1) Logistic Regression

Definition:

i) Logistic regression is a machine learning algorithm for classification. In this algorithm, the probabilities describing the possible outcomes of a single trial are modelled using a logistic function.

ii) The logistic function, also called the sigmoid function was developed by statisticians to describe properties of population growth in ecology, rising quickly and maxing out at the carrying capacity of the environment. It's an S-shaped curve that can take any real-valued number and map it into a value between 0 and 1, but never exactly at those limits.

A logistic function or logistic curve is a common "S" shape (sigmoid curve), with equation:

1 / (1 + e^-value)

Advantages:

Logistic regression is designed for this purpose (classification), and is most useful for understanding the influence of several independent variables on a single outcome variable.

Disadvantages:

Works only when the predicted variable is binary, assumes all predictors are independent of each other, and assumes data is free of missing values.

Example:

To predict whether an email is spam (1) or (0)

Whether the tumor is malignant (1) or not (0)

Syntax:

```
from sklearn.linear_model import LogisticRegression

model =
LogisticRegression(penalty='l2',tol=0.0001,C=1.0,random_state=None,max_iter=100,n_jobs=None)

model.fit(X_train,Y_train,)

predictions = mode.predict(X_test)
```

Parameters:

penalty :'l1' or 'l2', default: 'l2'

- L1 - Lasso Regression(sum of absolute weights)
- L2 (default) - Ridge Regression(SSE) - helps to solve overfitting.

tol : float, default: 1e-4

- Tolerance for stopping criteria.

C: float, default: 1.0

- Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.

random_state : int, RandomState instance or None, optional, default: None

max_iter : int, default: 100

- Maximum number of iterations taken for the solvers to converge.

n_jobs : int or None, optional (default=None)

- Number of CPU cores used when parallelizing over classes.

Methods:

decision_function(X) - Predict confidence scores for samples.

fit(X, y[, sample_weight]) - Fit the model according to the given training data.

get_params([deep]) - Get parameters for this estimator.

predict(X) - Predict class labels for samples in X.

predict_log_proba(X) - Log of probability estimates.

predict_proba(X) - Probability estimates.

score(X, y[, sample_weight]) - Returns the mean accuracy on the given test data and labels.

2) Naive Bayes

Definition:

i)Naive Bayes algorithm based on Bayes' theorem with the assumption of independence between every pair of features. Naive Bayes classifiers work well in many real-world situations such as document classification and spam filtering.

ii)Bayes' Theorem is a way of finding a probability when we know certain other probabilities.

The formula is:

P(A|B) = P(A)*P(B|A) / P(B)

Which tells us: how often A happens given that B happens, written P(A|B),

When we know: how often B happens given that A happens, written P(B|A)

and how likely A is on its own, written P(A)

and how likely B is on its own, written P(B)

Advantages:

This algorithm requires a small amount of training data to estimate the necessary parameters. Naive Bayes classifiers are extremely fast compared to more sophisticated methods.

Disadvantages:

Naive Bayes is is known to be a bad estimator.

Example:

Document Classification

Spam Filtering

Syntax:

```
from sklearn.naive_bayes import GaussianNB

model = GaussianNB(priors=None, var_smoothing=1e-09)

model.fit(X_train,Y_train,)

predictions = mode.predict(X_test)
```

Parameters:

priors : array-like, shape (n_classes)

- Prior probabilities of the classes. If specified the priors are not adjusted according to the data.

var_smoothing : float, optional (default=1e-9)

- Portion of the largest variance of all features that is added to variances for calculation stability

Methods:

fit(X, y[, sample_weight]) - Fit Gaussian Naive Bayes according to X, y

get_params([deep]) - Get parameters for this estimator.

partial_fit(X, y[, classes, sample_weight]) - Incremental fit on a batch of samples.

predict(X) - Perform classification on an array of test vectors X.

predict_log_proba(X) - Return log-probability estimates for the test vector X.

predict_proba(X) - Return probability estimates for the test vector X.

score(X, y[, sample_weight]) - Returns the mean accuracy on the given test data and labels.

3) Stochastic Gradient Descent

Definition:

Stochastic gradient descent is a simple and very efficient approach to fit linear models. It is particularly useful when the number of samples is very large. It supports different loss functions and penalties for classification.

Advantages:

Efficiency and ease of implementation.

Disadvantages:

Requires a number of hyper-parameters and it is sensitive to feature scaling.

Example:

Same as logistic Regression but it is more efficient.

Syntax:

```
from sklearn.linear_model import SGDClassifier

model = SGDClassifier(loss='hinge', penalty='l2', l1_ratio=0.15, tol=None, shuffle=True, random_state=None, learning_rate='optimal')

model.fit(X_train,Y_train,)

predictions = mode.predict(X_test)
```

Parameters:

loss : str, default: 'hinge'

- The loss function to be used. Defaults to 'hinge', which gives a linear SVM.
- The possible options are 'hinge', 'log', 'modified_huber', 'squared_hinge', 'perceptron', or a regression loss: 'squared_loss', 'huber', 'epsilon_insensitive', or 'squared_epsilon_insensitive'.
- 'log' loss gives logistic regression, a probabilistic classifier
- 'modified_huber' is another smooth loss that brings tolerance to outliers as well as probability estimates
- 'squared_hinge' is like hinge but is quadratically penalized
- 'perceptron' is the linear loss used by the perceptron algorithm.
- The other losses are designed for regression but can be useful in classification as well.

learning_rate : string, optional

- The learning rate schedule:
- 'constant':eta = eta0
- 'optimal': [default]
- eta = 1.0 / (alpha * (t + t0)) where t0 is chosen by a heuristic proposed by Leon Bottou.
- 'invscaling':eta = eta0 / pow(t, power_t)
- 'adaptive': eta = eta0, as long as the training keeps decreasing. Each time n_iter_no_change consecutive epochs fail to decrease the training loss by tol or fail to increase validation score by tol if early_stopping is True, the current learning rate is divided by 5.

Methods:

decision_function(X) - Predict confidence scores for samples.

fit(X, y[, coef_init, intercept_init, ...]) - Fit linear model with Stochastic Gradient Descent.

get_params([deep]) - Get parameters for this estimator.

partial_fit(X, y[, classes, sample_weight]) - Perform one epoch of stochastic gradient descent on given samples.

predict(X) - Predict class labels for samples in X.

score(X, y[, sample_weight]) - Returns the mean accuracy on the given test data and labels.

4) K-Nearest Neighbours

Definition:

Neighbours based classification is a type of lazy learning as it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the k nearest neighbours of each point.

Advantages:

This algorithm is simple to implement, robust to noisy training data, and effective if training data is large.

Disadvantages:

Need to determine the value of K and the computation cost is high as it needs to computer the distance of each instance to all the training samples.

Syntax:

from sklearn.linear_model import SGDClassifier

model = KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None)

model.fit(X_train,Y_train,)

predictions = mode.predict(X_test)

Parameters:

n_neighbors : int, optional (default = 5)

- Number of neighbors to use by default for kneighbors queries.
- weights : str or callable, optional (default = 'uniform')
    1. weight function used in prediction. Possible values:
    2. 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
    3. 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
    4. [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.
- algorithm : {'auto', 'ball_tree', 'kd_tree', 'brute'}, optional - Algorithm used to compute the nearest neighbors:
    1. 'ball_tree' will use BallTree
    2. 'kd_tree' will use KDTree
    3. 'brute' will use a brute-force search.
    4. 'auto' will attempt to decide the most appropriate algorithm based on the values passed to fit method

- leaf_size : int, optional (default = 30)
- Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p : integer, optional (default = 2)

1. Power parameter for the Minkowski metric.
2. p = 1, this is equivalent to using manhattan_distance (l1),
3. p = 2, and euclidean_distance (l2)
4. For arbitrary p, minkowski_distance (l_p) is used

metric : string or callable, default 'minkowski' - the distance metric to use for the tree.

1. The default metric is minkowski,
2. p=2 is equivalent to the standard Euclidean metric

Methods:

fit(X, y) - Fit the model using X as training data and y as target values

get_params([deep]) - Get parameters for this estimator.

kneighbors([X, n_neighbors, return_distance]) - Finds the K-neighbors of a point.

kneighbors_graph([X, n_neighbors, mode]) - Computes the (weighted) graph of k-Neighbors for points in X

predict(X) - Predict the class labels for the provided data

predict_proba(X) - Return probability estimates for the test data X.

score(X, y[, sample_weight]) - Returns the mean accuracy on the given test data and labels.

5) Decision Tree

Definition:

Given a data of attributes together with its classes, a decision tree produces a sequence of rules that can be used to classify the data.

Advantages:

Decision Tree is simple to understand and visualise, requires little data preparation, and can handle both numerical and categorical data.

Disadvantages:

Decision tree can create complex trees that do not generalise well, and decision trees can be unstable because small variations in the data might result in a completely different tree being generated.

Syntax:

from sklearn.tree import DecisionTreeClassifier

model = DecisionTreeClassifier(criterion='gini', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1, max_features=None, random_state=None, max_leaf_nodes=None)

model.fit(X_train,Y_train,)

predictions = mode.predict(X_test)

Parameters:

criterion : string, optional (default="gini") - The function to measure the quality of a split. Supported criteria are

1. "gini" for the Gini impurity - minimioze the propability of missclassification
2. "entropy" - defines the certainity of the decision(0 - completely certain, 1 - completely uncertain)

splitter : string, optional (default="best") - The strategy used to choose the split at each node. Supported strategies are

1. "best" to choose the best split
2. "random" to choose the best random split.

max_depth : int or None, optional (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, float, optional (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.

min_samples_leaf : int, float, optional (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.

max_features : int, float, string or None, optional (default=None) - 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 int(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).
- If “log2”, then max_features=log2(n_features).
- If None, then max_features=n_features.

max_leaf_nodes : int or None, optional (default=None)

- Grow a tree 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.

Methods:

apply(X[, check_input]) - Returns the index of the leaf that each sample is predicted as.

decision_path(X[, check_input]) - Return the decision path in the tree

fit(X, y[, sample_weight, check_input, …]) - Build a decision tree classifier from the training set (X, y).

get_params([deep]) - Get parameters for this estimator.

predict(X[, check_input]) - Predict class or regression value for X.

predict_log_proba(X) - Predict class log-probabilities of the input samples X.

predict_proba(X[, check_input]) - Predict class probabilities of the input samples X.

score(X, y[, sample_weight]) - Returns the mean accuracy on the given test data and labels.

6) Random Forest

Definition:

Random forest classifier is a meta-estimator that fits a number of decision trees on various sub-samples of datasets and uses average to improve the predictive accuracy of the model and controls over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement.

Advantages:

Reduction in over-fitting and random forest classifier is more accurate than decision trees in most cases.

Disadvantages:

Slow real time prediction, difficult to implement, and complex algorithm.

Syntax:

from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(n_estimators='warn', criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=True, oob_score=False,random_state=None,class_weight=None)

model.fit(X_train,Y_train,)

predictions = mode.predict(X_test)

Parameters:

n_estimators : integer, optional (default=10) - The number of trees in the forest.

bootstrap : boolean, optional (default=True) - Whether bootstrap samples are used when building trees. If False, the whole datset is used to build each tree.

oob_score : bool (default=False) -Whether to use out-of-bag samples to estimate the generalization accuracy.

class_weight : dict, list of dicts, "balanced", "balanced_subsample" or None, optional (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.

Methods:

apply(X) - Apply trees in the forest to X, return leaf indices.

decision_path(X) - Return the decision path in the forest

fit(X, y[, sample_weight]) - Build a forest of trees from the training set (X, y).

get_params([deep]) - Get parameters for this estimator.

predict(X) - Predict class for X.

predict_log_proba(X) - Predict class log-probabilities for X.

predict_proba(X) - Predict class probabilities for X.

score(X, y[, sample_weight]) - Returns the mean accuracy on the given test data and labels.

7) SupportVectorMachines:

Definition:

Support vector machine is a representation of the training data as points in space separated into categories by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall.

Advantages:

Effective in high dimensional spaces and uses a subset of training points in the decision function so it is also memory efficient.

Disadvantages:

The algorithm does not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation.

Syntax:

from sklearn.svm import SVC

model = SVC(C=1.0, kernel='rbf', degree=3,tol=0.001, cache_size=200, class_weight=None, verbose=False, max_iter=-1, decision_function_shape='ovr', random_state=None)

model.fit(X_train,Y_train,)

predictions = mode.predict(X_test)

Parameters:

C : float, optional (default=1.0) - Penalty parameter C of the error term.

kernel : string, optional (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, optional (default=3)

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

tol : float, optional (default=1e-3)

- Tolerance for stopping criterion.

cache_size : float, optional

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

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,
- original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2).

Methods:

decision_function(X) - Evaluates the decision function for the samples in X.

fit(X, y[, sample_weight]) - Fit the SVM model according to the given training data.

get_params([deep]) - Get parameters for this estimator.

predict(X) - Perform classification on samples in X.

score(X, y[, sample_weight]) - Returns the mean accuracy on the given test data and labels.