Machine Learning using Python : Implementation of Random Forest, SVM and MLP classifiers

Neelotpal Chakraborty

Department of Computer Science and Engineering

Jadavpur University

Classification: Random Forest

```
import pandas as pd
import numpy as np

# Dataset Preparation
dataset = pd.read_csv("E:/JU ML LAB/ML using Python/Datasets/wine.data.csv")

X = dataset.drop(['Class'], axis=1)
y = dataset['Class']

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.20,random_state=0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# Classification

from sklearn.ensemble import RandomForestClassifier

classifier = RandomForestClassifier(n_estimators=20,random_state=0)
classifier.fit(X_train,y_train)

y_pred = classifier.predict(X_test)
```

Classification: Random Forest

```python
# Evaluation of Classifier Performance

from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("-------------------------------------------------------")
print("-------------------------------------------------------")

print("Performance Evaluation:")
print(classification_report(y_test, y_pred))

print("-------------------------------------------------------")
print("-------------------------------------------------------")

print("Accuracy:")
print(accuracy_score(y_test, y_pred))
```

OUTPUT:

```
Confusion Matrix:
[[14  0  0]
 [ 2 13  1]
 [ 0  0  6]]
-------------------------------------------------------
-------------------------------------------------------
Performance Evaluation:
              precision    recall  f1-score   support

           1       0.88      1.00      0.93        14
           2       1.00      0.81      0.90        16
           3       0.86      1.00      0.92         6

    accuracy                           0.92        36
   macro avg       0.91      0.94      0.92        36
weighted avg       0.93      0.92      0.92        36

-------------------------------------------------------
-------------------------------------------------------
Accuracy:
0.9166666666666666
```

Random Forest Parameters

sklearn.ensemble.RandomForestClassifier

class `sklearn.ensemble.RandomForestClassifier`(*n_estimators=100, *, criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False, class_weight=None, ccp_alpha=0.0, max_samples=None*) ¶ [source]

Random Forest Parameters

Parameters:

n_estimators : *int, default=100*

The number of trees in the forest.

criterion : *{"gini", "entropy"}, default="gini"*

The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Note: this parameter is tree-specific.

max_depth : *int, default=None*

The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split : *int or float, default=2*

The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Random Forest Parameters

min_samples_leaf : *int or float, default=1*

The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

min_weight_fraction_leaf : *float, default=0.0*

The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features : *{"auto", "sqrt", "log2"}, int or float, default="auto"*

The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `round(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=sqrt(n_features)`.
- If "sqrt", then `max_features=sqrt(n_features)` (same as "auto").
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

Random Forest Parameters

max_leaf_nodes : *int, default=None*

Grow trees with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease : *float, default=0.0*

A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child.

`N`, `N_t`, `N_t_R` and `N_t_L` all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split : *float, default=None*

Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Random Forest Parameters

bootstrap : *bool, default=True*

Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.

oob_score : *bool, default=False*

Whether to use out-of-bag samples to estimate the generalization score. Only available if bootstrap=True.

n_jobs : *int, default=None*

The number of jobs to run in parallel. `fit`, `predict`, `decision_path` and `apply` are all parallelized over the trees. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details.

random_state : *int, RandomState instance or None, default=None*

Controls both the randomness of the bootstrapping of the samples used when building trees (if `bootstrap=True`) and the sampling of the features to consider when looking for the best split at each node (if `max_features < n_features`). See Glossary for details.

verbose : *int, default=0*

Controls the verbosity when fitting and predicting.

warm_start : *bool, default=False*

When set to `True`, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See the Glossary.

Random Forest Parameters

class_weight : *{"balanced", "balanced_subsample"}, dict or list of dicts, default=None*

Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}].

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

The "balanced_subsample" mode is the same as "balanced" except that weights are computed based on the bootstrap sample for every tree grown.

For multi-output, the weights of each column of y will be multiplied.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

ccp_alpha : *non-negative float, default=0.0*

Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than `ccp_alpha` will be chosen. By default, no pruning is performed. See Minimal Cost-Complexity Pruning for details.

Random Forest Parameters

max_samples : *int or float, default=None*

If bootstrap is True, the number of samples to draw from X to train each base estimator.

- If None (default), then draw `X.shape[0]` samples.
- If int, then draw `max_samples` samples.
- If float, then draw `max_samples * X.shape[0]` samples. Thus, `max_samples` should be in the interval `(0, 1)`.

Classification: Support Vector Machine (SVM)

```
#SVM for Classification

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Dataset Preparation
dataset = pd.read_csv("E:/JU ML LAB/ML using Python/Datasets/Mall_Customers.csv")
X = dataset.drop(['CustomerID','Gender'], axis=1)
y = dataset['Gender']

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20)

# Classification using simple SVM

from sklearn.svm import SVC
classifier = SVC(kernel='linear')
classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)

# Evaluation of Classifier Performance

from sklearn.metrics import classification_report, confusion_matrix
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("-----------------------------------------------------")
print("-----------------------------------------------------")

print("Performance Evaluation:")
print(classification_report(y_test, y_pred))
```

OUTPUT:

```
Confusion Matrix:
[[22  0]
 [18  0]]
------------------------------------------------------
------------------------------------------------------
Performance Evaluation:
              precision    recall  f1-score   support

      Female       0.55      1.00      0.71        22
        Male       0.00      0.00      0.00        18

    accuracy                           0.55        40
   macro avg       0.28      0.50      0.35        40
weighted avg       0.30      0.55      0.39        40
```

Classification: Support Vector Machine (SVM)

```
# Classification using simple SVM

from sklearn.svm import SVC
classifier = SVC(kernel='poly', degree=3)
classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)
```

OUTPUT:

```
Confusion Matrix:
[[23  1]
 [12  4]]
------------------------------------------------------
------------------------------------------------------
Performance Evaluation:
              precision    recall  f1-score   support

      Female       0.66      0.96      0.78        24
        Male       0.80      0.25      0.38        16

    accuracy                           0.68        40
   macro avg       0.73      0.60      0.58        40
weighted avg       0.71      0.68      0.62        40
```

Classification: Support Vector Machine (SVM)

```
# Classification using simple SVM

from sklearn.svm import SVC
classifier = SVC(kernel='poly', degree=2)
classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)
```

OUTPUT:

```
Confusion Matrix:
[[21  1]
 [17  1]]
-----------------------------------------------------
-----------------------------------------------------
Performance Evaluation:
              precision    recall  f1-score   support

      Female       0.55      0.95      0.70        22
        Male       0.50      0.06      0.10        18

    accuracy                           0.55        40
   macro avg       0.53      0.51      0.40        40
weighted avg       0.53      0.55      0.43        40
```

Classification: Support Vector Machine (SVM)

```
# Classification using SVM

from sklearn.svm import SVC
classifier = SVC(kernel='rbf')#Gaussian kernel
classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)
```

OUTPUT:

```
Confusion Matrix:
[[22  2]
 [11  5]]
----------------------------------------------------------
----------------------------------------------------------
Performance Evaluation:
              precision    recall  f1-score   support

      Female       0.67      0.92      0.77        24
        Male       0.71      0.31      0.43        16

    accuracy                           0.68        40
   macro avg       0.69      0.61      0.60        40
weighted avg       0.69      0.68      0.64        40
```

Classification: Support Vector Machine (SVM)

```
# Classification using SVM

from sklearn.svm import SVC
classifier = SVC(kernel='sigmoid')#Sigmoid kernel
classifier.fit(X_train, y_train)
```

OUTPUT:

```
Confusion Matrix:
[[28  0]
 [12  0]]
--------------------------------------------------
--------------------------------------------------
Performance Evaluation:
              precision    recall  f1-score   support

      Female       0.70      1.00      0.82        28
        Male       0.00      0.00      0.00        12

    accuracy                           0.70        40
   macro avg       0.35      0.50      0.41        40
weighted avg       0.49      0.70      0.58        40
```

SVM Parameters

sklearn.svm.SVC¶

class `sklearn.svm.SVC`(**, C=1.0, kernel='rbf', degree=3, gamma='scale', coef0=0.0, shrinking=True, probability=False, tol=0.001, cache_size=200, class_weight=None, verbose=False, max_iter=- 1, decision_function_shape='ovr', break_ties=False, random_state=None*) [source]

SVM Parameters

C : *float, default=1.0*

Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty.

kernel : *{'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'}, default='rbf'*

Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape `(n_samples, n_samples)`.

degree : *int, default=3*

Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma : *{'scale', 'auto'} or float, default='scale'*

Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

- if `gamma='scale'` (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma,
- if 'auto', uses 1 / n_features.

Changed in version 0.22: The default value of `gamma` changed from 'auto' to 'scale'.

SVM Parameters

coef0 : *float, default=0.0*

Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

shrinking : *bool, default=True*

Whether to use the shrinking heuristic. See the User Guide.

probability : *bool, default=False*

Whether to enable probability estimates. This must be enabled prior to calling `fit`, will slow down that method as it internally uses 5-fold cross-validation, and `predict_proba` may be inconsistent with `predict`. Read more in the User Guide.

tol : *float, default=1e-3*

Tolerance for stopping criterion.

cache_size : *float, default=200*

Specify the size of the kernel cache (in MB).

class_weight : *dict or 'balanced', default=None*

Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

verbose : *bool, default=False*

Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

SVM Parameters

max_iter : *int, default=-1*

Hard limit on iterations within solver, or -1 for no limit.

decision_function_shape : *{'ovo', 'ovr'}, default='ovr'*

Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2). However, one-vs-one ('ovo') is always used as multi-class strategy. The parameter is ignored for binary classification.

> *Changed in version 0.19:* decision_function_shape is 'ovr' by default.

New in version 0.17: decision_function_shape='ovr' is recommended.

> *Changed in version 0.17:* Deprecated *decision_function_shape='ovo' and None.*

break_ties : *bool, default=False*

If true, `decision_function_shape='ovr'`, and number of classes > 2, predict will break ties according to the confidence values of decision_function; otherwise the first class among the tied classes is returned. Please note that breaking ties comes at a relatively high computational cost compared to a simple predict.

New in version 0.22.

random_state : *int, RandomState instance or None, default=None*

Controls the pseudo random number generation for shuffling the data for probability estimates. Ignored when `probability` is False. Pass an int for reproducible output across multiple function calls. See Glossary.

Visualizing Performance Measures

```
# Classification using SVM

from sklearn.svm import SVC
classifier = SVC(kernel='linear',random_state=10,max_iter=5)
classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)

# Evaluation of Classifier Performance

from sklearn.metrics import classification_report, confusion_matrix
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("-----------------------------------------------------")
print("-----------------------------------------------------")

print("Performance Evaluation:")
print(classification_report(y_test, y_pred))
```

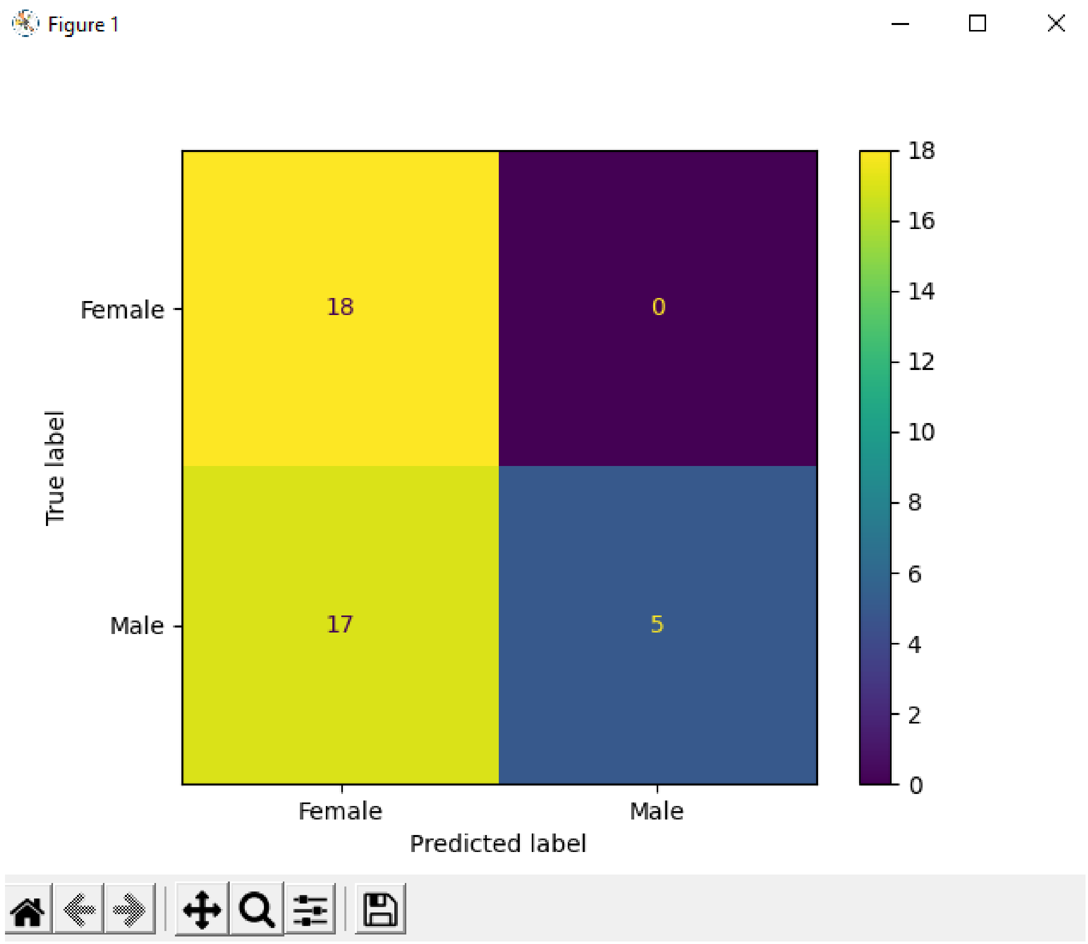
OUTPUT

```
Confusion Matrix:
[[18  0]
 [17  5]]
-----------------------------------------------------
-----------------------------------------------------
Performance Evaluation:
              precision    recall  f1-score   support

      Female       0.51      1.00      0.68        18
        Male       1.00      0.23      0.37        22

    accuracy                           0.57        40
   macro avg       0.76      0.61      0.52        40
weighted avg       0.78      0.57      0.51        40
```

Visualizing Performance Measures

```
from sklearn.metrics import plot_confusion_matrix
plot_confusion_matrix(classifier, X_test, y_test)
plt.show()
```

OUTPUT



Classification: MLP

```python
#MLP for Classification

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Dataset Preparation
dataset = pd.read_csv("E:/JU ML LAB/ML using Python/Datasets/Mall_Customers.csv")
X = dataset.drop(['CustomerID','Gender'], axis=1)
y = dataset['Gender']

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20)

# Classification using MLP

from sklearn.neural_network import MLPClassifier
classifier = MLPClassifier(hidden_layer_sizes=(10, 10, 10), max_iter=1000)
classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)

# Evaluation of Classifier Performance

from sklearn.metrics import classification_report, confusion_matrix
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("-----------------------------------------------------")
print("-----------------------------------------------------")

print("Performance Evaluation:")
print(classification_report(y_test, y_pred))
```

Output:

```
Confusion Matrix:
[[21  7]
 [ 6  6]]
-----------------------------------------------------
-----------------------------------------------------
Performance Evaluation:
              precision    recall  f1-score   support

      Female       0.78      0.75      0.76        28
        Male       0.46      0.50      0.48        12

    accuracy                           0.68        40
   macro avg       0.62      0.62      0.62        40
weighted avg       0.68      0.68      0.68        40
```

Classification: Parameter Tuning for MLP

sklearn.neural_network.MLPClassifier

class `sklearn.neural_network.MLPClassifier`(*hidden_layer_sizes=100, activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000*)

[source]

Classification: Parameter Tuning for MLP

hidden_layer_sizes : *tuple, length = n_layers - 2, default=(100,)*

The ith element represents the number of neurons in the ith hidden layer.

activation : *{'identity', 'logistic', 'tanh', 'relu'}, default='relu'*

Activation function for the hidden layer.

- 'identity', no-op activation, useful to implement linear bottleneck, returns f(x) = x
- 'logistic', the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)).
- 'tanh', the hyperbolic tan function, returns f(x) = tanh(x).
- 'relu', the rectified linear unit function, returns f(x) = max(0, x)

solver : *{'lbfgs', 'sgd', 'adam'}, default='adam'*

The solver for weight optimization.

- 'lbfgs' is an optimizer in the family of quasi-Newton methods.
- 'sgd' refers to stochastic gradient descent.
- 'adam' refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba

Note: The default solver 'adam' works pretty well on relatively large datasets (with thousands of training samples or more) in terms of both training time and validation score. For small datasets, however, 'lbfgs' can converge faster and perform better.

Classification: Parameter Tuning for MLP

alpha : *float, default=0.0001*

L2 penalty (regularization term) parameter.

batch_size : *int, default='auto'*

Size of minibatches for stochastic optimizers. If the solver is 'lbfgs', the classifier will not use minibatch. When set to "auto", `batch_size=min(200, n_samples)`

learning_rate : *{'constant', 'invscaling', 'adaptive'}, default='constant'*

Learning rate schedule for weight updates.

- 'constant' is a constant learning rate given by 'learning_rate_init'.
- 'invscaling' gradually decreases the learning rate at each time step 't' using an inverse scaling exponent of 'power_t'. effective_learning_rate = learning_rate_init / pow(t, power_t)
- 'adaptive' keeps the learning rate constant to 'learning_rate_init' as long as training loss keeps decreasing. Each time two consecutive epochs fail to decrease training loss by at least tol, or fail to increase validation score by at least tol if 'early_stopping' is on, the current learning rate is divided by 5.

Only used when `solver='sgd'`.

learning_rate_init : *double, default=0.001*

The initial learning rate used. It controls the step-size in updating the weights. Only used when solver='sgd' or 'adam'.

power_t : *double, default=0.5*

The exponent for inverse scaling learning rate. It is used in updating effective learning rate when the learning_rate is set to 'invscaling'. Only used when solver='sgd'.

Classification: Parameter Tuning for MLP

max_iter : *int, default=200*

Maximum number of iterations. The solver iterates until convergence (determined by 'tol') or this number of iterations. For stochastic solvers ('sgd', 'adam'), note that this determines the number of epochs (how many times each data point will be used), not the number of gradient steps.

shuffle : *bool, default=True*

Whether to shuffle samples in each iteration. Only used when solver='sgd' or 'adam'.

random_state : *int, RandomState instance, default=None*

Determines random number generation for weights and bias initialization, train-test split if early stopping is used, and batch sampling when solver='sgd' or 'adam'. Pass an int for reproducible results across multiple function calls. See Glossary.

tol : *float, default=1e-4*

Tolerance for the optimization. When the loss or score is not improving by at least `tol` for `n_iter_no_change` consecutive iterations, unless `learning_rate` is set to 'adaptive', convergence is considered to be reached and training stops.

verbose : *bool, default=False*

Whether to print progress messages to stdout.

warm_start : *bool, default=False*

When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See the Glossary.

Classification: Parameter Tuning for MLP

momentum : *float, default=0.9*

Momentum for gradient descent update. Should be between 0 and 1. Only used when solver='sgd'.

nesterovs_momentum : *bool, default=True*

Whether to use Nesterov's momentum. Only used when solver='sgd' and momentum > 0.

early_stopping : *bool, default=False*

Whether to use early stopping to terminate training when validation score is not improving. If set to true, it will automatically set aside 10% of training data as validation and terminate training when validation score is not improving by at least tol for `n_iter_no_change` consecutive epochs. The split is stratified, except in a multilabel setting. If early stopping is False, then the training stops when the training loss does not improve by more than tol for n_iter_no_change consecutive passes over the training set. Only effective when solver='sgd' or 'adam'

validation_fraction : *float, default=0.1*

The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True

beta_1 : *float, default=0.9*

Exponential decay rate for estimates of first moment vector in adam, should be in [0, 1). Only used when solver='adam'

beta_2 : *float, default=0.999*

Exponential decay rate for estimates of second moment vector in adam, should be in [0, 1). Only used when solver='adam'

epsilon : *float, default=1e-8*

Value for numerical stability in adam. Only used when solver='adam'

Classification: Parameter Tuning for MLP

n_iter_no_change : *int, default=10*

Maximum number of epochs to not meet `tol` improvement. Only effective when solver='sgd' or 'adam'

New in version 0.20.

max_fun : *int, default=15000*

Only used when solver='lbfgs'. Maximum number of loss function calls. The solver iterates until convergence (determined by 'tol'), number of iterations reaches max_iter, or this number of loss function calls. Note that number of loss function calls will be greater than or equal to the number of iterations for the `MLPClassifier`.

For more details visit the link below:

https://scikit-learn.org/