation necessary for RidgeClassifier?

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

Default:

```
1 ridge_classifier = RidgeClassifier(fit_intercept=True)
```

# How to make predictions on new data samples?

Use `predict` method to predict class labels for samples

**Step 1**: Arrange data for prediction in a feature matrix of shape (#samples, #features) or in sparse matrix format.

**Step 2**: Call predict method on classifier object with feature matrix as an argument.

```
1 # Predict labels for feature matrix X_test
2 y_pred = ridge_classifier.predict(X_test)
```

Other classifiers also use the same predict method.

RidgeClassifierCV implements
RidgeClassifier with built-in cross validation.

Let's implement perceptron classifier with Perceptron API.

# Perceptron classification

- It is a simple classification algorithm suitable for large-scale learning.

- Shares the same underlying implementation with `SGDClassifier`

```
Perceptron()
```

```
SGDClassifier(loss="perceptron", eta0=1,
learning_rate="constant", penalty=None)
```

Perceptron uses SGD for training.

# How to implement perceptron classifier?

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

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

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

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

Perceptron can be further customized with the following 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)

- Perceptron classifier can be trained in an iterative manner with partial_fit method

- Perceptron classifier can be initialized to the weights of the previous run by specifying warm_start = True in the constructor.

Let's implement logistic regression classifier
with LogisticRegression API.

# LogisticRegression API

- Implements logistic regression classifier, which is also known by a few different names like logit regression, maximum entropy classifier (maxent) and log-linear classifier.

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

- This implementation can fit
  - binary classification
  - one-vs-rest (OVR)
  - multinomial logistic regression

- Provision for $\ell_1, \ell_2$ or elastic-net regularization

# How to train a LogisticRegression classifier?

**Step 1**: Instantiate a classifier estimator without passing any arguments to it. This creates a logistic regression object.

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

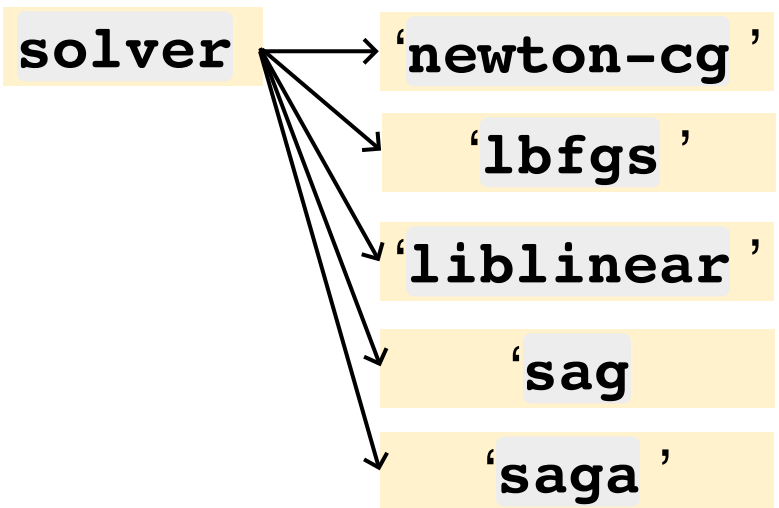
**Step 2**: Call fit method on logistic regression classifier object with training feature matrix and label vector as arguments

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

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.

# How to select solvers for Logistic Regression classifier?



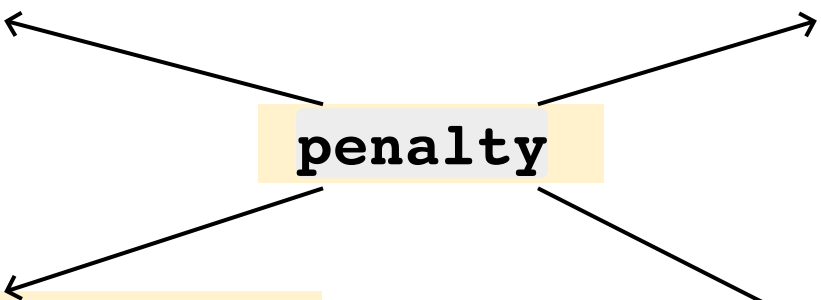

- For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga ' are faster for large ones.
- For unscaled datasets, '`liblinear`', '`lbfgs`' and '`newton-cg` ' are robust.
- For multiclass problems, only '`newton-cg`', '`sag`', '`saga`' and '`lbfgs` ' handle multinomial loss.
- '`liblinear` ' is limited to one-versus-rest schemes

By default, logistic regression uses lbfgs solver.

```
1 logit_classifier = LogisticRegression(solver='lbfgs')
```

# How to add regularization in Logistic Regression classifier?

- $l2$ - adds a L2 penalty term
- $l1$ - adds a L1 penalty term




- elasticnet - both L1 and L2 penalty terms are added
- none - no penalty is added

Regularization is applied by default because it improves numerical stability.

By default, it uses L2 penalty.

```
1 logit_classifier = LogisticRegression(penalty='l2')
```

- Not all the solvers supports all the penalties.
- Select appropriate solver for the desired penalty.
  - L2 penalty is supported by all solvers
  - L1 penalty is supported only by a few solvers.

| Solver | Penalty |
|---|---|
| ‘newton-cg ’ | [‘l2’, ‘none’] |
| ‘lbfgs ’ | [‘l2’, ‘none’] |
| ‘liblinear’ | [‘l1’, ‘l2’] |
| ‘sag’ | [‘l2’, ‘none’] |
| ‘saga ’ | [‘elasticnet’, ‘l1’, ‘l2’, ‘none’] |

# How to control amount of regularization in logistic regression?

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

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

- 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.

What purpose does it serve?

- 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.

This parameter is available in classifier estimators in sklearn.

Exercise: Read stack overflow discussion on this parameter.

LogisticRegressionCV implements logistic regression with in built cross validation support to find the best values of C and l1_ratio parameters according to the specified scoring attribute.

These classifiers can also be implemented with a generic SGDClassifier API by setting the loss parameter appropriately.

Let's study SGDClassifier API.

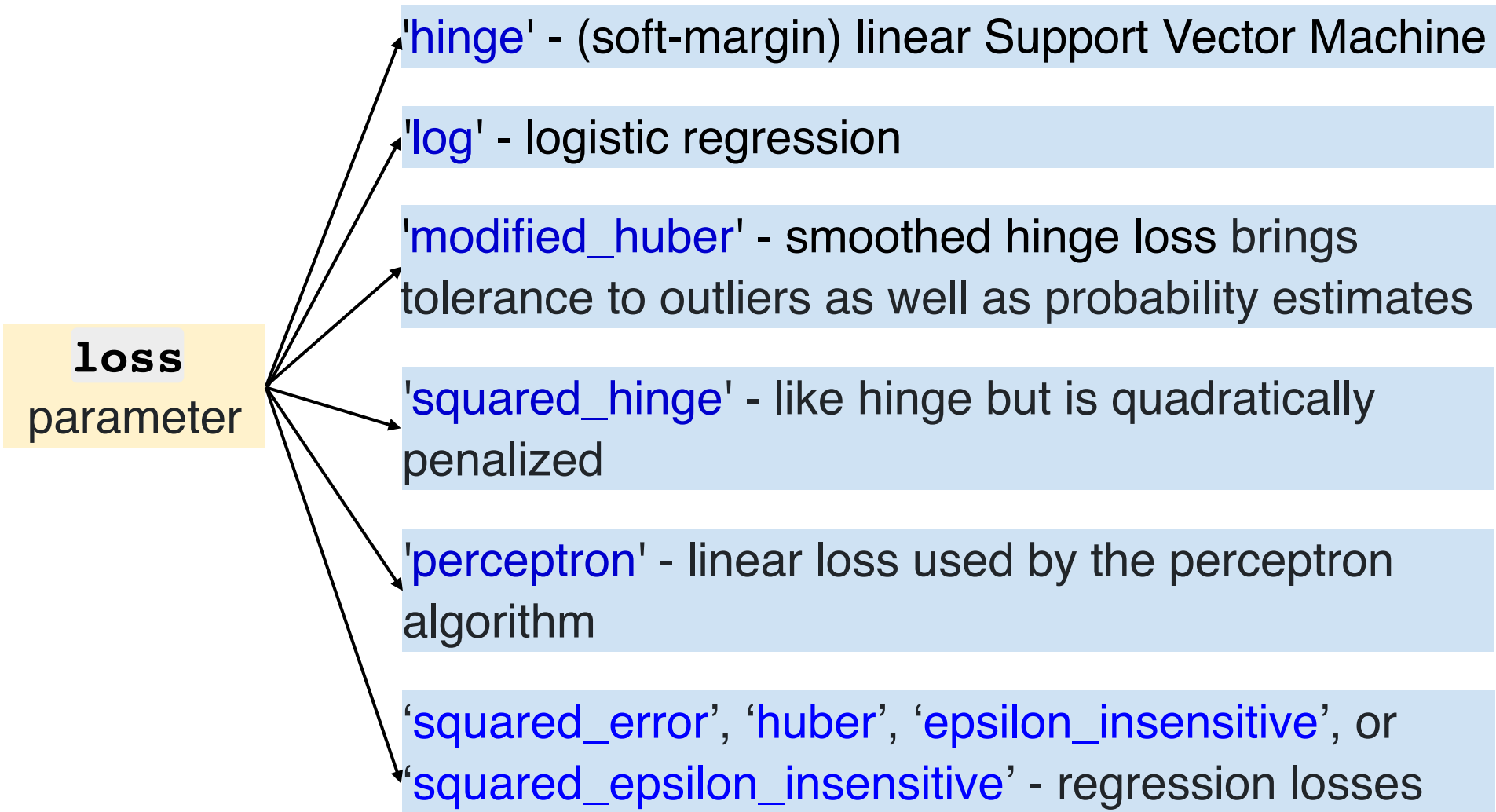
# SGDClassifier

- **SGD** is a simple yet very efficient approach to fitting linear classifiers under convex loss functions

- This API uses SGD as an optimization technique and can be applied to build a variety of linear classifiers by adjusting the loss parameter.

- 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 $10^5$ features. It also works with sparse machine learning problems
  - Text classification and natural language processing

We need to set loss parameter appropriately to build train classifier of our interest with SGDClassifier



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

- An instance of SGDClassifier might have an equivalent estimator in the scikit-learn API.

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

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

```
SGDClassifier(loss='hinge')
```

Linear Support vector machine

# How does SGDClassifier work?

- 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.
  - to standardize the features.

# How to use SGDClassifier for training a classifer?

**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.

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

Here we have used `log` loss that defines a logistic regression classifier.

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

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

# How to perform regularization in SGD classifier?

- $l2$ - adds a L2 penalty term
- $l1$ - adds a L1 penalty term

**penalty**

- elasticnet - Convex combination of L2 and L1

`(1 - l1_ratio) * L2 + l1_ratio * L1`

(`l1_ratio` controls the convex combination of L1 and L2 penalty. default=0.15)

Default:

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

**alpha**

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

# How to set maximum number of epochs for SGD Classifier?

The maximum number of passes over the training data (aka epochs) is an integer that can be set by the `max_iter` parameter.

```
1 SGD_classifier = SGDClassifier(max_iter=100)
```

Default:

**`max_iter = 1000`**

# Some common parameters between SGDClassifier and SGDRegressor

**`learning_rate`**

- ‘constant’
- ‘optimal’
- ‘invscaling’
- ‘adaptive’

**`warm_start`**

- ‘True’
- ‘False’

**`average`**

- SGDClassifier also supports averaged SGD (ASGD)

Stopping criteria

- **`tol`**
- **`n_iter_no_change`**
- **`max_iter`**
- **`early_stopping`**
- **`validation_fraction`**

# Summary

We learnt how to implement the following classifiers with sklearn APIs:

- Least square classification (RidgeClassifier)
- Perceptron (Perceptron)
- Logistic regression (LogisticRegression)

Alternatively we can use SGDClassifier with appropriate loss setting for implementing these classifiers:

- loss = `log` for logistic regression
- loss = `perceptron` for perceptron
- loss = `squared_error` for least square classification

Classification estimators implements a few common methods like fit, score, decision_function, and predict.

- These estimators can be readily used in multiclass setting.
- They support regularized loss function optimization.
- All classification estimators have ability to deal with class imbalance through class_weight parameter in the constructor.

# Part II: Multi-learning classification set up

Let's extend these classifiers to multi-learning (multi-class, multi-label & multi-output) settings.

# Basics of multiclass, multilabel and multioutput classification

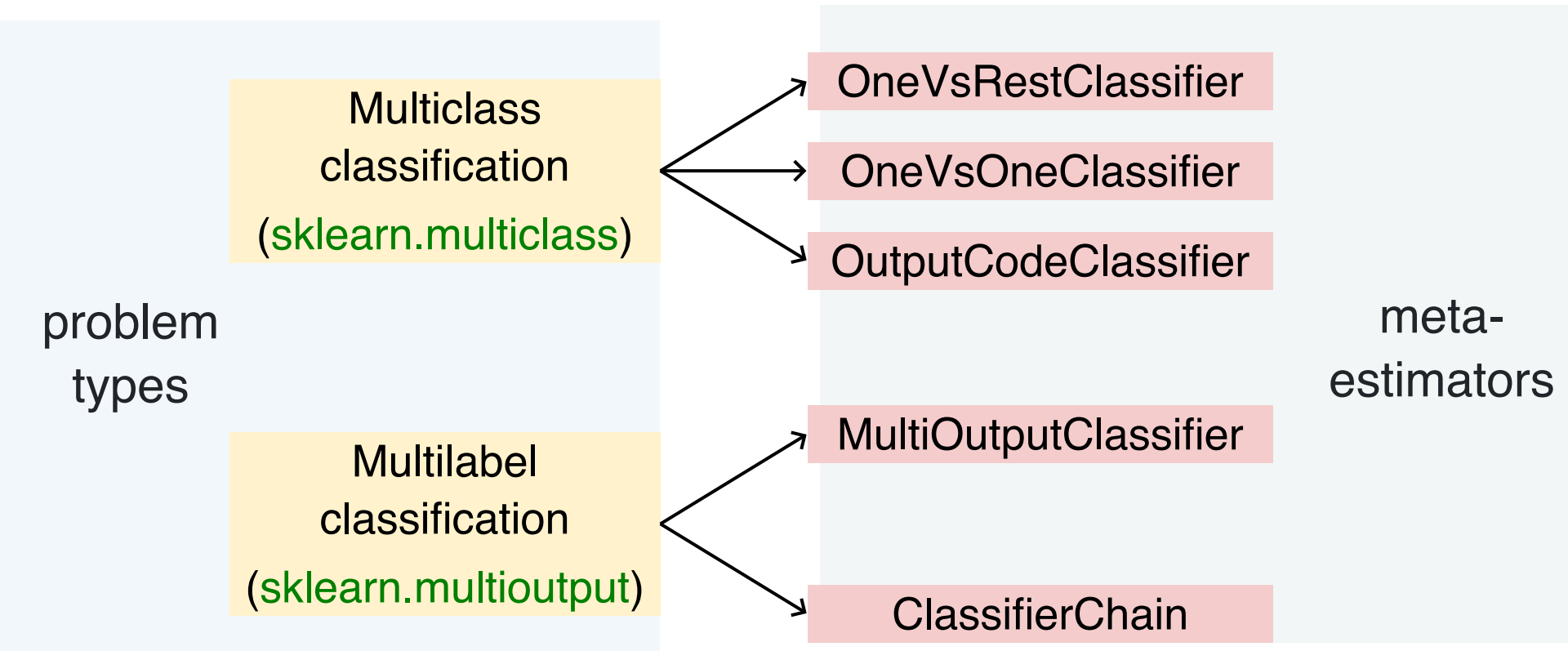
- Multiclass classification has exactly one output label and the total number of labels > 2.
- For more than one output, there are two types of classification models:

| Multilabel | Multiclass multioutput |
| --- | --- |
| total #labels = 2 | total #labels > 2 |

We will refer both these models as multi-label classification models, where # of output labels > 1.

Multiclass, multilabel, multioutput problems are referred to as multi-learning problems.

- 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 | Multiclass as OVO | Multiclass as OVR | Multilabel |
|---|---|---|---|

| | |
|---|---|
| Inherently multiclass | LogisticRegression (multi_class = 'multinomial')<br>LogisticRegressionCV (multi_class = 'multinomial')<br>RidgeClassifier<br>RidgeClassifierCV |
| Multiclass as OVR | LogisticRegression (multi_class = 'ovr')<br>LogisticRegressionCV (multi_class = 'ovr')<br>SGDClassifier<br>Perceptron |
| Multilabel | RidgeClassifier<br>RidgeClassifierCV |

First we will study multiclass APIs in sklearn.

# Multi-class classification

- Classification task with more than two classes.
- Each example is labeled with exactly one class

In Iris dataset,

- There are three class labels: setosa, versicolor and virginica.
- Each example has exactly one label of the three available class labels.
- Thus, this is an instance of a multi-class classification.

In MNIST digit recognition dataset,

- There are 10 class labels: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- Each example has exactly one label of the 10 available class labels.
- Thus, this is an instance of a multi-class classification.

# How to represent 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.

```
1 from sklearn.preprocessing import LabelBinarizer
2 y = np.array(['apple', 'pear', 'apple', 'orange'])
3 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]]
```

Let's say, you are given labels as part of the training set, how do we check if they are is suitable for multi-class classification?

- Use type_of_target to determine the type of the label.

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

- In case, $y$ is a vector with more than two discrete values, type_of_target returns multiclass.

type_of_target can determine different types
of multi-learning targets.

| target_type | y |
| --- | --- |
| ‘multiclass’ | • contains more than two discrete values<br>• not a sequence of sequences<br>• 1d or a column vector |
| ‘multiclass-multioutput’ | • 2d array that contains more than two discrete values<br>• not a sequence of sequences<br>• dimensions are of size > 1 |
| ‘multilabel-indicator’ | • label indicator matrix<br>• 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,<br>• sequence of sequences, or an array of non-sequence objects. |

# Examples

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'
```

Apart from these, there are three more types, type_of_target can determine targets corresponding to regression and binary classification.

- continuous - regression target
- continuous-multioutput - multi-output target
- binary - classification

All classifiers in scikit-learn perform multiclass classification out-of-the-box.

- Use sklearn.multiclass module only when you want to experiment with different multiclass strategies.
- Using different multi-class strategy than the one implemented by default may affect performance of classifier in terms of either generalization error or computational resource requirement.

# What are different multi-class classification strategies implemented in sklearn?

- One-vs-all or one-vs-rest (OVR)
- One-vs-One (OVA)

- OVR is implemented by OneVsRestClassifier API.
- OVA is implemented by OneVsOneClassifier API.

# OVR - OneVsRestClassifier

- Fits one classifier per class $c$ - $c$ vs not $c$.
- This approach is computationally efficient and requires only $k$ classifiers.
- The resulting model is interpretable.

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

- We need to supply estimator as an argument in the constructor.
- Support methods like other classifiers - fit, predict, predict_proba, partial_fit.

OneVsRest classifier also supports multilabel classification. We need to supply labels as indicator matrix of shape $(n, k)$.

# OVA - OneVsOneClassifier

- 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.

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

- We need to supply estimator as an argument in the constructor.
- Support methods like other classifiers - fit, predict, predict_proba, partial_fit.

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

# What is the difference between OVR and OVA?

**OneVsRestClassifier**

- Fits one classifier per class.
- For each classifier, the class is fitted against all the other classes.

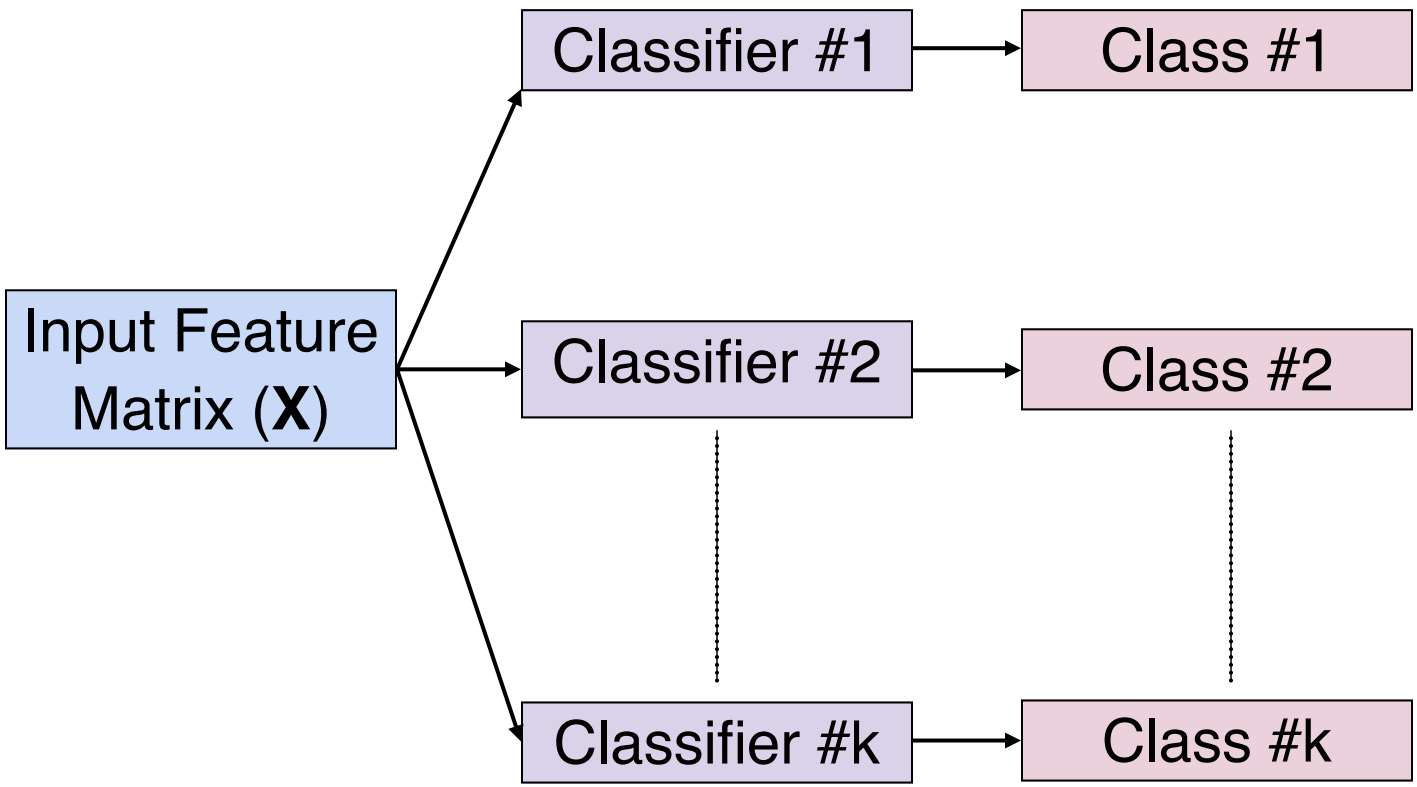
**OneVsOneClassifier**

- Fits one classifier per pair of classes.
- At prediction time, the class which received the most votes is selected.

Now we will learn how to perform multilabel and multi-output classification.

# How MultiOutputClassifier works?

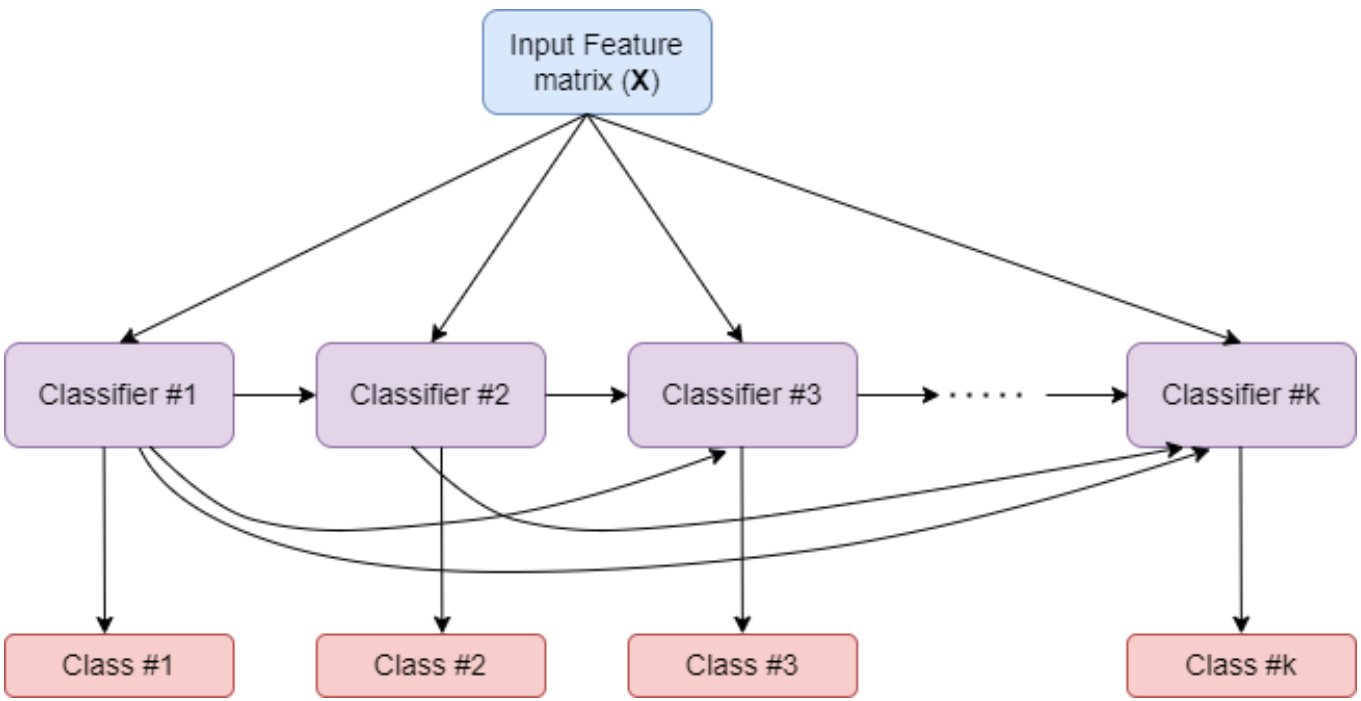
- Strategy consists of fitting one classifier per target.




- Allows multiple target variable classifications.

# How ClassifierChain works?

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



# Comparison of MultiOutputClassifier and ClassifierChain

| 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 $k-1$.<br>• These integers define the order of models in the chain. |

# Summary

- Different types of multi-learning setups: multi-class, multi-label, multi-output.
- type_of_target to determine the nature of supplied labels.
- Meta-estimators:
  - multi-class: One-vs-rest, one-vs-one
  - multi-label: Classifier chain and multi-output classifier

# Evaluating Classifiers

So far we learnt how to train classifiers for binary, multi-class and multi-label/output cases.

We will learn how to evaluate these classifiers with different scoring functions and with cross-validation.

We will also study how to set hyper-parameters for classifiers.

Many cross-validation and HPT methods discussed in the regression context are also applicable in classifiers.

- We will not repeat that discussion in this topic.
- Instead we will focus on only additional methods that are specific to classifiers.

# Stratified cross validation iterators

There may be issues like class imbalance in classification, which tend to impact the cross validation folds.

The overall class distribution and the ones in folds may be different and this has implications in effective model training.

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

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

- StratifiedKFold
- RepeatedStratifiedKFold
- StratifiedShuffleSplit

**Note**: Folds obtained via StratifiedShuffleSplit 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.

Now let's look at classification metrics implemented in sklearn.

# Classification metrics

sklearn.metrics implements a bunch of classification scoring metrics based on true labels and predicted labels as inputs.

`accuracy_score` `balanced_accuracy_score`

`top_k_accuracy_score` `roc_auc_score`

`precision_score` `recall_score` `f1_score`

score(actual_labels, predicted_labels)

# Confusion matrix

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

```
1 from sklearn.metrics import confusion_matrix
2 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

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

- From estimators

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

- From predictions

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

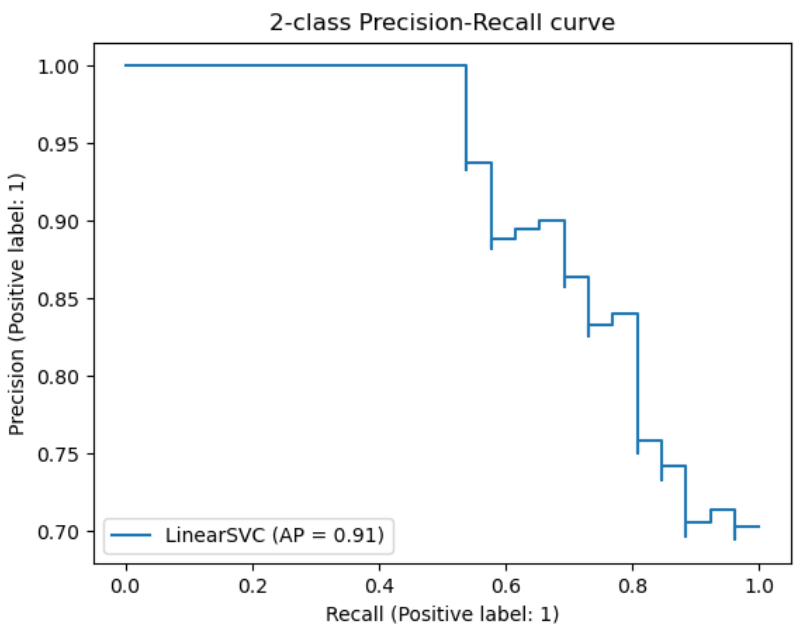
The `classification_report` function builds a text report showing the main classification metrics.

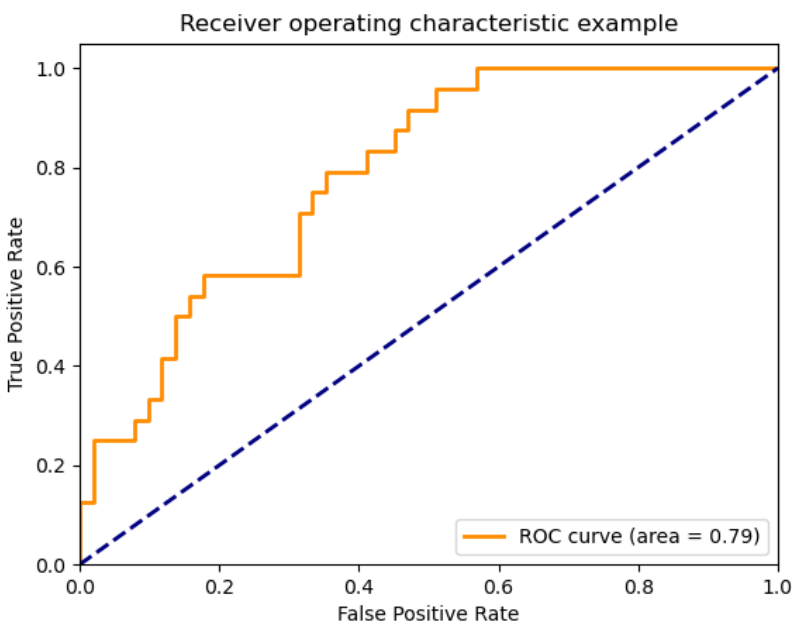
```
1 from sklearn.metrics import classification_report
2 print(classification_report(y_true, y_predicted))
```

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| class 0 | 0.67 | 1.00 | 0.80 | 2 |
| class 1 | 0.00 | 0.00 | 0.00 | 1 |
| class 2 | 1.00 | 0.50 | 0.67 | 2 |
| accuracy | | | 0.60 | 5 |
| macro avg | 0.56 | 0.50 | 0.49 | 5 |
| weighted avg | 0.67 | 0.60 | 0.59 | 5 |

# Classifier Performance across probability thresholds

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




```
1 from sklearn.metrics import roc_curve
2 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 |

# Summary

- Classification specific cross validation iterator based on stratification.
- Classification metrics
- Extending binary metrics to multi-learning set up.

# Naive Bayes in sci-kit learn

Dr. Ashish Tendulkar

IIT Madras

Machine Learning Practice

# Naive Bayes Classifier

# Naive Bayes classifier

- Naive Bayes classifier applies Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable.

For a given class variable $y$ and dependent feature vector $x_1$ through $x_m$,

the naive conditional independence assumption is given by:

$$P(x_i|y, x_1, ..., x_{i-1}, x_{i+1}, ..., x_m) = P(x_i|y)$$

Naive Bayes learners and classifiers can be extremely fast compared to more sophisticated methods.

# List of NB Classifiers

- Implemented in sklearn.naive_bayes module

GaussianNB

BernoulliNB

CategoricalNB

MultinomialNB

ComplementNB

- Implements fit method to estimate parameters of NB classifier with feature matrix and labels as inputs.
- The prediction is performed using predict method.

# Which NB to use if data is only numerical?

GaussianNB —— implements the Gaussian Naive Bayes algorithm for classification

Instantiate a GaussianNBClassifer estimator and then call fit method using X_train and y_train.

```
1 from sklearn.naive_bayes import GaussianNB
2 gnb = GaussianNB()
3 gnb.fit(X_train, y_train)
```

# Which NB to use if data is multinomially distributed?

MultinomialNB —— implements the naive Bayes algorithm for multinomially distributed data (text classification)

Instantiate a MultinomialNBClassifer estimator and then call fit method using X_train and y_train.

```
1 from sklearn.naive_bayes import MultinomialNB
2 mnb = MultinomialNB()
3 mnb.fit(X_train, y_train)
```

# What to do if data is imbalanced ?

ComplementNB —— implements the complement naive Bayes (CNB) algorithm.

Instantiate a ComplementNBClassifer estimator and then call fit method using X_train and y_train.

```
1 from sklearn.naive_bayes import ComplementNB
2 cnb = ComplementNB()
3 cnb.fit(X_train, y_train)
```

CNB regularly outperforms MNB (often by a considerable margin) on text classification tasks.

# What to do if data has multivariate Bernoulli distributions?

BernoulliNB

- implements the naive Bayes algorithm for data that is distributed according to multivariate Bernoulli distributions
- each feature is assumed to be a binary-valued (Bernoulli, boolean) variable

Instantiate a BernoulliNBClassifer estimator and then call fit method using X_train and y_train.

```
1 from sklearn.naive_bayes import BernoulliNB
2 bnb = BernoulliNB()
3 bnb.fit(X_train, y_train)
```

# What to do if data is categorical ?

CategoricalNB

implements the categorical naive Bayes algorithm suitable for classification with discrete features that are categorically distributed

assumes that each feature, which is described by the index $i$, has its own categorical distribution.

Instantiate a CategoricalNBClassifer estimator and then call fit method using X_train and y_train.

```
1 from sklearn.naive_bayes import CategoricalNB
2 canb = CategoricalNB()
3 canb.fit(X_train, y_train)
```

# K Nearest Neighbours

Dr. Ashish Tendulkar

IIT Madras

Machine Learning Practice

# Nearest neighbor classifier

- It is a type of instance-based learning or non-generalizing learning
  - does not attempt to construct a model
  - simply stores instances of the training data

- Classification is computed from a simple majority vote of the nearest neighbors of each point.

- Two different implementations of nearest neighbors classifiers are available.

1. KNeighborsClassifier
2. RadiusNeighborsClassifier

# How are KNeighborsClassifier and RadiusNeighborsClassifier different?

| KNeighborsClassifier | RadiusNeighborsClassifier |
| --- | --- |
| • learning based on the k nearest neighbors | • learning based on the number of neighbors within a fixed radius *r* of each training point |
| • most commonly used technique | • used in cases where the data is not uniformly sampled |
| • choice of the value *k* is highly data-dependent | • fixed value of *r* is specified, such that points in sparser neighborhoods use fewer nearest neighbors for the classification |

# How do you apply KNeighborsClassifier?

**Step 1**: Instantiate a KNeighborsClassifer estimator without passing any arguments to it to create a classifer object.

```
1 from sklearn.neighbors import KNeighborsClassifier
2 kneighbor_classifier = KNeighborsClassifier()
```

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

```
1 # Model training with feature matrix X_train and
2 # label vector or matrix y_train
3 kneighbor_classifier.fit(X_train, y_train)
```

# How do you specify the number of nearest neighbors in KNeighborsClassifier?

- Specify the number of nearest neighbors *K* from the training dataset using `n_neighbors` parameter.
  - value should be int.

```
1 kneighbor_classifier = KNeighborsClassifier(n_neighbors = 3)
```

## What is the default value of *K*?

`n_neighbors = 5`

# How do you assign weights to neighborhood in KNeighborsClassifier?

- It is better to weight the neighbors such that nearer neighbors contribute more to the fit.

**weights**

- 'uniform' : All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance.
  - closer neighbors of a query point will have a greater influence than neighbors which are further away.

Default:

```
1 kneighbor_classifier = KNeighborsClassifier(weights= 'uniform')
```

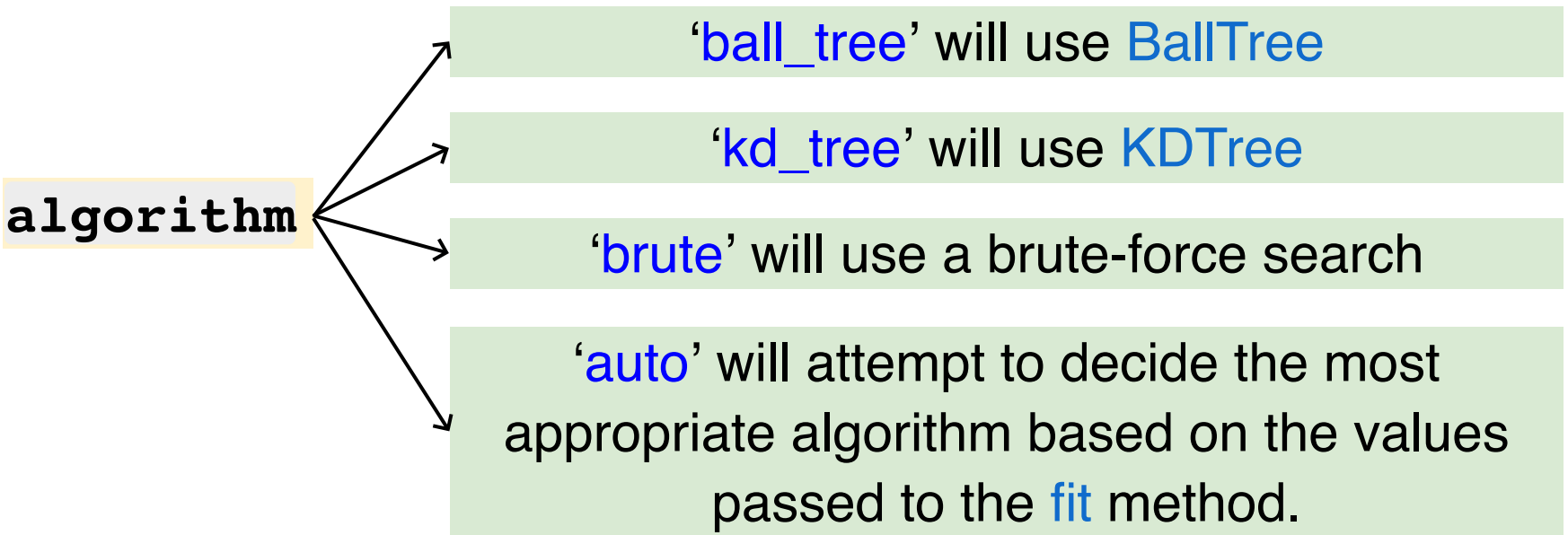
# Can we define our own weight values for KNeighborsClassifier?

- Yes, it is possible if you have an array of distances.
- **`weights`** parameter also accepts a user-defined function which takes an array of distances as input, and returns an array of the same shape containing the weights.

Example:

```
1 def user_weights(weights_array):
2     return weights_array
3
4 kneighbor_classifier = KNeighborsClassifier(weights=user_weights)
```

# Which algorithm is used to compute the nearest neighbors in KNeighborsClassifier?




Default:

```
1 kneighbor_classifier = KNeighborsClassifier(algorithm='auto')
```

# Some additional parmeters for tree algorithm in KNeighborsClassifier?

For 'ball_tree' and 'kd_tree' algorithms, there are some other parameters to be set.

**`leaf_size`**

- can affect the speed of the construction and query, as well as the memory required to store the tree
- default = 30

**`metric`**

- Distance metric to use for the tree
- It is either string or callable function
  - some metrics are listed below:
    - “euclidean”, “manhattan”, “chebyshev”, “minkowski”, “wminkowski”, “seuclidean”, “mahalanobis”
- default = 'minkowski'

**`p`**

- Power parameter for the Minkowski metric.
- default = 2

# How do you apply RadiusNeighborsClassifier?

**Step 1**: Instantiate a RadiusNeighborsClassifer estimator without passing any arguments to it to create a classifer object.

```
1 from sklearn.neighbors import RadiusNeighborsClassifier
2 radius_classifier = RadiusNeighborsClassifier()
```

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

```
1 # Model training with feature matrix X_train and
2 # label vector or matrix y_train
3 radius_classifier.fit(X_train, y_train)
```

# How do you specify the number of neighbors in RadiusNeighborsClassifier?

- The number of neighbors is specified within a fixed radius *r* of each training point using `radius` parameter.
- *r* is a float value.

```
1 radius_classifier = RadiusNeighborsClassifier(radius=1.0)
```

## What is the default value of *r* ?

**`r = 1.0`**

# Parameters for RadiusNeighborsClassifier

**`weights`**

- ‘uniform’
- ‘distance’
- [callable] function
- default = 'uniform'

**`algorithm`**

- ‘ball_tree’
- ‘kd_tree’
- ‘brute’
- ‘auto’
- default = *‘auto’*

**`leaf_size`**

default = 30

**`metric`**

default = 'minkowski'

**`p`**

default = 2

# Support Vector Machines

Dr. Ashish Tendulkar

IIT Madras

Machine Learning Practice

- In this week, we will study how to implement support vector machines for classification tasks with sklearn.

# Support Vector Machines

- Support Vector Machines (SVM) are a set of supervised learning methods used for classification, regression and outliers detection.

- SVM constructs a hyper-plane or set of hyper-planes in a high or infinite dimensional space, which can be used for classification, regression or other tasks.

- In sklearn, we have three methods to implement SVM.

SVC

NuSVC

These are similar methods but, accept slightly different sets of parameters.

Implementation is based on libsvm.

LinearSVC

Faster implementation of linear SVM classification with only linear kernel.

Implementation is based on liblinear.

# Training data

Array $X$ : holding the training samples

```
1 X = [[0, 0], [1, 1]]
```

shape $\rightarrow$ (n_samples, n_features)

Array $y$ : holding the class labels (strings or integers)

```
1 y = [0,1]
```

shape $\rightarrow$ (n_samples)

# How to implement SVC (C-Support Vector Classification)?

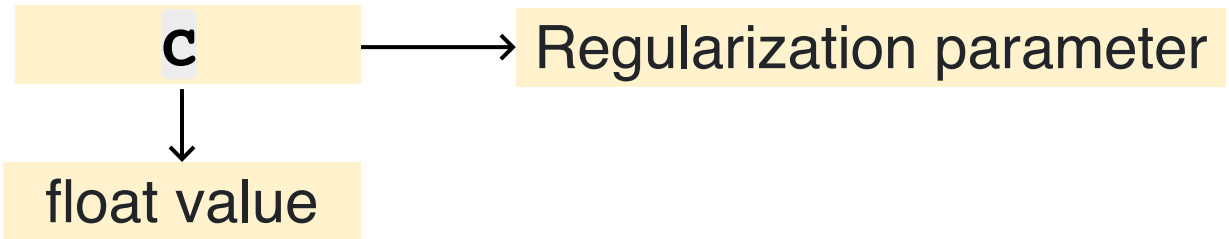
**Step 1**: Instantiate a SVC classifier estimator.

```
1 from sklearn.svm import SVC
2 SVC_classifier = SVC()
```

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

```
1 # Model training with feature matrix X_train and
2 # label vector or matrix y_train
3 SVC_classifier.fit(X_train, y_train)
```

# How to perform regularization in SVC classifier?



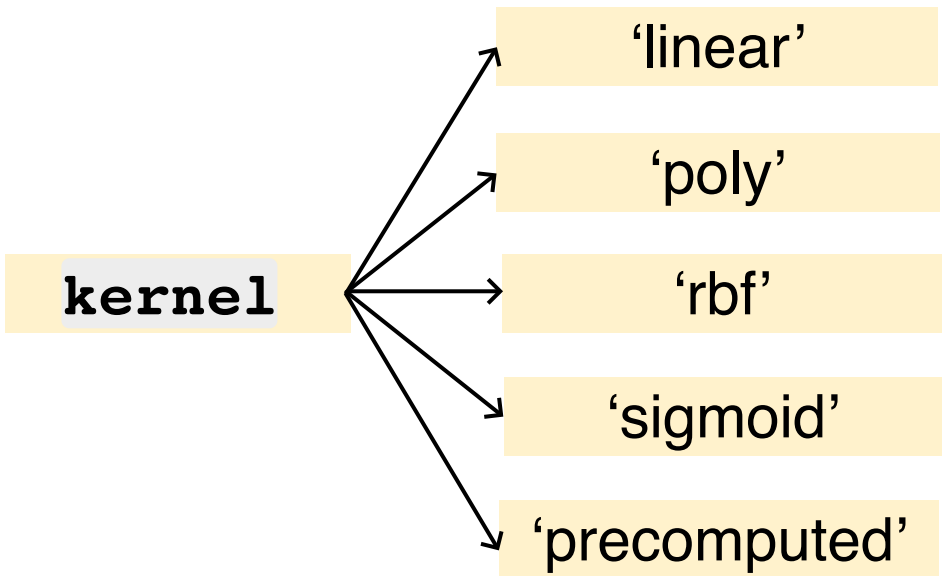

Default:

```
1 SVC_classifier = SVC(C=1.0)
```

Note:

- strength of the regularization is inversely proportional to C
- strictly positive
- penalty is a squared l2 penalty

# How to specify kernel type to be used in the algorithm ?




Default:

```
1 SVC_classifier = SVC(kernel = 'rbf')
```

- If **`kernel = poly`** , set **`degree`** (any integer value)
- If **`kernel = callable`** is given it is used to pre-compute the kernel matrix from data matrices

# How to set kernel coefficient for *'rbf'*, *'poly'* and *'sigmoid'* kernels?

**`gamma`**

- ‘scale’ → value of gamma = $\frac{1}{\text{number of features} * \text{X.Var()}}$
- ‘auto’ → value of gamma = $\frac{1}{\text{number of features}}$
- float value

Default:

```
1 SVC_classifier = SVC(gamma = 'scale')
```

- If **`kernel = 'poly'`** or **`'sigmoid'`** , set **`coef0`** which is an independent term in kernel function (any integer value)

# How to view support vectors?

After the classifier is fit on the training data, there are few attributes which reveal the details of support vectors.

```
1  from sklearn.svm import SVC
2  SVC_classifier = SVC()
3  clf = SVC_classifier.fit(X_train, y_train)
4
5  #to view indices of the support vectors
6  clf.support_
7
8  #to view the support vectors
9  clf.support_vectors_
10
11 #to view the number of support vectors for each class
12 clf.n_support_
```

# How to implement NuSVC ($\nu$-Support Vector Classification)?

**Step 1**: Instantiate a NuSVC classifier estimator.

```
1 from sklearn.svm import NuSVC
2 NuSVC_classifier = NuSVC()
```

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

```
1 # Model training with feature matrix X_train and
2 # label vector or matrix y_train
3 NuSVC_classifier.fit(X_train, y_train)
```

# What is the significance of $\nu$ in NuSVC?

Instead of C in SVC, $\nu$ is introduced in NuSVC to control the number of support vectors and margin errors.

$\nu$ is an upper bound on the fraction of margin errors and and a lower bound of the fraction of support vectors.

Value of $\nu$ should $\in (0, 1]$

Default: $\nu = 0.5$

Other parameters for NuSVC are same as that of SVC.

# How to implement LinearSVC (Linear Support Vector Classification)?

**Step 1**: Instantiate a LinearSVC classifier estimator.

```
1 from sklearn.svm import LinearSVC
2 LinearSVC_classifier = LinearSVC()
```

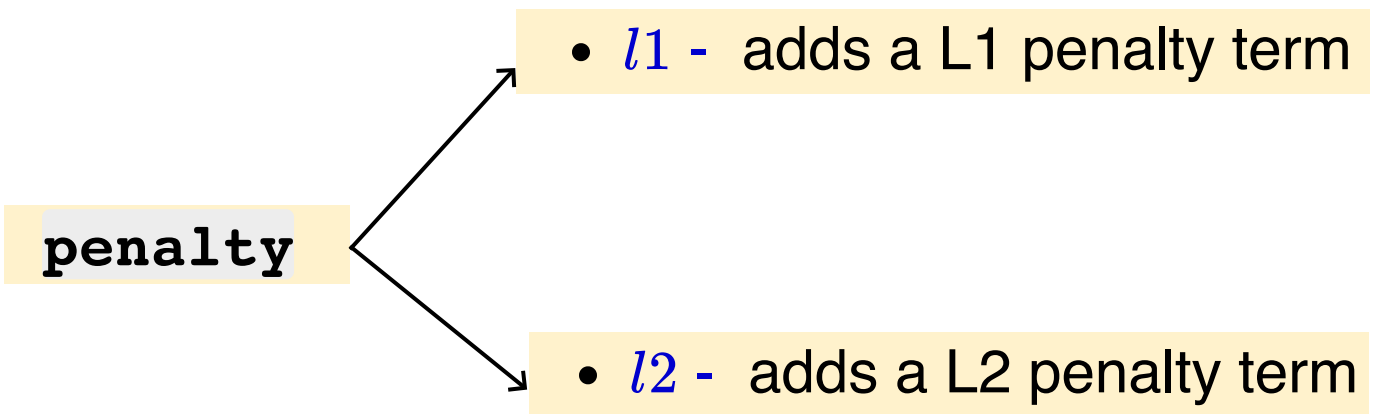
**Step 2**: Call fit method on SVC classifier object with training feature matrix and label vector as arguments.

```
1 # Model training with feature matrix X_train and
2 # label vector or matrix y_train
3 LinearSVC_classifier.fit(X_train, y_train)
```

# Advantages of LinearSVC

- It has more flexibility in the choice of penalties and loss functions since it is implemented in terms of liblinear.
- Scales better to large numbers of samples.
- Supports both dense and sparse input.

# How to provide penalty in LinearSVC classifier?




- $l1$ - leads to **`coef_`** vectors that are sparse.

Default:

```
1 LinearSVC_classifier = Linear_SVC(penalty = 'l2')
```

# How to choose loss functions in LinearSVC classifier?

**`loss`** parameter

- 'hinge' - standard SVM loss
- 'squared_hinge' - square of the hinge loss

Default:

```
1 LinearSVC_classifier = Linear_SVC(loss = 'squared_hinge')
```

Combination not supported:

**`penalty='l1'`** and **`loss='hinge'`**

# Some parameters in LinearSVC classifier

**`C`**

↓

Regularization parameter

**`dual`**

↓

- Select the algorithm to either solve the dual or primal optimization problem.
- When n_samples >n_features, prefer dual=False.

**`fit_intercept`**

↓

To calculate the intercept for the model.

# How to perform multi-class classification using SVM?

- SVC and NuSVC implement the "one-versus-one" approach for multi-class classification.

**`decision_function_shape`**

- 'ovo'
- 'ovr'

- LinearSVC implements "one-vs-the-rest" approach for multi-class classification.

**`multi_class`**

- 'ovr'
- 'crammer_singer'

# Advantages of SVM

- Effective in high dimensional spaces.
- Effective in cases where number of dimensions is greater than the number of samples.
- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
- Versatile: different Kernel Functions can be specified for the decision function.

# Disadvantages of SVM

- SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation.
- Avoid over-fitting in choosing Kernel functions if the number of features is much greater than the number of samples.

# Decision trees

Machine Learning Practice

Dr. Ashish Tendulkar

IIT Madras

# Quick recap

- Non-parametric supervised learning methods.
- Can learn classification and regression models.
- Predicts label based on rules inferred from the features in the training set.

# Tree algorithms

**ID3**
- ID3= Iterative Dichotomiser 3
- Creates a multiway tree

**C4.5**
- Successor to ID3
- Converts the trained trees into sets of if-then rules

**C5.0**
- Quinlan's latest version release under a proprietary license
- Uses less memory and builds smaller rulesets

**CART**
- Classification and Regression Trees
- Supports numerical target variables (regression) and does not compute rule sets

# sklearn implementation of trees

**scikit-learn** uses an optimized version of the **CART algorithm**; however, it does not support categorical variables for now

Classification `sklearn.tree.DecisionTreeClassifier`

Regression `sklearn.tree.DecisionTreeRegressor`

Both these estimators have the same set of parameters except for `criterion` used for tree splitting.

`splitter` `max_depth` `min_samples_split` `min_samples_leaf`

# sklearn tree parameters

`splitter` — Strategy for splitting at each node. `best` `random`

`max_depth` — Maximum depth of the tree.

`int`

When `None`, the tree expanded until all leaves are pure or they contain less than `min_samples_split` samples.

`min_samples_split` — The minimum number of samples required to split an internal node. `2`

`int` `float`

`min_samples_leaf` — The minimum number of samples required to be at a leaf node. `1`

`int` `float`

# sklearn tree parameters

criterion

Specifies function to measure the quality of a split.

| Classification | Regression |
| --- | --- |
| gini | squarred_error |
| entropy | friedman_mse |
| | absolute_error |
| | poisson |

# Tree visualization

`sklearn.tree.plot_tree`

| | |
|---|---|
| `decision_tree` | The decision tree to be plotted. |
| `max_depth` | The maximum depth of the representation. If `none`, the tree is fully generated. |
| `feature_names` | Names of each of the features. `none` |
| `class_names` | Names of each of the target classes in ascending numerical order. `none` |
| `label` | Whether to show informative labels for impurity. `none` |

# Avoiding overfitting of trees

## Pre-pruning

Uses hyper-parameter search like `GridSearchCV` for finding the best set of parameters.

## Post-pruning

First grows trees without any constraints and then uses `cost_complexity_pruning` with `max_depth` and `min_samples_split`.

# Tips for practical usage

- Decision trees tend to overfit data with a large number of features. Make sure that we have the right ratio of samples to number of features.
- Perform dimensionality reduction (PCA, or Feature Selection) on a data before using it for training the trees. It gives a better chance of finding discriminative features.
- Visualize the trained tree by using max_depth=3 as an initial tree depth to get a feel for the fitment and then increase the depth.
- Balance the dataset before training to prevent the tree from being biased toward the classes that are dominant.

- Use min_samples_split or min_samples_leaf to ensure that multiple samples influence every decision in the tree, by controlling which splits will be considered.
  - A very small number will usually mean the tree will overfit.
  - A large number will prevent the tree from learning the data.

# Neural Networks

Dr. Ashish Tendulkar

IIT Madras

Machine Learning Practice

- In this week, we will study how to implement Multilayer Perceptron neural network models for classification and regression tasks with sklearn.

# Multilayer Perceptron (MLP)

- It is a supervised learning algorithm.

- MLP learns a non-linear function approximator for either classification or regression depending on the given dataset.

- In sklearn, we implement MLP using:

1. MLPClassifier for classification
2. MLPRegressor for regression

- MLPClassifier supports multi-class classification by applying Softmax as the output function.
- It also supports supports multi-label classification in which a sample can belong to more than one class.

- MLPRegressor also supports multi-output regression, in which a sample can have more than one target.

# Training data

Array $X$ : holds the training samples

shape $\rightarrow$ (n_samples, n_features)

Array $y$ : holds the target

shape $\rightarrow$ (n_samples,)

# MLPClassifier

- How to implement MLPClassifier?

**Step 1**: Instantiate a MLP classifier estimator.

```
1 from sklearn.neural_network import MLPClassifier
2 MLP_clf = MLPClassifier()
```

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

```
1 # Model training with feature matrix X_train and
2 # label vector or matrix y_train
3 MLP_clf.fit(X_train, y_train)
```

# MLPClassifier

**Step 3**: After fitting (training), the model can make predictions for new samples (X_test) using two methods:

```
1 MLP_clf.predict(X_test)
2 MLP_clf.predict_proba(X_test)
```

**predict** →

- gives labels for new samples
- for example:

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

**predict_proba** →

- gives vector of probability estimates per sample
- for example:

```
array([1.967...e-04, 9.998...-01])
```

- MLPClassifier supports only the Cross-Entropy loss function

# How to set the number of hidden layers?

**`hidden_layer_sizes`**

- This parameter sets the number of layers and the number of neurons in each layer.

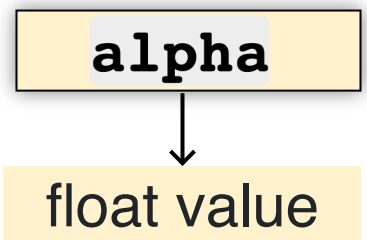
- It is a **tuple** where each element in the tuple represents the number of neurons at the $i$ th position where $i$ is the index of the tuple.
- The length of tuple denotes the total number of hidden layers in the network.

To create a 3 hidden layer neural network with 15 neurons in first layer, 10 neurons in second layer and 5 neurons in third layer:

```
1 MLPClassifier(hidden_layer_sizes=(15,10,5))
```

# How to perform regularization in MLPClassifier?

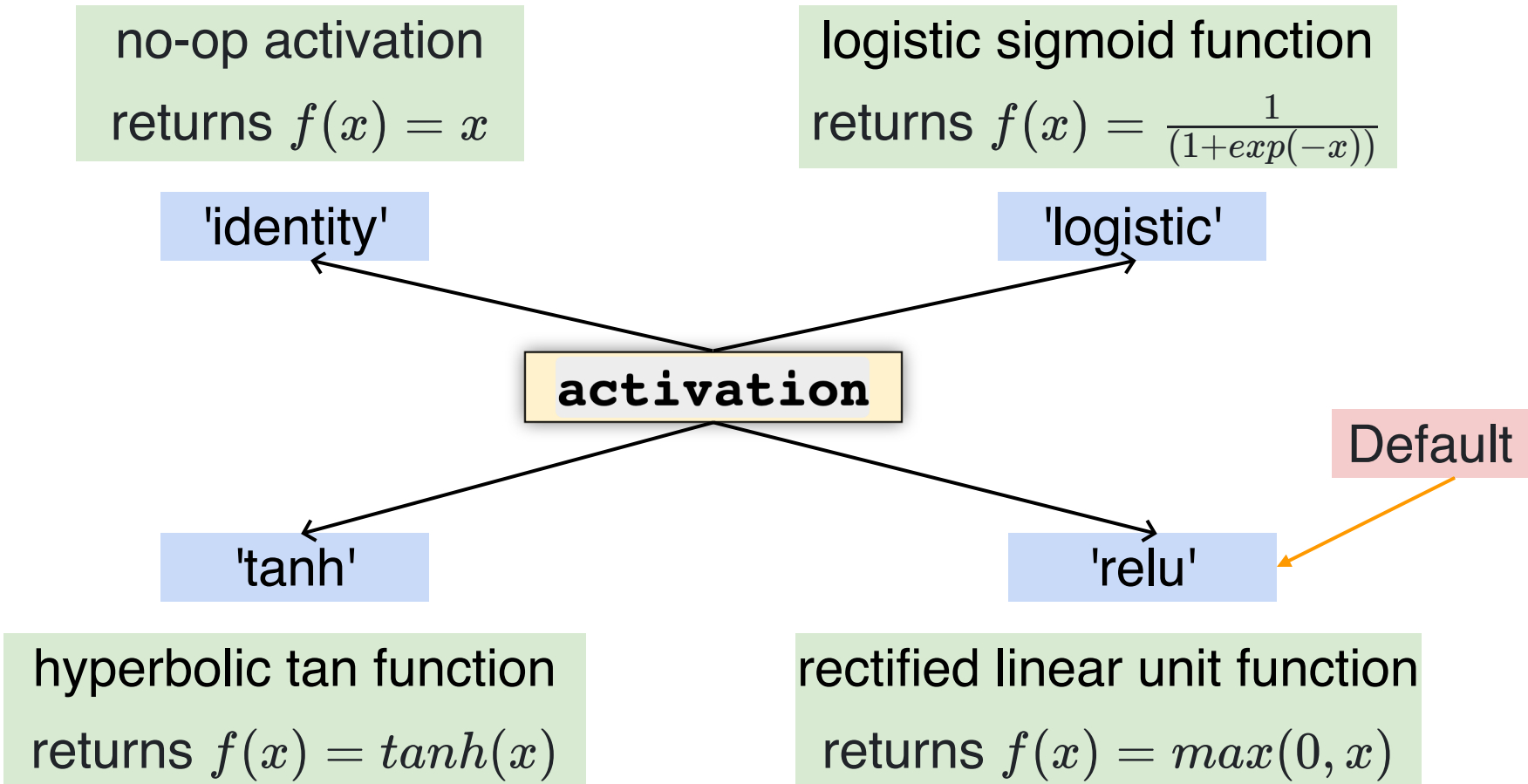
- The alpha parameter sets L2 penalty Regularization parameter




Default:

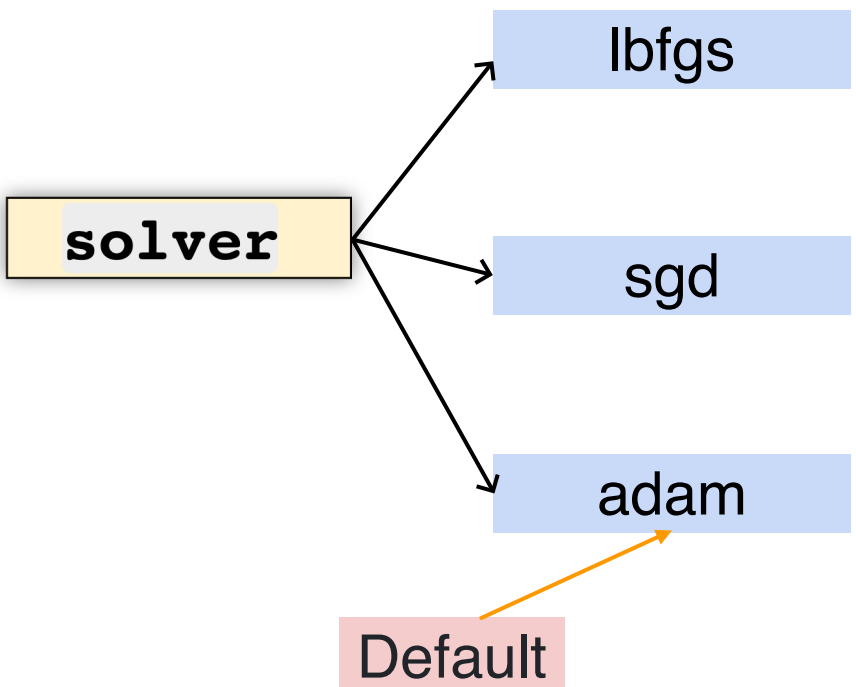
```
1 alpha = 0.0001
```

# How to set the activation function for the hidden layers?



# How to perform weight optimization in MLPClassifier?

- MLPClassifier optimizes the log-loss function using LBFGS or stochastic gradient descent




- If the **`solver`** is 'lbfgs', the classifier will not use minibatch.
- Size of minibatches can be set to other stochastic optimizers: **`batch_size`** (int)

- default batch_size is 'auto'.

```
1 batch_size=min(200, n_samples)
```

# How to view weight matrix coefficients of trained MLPClassifier?

**coefs_**

- It is a list of shape (n_layers - 1,)
- The $i$ th element in the list represents the weight matrix corresponding to layer $i$.

Example:

- "weights between input and first hidden layer:"

```
1 print(MLP_clf.coefs_[0])
```

- "weights between first hidden and second hidden layer:"

```
1 print(MLP_clf.coefs_[1])
```

```
weights between input and first hidden layer:
[[-0.14203691 -1.18304359 -0.85567518 -4.53250719 -0.60466275]
 [-0.69781111 -3.5850093  -0.26436018 -4.39161248  0.06644423]]

weights between first hidden and second hidden layer:
[[ 0.29179638 -0.14155284]
 [ 4.02666592 -0.61556475]
 [-0.51677234  0.51479708]
 [ 7.37215202 -0.31936965]
 [ 0.32920668  0.64428109]]
```

# How to view bias vector of trained MLPClassifier?

`intercepts_`

- It is a list of shape (n_layers - 1,)
- The $i$ th element in the list bias vector corresponding to layer $i + 1$.

Example:

- "Bias values for first hidden layer:"

```
1 print(MLP_clf.intercepts_[0])
```

- "Bias values for second hidden layer:"

```
1 print(MLP_clf.intercepts_[1])
```

```
Bias values for first hidden layer:
[-0.14962269 -0.59232707 -0.5472481   7.02667699 -0.87510813]

Bias values for second hidden layer:
[-3.61417672 -0.76834882]
```

# Some parameters in MLPClassifier

**`learning_rate`**

- 'constant'
- 'invscaling'
- 'adaptive'

default: 'constant'

**`learning_rate_init`**

float value

default: 0.001

**`power_t`**

float value

default: 0.5

**`max_iter`**

int value

default: 500

- `learning_rate` and `power_t` are used only for `solver = 'sgd'`
- `learning_rate_init` is used when `solver='sgd'` or ‘adam’.
- `shuffle` is used to shuffle samples in each iteration when `solver='sgd' or 'adam'`
- `momentum` is used for gradient descent update when `solver='sgd'`

# MLPRegressor

- MLPRegressor trains using backpropagation with no activation function in the output layer.
- Therefore, it uses the square error as the loss function, and the **output is a set of continuous values**.

**The parameters of MLPRegressor are the same as that of MLPClassifier.**

# How to implement MLPRegressor?

**Step 1**: Instantiate a MLP regressor estimator.

```
1 from sklearn.neural_network import MLPRegressor
2 MLP_reg = MLPRegressor()
```

**Step 2**: Call fit method on MLP regressor object with training feature matrix and label vector as arguments.

```
1 # Model training with feature matrix X_train and
2 # label vector or matrix y_train
3 MLP_reg.fit(X_train, y_train)
```

**Step 3**: After fitting (training), the model can make predictions for new samples (X_test):

```
1 MLP_reg.predict(X_test)
```

- returns predicted values for new samples
- for example:

```
array([-0.9..., -7.1...])
```

```
1 MLP_reg.score(X_test,y_test)
```

- returns $R^2$ score
- for example:

```
0.45678889
```

# Bagging and Boosting

Machine Learning Practice

Dr. Ashish Tendulkar

IIT Madras

# Part 2: Boosting

There are two boosting estimators:

- AdaBoost estimator
- Gradient boosting estimator

# AdaBoost estimator

Class: `sklearn.ensemble.AdaBoostClassifier`

Class: `sklearn.ensemble.AdaBoostRegressor`

## Class: sklearn.ensemble.AdaBoostClassifier

**base_estimator**

- Default estimator is DecisionTreeClassifier with depth = 1.

**n_estimators**

- Maximum number of estimators where boosting is terminated. The default value is 50.

**learning_rate**

- Weight applied to each classifier during boosting.
- Higher value here would increase contribution of individual classifiers.
- There is a trade-off between n_estimators and learning_rate.

## Class: sklearn.ensemble.AdaBoostRegressor

**base_estimator**

- Default estimator is DecisionTreeRegressor with depth = 3.

**n_estimators**

- Maximum number of estimators where boosting is terminated. The default value is 50.

**learning_rate**

- Weight applied to each regressor at each boosting iteration.
- Higher value here would increase contribution of individual regressor.
- There is a trade-off between n_estimators and learning_rate.

The main parameters to tune to obtain good results are

- **n_estimators** and
- Complexity of the base estimators (e.g. its depth **max_depth** or **min_samples_split**).

# Attributes of AdaBoost estimators

| | |
|---|---|
| `base_estimator_` | Base estimator of ensemble. |
| `estimators_` | Collection of fitted sub-estimators. |
| `estimator_weights_` | Weights for each estimator in ensemble. |
| `estimator_errors_` | Errors for each estimator in ensemble. |

# Gradient boosting estimators

Class: `sklearn.ensemble.GradientBoostingClassifier`

Class: `sklearn.ensemble.GradientBoostingRegressor`

There are two most important parameters of these estimators:

- `n_estimators`
- `learning_rates`

`sklearn.ensemble.GradientBoostingClassifier` supports both binary and multiclass classification.

We will directly demonstrate XGBoost through colab demonstration.

# Bagging and Boosting

Machine Learning Practice

Dr. Ashish Tendulkar

IIT Madras

# Contents

# Voting estimators

Class: `sklearn.ensemble.VotingClassifier`

Class: `sklearn.ensemble.VotingRegressor`

Both these estimators take the following **common parameters**:

`base_estimator` `weights`

Both these estimators implement the following **functions**:

`fit` `predict` `fit_transform` `score`

VotingClassifier takes an **additional argument**:

`voting` `hard` `soft`

# Bagging estimators

| | |
|---|---|
| Class: | sklearn.ensemble.BaggingClassifier |
| Class: | sklearn.ensemble.BaggingRegressor |

# Common parameters

| | | |
|---|---|---|
| base_estimator | default=None | base estimator to fit on random subsets of dataset |
| n_estimators | default=10 | number of base estimators in the ensemble |
| max_samples | default=1.0 | number of samples to draw from X to train each base estimator (**with** replacement by default) |
| max_features | default=1.0 | number of samples to draw from X to train each base estimator (**without** replacement by default) |
| bootstrap | default=True | Whether samples are drawn with replacement |

# Common parameters

| | | |
|---|---|---|
| bootstrap_features | default=False | Whether features are drawn with replacement |
| oob_score | default=False | Whether to use out-of-bag samples to estimate generalization error |

# Random forest estimators

Class: `sklearn.ensemble.RandomForestClassifier`

Class: `sklearn.ensemble.RandomForestRegressor`

The parameters can be classified as

- Bagging parameters
- Decision tree parameters

# Bagging parameters

- The number of trees are specified by `n_estimators`.
  - Default #trees for classification = 10
  - Default #trees for regression = 100
- `bootstrap` specifies whether to use bootstrap samples for training.
  - `True`: bootstrapped samples are used.
  - `False`: whole dataset is used.
- `oob_score` specifies whether to use out-of-bag samples for estimating generalization error. It is only available when `bootstrap` = `True`.

# Bagging parameters

- `max_samples` specifies the number of samples to be drawn while bootstrapping.
  - `None` : Use all samples in the training data.
  - `int` : Use `max_samples` samples from the training data.
  - `float` : Use `max_samples*total number of samples from training data`

    The value should be between 0 and 1.

- `random_state` controls randomness of features and samples selected during bootstrap.

- The number of features to be considered while splitting is specified by `max_features` .
  - `auto` , `sqrt` , `log2` , `int` , `float`

| **Value** | **max_features** |
|---|---|
| int | value specified |
| float | value * # features |
| auto | sqrt(#features) |
| sqrt | sqrt(#features) |
| log2 | log2(#features) |
| None | #features |

# Decision tree parameters

- The criteria for splitting the node is specified through `criterion`.
  - Default for classification: `gini`
  - Default for regression: `squared_error`

- The depth of the tree is controlled by `max_depth`. The default value is `None`, which means the tree will be grown until all leaf nodes are pure or until leaves contain less than `min_samples_splits` samples.

- We will continue to split the internal node until they contain `min_samples_splits` samples.
  - Whenever it is specified as an integer, then it is considered as a number.
  - Whenever it is specified as a float, and the `min_samples_splits` is calculated as $\texttt{min\_samples\_splits} \times n$.

- The tree growth can also be controlled by `min_impurity_decrease` parameter.
  - A node will be split if it reduces impurity at least by the value specified in this parameter.

- The complexity of tree can also be controlled by `ccp_alpha` parameter through minimal cost complexity pruning procedure.

# Trained random forest estimators

- `estimators_` member variable contains a collection of fitted estimators.

- `feature_importances_` member variable contains a list of important features.

# Training and inference for random forest

- `fit` builds forest of trees from the training dataset with the specified parameters.
- `decision_path` returns decision path in the forest.
- `predict` returns class label in classification and output value in regression.
- `predict_proba` and `predict_log_proba` returns probabilities and their logs for classification set up.