USING SCIKIT-LEARN

Introduction and Installation

Members:

Naveen Kumar MR
Mithun G
Myil Vaughanan V L
Pranav Polavarapu

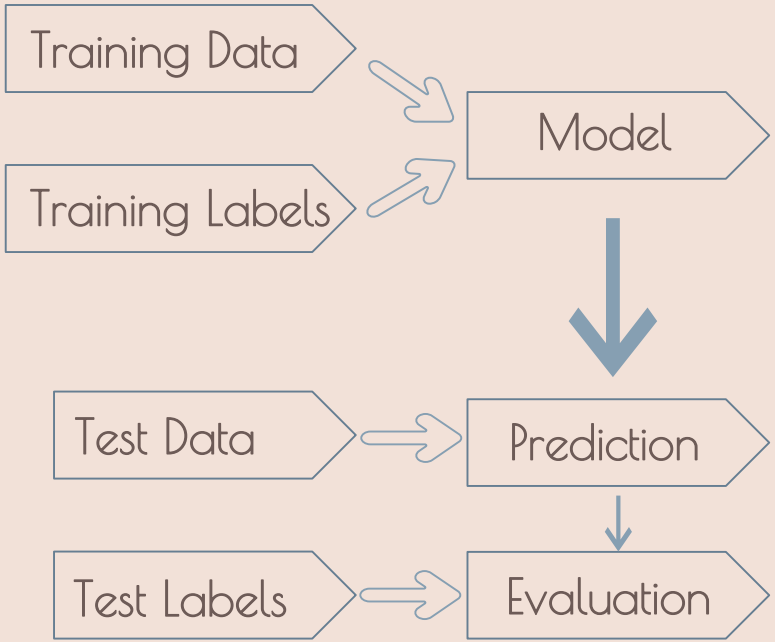
TABLE OF CONTENTS

TABLE OF CONTENTS

01

Types of Machine Learning

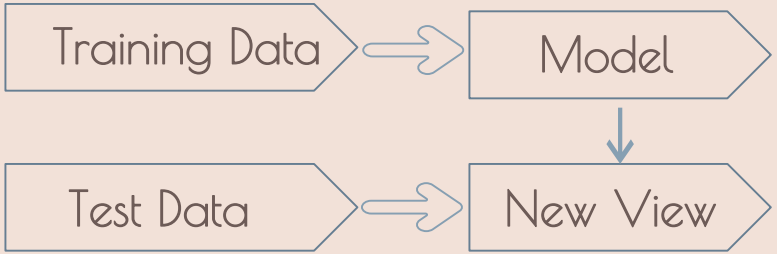
Supervised Machine Learning



Training

Generalization

Unsupervised Machine Learning



Eg: pca = PCA()

pca.fit(X_train)

X_new = pca.transform(X_test)

02

Intro to sklearn

INTRODUCTION

Scikit-learn is an open source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection and evaluation, and many other utilities.

What is sklearn

Scikit-learn (Sklearn) is the most useful and robust library for machine learning in Python. It provides a selection of efficient tools for machine learning and statistical modeling including classification, regression, clustering and dimensionality reduction via a consistence interface in Python. This library, which is largely written in Python, is built upon NumPy, SciPy and Matplotlib.

Basic API

estimator.fit(X , [y])

estimator.predict	estimator.transform
Classification Regression Clustering	Preprocessing Dimensionality reduction Feature selection Feature extraction

Example of Estimator API

1)Importing the Model:

```
from sklearn.linear_model import LinearRegression
```

2)Estimator parameter:

```
model = LinearRegression(normalize = True)
```

3)Split data:

```
form sklearn.cross_validation import train_test_split
X, y = np.arrange(10).reshape( (5,2) ).range(5)
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size = 0.3 )
```

Example of Estimator API

4)Fit the Model:

```
model.fit( X_train, y_train )
```

5)Predicting the test data:

```
Predictions = model.predict( X_test )
```

6)Evaluation is done at the last

Note : model.predict_probe() - It returns the probability that a new obstruction has each label
model.score() - measuring the accuracy of the model against the training data

03

Histroy of sklearn

Origin of sklearn

It was originally called scikits.learn and was initially developed by David Cournapeau as a Google summer of code project in 2007. Later, in 2010, Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, and Vincent Michel, from FIRCA (French Institute for Research in Computer Science and Automation), took this project at another level and made the first public release (v0.1 beta) on 1st Feb. 2010..

Version History

Versions (sklearn)	Year	Major changes
0.18.0	Sept 2016	Model Selection Enhancements and API Changes
0.19.0	July 2017	Added multioutput.ClassifierChain
0.20.0	Sept 2018	Bug fixes, Changed models & Enchanments
0.21.0	May 2019	Added cluster.OPTICS algorithm
0.22.0	Dec 2019	Bug fixes
0.23.0	May 2020	The critical parts of cluster.KMeans have a more optimized implementation.
0.24.0	Jan 2021	Added ensemble.HistGradientBoostingRegressor and ensemble.HistGradientBoostingClassifier
1.0	Sept 2021	calibration.CalibrationDisplay added to plot calibration curves.

04

Installation

Installation Prerequisites

Before we start using scikit-learn latest release, we require the following –

- Python (>=3.5)
- NumPy (>= 1.11.0)
- Scipy (>= 0.17.0)li
- Joblib (>= 0.11)
- Matplotlib (>= 1.5.1) is required for Sklearn plotting capabilities.
- Pandas (>= 0.18.0) is required for some of the scikit-learn examples using data structure and analysis.

Installation

Using pip

Following command can be used to install scikit-learn via pip –

```
$ pip install –U scikit-learn
```

Using conda

Following command can be used to install scikit-learn via conda –

```
$ conda install scikit-learn
```

05

Pros and Cons

Pros and cons

Pros:

The library is distributed under the BSD license, making it free with minimum legal and licensing restrictions.
It is easy to use.
The scikit-learn library is very versatile and handy and serves real-world purposes like the prediction of consumer behavior, the creation of neuroimages, etc.
Scikit-learn is backed and updated by numerous authors, contributors, and a vast international online community.
The scikit-learn website provides elaborate API documentation for users who want to integrate the algorithms with their platforms.

Con:

It is not the best choice for in-depth learning

06

Modules in sklearn

sklearn.datasets

The sklearn.datasets module includes utilities to load datasets, including methods to load and fetch popular reference datasets. It also features some artificial data generators.

Datasets - Loaders

datasets.load_boston(*[, return_X_y]) DEPRECATED: load_boston is deprecated in 1.0 and will be removed in 1.2.

datasets.load_breast_cancer(*[, return_X_y, ...]) Load and return the breast cancer wisconsin dataset (classification).

datasets.load_diabetes(*[, return_X_y, as_frame]) Load and return the diabetes dataset (regression).

datasets.load_digits(*[, n_class, ...]) Load and return the digits dataset (classification).

datasets.load_files(container_path, *[, ...]) Load text files with categories as subfolder names.

datasets.load_iris(*[, return_X_y, as_frame]) Load and return the iris dataset (classification).

datasets.load_linnerud(*[, return_X_y, as_frame]) Load and return the physical exercise Linnerud dataset.

datasets.load_sample_image(image_name) Load the numpy array of a single sample image

datasets.load_sample_images() Load sample images for image manipulation.

datasets.load_svmlight_file(f, *[, ...]) Load datasets in the svmlight / libsvm format into sparse CSR matrix

datasets.load_svmlight_files(files, *[, ...]) Load dataset from multiple files in SVMlight format

datasets.load_wine(*[, return_X_y, as_frame]) Load and return the wine dataset (classification).

Example: sklearn.datasets.load_iris

```
>>> from sklearn.datasets import load_iris
>>> data = load_iris()
>>> data.target[[10, 25, 50]]
array([0, 0, 1])
>>> list(data.target_names)
['setosa', 'versicolor', 'virginica']
```

Datasets - Sample generators

datasets.make_biclusters(shape, n_clusters, *) Generate an array with constant block diagonal structure for biclustering.

datasets.make_blobs([n_samples, n_features, ...]) Generate isotropic Gaussian blobs for clustering.

datasets.make_checkerboard(shape, n_clusters, *) Generate an array with block checkerboard structure for biclustering.

datasets.make_circles([n_samples, shuffle, ...]) Make a large circle containing a smaller circle in 2d.

datasets.make_classification([n_samples, ...]) Generate a random n-class classification problem.

datasets.make_friedman1([n_samples, ...]) Generate the "Friedman #1" regression problem..

datasets.make_gaussian_quantiles(*[, mean, ...]) Generate isotropic Gaussian and label samples by quantile.

datasets.make_hastie_10_2([n_samples, ...] Generates data for binary classification used in Hastie et al. 2009, Example 10.2.

datasets.make_low_rank_matrix([n_samples, ...]) Generate a mostly low rank matrix with bell-shaped singular values.

datasets.make_moons([n_samples, shuffle, ...]) Make two interleaving half circles.

Example: sklearn.datasets.make_blobs

```
>>> from sklearn.datasets import make_blobs
>>> X, y = make_blobs(n_samples=10, centers=3, n_features=2,
...                   random_state=0)
>>> print(X.shape)
(10, 2)
>>> y
array([0, 0, 1, 0, 2, 2, 2, 1, 1, 0])
>>> X, y = make_blobs(n_samples=[3, 3, 4], centers=None,
n_features=2,
...                   random_state=0)
>>> print(X.shape)
(10, 2)
>>> y
array([0, 1, 2, 0, 2, 2, 2, 1, 1, 0])
```

sklearn.exceptions

The sklearn.exceptions module includes all custom warnings and error classes used across scikit-learn.

Exceptions and warnings

exceptions.ConvergenceWarning - Custom warning to capture convergence problems

exceptions.DataConversionWarning - Warning used to notify implicit data conversions happening in the code.

exceptions.DataDimensionalityWarning - Custom warning to notify potential issues with data dimensionality.

exceptions.EfficiencyWarning - Warning used to notify the user of inefficient computation.

exceptions.FitFailedWarning - Warning class used if there is an error while fitting the estimator.

exceptions.NotFittedError - Exception class to raise if estimator is used before fitting.

exceptions.UndefinedMetricWarning - Warning used when the metric is invalid

sklearn.pipeline

The sklearn.pipeline module implements utilities to build a composite estimator, as a chain of transforms and estimators.

Pipeline

pipeline.FeatureUnion(transformer_list, *[, ...]) Concatenates results of multiple transformer objects.

pipeline.Pipeline(steps, *[, memory, verbose]) Pipeline of transforms with a final estimator.

pipeline.make_pipeline(*steps[, memory, verbose]) Construct a Pipeline from the given estimators.

pipeline.make_union(*transformers[, n_jobs, ...]) Construct a FeatureUnion from the given transformers

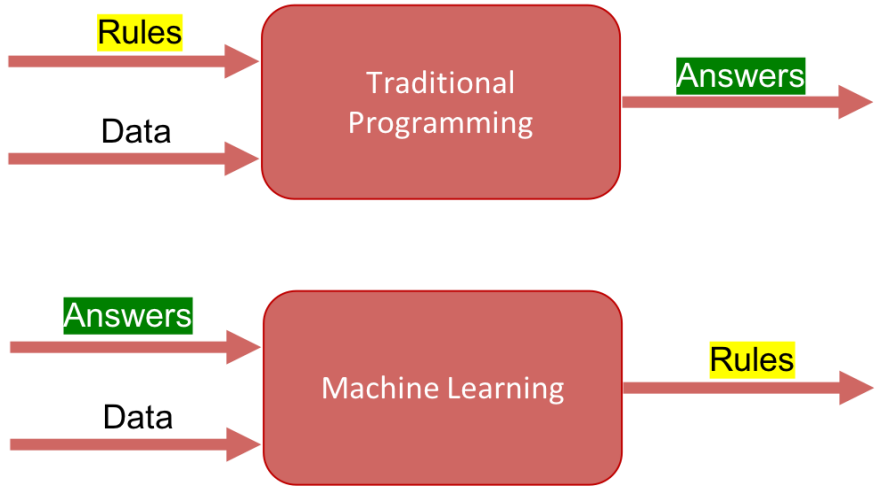
Example: sklearn.pipeline.FeatureUnion

```
>>> from sklearn.pipeline import FeatureUnion
>>> from sklearn.decomposition import PCA, TruncatedSVD
>>> union = FeatureUnion([("pca", PCA(n_components=1)),
...                       ("svd", TruncatedSVD(n_components=2))])
>>> X = [[0., 1., 3], [2., 2., 5]]
>>> union.fit_transform(X)
array([[ 1.5       ,  3.0...,  0.8...],
       [-1.5       ,  5.7..., -0.4...]])
```

Machine Learning in Python

Introduction to ML

- From data science perspective we can define ML as a mean of building models of data.
- There are mainly 2 types of ML techniques:
  1) Supervised Learning
  2) Unsupervised Learning



Supervised Learning

The training data you feed to the algorithm includes the desired solutions or the labels

Classification and regression problems are some of the supervised learning algorithms

Examples for supervised learning algorithms: Linear regression, KNN, Logistic regression, Gradient Descent, SVM etc

Unsupervised Learning

- Here, the training data is unlabeled
- The system tries to learn by itself
- These models includes tasks such as clustering and dimensionality reduction
- Examples of unsupervised learning algorithms:- K means, PCA, Hierarchical cluster algorithm

1.1. Linear Models

- 1.1.1. Ordinary Least Squares
- 1.1.2. Ridge regression and classification
- 1.1.3. Lasso
- 1.1.4. Multi-task Lasso
- 1.1.5. Elastic-Net
- 1.1.6. Multi-task Elastic-Net
- 1.1.7. Least Angle Regression
- 1.1.8. LARS Lasso
- 1.1.9. Orthogonal Matching Pursuit (OMP)
- 1.1.10. Bayesian Regression
- 1.1.11. Logistic regression
- 1.1.12. Generalized Linear Regression
- 1.1.13. Stochastic Gradient Descent - SGD
- 1.1.14. Perceptron
- 1.1.15. Passive Aggressive Algorithms
- 1.1.16. Robustness regression: outliers and modeling errors
- 1.1.17. Quantile Regression
- 1.1.18. Polynomial regression: extending linear models with basis functions

List of algorithms supported in Linear models

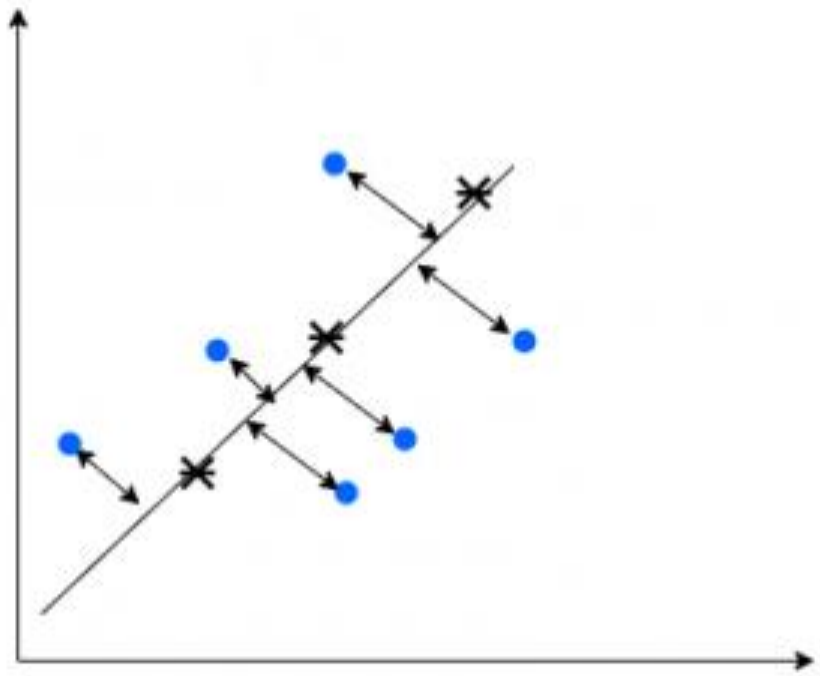
Linear models class

- The following are a set of methods intended for regression in which the target value is expected to be a linear combination of the features.
- Regression is the process of predicting a continuous values using some other variables.

Ordinary Least squares

- Ordinary Least square method is one of the mathematical approaches used for Linear regression
- We will use the OLS method, which is fitted by minimizing the sum of squares of the residuals
- `LinearRegression` will take in its `fit` method arrays X, y and will store the coefficients w of the linear model in its `coef_` member:

```
>>> from sklearn import linear_model
>>> reg = linear_model.LinearRegression()
>>> reg.fit([[0, 0], [1, 1], [2, 2]], [0, 1, 2])
LinearRegression()
>>> reg.coef_
array([0.5, 0.5])
```



Parameters & Attributes

- *class* sklearn.linear_model.LinearRegression(*, *fit_intercept=True*, *normalize='deprecated'*, *copy_X=True*, *n_jobs=None*, *positive=False*)

Attributes:

coef_ : *array of shape (n_features,) or (n_targets, n_features)*
Estimated coefficients for the linear regression problem. If multiple targets are passed during the fit (y 2D), this is a 2D array of shape (n_targets, n_features), while if only one target is passed, this is a 1D array of length n_features.

rank_ : *int*
Rank of matrix `X`. Only available when `X` is dense.

singular_ : *array of shape (min(X, y),)*
Singular values of `X`. Only available when `X` is dense.

intercept_ : *float or array of shape (n_targets,)*
Independent term in the linear model. Set to 0.0 if `fit_intercept = False`.

n_features_in_ : *int*
Number of features seen during fit.

New in version 0.24.

feature_names_in_ : *ndarray of shape (`n_features_in_`,)*
Names of features seen during fit. Defined only when `X` has feature names that are all strings.

Attributes available in linear_model.Linear_regression

Nearest Neighbors class

sklearn.neighbors provides functionality for unsupervised and supervised neighbors-based learning methods.

Supervised neighbors-based learning comes in two flavors: classification for data with discrete labels, and regression for data with continuous labels.

The principle behind nearest neighbor methods is to find a predefined number of training samples closest in distance to the new point and predict the label from these. The number of samples can be a user-defined constant (k-nearest neighbor learning)

The distance can, in general, be any metric measure: standard Euclidean distance is the most common choice. Neighbors-based methods are known as *non-generalizing* machine learning methods, since they simply "remember" all its training data

K – Nearest Neighbor Algorithm

- KNN algorithm is a classification algorithm that takes a bunch of labelled points and uses them to learn how to label other points
- The algorithm classifies cases based on their similarity to other classes, the data points which are nearer to each other is called neighbors
- It is based on 'similar cases with same class labels are near to each other'
- So, the distance between 2 cases is a measure of dissimilarity

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from sklearn.neighbors import KNeighborsClassifier
>>> neigh = KNeighborsClassifier(n_neighbors=3)
>>> neigh.fit(X, y)
KNeighborsClassifier(...)
>>> print(neigh.predict([[1.1]]))
[0]
>>> print(neigh.predict_proba([[0.9]]))
[[0.666... 0.333...]]
```

Attributes:

classes_ : *array of shape (n_classes,)*
Class labels known to the classifier

effective_metric_ : *str or callble*
The distance metric used. It will be same as the `metric` parameter or a synonym of it, e.g. 'euclidean' if the `metric` parameter set to 'minkowski' and `p` parameter set to 2.

effective_metric_params_ : *dict*
Additional keyword arguments for the metric function. For most metrics will be same with `metric_params` parameter, but may also contain the `p` parameter value if the `effective_metric_` attribute is set to 'minkowski'.

n_features_in_ : *int*
Number of features seen during fit.

New in version 0.24.

feature_names_in_ : *ndarray of shape (`n_features_in_`,)*
Names of features seen during fit. Defined only when `X` has feature names that are all strings.

New in version 1.0.

n_samples_fit_ : *int*
Number of samples in the fitted data.

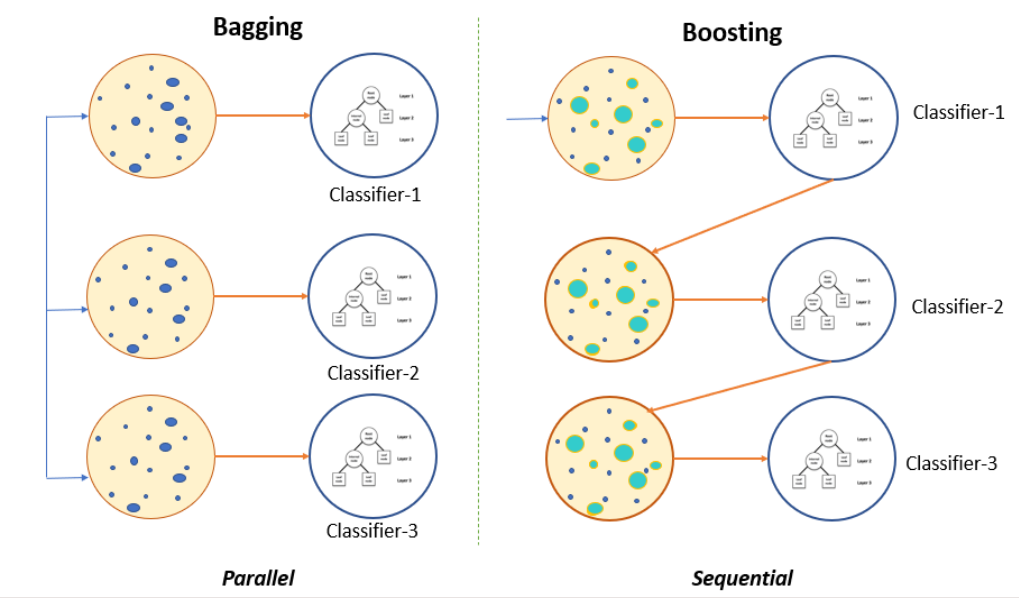
outputs_2d_ : *bool*
False when `y`'s shape is (n_samples,) or (n_samples, 1) during fit otherwise True.

Attributes available in neighbors.KNeighborsClassifier

Parameters and Attributes

- *class* sklearn.neighbors.KNeighborsClassifier(*n_neighbors=5*, *, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*

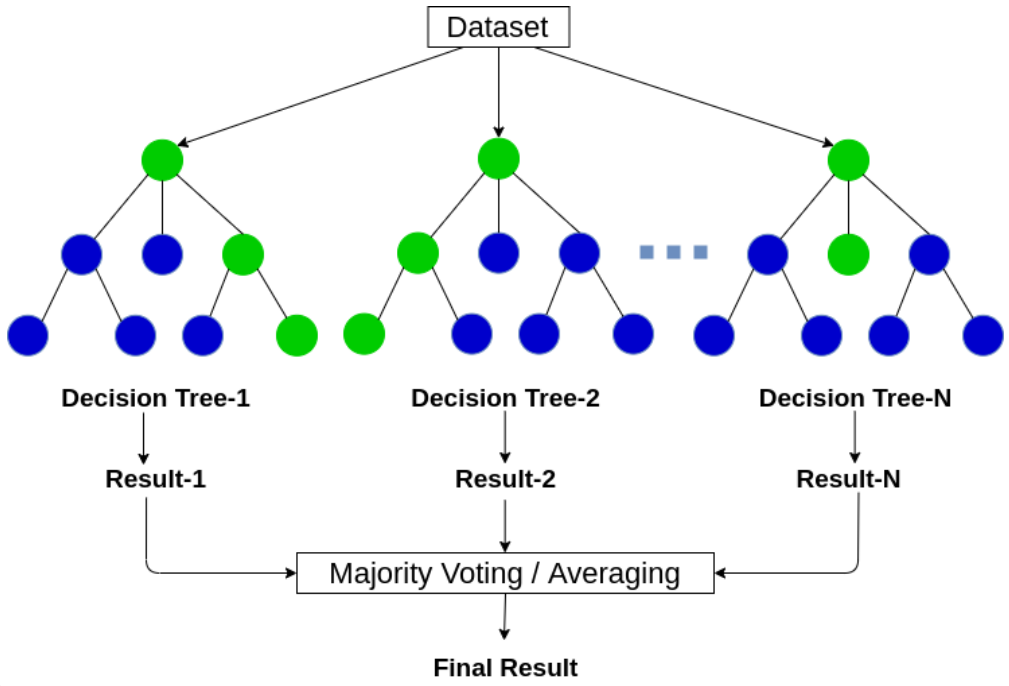
Ensemble methods



- The goal of **ensemble methods** is to combine the predictions of several base estimators built with a given learning algorithm in order to improve generalizability / robustness over a single estimator.
- Two families of ensemble methods are usually distinguished:
- In **averaging methods**, the driving principle is to build several estimators independently and then to average their predictions. On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced.
- **Examples:** Bagging methods, Forests of randomized trees, …
- By contrast, in **boosting methods**, base estimators are built sequentially, and one tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble.
- **Examples:** AdaBoost, Gradient Tree Boosting, …

Forests of randomized trees algorithm

- The forest it builds, is an ensemble of decision trees, trained with bagging method. The idea of this method is to combine learning models increases the overall result
- Random Forest adds additional randomness to the model, while growing the trees. Instead of searching for the most important feature while splitting a node.
- Random forest can be used for both classification and regression techniques by using a method called as Bootstrap Aggregation



Attributes:

base_estimator_ : *DecisionTreeClassifier*
The child estimator template used to create the collection of fitted sub-estimators.

estimators_ : *list of DecisionTreeClassifier*
The collection of fitted sub-estimators.

classes_ : *ndarray of shape (n_classes,) or a list of such arrays*
The classes labels (single output problem), or a list of arrays of class labels (multi-output problem).

n_classes_ : *int or list*
The number of classes (single output problem), or a list containing the number of classes for each output (multi-output problem).

`n_features_` **: *int***
DEPRECATED: Attribute `n_features_` was deprecated in version 1.0 and will be removed in 1.2.

n_features_in_ : *int*
Number of features seen during fit.

New in version 0.24.

feature_names_in_ : *ndarray of shape (`n_features_in_`,)*
Names of features seen during fit. Defined only when `X` has feature names that are all strings. .. versionadded:: 1.0

n_outputs_ : *int*
The number of outputs when `fit` is performed.

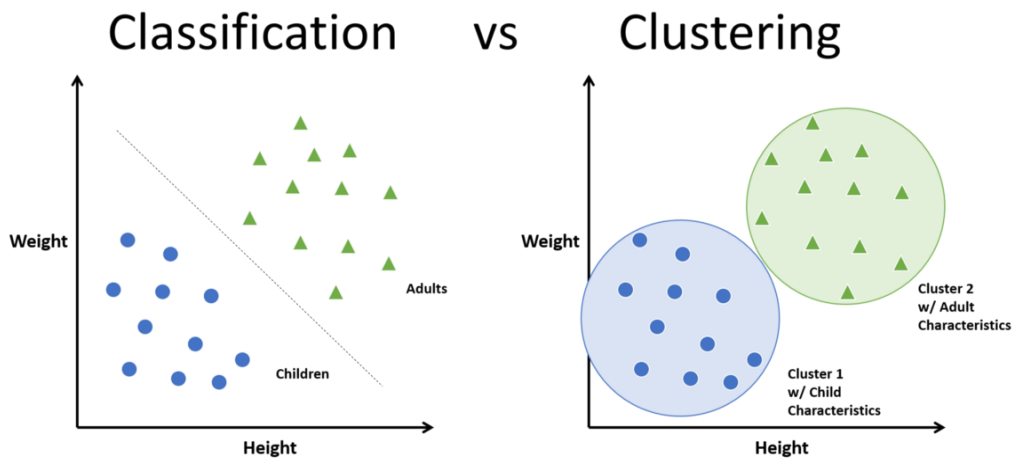
Attributes available in ensemble.RandomForestClassifier

Parameters and Attributes

- *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, 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*)

Cluster method

- Clustering is an unsupervised machine learning approach
- Cluster – a group of objects that are like the other objects in the cluster, and dissimilar to datapoints in other clusters
- Each clustering algorithm comes in two variants: a class, that implements the `fit` method to learn the clusters on train data, and a function, that, given train data, returns an array of integer labels corresponding to the different clusters. For the class, the labels over the training data can be found in the `labels_` attribute.



K means clustering algorithm

- It is a partition based clustering which means it divides the data into k clusters without any cluster internal structure without any labels
- Objects within a cluster are homogeneous and the objects across the clusters are heterogeneous
- The distance of samples from each other is used to shape the cluster
- So, we can clearly say that K-means tries to minimizes the intra-cluster and tries to maximize the inter-cluster

```
>>> from sklearn.cluster import KMeans
>>> import numpy as np
>>> X = np.array([[1, 2], [1, 4], [1, 0],
...               [10, 2], [10, 4], [10, 0]])
>>> kmeans = KMeans(n_clusters=2, random_state=0).fit(X)
>>> kmeans.labels_
array([1, 1, 1, 0, 0, 0], dtype=int32)
>>> kmeans.predict([[0, 0], [12, 3]])
array([1, 0], dtype=int32)
>>> kmeans.cluster_centers_
array([[10.,  2.],
       [ 1.,  2.]])
```

Parameters and Attributes

Attributes:

cluster_centers_ : *ndarray of shape (n_clusters, n_features)*
Coordinates of cluster centers. If the algorithm stops before fully converging (see `tol` and `max_iter`), these will not be consistent with `labels_`.

labels_ : *ndarray of shape (n_samples,)*
Labels of each point

inertia_ : *float*
Sum of squared distances of samples to their closest cluster center, weighted by the sample weights if provided.

n_iter_ : *int*
Number of iterations run.

n_features_in_ : *int*
Number of features seen during fit.

New in version 0.24.

feature_names_in_ : *ndarray of shape (`n_features_in_`,)*
Names of features seen during fit. Defined only when `X` has feature names that are all strings.

New in version 1.0.

Attributes available in cluster.Kmeans

- *class* sklearn.cluster.KMeans(*n_clusters=8*, *, *init='k-means++'*, *n_init=10*, *max_iter=300*, *tol=0.0001*, *verbose=0*, *random_state=None*, *copy_x=True*, *algorithm='auto'*)

Contents of the Presentation

What is data Preprocessing?

- Data preprocessing is a data mining technique which is used to transform the raw data in a useful and efficient format.
- In our presentation we will be looking into few of the methods involved in preprocessing:
    1. Standardization
    2. Normalization
    3. Encoding
    4. Discretization
    5. Imputation of missing values

Module for Preprocessing

For Data Preprocessing we will be making use of sklearn.preprocessing class which is present in the sklearn library

Standardization – z-score Standardization

- Standardization of datasets is a common requirement for many machine learning estimators implemented in scikit-learn
- they might behave badly if the individual features do not more or less look like standard normally distributed data: Gaussian with zero mean and unit variance.
- The preprocessing module provides the StandardScaler utility class, which is a quick and easy way to perform scaling

Normalization

Normalization is the process of scaling individual samples to have unit norm.

This process can be useful if you plan to use a quadratic form such as the dot-product or any other kernel to quantify the similarity of any pair of samples.

The function normalize provides a quick and easy way to perform this operation on a single array-like dataset, either using the l1, l2, or max norms

The preprocessing module further provides a utility class Normalizer that implements the same operation using the Transformer API (even though the fit method is useless in this case: the class is stateless as this operation treats samples independently)

Encoding categorical features – OrdinalEncoder

- To convert categorical features to such integer codes, we can use the OrdinalEncoder.
- This estimator transforms each categorical feature to one new feature of integers (0 to n_categories - 1)
- Such integer representation can, however, not be used directly with all scikit-learn estimators, as these expect continuous input, and would interpret the categories as being ordered, which is often not desired (i.e. the set of browsers was ordered arbitrarily).

Encoding categorical features – OneHotEncoder

- Another possibility to convert categorical features to features that can be used with scikit-learn estimators is to use a one-of-K, also known as one-hot or dummy encoding.
- This type of encoding can be obtained with the OneHotEncoder, which transforms each categorical feature with n_categories possible values into n_categories binary features, with one of them 1, and all others 0.

Discretization

- Discretization (otherwise known as quantization or binning) provides a way to partition continuous features into discrete values.
- Certain datasets with continuous features may benefit from discretization, because discretization can transform the dataset of continuous attributes to one with only nominal attributes.
- The two types of discretizatisers:
    1. K-bins discretization
    2. Feature binarization

Imputation of missing values

- The SimpleImputer class provides basic strategies for imputing missing values. Missing values can be imputed with a provided constant value, or using the statistics (mean, median or most frequent) of each column in which the missing values are located. This class also allows for different missing values encodings.

TABLE OF CONTENTS

07

Train-Test split & Cross-Validation

Train_Test_Split

- Building an optimum model which **neither underfits nor overfits** the dataset takes effort. To know the performance of our model on unseen data, we can ***split*** the dataset into train and test sets and perform ***cross-validation.***

- To know the performance of a model, we should test it on **unseen data**. For that purpose, we partition dataset into training set (around 70 to 90% of the data) and test set (10 to 30%). In, **sklearn** we use ***train_test_split*** function from ***sklearn.model_selection.***

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2, random_state=4)
print ('Train set:', X_train.shape,  y_train.shape)
print ('Test set:', X_test.shape,  y_test.shape)
```

```
Train set: (800, 11) (800, 1)
Test set: (200, 11) (200, 1)
```

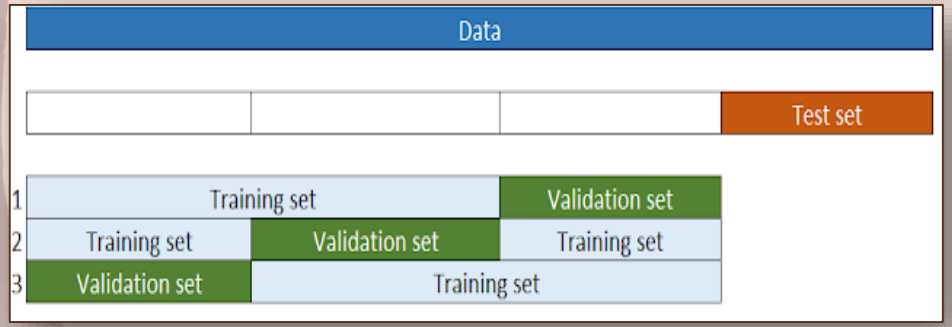
```
type(y_test)
```

```
pandas.core.frame.DataFrame
```

Cross-validation

- During **Hyperparameter tuning,** to know if the value of hyperparameter that we chose is **optimal or not**, we must run the model on test set. But if we use the test set more than once, then the information from test dataset *leaks to the model*. This leads to over-fitting or byhearting the value of dependent variable. To avoid that, we **use cross-validation**.

- We use one more test set, that is called **validation set** to tune the hyperparameters. Following picture depicts the 3-fold CV. **K-fold CV** corresponds to *subdividing the dataset into k folds* such that each fold gets the chance to be in both training set and validation set.




```python
# scikit-learn k-fold cross-validation
from numpy import array
from sklearn.model_selection import KFold
# data sample
data = array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
# prepare cross validation
kfold = KFold(3, True)
gen=kfold.split(data)
print(next(gen))
print(next(gen))
print(next(gen))
# enumerate splits
for train, test in kfold.split(data):
 print('train: %s, test: %s' % (data[train], data[test]))
```

```
(array([1, 2, 3, 5]), array([0, 4]))
(array([0, 2, 3, 4]), array([1, 5]))
(array([0, 1, 4, 5]), array([2, 3]))
train: [0.2 0.3 0.4 0.5], test: [0.1 0.6]
train: [0.1 0.3 0.5 0.6], test: [0.2 0.4]
train: [0.1 0.2 0.4 0.6], test: [0.3 0.5]
```

08

Hyperparameter Tuning

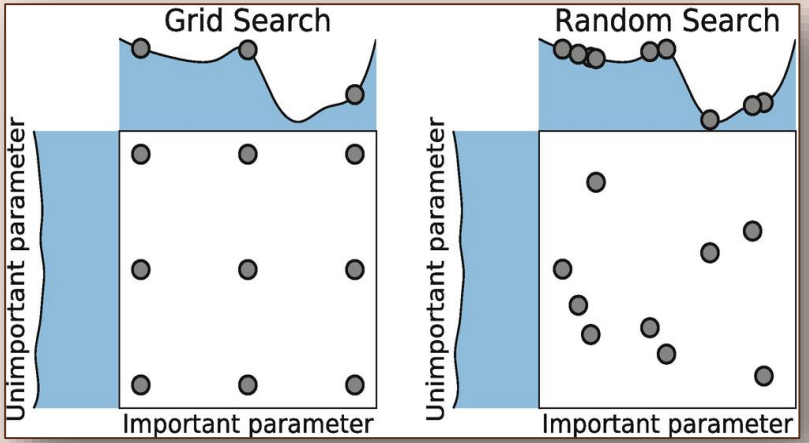
Hyperparameters

- Like an analog Radio, ML models have all sorts of *knobs and dials* you can tune, called as **Hyperparameters**.
- Few examples are *Learning Rate, Regularization Strength, No(nodes) in CNN, kernels in SVM, etc.;*
- Choosing right Models & tuning any knobs and dials associated with the model is very crucial - one wrong choice, and *your accuracy can vary*.
- In Scikit-learn, this can be done with **grid search** and **random search**.

The general idea behind both algorithms is that you:

a) Define a set of hyperparameters you want to tune
b) Give these hyperparameters to the grid search or random search
c) These algorithms then automatically examine the hyperparameter search space and attempt to find the optimal values that maximize accuracy

Tuning using sklearn




```python
from sklearn.model_selection import GridSearchCV

grid_params = {
    'n_neighbors': [3,5,11,19],
    'weights': ['uniform', 'distance'],
    'metric':['euclidean', 'manhattan']
}

gs = GridSearchCV(
     KNeighborsClassifier(),
     grid_params,
     verbose = 1,
     cv = 3,
     n_jobs = -1
     )

gs_results = gs.fit(X_train, y_train)
```

~Grid Search Cross-Validation **example**

09
Evaluation Metrics

Evaluation Metrics

- As we train your ML Model, we want to **assess how good** it is. Interestingly, there are many ways of evaluating the *performance*. Most data scientists that use Python for predictive modeling use the scikit-learn Python package. Scikit-learn contains **many built-in functions** for **analyzing the performance** of models.

- The metrics that you choose to evaluate your machine learning algorithms are very important.

- **Choice of metrics** influences how the performance of machine learning algorithms is measured and compared. They influence how you **weight the importance** of different characteristics in the results and your ultimate choice of which algorithm to choose.

- All the available Metrics for different ML Models can be found here:

https://scikit-learn.org/stable/modules/model_evaluation.html

Few examples...

Classification Metrics

Accuracy Score:

```
>>> knn.score(X_test, y_test)
>>> from sklearn.metrics import accuracy_score
>>> accuracy_score(y_test, y_pred)
```

Classification Report:

```
>>> from sklearn.metrics import classification_report
>>> print(classification_report(y_test, y_pred)))
```

Confusion Matrix:

```
>>> from sklearn.metrics import confusion_matrix
>>> print(confusion_matrix(y_test, y_pred)))
```

- A **confusion matrix** is a technique for **summarizing** the performance of a classification algorithm.
- **Accuracy Score** is ***The sum of true positives and true negatives divided by the total number of samples***. This is only accurate if the model is balanced. It will give inaccurate results if there is a class imbalance.
- A **classification report** is a performance evaluation metric in machine learning. It is used to show *the precision, recall, F1 Score, and support* of your trained classification model.

Regression Metrics

- **MAE (Mean absolute error)** is the difference between the original and predicted values extracted by averaged the absolute difference over the data set.

- **MSE (Mean Squared Error)** is the difference between the original and predicted values extracted by squared the average difference over the data set.

- **R squared**, also called coefficient of determination, measures the degree of interrelation and dependence between two variables.

```
#Mean Absolute Error
>>> from sklearn.metrics import mean_absolute_error
>>> y_true = [3, -0.5, 2])
>>> mean_absolute_error(y_true, y_pred))
#Mean Squared Error
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(y_test, y_pred))
#R2 Score
>>> from sklearn.metrics import r2_score
>>> r2_score(y_true, y_pred))
```

Clustering Metrics

```
#Adjusted Rand Index
>>> from sklearn.metrics import adjusted_rand_score
>>> adjusted_rand_score(y_true, y_pred))

#Homogeneity
>>> from sklearn.metrics import homogeneity_score
>>> homogeneity_score(y_true, y_pred))

#V-measure
>>> from sklearn.metrics import v_measure_score
>>> metrics.v_measure_score(y_true, y_pred))
```

- The **Adjusted Rand Index (ARI)** is frequently used in cluster validation since it is a measure of agreement between two partitions: one given by the clustering process and the other defined by external criteria
- **Homogeneity** describes the closeness of the clustering algorithm to this perfection.
- The **V-measure** is the harmonic mean between homogeneity and completeness.

Cross Validation

- The **cross_val_score()** function will be used to perform the evaluation, taking the dataset and cross-validation configuration and returning a list of scores calculated for each fold
- For the **cross_val_score()** , you are using the average of the output, which will be affected by the number of folds because then it may have some folds which may have high error (not fit correctly).

```
#Cross-Validation
>>> print(cross_val_score(knn, X_train, y_train, cv=4))
>>> print(cross_val_score(lr, X, y, cv=2)
```

```
              precision    recall  f1-score   support

         0.0       0.77      0.87      0.82       162
         1.0       0.71      0.55      0.62        92

avg / total       0.75      0.76      0.75       254
```

```
>>> from sklearn.metrics import mean_absolute_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_absolute_error(y_true, y_pred)
0.5
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> mean_absolute_error(y_true, y_pred)
0.75
>>> mean_absolute_error(y_true, y_pred, multioutput='raw_values')
array([0.5, 1. ])
>>> mean_absolute_error(y_true, y_pred, multioutput=[0.3, 0.7])
0.85...
```

Classification metrics:

- Accuracy.
- Log Loss.
- Area Under ROC Curve.

Convenience methods for classification prediction results:

- Confusion Matrix.
- Classification Report.

Regression metrics:

- Mean Absolute Error.
- Mean Squared Error.
- R^2.