

IIT Madras
ONLINE DEGREE

Machine Learning Practice
Dr. Ashish Tendulkar
Indian Institute of Technology, Madras
Classification Functions of sci-kit Learn

(Refer Slide Time: 0:10)



Classification functions in sci-kit learn

Dr. Ashish Tendulkar

IIT Madras

Machine Learning Practice



Namaste. Welcome to the next video of machine learning practice course. In this video, we will study classification functions in sci-kit learn.

(Refer Slide Time: 0:21)



- 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

In this week, will study sklearn functionality for implementing classification algorithms. We will cover a sklearn API's for specific classification algorithms like least squares classification, perceptron and logistic regression classifier.

We will study these algorithms with regularization and we will also study how to extend these algorithms to multiclass, multilabel and multi-output settings. We will also cover sklearn API's for different 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 cross validation strategies that are specific to classification. And we will study those cross validation strategies in this slide deck.

(Refer Slide Time: 1:25)



Part I: sklearn API for classification

In this video, we will cover a sklearn API's for classification.

(Refer Slide Time: 1:34)



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

Generic	Specific
<ul style="list-style-type: none">• SGD classifier	<ul style="list-style-type: none">• Logistic regression• Perceptron• Ridge classifier (for LSC)• K-nearest neighbours (KNNs)• Support vector machines (SVMs)• Naive Bayes
Uses gradient descent for opt Need to specify loss function	Specialized solvers for opt

There are broadly two types of API's based on their functionality. One is generic and second is specific. SGD classifier is an example of a generic API whereas there are specific API's for logistic regression, perceptron and ridge classifier, which is used for implementing least square classification. There are also API specific API's available for K-nearest neighbours, support vector machines and Naive Bayes classifiers which will be studied in later weeks.

In case of generic KPIs, we use stochastic gradient descent for optimization whereas in specific API's, we use specialized solvers for optimization. The generic API is customized to specific API's by specifying the loss function and we will see exactly how this is done later in this video.

(Refer Slide Time: 2:44)



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

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

(Refer Slide Time: 2:58)



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.

Model training is done through the fit method that takes feature metrics and label as inputs along with some other optional arguments. Then there are a couple of methods for prediction. One is predict that takes feature metrics as an input and predicts class labels for the samples. And then, there is decision_function which also takes feature metrics as an input and predicts confidence score for samples.

Then there is a score method for evaluating the classifier that takes feature metrics and label as input. And there is an optional argument about sample rate. Score returns mean accuracy on the given test set and labels.

(Refer Slide Time: 3:57)



There are 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.

There are a few common miscellaneous methods like get params that gets the parameter for the estimator, set params that sets the parameter of the estimator then densify that converts coefficient matrix returns a format and specify which converts coefficient matrix to sparse format.

(Refer Slide Time: 4:17)



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



Now let us study how to implement different classifiers with the sklearn API's.

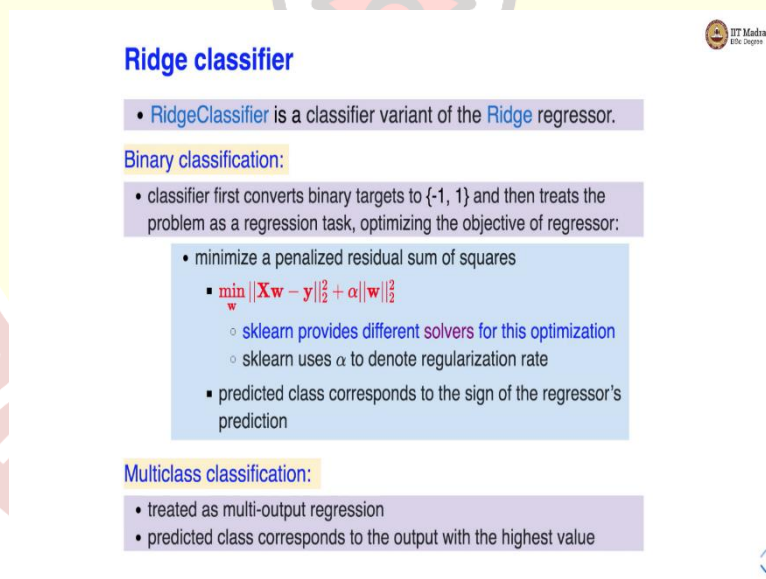
(Refer Slide Time: 4:25)



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

Let us start with implementation of least squares classification with ridge classifier API.

(Refer Slide Time: 4:34)



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 α to denote regularization rate
 - predicted class corresponds to the sign of the regressor's prediction

Multiclass classification:

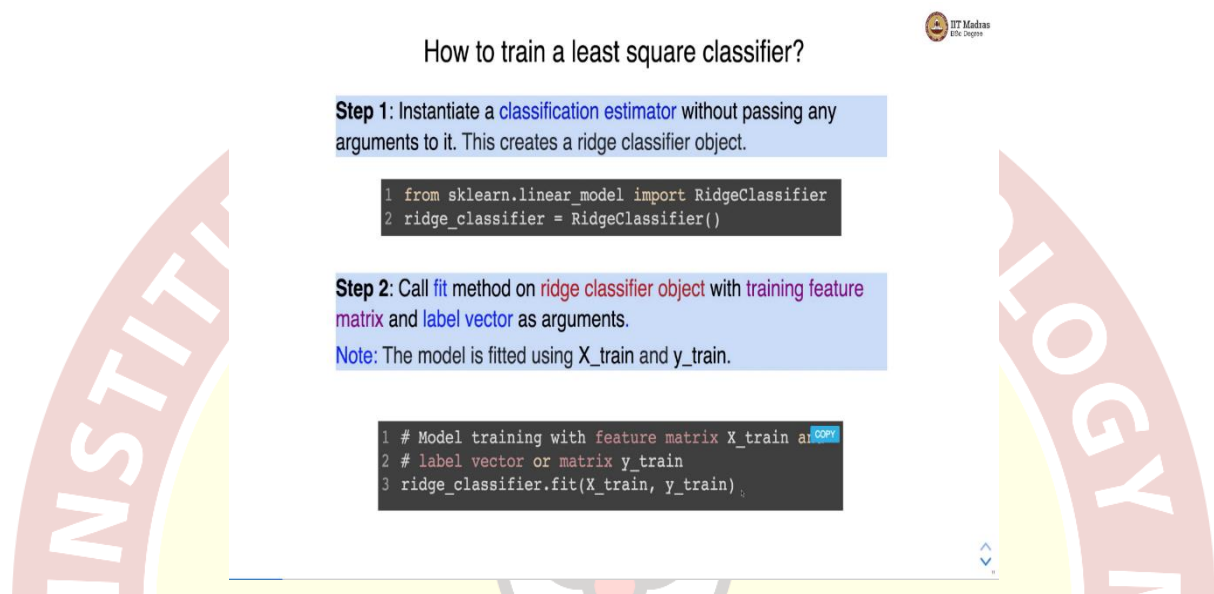
- treated as multi-output regression
- predicted class corresponds to the output with the highest value

Ridge classifier is a classifier variant of the ridge regressor. It can be used in binary and multi class classification. In case of binary classification, it first converts a binary target to -1 and +1 and then treats the problem as a regression task. It optimizes the objective of the regression. So, the ridge classifier minimizes a penalized residual sum of squares.

So, this is the predicted label, this actual label and we take square of the difference and we penalize it by let us say the second norm of the weight vector which is scaled by α which is the regularization rate. And we find out the weight vector \mathbf{w} which minimizes the whole expression. So, sklearn provides different solvers for this optimization.

The predicted class corresponds to the sign of regressors prediction. So, if regressor predicts a negative value, the predicted class will be -1. And if the regressor produces a regressor predicts positive value the predicted class would be + 1. Ridge classifier can also be applied in multiclass classification. It treats the multiclass classification problem as a multi-output regression problem. The predicted class corresponds to the output with the highest value.

(Refer Slide Time: 6:17)



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)
```

Let us see how to train a least square classifier. We first instantiate a classification estimator without passing any arguments to it which creates a basic ridge classifier object without any customization. This uses all the default parameters. So, ridge classifier can be imported from `sklearn.linear_model` module. So, we have ridge classifier object created in the second line.

Then we call the fit method on the ridge classifier object with training feature matrix and label vector as arguments. Again, in case of binary classification, it will be a label vector but in case of multi-class classification, it will be a labelled matrix. So, you simply call the fit on the ridge classifier object by passing the feature matrix `x_train` and label vector or matrix which is `y_train`.

So, the model is fitted with or we learn the parameters of the model by looking at the feature matrix and the labels supplied as part of the training.

(Refer Slide Time: 7:46)

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.

Let us see how to set regularization rate in sklearn. So, we use α to control the regularization rate. We need to set α to a float value and the default value for α is point 1. α should be positive and larger value of α specifies stronger regularization. So, we simply set the value of α in the ridge classifier constructor by setting the desired value for controlling the regularization rate.

(Refer Slide Time: 8:31)

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 <code>scipy.linalg.solve</code> function to obtain the closed-form solution
sparse_cg	uses the conjugate gradient solver of <code>scipy.sparse.linalg.cg</code> .
lsqr	uses the dedicated regularized least-squares routine <code>scipy.sparse.linalg.lsqr</code> 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 <code>scipy.optimize.minimize</code> . can be used only when coefficients are forced to be positive.

Let us see what are the options available to us for solving the optimization problem in ridge classification. We can use one of the seven solvers that are out here `svd`, `cholesky`, `sparse_cg` which is conjugate gradient, `lsqr`, `sag` or `saga` and `lbfgs`. `svd` uses a singular value decomposition of the feature matrix to compute the ridge coefficients. `cholesky` uses the `scipy.linalg.solve` function to obtain a closed form solution.

`sparse_cg` uses conjugate gradient solver of `scipy.sparse.linalg.cg`. `lsqr` uses dedicated regularized least square routine and `lsqr` is the fastest solver. `Sag` and `saga` uses stochastic average gradient descent iterative procedure. `Saga` is unbiased and more flexible version of `sag`.

lbfgs uses lbfgs implementation from scipy. It can be used only when coefficients are forced to be positive.

(Refer Slide Time: 10:03)



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.

We can specify different solvers depending on the problem at hand. For large scale data, we use sparse_cg solver. When both number of samples and number of features are large, we use sag or saga solver. The fast convergence is only guaranteed when features are approximately on the same scale.

(Refer Slide Time: 10:34)



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`.

So, there is a facility where we can make ridge classifier select the solver automatically. So, we have to set the solver argument to auto in the ridge classifier constructor and auto chooses solver automatically based on the type of the data. So, the default choice for solver is always auto.

(Refer Slide Time: 11:02)

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)
```

Is intercept estimation necessary for each classifier? This could be one of the question that might come to your mind. So, if the data is already centered, the intercept estimation is not necessary. In that case, we set **fit_intercept** argument in the constructor to false. By default, the **fit_intercept** argument is true, so we have to set it to false in case we do not want ridge classifier to perform estimation computation.

(Refer Slide Time: 11:39)

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.

Let us see how to make prediction on the new data. We use **predict** method to predict class labels for samples. We first arrange the data for prediction in a feature matrix of the shape number of samples from a number of features or in sparse matrix format. And then, we call the **predict** method on classifier object with feature matrix as an argument.

So, we already had created a ridge classifier object simply called the **predict** method on that by passing the feature matrix and we get the class labels as an output. So, other classifiers also use the same **predict** method.

(Refer Slide Time: 12:31)



`RidgeClassifierCV` implements
`RidgeClassifier` with built-in cross validation.

So, there is another variant of ridge classifier which is ridge classifier CV that implements ridge classifier with built in cross validation.

(Refer Slide Time: 12:43)



Let's implement `perceptron classifier` with
`Perceptron` API.

Now let us implement perceptron classifier with perceptron API.

(Refer Slide Time: 12:51)

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.

Perceptron classification is the simplest classification algorithm that is suitable for large scale learning. It shares the same underlying implementation with SGD classifier. So, we can also implement perceptron with SGD classifier by setting the loss to perceptron setting the initial value of the learning rate to 1 and learning rate=constant and without setting any penalty.

Perceptron uses a SGD for training. So, these two definitions or these two instantiations are equivalent, we can either use perceptron or we can use SGD classifier with this particular parameterization where we are setting loss to perceptron initial learning rate to 1, learning rate schedule to constant and penalty to none. Then this SGD classifier instance would train a perceptron model.

(Refer Slide Time: 14:03)

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)
```

So, we can implement a perceptron classifier by first importing perceptron from sklearn.linear module and then we instantiate a perceptron classifier. Then we call the fit method on the perceptron estimator object with training feature matrix and labels as argument and this is very similar to what we did in ridge classifier.

(Refer Slide Time: 14:31)

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)

So, perceptron can be further customized with the following parameters. We can set penalty to let us say, l_2 or l_1 . We can also set l_1_ratio . In case of elastic neck penalty. We can set the regularization rate α . We can also set the early stopping criteria. We can also control whether we want to fit the intercept we can specify the max iteration we can also specify a number of iterations with no change in order to set early stopping criteria and then we can also set tolerance and initial learning rate and validation fraction.

So, you may recall that most of the arguments are very similar to what arguments you used in a SGD regressor during linear regression module.

(Refer Slide Time: 15:37)

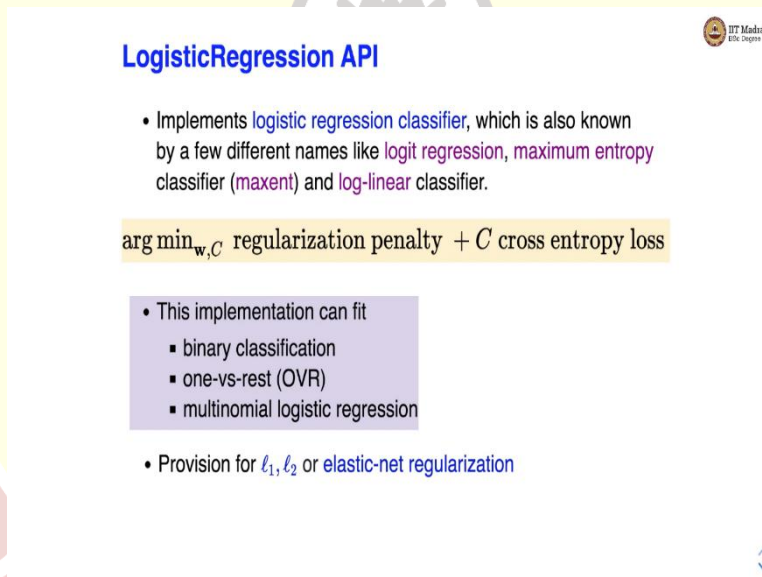
- Perceptron classifier can be trained in an **iterative manner** with **partial_fit** method
- Perceptron classifier can be initialized to the weights of the previous run by specifying **warm_start = True** in the constructor.

So, perceptron classifier can also 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. So, these two features help us to use perceptron classifier in large scale learning.

(Refer Slide Time: 16:02)

Let's implement logistic regression classifier
with `LogisticRegression` API.

Let us implement logistic regression classifier with logistic regression API.
(Refer Slide Time: 16:08)



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`.

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

- This implementation can fit
 - binary classification
 - one-vs-rest (OVR)
 - multinomial logistic regression
- Provision for ℓ_1 , ℓ_2 or `elastic-net regularization`

Logistic regression API implements logistic regression classifier and logistic regression classifier is also known by different names. There are names like logistic regression, maximum entropy classifier or maxent and log linear classifier. All of them refer to logistic regression. Logistic regression API basically optimizes the following expression.

So, it tries to find out the value of w and C that minimizes the regularization penalty + C times the cross entropy loss. So, this expression is slightly different than what we studied in machine learning techniques course about the logistic regression. Instead of applying λ which is regularization rate to the regularization penalty, it instead applies c or it instead scales the cross entropy loss.

So, this C over here is inverse of λ or the regularization rate. So, this particular logistic regression API can implement binary classification. Then it can also does multiclass classification with one versus straight setting and it can also implement multinomial logistic regression. It also has provision for ℓ_1 , ℓ_2 and elastic net regularization.

(Refer Slide Time: 17:57)

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)
```

So, how do we train a logistic regression classifier? We first instantiate a classifier or classification estimator without passing any arguments which creates a basic logistic regression classifier object. Then, we call the fit method with training feature matrix and label as argument. So, that helps us to train the logistic regression classifier.

(Refer Slide Time: 18:25)



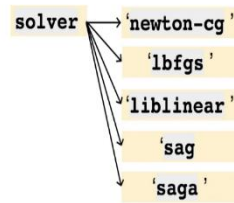
Logistic regression uses `specific algorithms` for solving the optimization problem in training. These algorithms are known as `solvers`.

The `choice` of the solver depends on the `classification problem set up` such as `size of the dataset`, `number of features` and `labels`.

So, logistic regression uses specific algorithms for solving the optimization problem in the training. These algorithms are known as solvers. The choice of solvers depends on the classification problem setup. It depends on the size of data set, number of features and labels.

(Refer Slide Time: 18:47)

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')
```

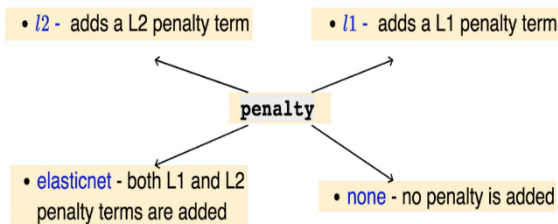
Let us look at different solver options for logistic regression classifier. There are five different solvers that we can use, namely, newton-cg and lbfgs, liblinear, sag and saga. These are basically different optimization algorithms. And as part of practice goals, you need to use how to use the solvers and also find out what are the suitable situations for using these solvers.

You are not expected to know the details of the optimization procedures behind each of the solvers. So, for a small data set liblinear is a good choice. Whereas sag and saga are faster for large datasets. For unscaled data set, liblinear and lbfgs or newton-cg are robust. And remember that sag and saga uses stochastic gradient descent which is sensitive to the feature scaling and hence they are not suitable for unscaled data sets.

If you want to solve the multi class classification problem with multinomial loss, then liblinear does not support that case. So, in order to use multinomial loss, you have to use newton-cg and lbfgs, sag or saga as the solvers. liblinear solver only supports the one-versus-rest scheme for multiclass classification.

By default, logistic regression uses lbfgs solver and you can specify the solver, the desired solver in the constructor of the logistic regression object.

How to add regularization in Logistic Regression classifier?



Regularization is applied by default because it improves numerical stability.

By default, it uses L2 penalty.

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

So, let us see how to add regularization in logistic regression classifier. Logistic Regression supports l_1 , l_2 and elastic net penalties. And if you do not want to use any regularization, we will have to set a penalty to none. So, regularization is applied by default because it improves the numerical stability. By default, logistic regression classifier uses l_2 penalty and we can change this penalty to desert penalty by setting the penalty argument in the constructor of logistic regression classifier.

(Refer Slide Time: 21:25)

- 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']

Again, note that not all solver supports all penalties. You have to select appropriate solver for the desired penalty. So, l_2 penalty is supported by all solvers whereas l_1 penalty is supported only by few solvers. So, we can see that element penalty is supported only by liblinear and saga. So, elasticnet penalty is only supported by saga. So, if you want to use elastic penalty, then you have to use saga, if you want to use l_1 penalty, then you have to use either liblinear or saga solvers. For l_2 , you can use any of the solvers all the solvers support into penalty.

(Refer Slide Time: 22:13)

How to control amount of regularization in logistic regression?

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

- Recall

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

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

Let us check out how to control the amount of regularization in logistic regression. So, as i said earlier, the regularization is controlled by a parameter C which is inverse of regularization rate. So, recall that logistic regression optimizes the following equation which is regularization penalty + C times the cross entropy loss.

It finds out w and c that minimizes this particular expression. So, the value of C is specified in the constructor and it must be positive. The smaller value of C leads to stronger regularization, whereas, larger value of C leads to weaker regularization. And remember that C is inverse of regularization rate. That is why strong value means stronger regularization and large value means weaker regularization.

(Refer Slide Time: 23:24)

LogisticRegression classifier has a **class_weight** parameter in its constructor.

What purpose does it serve? *

- Handles **class imbalance** 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.

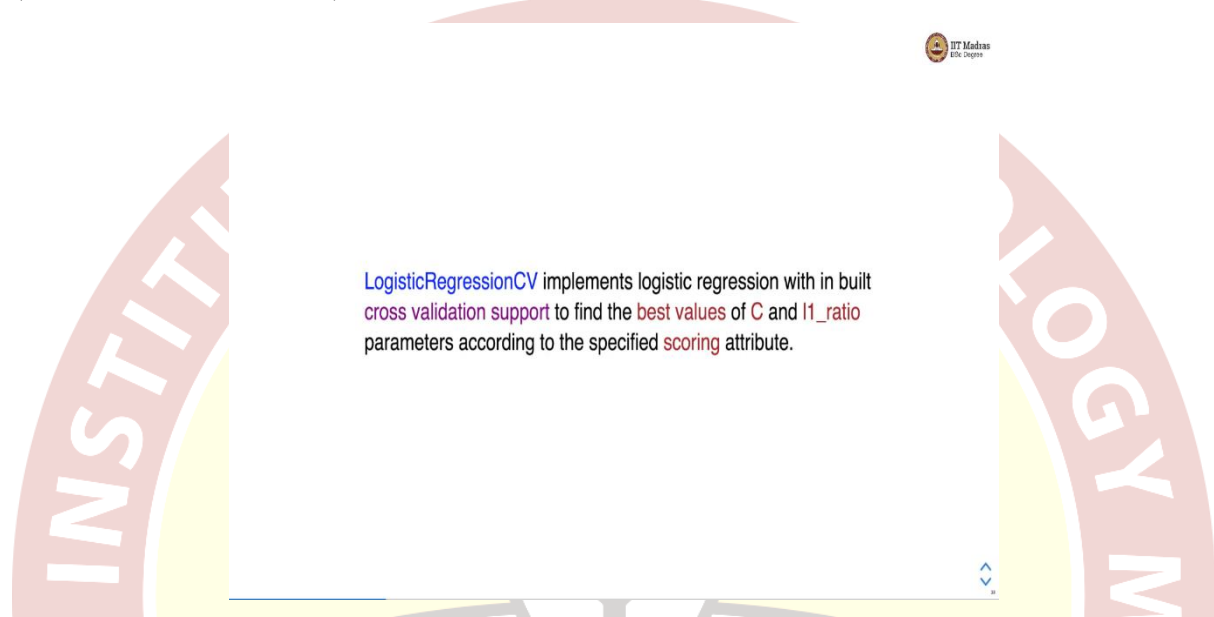
Exercise: Read [stack overflow discussion](#) on this parameter.

So, logistic regression classifier also has class_weight parameter in its constructor. So, what purpose does it serve? So, class_weight parameter helps us to handle the class imbalance with differential class rates. So, the mistakes in a class are penalized by the class rate. Higher value would mean higher emphasis on the class.

This parameter is also available in other classifiers in a sklearn. So, this is one way in which we can handle the class imbalance problem. In earlier weeks, we have also seen a couple of other ways to handle class imbalance either through sampling and through specialized method like smart.

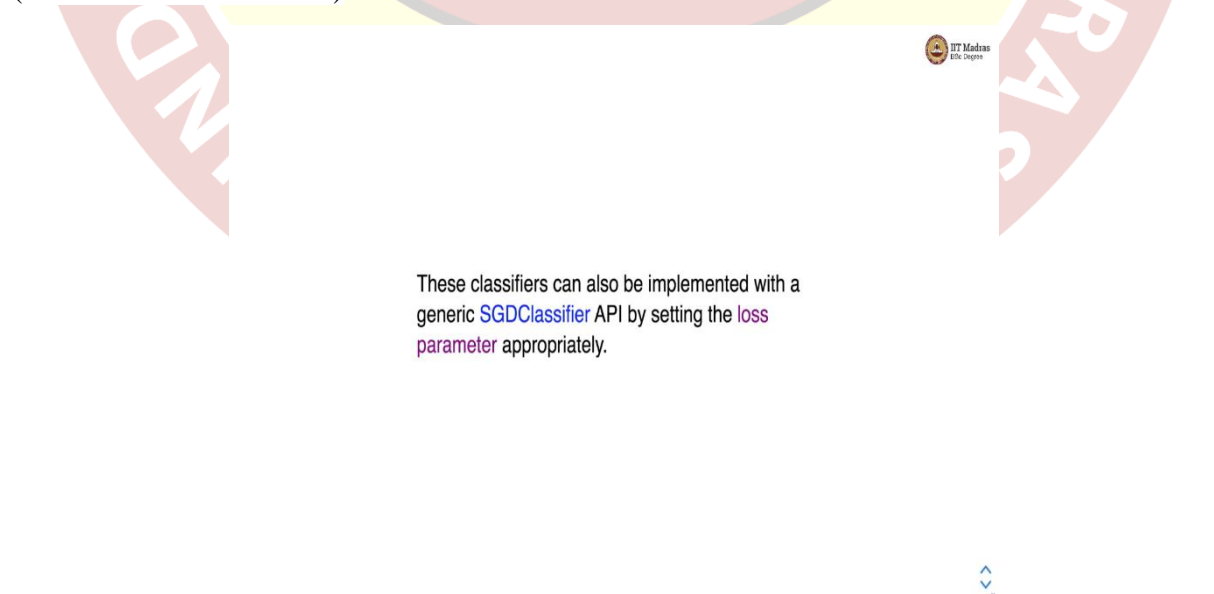
So, there is a very interesting Stack workflow discussion on this parameter and I would recommend you to read this discussion to understand the significance of this parameter in a better manner.

(Refer Slide Time: 24:31)



So, there is another variation of logistic regression which is logistic regression CV that implements logistic regression with built-in cross validation support. It helps us to find the best value of C and l1_ratio parameters according to specified scoring attributes. We will look at logistic regression CV in greater detail later in the module selection.

(Refer Slide Time: 24:58)



So now, up until this point, what we did is we basically looked at specific classifiers for least squares classification, for logistic regression and for perceptron. We can also implement these

classifiers with a generic SGD classifier API. And in order to implement a specific classifier, we need to set the last parameter of the SGD classifier appropriately.

(Refer Slide Time: 25:32)

SGDClassifier

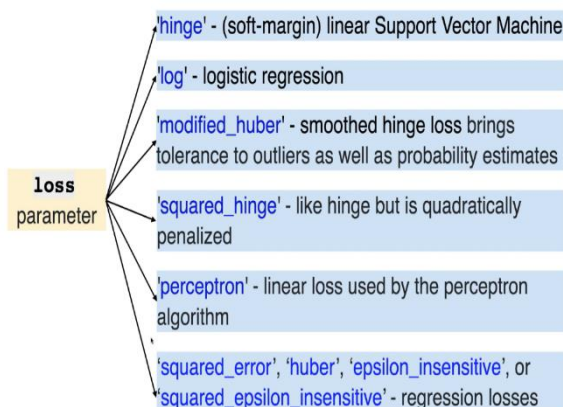
- **SGD** is a simple yet very efficient approach to fitting linear classifiers under convex loss functions
- This API uses SGD as an optimization technique and can be applied to build a variety of linear classifiers by adjusting the loss parameter.
- It supports multi-class classification by combining 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

Let us study SGD classifier API. SGD classifier uses SGD as an optimization technique. SGD is a simple yet very efficient approach to fitting linear classifiers under convex loss function. So, we can use this API to build a variety of linear classifiers by adjusting the loss parameters. SGD classifier easily scales up to large scale problems.

Problems with more than 10^5 training examples and 10^5 features can be easily handled with SGD classifier. It also works with sparse learning problems. It is useful for text classification and natural language processing. It also supports multiclass classification by combining multiple binary classifiers in one versus all schemes.

(Refer Slide Time: 26:32)

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



So, the way we can customize the SGD classifier is by setting the appropriate loss functions. Let us look at different loss parameters or different options for loss parameters. We can use hinge loss, log loss, modified huber laws, squared hinge loss, perceptron loss and squared error

loss. The hinge loss is used for linear support vector machine. The log loss when we said the loss log, we are effectively training a logistic regression model.

When we said the lost modified huber loss, we get smooth hinge loss that is tolerant to outliers as well as it gets its probability estimates in case of linear support vector machines. So, squared hinge loss it just like hinge loss, but it is quadratically penalized. Perceptron loss when we set loss parameter to perceptron, we are effectively training the perceptron algorithm.

When we said the loss to either squared error huber loss, epsilon insensitive loss or squared epsilon insensitive loss, we are basically using the SGD classifier to train the least square classification. So, these are all the losses that are used in regression and when we set the loss to one of these losses, we are effectively training the least square classification.

(Refer Slide Time: 28:17)

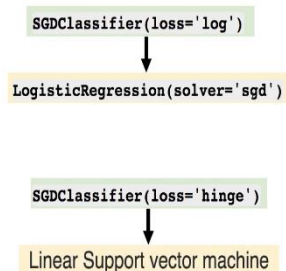


By default `SGDClassifier` uses `hinge loss` and hence trains `linear support vector machine classifier`.

By default, SGD classifier uses hinge loss and hence the trains linear support vector machine classifier.

(Refer Slide Time: 28:27)

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



Again, in other words, an instance of SGD classifier might have an equivalent estimator in sklearn API. For example, when we set the loss log, it is equivalent to logistic regression API with solver =SGD. When we set SGD classifier loss to hinge, you basically get a linear support vector machine.

(Refer Slide Time: 28:53)

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: | Disadvantages: |
|--|---|
| <ul style="list-style-type: none">• Efficiency.• Ease of implementation | <ul style="list-style-type: none">• 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](#).

Let us see how SGD classifier works. So, SGD classifier implements stochastic gradient descent procedure. The gradient of the loss is estimated with one sample at a time and the model is updated along the way with decreasing learning rate. SGD classifier is efficient and it is easy to implement.

On the flip side, it has a number of hyper parameters that we need to tune in order to get it working and SGD classifier is also sensitive to feature scaling. So, it is important to shuffle the training data before fitting the model and it is also important to standardize the features because SGD classifier works well with the standardized features or when all the features are on the same scale.

(Refer Slide Time: 29:49)

How to use `SGDClassifier` for training a classifier?

Step 1: Instantiate a `SGDClassifier` 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](#).

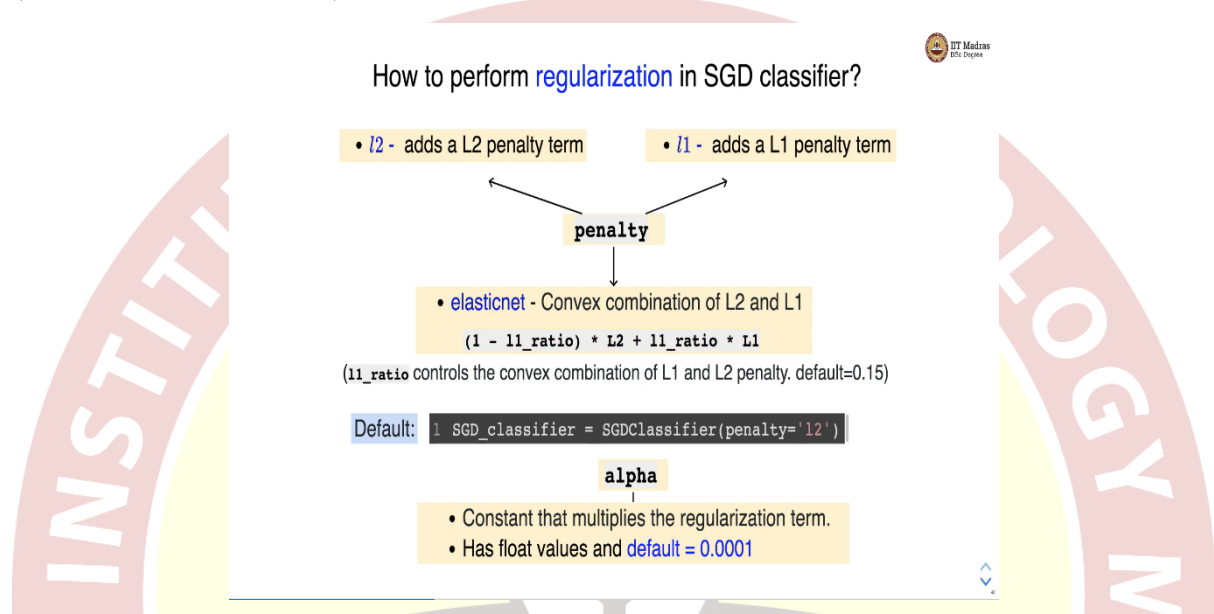
Step 2: Call `fit` method on `SGD classifier object` with [training feature matrix](#) and [label vector](#) as arguments.

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


So, we can instantiate SGD classifier estimator by setting appropriate loss parameter to define the classifier of interest. By default, it uses hinge loss that is used to train linear support vector machine model. If you want to use SGD classifier to train let us say logistic regression, we have to set the loss to log and the log loss defines the logistic regression classifier.

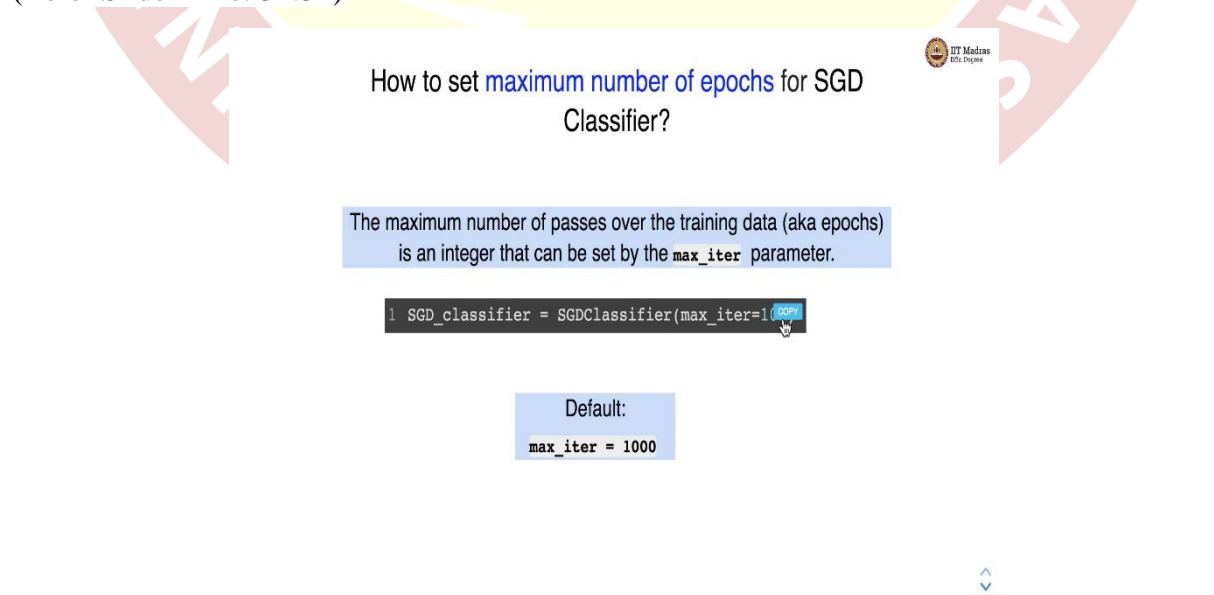
We call the fit method on SGD classifier object just like any other classification, just like any other classifier estimators. We sent the training feature matrix and label vectors or matrix as arguments.

(Refer Slide Time: 30:38)



We can also perform regularization in SGD classifier, we can use either l_2 , l_1 or elasticnet regularization. We have already seen elasticnet regularization, while we were studying linear regression module. So, in case of elastic net regularization, l_1_ratio controls the convex combination of l_1 and l_2 penalty, the default value of l_1_ratio is 0.15. So, by default, SGD classifier uses l_2 penalty. We can control the amount of penalty through parameter α . And α is a float value and the default is 10^{-4} .

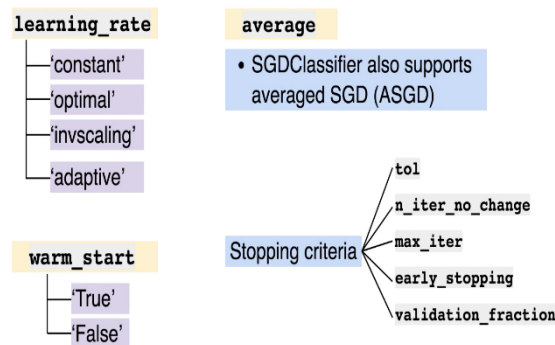
(Refer Slide Time: 31:34)



Let us see how to set the maximum number of epochs for SGD classifier just like in SGD regressor we can use max iter parameter here. The default number of iteration are 1000, we can set max iter to any number of iterations that we desire the SGD classifier to train.

(Refer Slide Time: 32:02)

Some common parameters between SGDClassifier and SGDRegressor



There are some parameters that are common between SGD classifier and regressor. There are parameters like learning rate, then average SGD, warm start and stopping criteria. These parameters are exactly like how we used initially regressor. If you want to know more about these parameters, I would recommend you to look at to have a relook at the SGD regressor video in the linear regression module.

We can set the learning rate to constant optimal inverse scaling or adaptive. We can also set the average parameter to true or false that helps us to inform the SGD classifier whether you want to use average SGD. We can set warm start to true or false. Warm start helps us to initialize SGD with the parameters learned in the previous iteration.

And there are a bunch of parameters that helps us to specify the stopping criteria. The parameters like tolerance n_iter_no_change, max_iter or early stopping and validation traction all these parameters help us to specify the stopping criteria.

(Refer Slide Time: 33:40)



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](#).

In this video, we learned how to implement different classifiers with sklearn API's. We learned how to implement least squares classifier, perceptron classifier and logistic regression classifier with specific API's like ridge classifier, perceptron and logistic regression. We can also implement these classifiers with a generic SGD classifier by appropriately setting the loss parameter.

For example, we can implement a logistic regression classification model with the `loss=log`. We can implement perceptron classifier by setting the `loss=perceptron` and we can learn least square classification model by setting the loss to `squared_error`. We also learned that classification estimators implement a few common methods.

(Refer Slide Time: 34:39)



- 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.

These estimators can also be readily used in multiclass setting. They support regularized loss function optimization. And finally, all classification estimators have the ability to deal with class imbalance through `class_weight` parameter in the constructor. In the next video we will

implement or we will extend these estimators in multiclass setting. So, now you know how to implement different classifiers with sklearn API's

