ml_algos

January 15, 2021

#

Welcome to Supervised Learning

##

Part 4: Non-linear supervised machine learning algorithms

##

Instructor: Andras Zsom

###

https://github.com/azsom/Supervised-Learning

0.1 The topic of the course series: supervised Machine Learning (ML)

- how to build an ML pipeline from beginning to deployment
- we assume you already performed data cleaning
- this is the fourth course out of 6 courses
  - Part 1: Introduction to machine learning and the bias-variance tradeoff
  - Part 2: How to prepare your data for supervised machine learning**
  - Part 3: Evaluation metrics in supervised machine learning
  - **Part 4: Non-linear supervised machine learning algorithms**
  - Part 5: Missing data in supervised ML
  - Part 6: Interpretability
- you can complete the courses in sequence or complete individual courses based on your interest

0.1.1 Structured data

X	feature_1	feature_2	...	feature_j	...	feature_m	Y
data_point_1	x_11	x_12	...	x_1j	...	x_1m	**y_1**
data_point_2	x_21	x_22	...	x_2j	...	x_2m	**y_2**
...	...	...	...	...	...	...	**...**
data_point_i	x_i1	x_i2	...	x_ij	...	x_im	**y_i**
...	...	...	...	...	...	...	**...**
data_point_n	x_n1	x_n2	...	x_nj	...	x_nm	**y_n**

0.1.2 Learning objectives of this course

By the end of the course, you will be able to - Summarize how each algorithm works (KNN, SVM, RF, XGBoost) - Describe which hyperparameters need to be tuned and what range the values should have - Apply the algorithms in regression and classification - Visualize the predictions of toy datasets - Summarize under what circumstances a certain algorithm is expected to perform well or poorly and why

0.2 Which ML algorithm to try on your dataset?

- there is no algo that performs well under all conditions!
- try as many as you can ato figure out which one performs best on your dataset
- always start with linear or logistic regression - check out the first course in the course series!
- then try as many other ML lagorithms as you can
- only exclude an ML algorithm if you have a very good reason to believe it wouldn't work well!
- but you might be able to exclude some algos in advance
  - large dataset (>1e6 points)
  - more features than points
- other than predictive power, what else is important for you?
  - how the model behaves with respect to outliers?
  - does the prediction varies smoothly with the feature values?
  - can the model capture non-linear dependencies?
  - is the model easy to interpret for a human?

1 Module 1: K-Nearest Neighbors

1.0.1 Learning objectives of this module:

- Summarize how KNN works
- Describe which hyperparameters need to be tuned and what range the values should have
- Apply the algorithms in regression and classification
- Visualize the predictions of toy datasets
- Summarize under what circumstances a certain algorithm is expected to perform well or poorly and why

1.1 KNNs

- instance-based learning
- the feature values of a point is treated as a coordinate in an m dimensional space (m is the number of features)
- a distance metric (like euclidian or manhattan) is used to determine how far each point in the training set is from the point we want to predict
- collect the target variabe of k nearest points in the training set
- in classification: the predicted class is determined by the majority vote of target variable, the predicted probability is determined by the class ratios in the target variable
- in regression: the prediction is the mean of the target variable

1.2 KNN hyperparameters to tune

- read through the manual for detailed info
- n_neighbors: the number of neighbors to use
  - extreme cases:
    - n_neighbors = 1, the prediction is based on the nearest point only, this means 0 or 100% predicted probabilities in classification and a perfect score on the training set
    - n_neighbors = n (where n is the number of points in the training set), all points contribute to the prediction, usually not desired because it generates a high bias model
  - I recommend exploring values like [1, 3, 10, 30, 100, ...] the max value should be near n
- weights: 'uniform' or 'distance'
  - uniform: all nearest neighbors contribute equally to the prediction
  - distance: neighbors closer to the point contribute with a larger weight
- metric: the distance metric to use
  - minkowski with p = 1 is the manhattan distance and p = 2 is the euclidian distance
  - see here for a complete list

1.3 KNN regression

[1]:
```
# let's create a simple toy dataset

import numpy as np
np.random.seed(10)
def true_fun(X):
    return np.cos(1.5 * np.pi * X)

n_samples = 30

X = np.random.rand(n_samples)
y = true_fun(X) + np.random.randn(n_samples) * 0.1

X_new = np.linspace(-0.5, 1.5, 2000)
```

[2]:
```
from sklearn.neighbors import KNeighborsRegressor
help(KNeighborsRegressor)
```

```
Help on class KNeighborsRegressor in module sklearn.neighbors._regression:

class KNeighborsRegressor(sklearn.neighbors._base.NeighborsBase,
sklearn.neighbors._base.KNeighborsMixin,
sklearn.neighbors._base.SupervisedFloatMixin, sklearn.base.RegressorMixin)
 |  KNeighborsRegressor(n_neighbors=5, *, weights='uniform', algorithm='auto',
leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None,
**kwargs)
 |
 |  Regression based on k-nearest neighbors.
```

```
|
|  The target is predicted by local interpolation of the targets
|  associated of the nearest neighbors in the training set.
|
|  Read more in the :ref:`User Guide <regression>`.
|
|  .. versionadded:: 0.9
|
|  Parameters
|  ----------
|  n_neighbors : int, default=5
|      Number of neighbors to use by default for :meth:`kneighbors` queries.
|
|  weights : {'uniform', 'distance'} or callable, default='uniform'
|      weight function used in prediction.  Possible values:
|
|      - 'uniform' : uniform weights.  All points in each neighborhood
|        are weighted equally.
|      - '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.
|      - [callable] : a user-defined function which accepts an
|        array of distances, and returns an array of the same shape
|        containing the weights.
|
|      Uniform weights are used by default.
|
|  algorithm : {'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'
|      Algorithm used to compute the nearest neighbors:
|
|      - 'ball_tree' will use :class:`BallTree`
|      - 'kd_tree' will use :class:`KDTree`
|      - 'brute' will use a brute-force search.
|      - 'auto' will attempt to decide the most appropriate algorithm
|        based on the values passed to :meth:`fit` method.
|
|      Note: fitting on sparse input will override the setting of
|      this parameter, using brute force.
|
|  leaf_size : int, 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 : int, default=2
|      Power parameter for the Minkowski metric. When p = 1, this is
|      equivalent to using manhattan_distance (l1), and euclidean_distance
```

```
|      (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.
|
|  metric : str or callable, default='minkowski'
|      the distance metric to use for the tree.  The default metric is
|      minkowski, and with p=2 is equivalent to the standard Euclidean
|      metric. See the documentation of :class:`DistanceMetric` for a
|      list of available metrics.
|      If metric is "precomputed", X is assumed to be a distance matrix and
|      must be square during fit. X may be a :term:`sparse graph`,
|      in which case only "nonzero" elements may be considered neighbors.
|
|  metric_params : dict, default=None
|      Additional keyword arguments for the metric function.
|
|  n_jobs : int, default=None
|      The number of parallel jobs to run for neighbors search.
|      ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
|      ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
|      for more details.
|      Doesn't affect :meth:`fit` method.
|
|  Attributes
|  ----------
|  effective_metric_ : str or callable
|      The distance metric to use. 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'.
|
|  Examples
|  --------
|  >>> X = [[0], [1], [2], [3]]
|  >>> y = [0, 0, 1, 1]
|  >>> from sklearn.neighbors import KNeighborsRegressor
|  >>> neigh = KNeighborsRegressor(n_neighbors=2)
|  >>> neigh.fit(X, y)
|  KNeighborsRegressor(…)
|  >>> print(neigh.predict([[1.5]]))
|  [0.5]
|
|  See also
|  --------
|  NearestNeighbors
```

```
 |  RadiusNeighborsRegressor
 |  KNeighborsClassifier
 |  RadiusNeighborsClassifier
 |
 |  Notes
 |  -----
 |  See :ref:`Nearest Neighbors <neighbors>` in the online documentation
 |  for a discussion of the choice of ``algorithm`` and ``leaf_size``.
 |
 |  .. warning::
 |
 |     Regarding the Nearest Neighbors algorithms, if it is found that two
 |     neighbors, neighbor `k+1` and `k`, have identical distances but
 |     different labels, the results will depend on the ordering of the
 |     training data.
 |
 |  https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm
 |
 |  Method resolution order:
 |      KNeighborsRegressor
 |      sklearn.neighbors._base.NeighborsBase
 |      sklearn.base.MultiOutputMixin
 |      sklearn.base.BaseEstimator
 |      sklearn.neighbors._base.KNeighborsMixin
 |      sklearn.neighbors._base.SupervisedFloatMixin
 |      sklearn.base.RegressorMixin
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, n_neighbors=5, *, weights='uniform', algorithm='auto',
leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None,
**kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  predict(self, X)
 |      Predict the target for the provided data
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_queries, n_features),                 or
(n_queries, n_indexed) if metric == 'precomputed'
 |          Test samples.
 |
 |      Returns
 |      -------
 |      y : ndarray of shape (n_queries,) or (n_queries, n_outputs), dtype=int
 |          Target values.
```

```
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __abstractmethods__ = frozenset()
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from sklearn.base.MultiOutputMixin:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.base.BaseEstimator:
|
|  __getstate__(self)
|
|  __repr__(self, N_CHAR_MAX=700)
|      Return repr(self).
|
|  __setstate__(self, state)
|
|  get_params(self, deep=True)
|      Get parameters for this estimator.
|
|      Parameters
|      ----------
|      deep : bool, default=True
|          If True, will return the parameters for this estimator and
|          contained subobjects that are estimators.
|
|      Returns
|      -------
|      params : mapping of string to any
|          Parameter names mapped to their values.
|
|  set_params(self, **params)
|      Set the parameters of this estimator.
|
|      The method works on simple estimators as well as on nested objects
|      (such as pipelines). The latter have parameters of the form
|      ``<component>__<parameter>`` so that it's possible to update each
|      component of a nested object.
|
|      Parameters
```

```
 |      ----------
 |      **params : dict
 |          Estimator parameters.
 |
 |      Returns
 |      -------
 |      self : object
 |          Estimator instance.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.neighbors._base.KNeighborsMixin:
 |
 |  kneighbors(self, X=None, n_neighbors=None, return_distance=True)
 |      Finds the K-neighbors of a point.
 |      Returns indices of and distances to the neighbors of each point.
 |
 |      Parameters
 |      ----------
 |      X : array-like, shape (n_queries, n_features),                 or
(n_queries, n_indexed) if metric == 'precomputed'
 |          The query point or points.
 |          If not provided, neighbors of each indexed point are returned.
 |          In this case, the query point is not considered its own neighbor.
 |
 |      n_neighbors : int
 |          Number of neighbors to get (default is the value
 |          passed to the constructor).
 |
 |      return_distance : boolean, optional. Defaults to True.
 |          If False, distances will not be returned
 |
 |      Returns
 |      -------
 |      neigh_dist : array, shape (n_queries, n_neighbors)
 |          Array representing the lengths to points, only present if
 |          return_distance=True
 |
 |      neigh_ind : array, shape (n_queries, n_neighbors)
 |          Indices of the nearest points in the population matrix.
 |
 |      Examples
 |      --------
 |      In the following example, we construct a NearestNeighbors
 |      class from an array representing our data set and ask who's
 |      the closest point to [1,1,1]
 |
 |      >>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
 |      >>> from sklearn.neighbors import NearestNeighbors
```

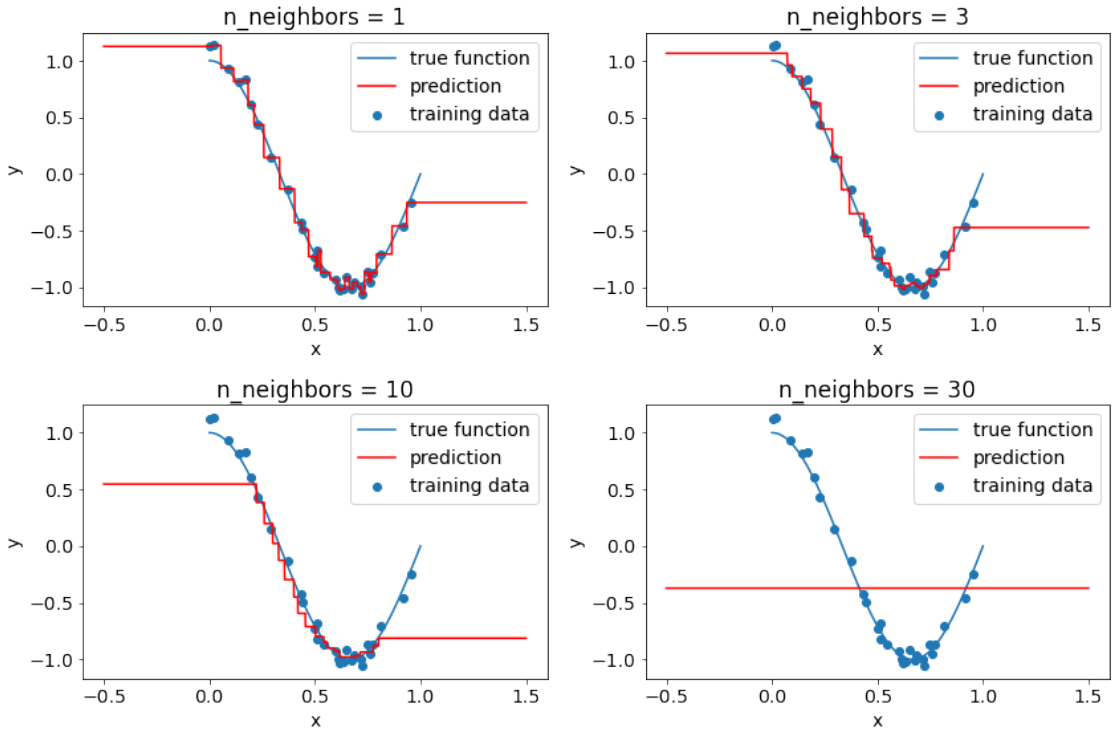
```
 |      >>> neigh = NearestNeighbors(n_neighbors=1)
 |      >>> neigh.fit(samples)
 |      NearestNeighbors(n_neighbors=1)
 |      >>> print(neigh.kneighbors([[1., 1., 1.]]))
 |      (array([[0.5]]), array([[2]]))
 |
 |      As you can see, it returns [[0.5]], and [[2]], which means that the
 |      element is at distance 0.5 and is the third element of samples
 |      (indexes start at 0). You can also query for multiple points:
 |
 |      >>> X = [[0., 1., 0.], [1., 0., 1.]]
 |      >>> neigh.kneighbors(X, return_distance=False)
 |      array([[1],
 |             [2]]…)
 |
 |  kneighbors_graph(self, X=None, n_neighbors=None, mode='connectivity')
 |      Computes the (weighted) graph of k-Neighbors for points in X
 |
 |      Parameters
 |      ----------
 |      X : array-like, shape (n_queries, n_features),                 or
(n_queries, n_indexed) if metric == 'precomputed'
 |          The query point or points.
 |          If not provided, neighbors of each indexed point are returned.
 |          In this case, the query point is not considered its own neighbor.
 |
 |      n_neighbors : int
 |          Number of neighbors for each sample.
 |          (default is value passed to the constructor).
 |
 |      mode : {'connectivity', 'distance'}, optional
 |          Type of returned matrix: 'connectivity' will return the
 |          connectivity matrix with ones and zeros, in 'distance' the
 |          edges are Euclidean distance between points.
 |
 |      Returns
 |      -------
 |      A : sparse graph in CSR format, shape = [n_queries, n_samples_fit]
 |          n_samples_fit is the number of samples in the fitted data
 |          A[i, j] is assigned the weight of edge that connects i to j.
 |
 |      Examples
 |      --------
 |      >>> X = [[0], [3], [1]]
 |      >>> from sklearn.neighbors import NearestNeighbors
 |      >>> neigh = NearestNeighbors(n_neighbors=2)
 |      >>> neigh.fit(X)
 |      NearestNeighbors(n_neighbors=2)
```

```
|      >>> A = neigh.kneighbors_graph(X)
|      >>> A.toarray()
|      array([[1., 0., 1.],
|             [0., 1., 1.],
|             [1., 0., 1.]])
|
|      See also
|      --------
|      NearestNeighbors.radius_neighbors_graph
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.neighbors._base.SupervisedFloatMixin:
|
|  fit(self, X, y)
|      Fit the model using X as training data and y as target values
|
|      Parameters
|      ----------
|      X : {array-like, sparse matrix, BallTree, KDTree}
|          Training data. If array or matrix, shape [n_samples, n_features],
|          or [n_samples, n_samples] if metric='precomputed'.
|
|      y : {array-like, sparse matrix}
|          Target values, array of float values, shape = [n_samples]
|           or [n_samples, n_outputs]
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.base.RegressorMixin:
|
|  score(self, X, y, sample_weight=None)
|      Return the coefficient of determination R^2 of the prediction.
|
|      The coefficient R^2 is defined as (1 - u/v), where u is the residual
|      sum of squares ((y_true - y_pred) ** 2).sum() and v is the total
|      sum of squares ((y_true - y_true.mean()) ** 2).sum().
|      The best possible score is 1.0 and it can be negative (because the
|      model can be arbitrarily worse). A constant model that always
|      predicts the expected value of y, disregarding the input features,
|      would get a R^2 score of 0.0.
|
|      Parameters
|      ----------
|      X : array-like of shape (n_samples, n_features)
|          Test samples. For some estimators this may be a
|          precomputed kernel matrix or a list of generic objects instead,
|          shape = (n_samples, n_samples_fitted),
|          where n_samples_fitted is the number of
|          samples used in the fitting for the estimator.
```

```
|
|       y : array-like of shape (n_samples,) or (n_samples, n_outputs)
|           True values for X.
|
|       sample_weight : array-like of shape (n_samples,), default=None
|           Sample weights.
|
|       Returns
|       -------
|       score : float
|           R^2 of self.predict(X) wrt. y.
|
|       Notes
|       -----
|       The R2 score used when calling ``score`` on a regressor uses
|       ``multioutput='uniform_average'`` from version 0.23 to keep consistent
|       with default value of :func:`~sklearn.metrics.r2_score`.
|       This influences the ``score`` method of all the multioutput
|       regressors (except for
|       :class:`~sklearn.multioutput.MultiOutputRegressor`).
```

[3]:
```
# let's train a couple of KNNs with various n_neighbors and uniform weight
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rcParams.update({'font.size': 14})


plt.figure(figsize=(12,8))

plt.subplot(2,2,1)
plt.scatter(X,y,label='training data')
plt.plot(np.linspace(0, 1, 100),true_fun(np.linspace(0, 1, 100)),label='true␣
↪function')
reg = KNeighborsRegressor(n_neighbors=1,weights='uniform')
reg.fit(X[:, np.newaxis],y)
y_new = reg.predict(X_new[:, np.newaxis])
plt.plot(X_new,y_new,'r',label='prediction')
plt.xlabel('x')
plt.ylabel('y')
plt.title('n_neighbors = 1')
plt.legend()

plt.subplot(2,2,2)
plt.scatter(X,y,label='training data')
plt.plot(np.linspace(0, 1, 100),true_fun(np.linspace(0, 1, 100)),label='true␣
↪function')
```

```
reg = KNeighborsRegressor(n_neighbors=3,weights='uniform')
reg.fit(X[:, np.newaxis],y)
y_new = reg.predict(X_new[:, np.newaxis])
plt.plot(X_new,y_new,'r',label='prediction')
plt.xlabel('x')
plt.ylabel('y')
plt.title('n_neighbors = 3')
plt.legend()

plt.subplot(2,2,3)
plt.scatter(X,y,label='training data')
plt.plot(np.linspace(0, 1, 100),true_fun(np.linspace(0, 1, 100)),label='true␣
 ↪function')
reg = KNeighborsRegressor(n_neighbors=10,weights='uniform')
reg.fit(X[:, np.newaxis],y)
y_new = reg.predict(X_new[:, np.newaxis])
plt.plot(X_new,y_new,'r',label='prediction')
plt.xlabel('x')
plt.ylabel('y')
plt.title('n_neighbors = 10')
plt.legend()


plt.subplot(2,2,4)
plt.scatter(X,y,label='training data')
plt.plot(np.linspace(0, 1, 100),true_fun(np.linspace(0, 1, 100)),label='true␣
 ↪function')
reg = KNeighborsRegressor(n_neighbors=30,weights='uniform')
reg.fit(X[:, np.newaxis],y)
y_new = reg.predict(X_new[:, np.newaxis])
plt.plot(X_new,y_new,'r',label='prediction')
plt.xlabel('x')
plt.ylabel('y')
plt.title('n_neighbors = 30')
plt.legend()

plt.tight_layout()
plt.savefig('figures/kneighbors_uni_reg.png',dpi=300)
plt.show()
```



1.4 KNN in classification

[4]:
```
# let's create a toy dataset with two features
from sklearn.datasets import make_moons

# create the data
X,y = make_moons(noise=0.2, random_state=1,n_samples=200)
```

[5]:
```
from sklearn.neighbors import KNeighborsClassifier
help(KNeighborsClassifier)
```

```
Help on class KNeighborsClassifier in module sklearn.neighbors._classification:

class KNeighborsClassifier(sklearn.neighbors._base.NeighborsBase,
sklearn.neighbors._base.KNeighborsMixin,
sklearn.neighbors._base.SupervisedIntegerMixin, sklearn.base.ClassifierMixin)
 |  KNeighborsClassifier(n_neighbors=5, *, weights='uniform', algorithm='auto',
leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None,
**kwargs)
 |
 |  Classifier implementing the k-nearest neighbors vote.
 |
 |  Read more in the :ref:`User Guide <classification>`.
 |
```

```
|  Parameters
|  ----------
|  n_neighbors : int, default=5
|      Number of neighbors to use by default for :meth:`kneighbors` queries.
|
|  weights : {'uniform', 'distance'} or callable, default='uniform'
|      weight function used in prediction.  Possible values:
|
|      - 'uniform' : uniform weights.  All points in each neighborhood
|        are weighted equally.
|      - '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.
|      - [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'}, default='auto'
|      Algorithm used to compute the nearest neighbors:
|
|      - 'ball_tree' will use :class:`BallTree`
|      - 'kd_tree' will use :class:`KDTree`
|      - 'brute' will use a brute-force search.
|      - 'auto' will attempt to decide the most appropriate algorithm
|        based on the values passed to :meth:`fit` method.
|
|      Note: fitting on sparse input will override the setting of
|      this parameter, using brute force.
|
|  leaf_size : int, 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 : int, default=2
|      Power parameter for the Minkowski metric. When p = 1, this is
|      equivalent to using manhattan_distance (l1), and euclidean_distance
|      (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.
|
|  metric : str or callable, default='minkowski'
|      the distance metric to use for the tree.  The default metric is
|      minkowski, and with p=2 is equivalent to the standard Euclidean
|      metric. See the documentation of :class:`DistanceMetric` for a
|      list of available metrics.
|      If metric is "precomputed", X is assumed to be a distance matrix and
|      must be square during fit. X may be a :term:`sparse graph`,
|      in which case only "nonzero" elements may be considered neighbors.
```

```
|
|  metric_params : dict, default=None
|      Additional keyword arguments for the metric function.
|
|  n_jobs : int, default=None
|      The number of parallel jobs to run for neighbors search.
|      ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
|      ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
|      for more details.
|      Doesn't affect :meth:`fit` method.
|
|  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'.
|
|  outputs_2d_ : bool
|      False when `y`'s shape is (n_samples, ) or (n_samples, 1) during fit
|      otherwise True.
|
|  Examples
|  --------
|  >>> 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]]))
|  [[0.66666667 0.33333333]]
|
|  See also
|  --------
|  RadiusNeighborsClassifier
|  KNeighborsRegressor
```

```
 |  RadiusNeighborsRegressor
 |  NearestNeighbors
 |
 |  Notes
 |  -----
 |  See :ref:`Nearest Neighbors <neighbors>` in the online documentation
 |  for a discussion of the choice of ``algorithm`` and ``leaf_size``.
 |
 |  .. warning::
 |
 |     Regarding the Nearest Neighbors algorithms, if it is found that two
 |     neighbors, neighbor `k+1` and `k`, have identical distances
 |     but different labels, the results will depend on the ordering of the
 |     training data.
 |
 |  https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm
 |
 |  Method resolution order:
 |      KNeighborsClassifier
 |      sklearn.neighbors._base.NeighborsBase
 |      sklearn.base.MultiOutputMixin
 |      sklearn.base.BaseEstimator
 |      sklearn.neighbors._base.KNeighborsMixin
 |      sklearn.neighbors._base.SupervisedIntegerMixin
 |      sklearn.base.ClassifierMixin
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, n_neighbors=5, *, weights='uniform', algorithm='auto',
leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None,
**kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  predict(self, X)
 |      Predict the class labels for the provided data.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_queries, n_features),                 or
(n_queries, n_indexed) if metric == 'precomputed'
 |          Test samples.
 |
 |      Returns
 |      -------
 |      y : ndarray of shape (n_queries,) or (n_queries, n_outputs)
 |          Class labels for each data sample.
 |
```

```
 |  predict_proba(self, X)
 |      Return probability estimates for the test data X.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_queries, n_features),                 or
(n_queries, n_indexed) if metric == 'precomputed'
 |          Test samples.
 |
 |      Returns
 |      -------
 |      p : ndarray of shape (n_queries, n_classes), or a list of n_outputs
 |          of such arrays if n_outputs > 1.
 |          The class probabilities of the input samples. Classes are ordered
 |          by lexicographic order.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __abstractmethods__ = frozenset()
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.MultiOutputMixin:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |
 |      Parameters
 |      ----------
 |      deep : bool, default=True
 |          If True, will return the parameters for this estimator and
 |          contained subobjects that are estimators.
```

```
 |
 |      Returns
 |      -------
 |      params : mapping of string to any
 |          Parameter names mapped to their values.
 |
 |  set_params(self, **params)
 |      Set the parameters of this estimator.
 |
 |      The method works on simple estimators as well as on nested objects
 |      (such as pipelines). The latter have parameters of the form
 |      ``<component>__<parameter>`` so that it's possible to update each
 |      component of a nested object.
 |
 |      Parameters
 |      ----------
 |      **params : dict
 |          Estimator parameters.
 |
 |      Returns
 |      -------
 |      self : object
 |          Estimator instance.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.neighbors._base.KNeighborsMixin:
 |
 |  kneighbors(self, X=None, n_neighbors=None, return_distance=True)
 |      Finds the K-neighbors of a point.
 |      Returns indices of and distances to the neighbors of each point.
 |
 |      Parameters
 |      ----------
 |      X : array-like, shape (n_queries, n_features),                or
(n_queries, n_indexed) if metric == 'precomputed'
 |          The query point or points.
 |          If not provided, neighbors of each indexed point are returned.
 |          In this case, the query point is not considered its own neighbor.
 |
 |      n_neighbors : int
 |          Number of neighbors to get (default is the value
 |          passed to the constructor).
 |
 |      return_distance : boolean, optional. Defaults to True.
 |          If False, distances will not be returned
 |
 |      Returns
 |      -------
```

```
 |      neigh_dist : array, shape (n_queries, n_neighbors)
 |          Array representing the lengths to points, only present if
 |          return_distance=True
 |
 |      neigh_ind : array, shape (n_queries, n_neighbors)
 |          Indices of the nearest points in the population matrix.
 |
 |      Examples
 |      --------
 |      In the following example, we construct a NearestNeighbors
 |      class from an array representing our data set and ask who's
 |      the closest point to [1,1,1]
 |
 |      >>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
 |      >>> from sklearn.neighbors import NearestNeighbors
 |      >>> neigh = NearestNeighbors(n_neighbors=1)
 |      >>> neigh.fit(samples)
 |      NearestNeighbors(n_neighbors=1)
 |      >>> print(neigh.kneighbors([[1., 1., 1.]]))
 |      (array([[0.5]]), array([[2]]))
 |
 |      As you can see, it returns [[0.5]], and [[2]], which means that the
 |      element is at distance 0.5 and is the third element of samples
 |      (indexes start at 0). You can also query for multiple points:
 |
 |      >>> X = [[0., 1., 0.], [1., 0., 1.]]
 |      >>> neigh.kneighbors(X, return_distance=False)
 |      array([[1],
 |             [2]]…)
 |
 |  kneighbors_graph(self, X=None, n_neighbors=None, mode='connectivity')
 |      Computes the (weighted) graph of k-Neighbors for points in X
 |
 |      Parameters
 |      ----------
 |      X : array-like, shape (n_queries, n_features),                 or
(n_queries, n_indexed) if metric == 'precomputed'
 |          The query point or points.
 |          If not provided, neighbors of each indexed point are returned.
 |          In this case, the query point is not considered its own neighbor.
 |
 |      n_neighbors : int
 |          Number of neighbors for each sample.
 |          (default is value passed to the constructor).
 |
 |      mode : {'connectivity', 'distance'}, optional
 |          Type of returned matrix: 'connectivity' will return the
 |          connectivity matrix with ones and zeros, in 'distance' the
```

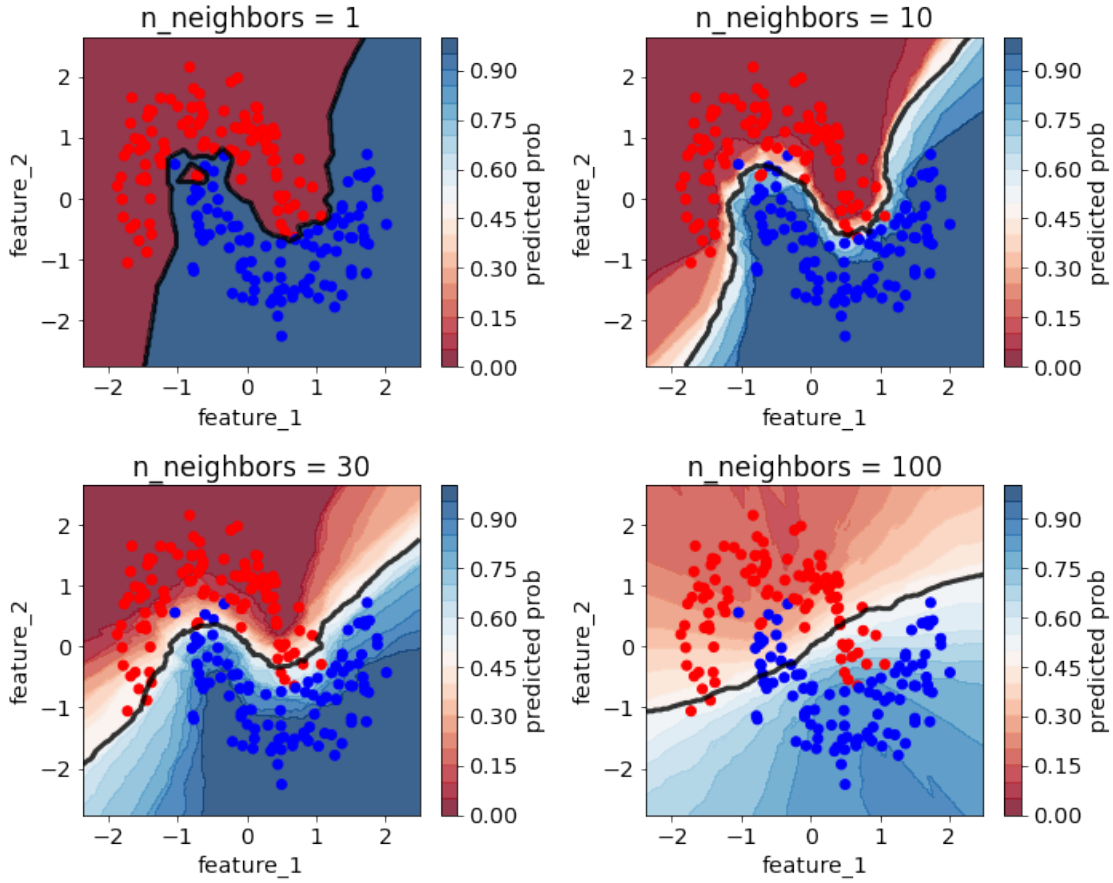
```
|          edges are Euclidean distance between points.
|
|      Returns
|      -------
|      A : sparse graph in CSR format, shape = [n_queries, n_samples_fit]
|          n_samples_fit is the number of samples in the fitted data
|          A[i, j] is assigned the weight of edge that connects i to j.
|
|      Examples
|      --------
|      >>> X = [[0], [3], [1]]
|      >>> from sklearn.neighbors import NearestNeighbors
|      >>> neigh = NearestNeighbors(n_neighbors=2)
|      >>> neigh.fit(X)
|      NearestNeighbors(n_neighbors=2)
|      >>> A = neigh.kneighbors_graph(X)
|      >>> A.toarray()
|      array([[1., 0., 1.],
|             [0., 1., 1.],
|             [1., 0., 1.]])
|
|      See also
|      --------
|      NearestNeighbors.radius_neighbors_graph
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.neighbors._base.SupervisedIntegerMixin:
|
|  fit(self, X, y)
|      Fit the model using X as training data and y as target values
|
|      Parameters
|      ----------
|      X : {array-like, sparse matrix, BallTree, KDTree}
|          Training data. If array or matrix, shape [n_samples, n_features],
|          or [n_samples, n_samples] if metric='precomputed'.
|
|      y : {array-like, sparse matrix}
|          Target values of shape = [n_samples] or [n_samples, n_outputs]
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.base.ClassifierMixin:
|
|  score(self, X, y, sample_weight=None)
|      Return the mean accuracy on the given test data and labels.
|
|      In multi-label classification, this is the subset accuracy
|      which is a harsh metric since you require for each sample that
```

```
|       each label set be correctly predicted.
|
|       Parameters
|       ----------
|       X : array-like of shape (n_samples, n_features)
|           Test samples.
|
|       y : array-like of shape (n_samples,) or (n_samples, n_outputs)
|           True labels for X.
|
|       sample_weight : array-like of shape (n_samples,), default=None
|           Sample weights.
|
|       Returns
|       -------
|       score : float
|           Mean accuracy of self.predict(X) wrt. y.
```

[6]:
```
from matplotlib.colors import ListedColormap
from sklearn.preprocessing import StandardScaler

matplotlib.rcParams.update({'font.size': 14})

X = StandardScaler().fit_transform(X)

h = .02  # step size in the mesh

x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

plt.figure(figsize=(10,8))
cm_bright = ListedColormap(['#FF0000', '#0000FF'])
cm = plt.cm.RdBu

plt.subplot(2,2,1)
clf = KNeighborsClassifier(n_neighbors = 1)
clf.fit(X,y)
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cm, alpha=.8,vmin=0,vmax=1,levels=np.arange(0,1.
 ↪05,0.05))
plt.colorbar(label='predicted prob')
```

```
plt.contour(xx, yy, Z, alpha=.8,vmin=0,vmax=1,levels=[0.
 ↪5],colors=['k'],linewidths=3)
plt.scatter(X[:, 0], X[:, 1], c=y,cmap=cm_bright)
plt.xlabel('feature_1')
plt.ylabel('feature_2')
plt.title('n_neighbors = 1')

plt.subplot(2,2,2)
clf = KNeighborsClassifier(n_neighbors = 10)
clf.fit(X,y)
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cm, alpha=.8,vmin=0,vmax=1,levels=np.arange(0,1.
 ↪05,0.05))
plt.colorbar(label='predicted prob')
plt.contour(xx, yy, Z, alpha=.8,vmin=0,vmax=1,levels=[0.
 ↪5],colors=['k'],linewidths=3)
plt.scatter(X[:, 0], X[:, 1], c=y,cmap=cm_bright)
plt.xlabel('feature_1')
plt.ylabel('feature_2')
plt.title('n_neighbors = 10')

plt.subplot(2,2,3)
clf = KNeighborsClassifier(n_neighbors = 30)
clf.fit(X,y)
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cm, alpha=.8,vmin=0,vmax=1,levels=np.arange(0,1.
 ↪05,0.05))
plt.colorbar(label='predicted prob')
plt.contour(xx, yy, Z, alpha=.8,vmin=0,vmax=1,levels=[0.
 ↪5],colors=['k'],linewidths=3)
plt.scatter(X[:, 0], X[:, 1], c=y,cmap=cm_bright)
plt.xlabel('feature_1')
plt.ylabel('feature_2')
plt.title('n_neighbors = 30')

plt.subplot(2,2,4)
clf = KNeighborsClassifier(n_neighbors = 100)
clf.fit(X,y)
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cm, alpha=.8,vmin=0,vmax=1,levels=np.arange(0,1.
 ↪05,0.05))
```

```
plt.colorbar(label='predicted prob')
plt.contour(xx, yy, Z, alpha=.8,vmin=0,vmax=1,levels=[0.
 ↪5],colors=['k'],linewidths=3)
plt.scatter(X[:, 0], X[:, 1], c=y,cmap=cm_bright)
plt.xlabel('feature_1')
plt.ylabel('feature_2')
plt.title('n_neighbors = 100')


plt.tight_layout()

plt.savefig('figures/kneighbors_uni_clf.png',dpi=300)
plt.show()
```



1.5 KNN notes

- it works best if the number of features is much smaller than the number of points
- it is OK to not try this ML algorithm if the number of features is similar or larger than the number of points

- versatile technique and it can capture complex non-linearities
- the prediction is not a smoothly varying function of the features
- the CPU training time is OK on large datasets
- in regression, outliers are reasonably extrapolated

2 Module 2: Support Vector Machines

2.0.1 Learning objectives of this module:

- Summarize how SVM works
- Describe which hyperparameters need to be tuned and what range the values should have
- Apply the algorithms in regression and classification
- Visualize the predictions of toy datasets
- Summarize under what circumstances a certain algorithm is expected to perform well or poorly and why

2.1 Support Vector Machine

- very versatile technique, it comes in lots of flavors/types, read more about it here
- SVM classifier motivation
  - points in n dimensional space with class 0 and 1
  - we want to find the (n-1) dimensional hyperplane that best separates the points
  - this hyperplane is our (linear) decision boundary
- we cover SVMs with radial basis functions (rbf)
  - we apply a kernel function (a non-linear transformation) to the data points
  - the kernel function basically "smears" the points
  - gaussian rbf kernel: $\exp(-\gamma(|x - x'|)^2)$ where $\gamma > 0$

2.2 SVR

[7]:
```
import numpy as np
np.random.seed(10)
def true_fun(X):
    return np.cos(1.5 * np.pi * X)

n_samples = 30

X = np.random.rand(n_samples)
y = true_fun(X) + np.random.randn(n_samples) * 0.1

X_new = np.linspace(-0.5, 1.5, 2000)
```

[8]:
```
from sklearn.svm import SVR
help(SVR)
```

```
Help on class SVR in module sklearn.svm._classes:

class SVR(sklearn.base.RegressorMixin, sklearn.svm._base.BaseLibSVM)
```

```
 |  SVR(*, kernel='rbf', degree=3, gamma='scale', coef0=0.0, tol=0.001, C=1.0,
epsilon=0.1, shrinking=True, cache_size=200, verbose=False, max_iter=-1)
 |
 |  Epsilon-Support Vector Regression.
 |
 |  The free parameters in the model are C and epsilon.
 |
 |  The implementation is based on libsvm. The fit time complexity
 |  is more than quadratic with the number of samples which makes it hard
 |  to scale to datasets with more than a couple of 10000 samples. For large
 |  datasets consider using :class:`sklearn.svm.LinearSVR` or
 |  :class:`sklearn.linear_model.SGDRegressor` instead, possibly after a
 |  :class:`sklearn.kernel_approximation.Nystroem` transformer.
 |
 |  Read more in the :ref:`User Guide <svm_regression>`.
 |
 |  Parameters
 |  ----------
 |  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 precompute the kernel matrix.
 |
 |  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.
 |
 |      .. versionchanged:: 0.22
 |         The default value of ``gamma`` changed from 'auto' to 'scale'.
 |
 |  coef0 : float, default=0.0
 |      Independent term in kernel function.
 |      It is only significant in 'poly' and 'sigmoid'.
 |
 |  tol : float, default=1e-3
 |      Tolerance for stopping criterion.
 |
 |  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.
|
|  epsilon : float, default=0.1
|       Epsilon in the epsilon-SVR model. It specifies the epsilon-tube
|       within which no penalty is associated in the training loss function
|       with points predicted within a distance epsilon from the actual
|       value.
|
|  shrinking : bool, default=True
|      Whether to use the shrinking heuristic.
|      See the :ref:`User Guide <shrinking_svm>`.
|
|  cache_size : float, default=200
|      Specify the size of the kernel cache (in MB).
|
|  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.
|
|  max_iter : int, default=-1
|      Hard limit on iterations within solver, or -1 for no limit.
|
|  Attributes
|  ----------
|  support_ : ndarray of shape (n_SV,)
|      Indices of support vectors.
|
|  support_vectors_ : ndarray of shape (n_SV, n_features)
|      Support vectors.
|
|  dual_coef_ : ndarray of shape (1, n_SV)
|      Coefficients of the support vector in the decision function.
|
|  coef_ : ndarray of shape (1, n_features)
|      Weights assigned to the features (coefficients in the primal
|      problem). This is only available in the case of a linear kernel.
|
|      `coef_` is readonly property derived from `dual_coef_` and
|      `support_vectors_`.
|
|  fit_status_ : int
|      0 if correctly fitted, 1 otherwise (will raise warning)
|
|  intercept_ : ndarray of shape (1,)
|      Constants in decision function.
|
```

```
 |  Examples
 |  --------
 |  >>> from sklearn.svm import SVR
 |  >>> from sklearn.pipeline import make_pipeline
 |  >>> from sklearn.preprocessing import StandardScaler
 |  >>> import numpy as np
 |  >>> n_samples, n_features = 10, 5
 |  >>> rng = np.random.RandomState(0)
 |  >>> y = rng.randn(n_samples)
 |  >>> X = rng.randn(n_samples, n_features)
 |  >>> regr = make_pipeline(StandardScaler(), SVR(C=1.0, epsilon=0.2))
 |  >>> regr.fit(X, y)
 |  Pipeline(steps=[('standardscaler', StandardScaler()),
 |                  ('svr', SVR(epsilon=0.2))])
 |
 |
 |  See also
 |  --------
 |  NuSVR
 |      Support Vector Machine for regression implemented using libsvm
 |      using a parameter to control the number of support vectors.
 |
 |  LinearSVR
 |      Scalable Linear Support Vector Machine for regression
 |      implemented using liblinear.
 |
 |  Notes
 |  -----
 |  **References:**
 |  `LIBSVM: A Library for Support Vector Machines
 |  <http://www.csie.ntu.edu.tw/~cjlin/papers/libsvm.pdf>`__
 |
 |  Method resolution order:
 |      SVR
 |      sklearn.base.RegressorMixin
 |      sklearn.svm._base.BaseLibSVM
 |      sklearn.base.BaseEstimator
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, *, kernel='rbf', degree=3, gamma='scale', coef0=0.0,
tol=0.001, C=1.0, epsilon=0.1, shrinking=True, cache_size=200, verbose=False,
max_iter=-1)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
```

```
|
|  probA_
|
|  probB_
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __abstractmethods__ = frozenset()
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.base.RegressorMixin:
|
|  score(self, X, y, sample_weight=None)
|      Return the coefficient of determination R^2 of the prediction.
|
|      The coefficient R^2 is defined as (1 - u/v), where u is the residual
|      sum of squares ((y_true - y_pred) ** 2).sum() and v is the total
|      sum of squares ((y_true - y_true.mean()) ** 2).sum().
|      The best possible score is 1.0 and it can be negative (because the
|      model can be arbitrarily worse). A constant model that always
|      predicts the expected value of y, disregarding the input features,
|      would get a R^2 score of 0.0.
|
|      Parameters
|      ----------
|      X : array-like of shape (n_samples, n_features)
|          Test samples. For some estimators this may be a
|          precomputed kernel matrix or a list of generic objects instead,
|          shape = (n_samples, n_samples_fitted),
|          where n_samples_fitted is the number of
|          samples used in the fitting for the estimator.
|
|      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
|          True values for X.
|
|      sample_weight : array-like of shape (n_samples,), default=None
|          Sample weights.
|
|      Returns
|      -------
|      score : float
|          R^2 of self.predict(X) wrt. y.
|
|      Notes
|      -----
|      The R2 score used when calling ``score`` on a regressor uses
|      ``multioutput='uniform_average'`` from version 0.23 to keep consistent
```

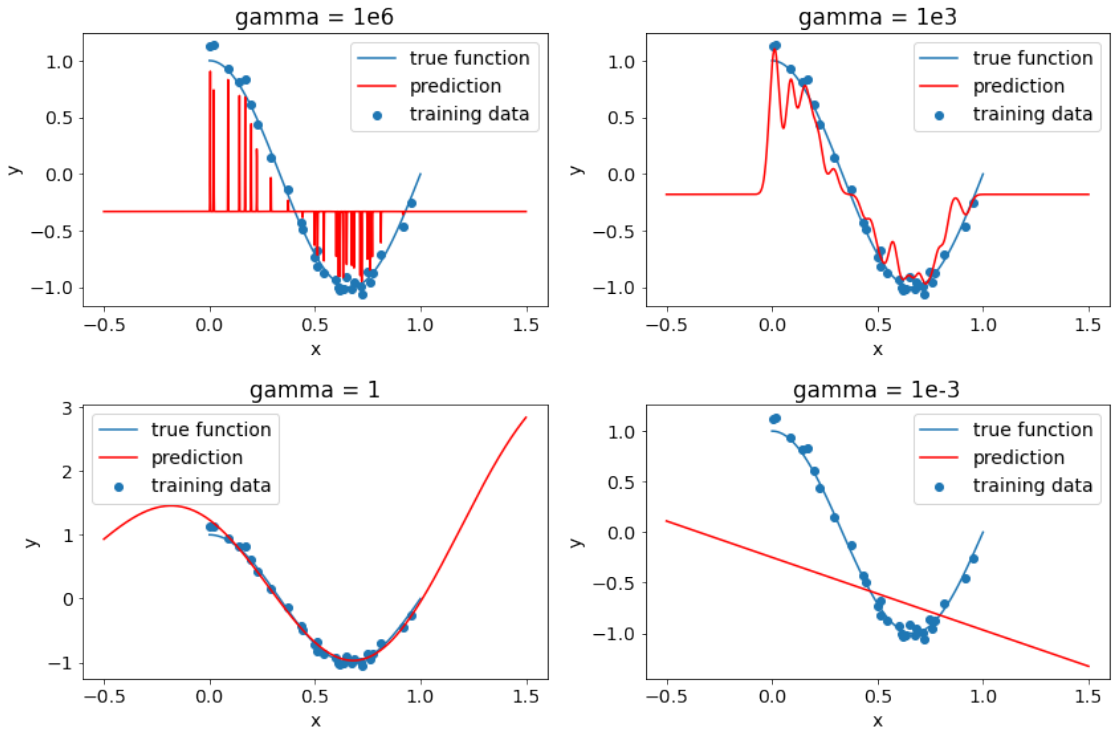
```
 |      with default value of :func:`~sklearn.metrics.r2_score`.
 |      This influences the ``score`` method of all the multioutput
 |      regressors (except for
 |      :class:`~sklearn.multioutput.MultiOutputRegressor`).
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.RegressorMixin:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.svm._base.BaseLibSVM:
 |
 |  fit(self, X, y, sample_weight=None)
 |      Fit the SVM model according to the given training data.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features)
or (n_samples, n_samples)
 |          Training vectors, where n_samples is the number of samples
 |          and n_features is the number of features.
 |          For kernel="precomputed", the expected shape of X is
 |          (n_samples, n_samples).
 |
 |      y : array-like of shape (n_samples,)
 |          Target values (class labels in classification, real numbers in
 |          regression)
 |
 |      sample_weight : array-like of shape (n_samples,), default=None
 |          Per-sample weights. Rescale C per sample. Higher weights
 |          force the classifier to put more emphasis on these points.
 |
 |      Returns
 |      -------
 |      self : object
 |
 |      Notes
 |      -----
 |      If X and y are not C-ordered and contiguous arrays of np.float64 and
 |      X is not a scipy.sparse.csr_matrix, X and/or y may be copied.
 |
 |      If X is a dense array, then the other methods will not support sparse
 |      matrices as input.
```

```
|
|  predict(self, X)
|      Perform regression on samples in X.
|
|      For an one-class model, +1 (inlier) or -1 (outlier) is returned.
|
|      Parameters
|      ----------
|      X : {array-like, sparse matrix} of shape (n_samples, n_features)
|          For kernel="precomputed", the expected shape of X is
|          (n_samples_test, n_samples_train).
|
|      Returns
|      -------
|      y_pred : ndarray of shape (n_samples,)
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from sklearn.svm._base.BaseLibSVM:
|
|  coef_
|
|  n_support_
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.base.BaseEstimator:
|
|  __getstate__(self)
|
|  __repr__(self, N_CHAR_MAX=700)
|      Return repr(self).
|
|  __setstate__(self, state)
|
|  get_params(self, deep=True)
|      Get parameters for this estimator.
|
|      Parameters
|      ----------
|      deep : bool, default=True
|          If True, will return the parameters for this estimator and
|          contained subobjects that are estimators.
|
|      Returns
|      -------
|      params : mapping of string to any
|          Parameter names mapped to their values.
|
|  set_params(self, **params)
```

```
|      Set the parameters of this estimator.
|
|      The method works on simple estimators as well as on nested objects
|      (such as pipelines). The latter have parameters of the form
|      ``<component>__<parameter>`` so that it's possible to update each
|      component of a nested object.
|
|      Parameters
|      ----------
|      **params : dict
|          Estimator parameters.
|
|      Returns
|      -------
|      self : object
|          Estimator instance.
```

[9]:
```
matplotlib.rcParams.update({'font.size': 14})


plt.figure(figsize=(12,8))

plt.subplot(2,2,1)
plt.scatter(X,y,label='training data')
plt.plot(np.linspace(0, 1, 100),true_fun(np.linspace(0, 1, 100)),label='true␣
 ↪function')
reg = SVR(gamma = 1000000, C = 100)
reg.fit(X[:, np.newaxis],y)
y_new = reg.predict(X_new[:, np.newaxis])
plt.plot(X_new,y_new,'r',label='prediction')
plt.xlabel('x')
plt.ylabel('y')
plt.title('gamma = 1e6')
plt.legend()

plt.subplot(2,2,2)
plt.scatter(X,y,label='training data')
plt.plot(np.linspace(0, 1, 100),true_fun(np.linspace(0, 1, 100)),label='true␣
 ↪function')
reg = SVR(gamma = 1000, C = 100)
reg.fit(X[:, np.newaxis],y)
y_new = reg.predict(X_new[:, np.newaxis])
plt.plot(X_new,y_new,'r',label='prediction')
plt.xlabel('x')
plt.ylabel('y')
plt.title('gamma = 1e3')
```

```
plt.legend()

plt.subplot(2,2,3)
plt.scatter(X,y,label='training data')
plt.plot(np.linspace(0, 1, 100),true_fun(np.linspace(0, 1, 100)),label='true␣
 ↪function')
reg = SVR(gamma = 1, C = 100)
reg.fit(X[:, np.newaxis],y)
y_new = reg.predict(X_new[:, np.newaxis])
plt.plot(X_new,y_new,'r',label='prediction')
plt.xlabel('x')
plt.ylabel('y')
plt.title('gamma = 1')
plt.legend()


plt.subplot(2,2,4)
plt.scatter(X,y,label='training data')
plt.plot(np.linspace(0, 1, 100),true_fun(np.linspace(0, 1, 100)),label='true␣
 ↪function')
reg = SVR(gamma = 0.001, C = 100)
reg.fit(X[:, np.newaxis],y)
y_new = reg.predict(X_new[:, np.newaxis])
plt.plot(X_new,y_new,'r',label='prediction')
plt.xlabel('x')
plt.ylabel('y')
plt.title('gamma = 1e-3')
plt.legend()

plt.tight_layout()
plt.savefig('figures/SVM_reg.png',dpi=300)
plt.show()
```



2.3 SVC

[10]:
```
from sklearn.datasets import make_moons

# create the data
X,y = make_moons(noise=0.2, random_state=1,n_samples=200)
```

[11]:
```
from sklearn.svm import SVC
help(SVC)
```

```
Help on class SVC in module sklearn.svm._classes:

class SVC(sklearn.svm._base.BaseSVC)
 |  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)
 |
 |  C-Support Vector Classification.
 |
 |  The implementation is based on libsvm. The fit time scales at least
 |  quadratically with the number of samples and may be impractical
 |  beyond tens of thousands of samples. For large datasets
 |  consider using :class:`sklearn.svm.LinearSVC` or
```

```
|  :class:`sklearn.linear_model.SGDClassifier` instead, possibly after a
|  :class:`sklearn.kernel_approximation.Nystroem` transformer.
|
|  The multiclass support is handled according to a one-vs-one scheme.
|
|  For details on the precise mathematical formulation of the provided
|  kernel functions and how `gamma`, `coef0` and `degree` affect each
|  other, see the corresponding section in the narrative documentation:
|  :ref:`svm_kernels`.
|
|  Read more in the :ref:`User Guide <svm_classification>`.
|
|  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.
|
|      .. versionchanged:: 0.22
|         The default value of ``gamma`` changed from 'auto' to 'scale'.
|
|  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 :ref:`User Guide <shrinking_svm>`.
```

```
|
|  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 :ref:`User Guide <scores_probabilities>`.
|
|  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.
|
|  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.
|
|      .. versionchanged:: 0.19
|          decision_function_shape is 'ovr' by default.
|
|      .. versionadded:: 0.17
|         *decision_function_shape='ovr'* is recommended.
|
|      .. versionchanged:: 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,
```

```
|      :term:`predict` will break ties according to the confidence values of
|      :term:`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.
|
|      .. versionadded:: 0.22
|
|  random_state : int or RandomState instance, 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 :term:`Glossary <random_state>`.
|
|  Attributes
|  ----------
|  support_ : ndarray of shape (n_SV,)
|      Indices of support vectors.
|
|  support_vectors_ : ndarray of shape (n_SV, n_features)
|      Support vectors.
|
|  n_support_ : ndarray of shape (n_class,), dtype=int32
|      Number of support vectors for each class.
|
|  dual_coef_ : ndarray of shape (n_class-1, n_SV)
|      Dual coefficients of the support vector in the decision
|      function (see :ref:`sgd_mathematical_formulation`), multiplied by
|      their targets.
|      For multiclass, coefficient for all 1-vs-1 classifiers.
|      The layout of the coefficients in the multiclass case is somewhat
|      non-trivial. See the :ref:`multi-class section of the User Guide
|      <svm_multi_class>` for details.
|
|  coef_ : ndarray of shape (n_class * (n_class-1) / 2, n_features)
|      Weights assigned to the features (coefficients in the primal
|      problem). This is only available in the case of a linear kernel.
|
|      `coef_` is a readonly property derived from `dual_coef_` and
|      `support_vectors_`.
|
|  intercept_ : ndarray of shape (n_class * (n_class-1) / 2,)
|      Constants in decision function.
|
|  fit_status_ : int
|      0 if correctly fitted, 1 otherwise (will raise warning)
|
|  classes_ : ndarray of shape (n_classes,)
|      The classes labels.
```

```
|
|  probA_ : ndarray of shape (n_class * (n_class-1) / 2)
|  probB_ : ndarray of shape (n_class * (n_class-1) / 2)
|      If `probability=True`, it corresponds to the parameters learned in
|      Platt scaling to produce probability estimates from decision values.
|      If `probability=False`, it's an empty array. Platt scaling uses the
|      logistic function
|      ``1 / (1 + exp(decision_value * probA_ + probB_))``
|      where ``probA_`` and ``probB_`` are learned from the dataset [2]_. For
|      more information on the multiclass case and training procedure see
|      section 8 of [1]_.
|
|  class_weight_ : ndarray of shape (n_class,)
|      Multipliers of parameter C for each class.
|      Computed based on the ``class_weight`` parameter.
|
|  shape_fit_ : tuple of int of shape (n_dimensions_of_X,)
|      Array dimensions of training vector ``X``.
|
|  Examples
|  --------
|  >>> import numpy as np
|  >>> from sklearn.pipeline import make_pipeline
|  >>> from sklearn.preprocessing import StandardScaler
|  >>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
|  >>> y = np.array([1, 1, 2, 2])
|  >>> from sklearn.svm import SVC
|  >>> clf = make_pipeline(StandardScaler(), SVC(gamma='auto'))
|  >>> clf.fit(X, y)
|  Pipeline(steps=[('standardscaler', StandardScaler()),
|                  ('svc', SVC(gamma='auto'))])
|
|  >>> print(clf.predict([[-0.8, -1]]))
|  [1]
|
|  See also
|  --------
|  SVR
|      Support Vector Machine for Regression implemented using libsvm.
|
|  LinearSVC
|      Scalable Linear Support Vector Machine for classification
|      implemented using liblinear. Check the See also section of
|      LinearSVC for more comparison element.
|
|  References
|  ----------
|  .. [1] `LIBSVM: A Library for Support Vector Machines
```

```
 |      <http://www.csie.ntu.edu.tw/~cjlin/papers/libsvm.pdf>`_
 |
 |  .. [2] `Platt, John (1999). "Probabilistic outputs for support vector
 |      machines and comparison to regularizedlikelihood methods."
 |      <http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.1639>`_
 |
 |  Method resolution order:
 |      SVC
 |      sklearn.svm._base.BaseSVC
 |      sklearn.base.ClassifierMixin
 |      sklearn.svm._base.BaseLibSVM
 |      sklearn.base.BaseEstimator
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, *, 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)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __abstractmethods__ = frozenset()
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.svm._base.BaseSVC:
 |
 |  decision_function(self, X)
 |      Evaluates the decision function for the samples in X.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |
 |      Returns
 |      -------
 |      X : ndarray of shape (n_samples, n_classes * (n_classes-1) / 2)
 |          Returns the decision function of the sample for each class
 |          in the model.
 |          If decision_function_shape='ovr', the shape is (n_samples,
 |          n_classes).
 |
 |      Notes
 |      -----
 |      If decision_function_shape='ovo', the function values are proportional
```

```
 |      to the distance of the samples X to the separating hyperplane. If the
 |      exact distances are required, divide the function values by the norm of
 |      the weight vector (``coef_``). See also `this question
 |      <https://stats.stackexchange.com/questions/14876/
 |      interpreting-distance-from-hyperplane-in-svm>`_ for further details.
 |      If decision_function_shape='ovr', the decision function is a monotonic
 |      transformation of ovo decision function.
 |
 |  predict(self, X)
 |      Perform classification on samples in X.
 |
 |      For an one-class model, +1 or -1 is returned.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features) or
(n_samples_test, n_samples_train)
 |          For kernel="precomputed", the expected shape of X is
 |          (n_samples_test, n_samples_train).
 |
 |      Returns
 |      -------
 |      y_pred : ndarray of shape (n_samples,)
 |          Class labels for samples in X.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.svm._base.BaseSVC:
 |
 |  predict_log_proba
 |      Compute log probabilities of possible outcomes for samples in X.
 |
 |      The model need to have probability information computed at training
 |      time: fit with attribute `probability` set to True.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features) or
(n_samples_test, n_samples_train)
 |          For kernel="precomputed", the expected shape of X is
 |          (n_samples_test, n_samples_train).
 |
 |      Returns
 |      -------
 |      T : ndarray of shape (n_samples, n_classes)
 |          Returns the log-probabilities of the sample for each class in
 |          the model. The columns correspond to the classes in sorted
 |          order, as they appear in the attribute :term:`classes_`.
 |
```

```
|      Notes
|      -----
|      The probability model is created using cross validation, so
|      the results can be slightly different than those obtained by
|      predict. Also, it will produce meaningless results on very small
|      datasets.
|
|  predict_proba
|      Compute probabilities of possible outcomes for samples in X.
|
|      The model need to have probability information computed at training
|      time: fit with attribute `probability` set to True.
|
|      Parameters
|      ----------
|      X : array-like of shape (n_samples, n_features)
|          For kernel="precomputed", the expected shape of X is
|          [n_samples_test, n_samples_train]
|
|      Returns
|      -------
|      T : ndarray of shape (n_samples, n_classes)
|          Returns the probability of the sample for each class in
|          the model. The columns correspond to the classes in sorted
|          order, as they appear in the attribute :term:`classes_`.
|
|      Notes
|      -----
|      The probability model is created using cross validation, so
|      the results can be slightly different than those obtained by
|      predict. Also, it will produce meaningless results on very small
|      datasets.
|
|  probA_
|
|  probB_
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.base.ClassifierMixin:
|
|  score(self, X, y, sample_weight=None)
|      Return the mean accuracy on the given test data and labels.
|
|      In multi-label classification, this is the subset accuracy
|      which is a harsh metric since you require for each sample that
|      each label set be correctly predicted.
|
|      Parameters
```

```
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Test samples.
 |
 |      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
 |          True labels for X.
 |
 |      sample_weight : array-like of shape (n_samples,), default=None
 |          Sample weights.
 |
 |      Returns
 |      -------
 |      score : float
 |          Mean accuracy of self.predict(X) wrt. y.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.ClassifierMixin:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.svm._base.BaseLibSVM:
 |
 |  fit(self, X, y, sample_weight=None)
 |      Fit the SVM model according to the given training data.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features)
or (n_samples, n_samples)
 |          Training vectors, where n_samples is the number of samples
 |          and n_features is the number of features.
 |          For kernel="precomputed", the expected shape of X is
 |          (n_samples, n_samples).
 |
 |      y : array-like of shape (n_samples,)
 |          Target values (class labels in classification, real numbers in
 |          regression)
 |
 |      sample_weight : array-like of shape (n_samples,), default=None
 |          Per-sample weights. Rescale C per sample. Higher weights
 |          force the classifier to put more emphasis on these points.
 |
 |      Returns
```

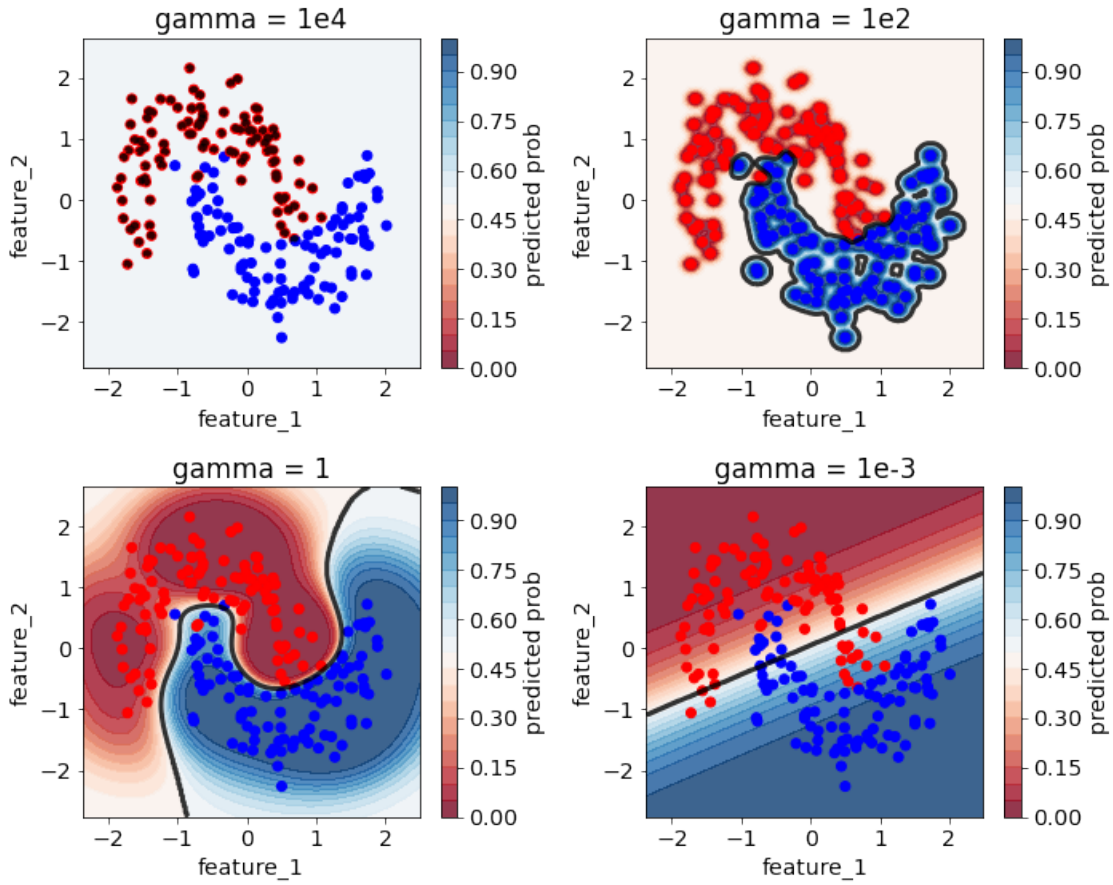
```
|      -------
|      self : object
|
|      Notes
|      -----
|      If X and y are not C-ordered and contiguous arrays of np.float64 and
|      X is not a scipy.sparse.csr_matrix, X and/or y may be copied.
|
|      If X is a dense array, then the other methods will not support sparse
|      matrices as input.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from sklearn.svm._base.BaseLibSVM:
|
|  coef_
|
|  n_support_
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.base.BaseEstimator:
|
|  __getstate__(self)
|
|  __repr__(self, N_CHAR_MAX=700)
|      Return repr(self).
|
|  __setstate__(self, state)
|
|  get_params(self, deep=True)
|      Get parameters for this estimator.
|
|      Parameters
|      ----------
|      deep : bool, default=True
|          If True, will return the parameters for this estimator and
|          contained subobjects that are estimators.
|
|      Returns
|      -------
|      params : mapping of string to any
|          Parameter names mapped to their values.
|
|  set_params(self, **params)
|      Set the parameters of this estimator.
|
|      The method works on simple estimators as well as on nested objects
|      (such as pipelines). The latter have parameters of the form
|      ``<component>__<parameter>`` so that it's possible to update each
```

```
|      component of a nested object.
|
|      Parameters
|      ----------
|      **params : dict
|          Estimator parameters.
|
|      Returns
|      -------
|      self : object
|          Estimator instance.
```

[12]:
```
matplotlib.rcParams.update({'font.size': 14})

X = StandardScaler().fit_transform(X)

h = .02  # step size in the mesh

x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

plt.figure(figsize=(10,8))
cm_bright = ListedColormap(['#FF0000', '#0000FF'])
cm = plt.cm.RdBu

plt.subplot(2,2,1)
clf = SVC(gamma = 1e4, C = 100, probability=True)
clf.fit(X,y)
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cm, alpha=.8,vmin=0,vmax=1,levels=np.arange(0,1.
 ↪05,0.05))
plt.colorbar(label='predicted prob')
plt.contour(xx, yy, Z, alpha=.8,vmin=0,vmax=1,levels=[0.
 ↪5],colors=['k'],linewidths=3)
plt.scatter(X[:, 0], X[:, 1], c=y,cmap=cm_bright)
plt.xlabel('feature_1')
plt.ylabel('feature_2')
plt.title('gamma = 1e4')

plt.subplot(2,2,2)
clf = SVC(gamma = 1e2, C = 100, probability=True)
clf.fit(X,y)
```

```
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cm, alpha=.8,vmin=0,vmax=1,levels=np.arange(0,1.
 ↪05,0.05))
plt.colorbar(label='predicted prob')
plt.contour(xx, yy, Z, alpha=.8,vmin=0,vmax=1,levels=[0.
 ↪5],colors=['k'],linewidths=3)
plt.scatter(X[:, 0], X[:, 1], c=y,cmap=cm_bright)
plt.xlabel('feature_1')
plt.ylabel('feature_2')
plt.title('gamma = 1e2')

plt.subplot(2,2,3)
clf = SVC(gamma = 1e0, C = 100, probability=True)
clf.fit(X,y)
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cm, alpha=.8,vmin=0,vmax=1,levels=np.arange(0,1.
 ↪05,0.05))
plt.colorbar(label='predicted prob')
plt.contour(xx, yy, Z, alpha=.8,vmin=0,vmax=1,levels=[0.
 ↪5],colors=['k'],linewidths=3)
plt.scatter(X[:, 0], X[:, 1], c=y,cmap=cm_bright)
plt.xlabel('feature_1')
plt.ylabel('feature_2')
plt.title('gamma = 1')

plt.subplot(2,2,4)
clf = SVC(gamma = 1e-3, C = 100, probability=True)
clf.fit(X,y)
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cm, alpha=.8,vmin=0,vmax=1,levels=np.arange(0,1.
 ↪05,0.05))
plt.colorbar(label='predicted prob')
plt.contour(xx, yy, Z, alpha=.8,vmin=0,vmax=1,levels=[0.
 ↪5],colors=['k'],linewidths=3)
plt.scatter(X[:, 0], X[:, 1], c=y,cmap=cm_bright)
plt.xlabel('feature_1')
plt.ylabel('feature_2')
plt.title('gamma = 1e-3')
```

```
plt.tight_layout()

plt.savefig('figures/SVM_clf.png',dpi=300)
plt.show()
```



2.4 SVM notes

- explore other kernels too!
- it works best on small to medium-sized datasets
  - the training time increases non-linearly with the number of points
  - one model is trained on one CPU, this technique cannot be parallelized
  - if you dataset has more than 1e5 points, the training time can be hours or days
- very versatile technique with linear and various non-linear kernels
- the prediction is a smoothly varying function of the features
- in regression, outliers might be extrapolated in a wild non-linear fashion
- in classification, model is unsure about outliers

3 Module 3: Random Forest

3.0.1 Learning objectives of this module:

- Summarize how RF works
- Describe which hyperparameters need to be tuned and what range the values should have
- Apply the algorithms in regression and classification
- Visualize the predictions of toy datasets
- Summarize under what circumstances a certain algorithm is expected to perform well or poorly and why

##

Decision trees and random forests

- Decision tree: the data is split according to certain features
- Here is an example tree fitted to data:

- Trees have nodes and leaves.
- The critical values and features in the nodes are determined automatically by minimizing a cost function.

- Random forest: ensemble of random decision trees
- Each tree sees a random subset of the training data, that's why the forest is random.

3.1 A decision tree in regression

[13]:
```
import numpy as np
np.random.seed(10)
def true_fun(X):
    return np.cos(1.5 * np.pi * X)

n_samples = 30

X = np.random.rand(n_samples)
y = true_fun(X) + np.random.randn(n_samples) * 0.1

X_new = np.linspace(0, 1, 1000)
```

[14]:
```
from sklearn.ensemble import RandomForestRegressor
help(RandomForestRegressor)
```

```
Help on class RandomForestRegressor in module sklearn.ensemble._forest:

class RandomForestRegressor(ForestRegressor)
 |  RandomForestRegressor(n_estimators=100, *, criterion='mse', 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, ccp_alpha=0.0, max_samples=None)
 |
```

```
|  A random forest regressor.
|
|  A random forest is a meta estimator that fits a number of classifying
|  decision trees on various sub-samples of the dataset and uses averaging
|  to improve the predictive accuracy and control over-fitting.
|  The sub-sample size is controlled with the `max_samples` parameter if
|  `bootstrap=True` (default), otherwise the whole dataset is used to build
|  each tree.
|
|  Read more in the :ref:`User Guide <forest>`.
|
|  Parameters
|  ----------
|  n_estimators : int, default=100
|      The number of trees in the forest.
|
|      .. versionchanged:: 0.22
|         The default value of ``n_estimators`` changed from 10 to 100
|         in 0.22.
|
|  criterion : {"mse", "mae"}, default="mse"
|      The function to measure the quality of a split. Supported criteria
|      are "mse" for the mean squared error, which is equal to variance
|      reduction as feature selection criterion, and "mae" for the mean
|      absolute error.
|
|      .. versionadded:: 0.18
|         Mean Absolute Error (MAE) criterion.
|
|  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.
|
|      .. versionchanged:: 0.18
|         Added float values for fractions.
|
|  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.
|
|      .. versionchanged:: 0.18
|         Added float values for fractions.
|
|  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
|        `int(max_features * n_features)` features are considered at each
|        split.
|      - If "auto", then `max_features=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`.
|
|      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.
|
|  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.
|
|      .. versionadded:: 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.
|
|      .. deprecated:: 0.19
|         ``min_impurity_split`` has been deprecated in favor of
|         ``min_impurity_decrease`` in 0.19. The default value of
|         ``min_impurity_split`` has changed from 1e-7 to 0 in 0.23 and it
|         will be removed in 0.25. Use ``min_impurity_decrease`` instead.
|
|  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 R^2 on unseen data.
|
|  n_jobs : int, default=None
|      The number of jobs to run in parallel. :meth:`fit`, :meth:`predict`,
|      :meth:`decision_path` and :meth:`apply` are all parallelized over the
|      trees. ``None`` means 1 unless in a :obj:`joblib.parallel_backend`
|      context. ``-1`` means using all processors. See :term:`Glossary
|      <n_jobs>` for more details.
|
|  random_state : int or RandomState, 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 :term:`Glossary <random_state>` 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 :term:`the Glossary <warm_start>`.
|
```

```
|  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
|      :ref:`minimal_cost_complexity_pruning` for details.
|
|      .. versionadded:: 0.22
|
|  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)`.
|
|      .. versionadded:: 0.22
|
|  Attributes
|  ----------
|  base_estimator_ : DecisionTreeRegressor
|      The child estimator template used to create the collection of fitted
|      sub-estimators.
|
|  estimators_ : list of DecisionTreeRegressor
|      The collection of fitted sub-estimators.
|
|  feature_importances_ : ndarray of shape (n_features,)
|      The impurity-based feature importances.
|      The higher, the more important the feature.
|      The importance of a feature is computed as the (normalized)
|      total reduction of the criterion brought by that feature.  It is also
|      known as the Gini importance.
|
|      Warning: impurity-based feature importances can be misleading for
|      high cardinality features (many unique values). See
|      :func:`sklearn.inspection.permutation_importance` as an alternative.
|
|  n_features_ : int
|      The number of features when ``fit`` is performed.
|
|  n_outputs_ : int
|      The number of outputs when ``fit`` is performed.
|
|  oob_score_ : float
|      Score of the training dataset obtained using an out-of-bag estimate.
|      This attribute exists only when ``oob_score`` is True.
```

```
|
|  oob_prediction_ : ndarray of shape (n_samples,)
|      Prediction computed with out-of-bag estimate on the training set.
|      This attribute exists only when ``oob_score`` is True.
|
|  See Also
|  --------
|  DecisionTreeRegressor, ExtraTreesRegressor
|
|  Notes
|  -----
|  The default values for the parameters controlling the size of the trees
|  (e.g. ``max_depth``, ``min_samples_leaf``, etc.) lead to fully grown and
|  unpruned trees which can potentially be very large on some data sets. To
|  reduce memory consumption, the complexity and size of the trees should be
|  controlled by setting those parameter values.
|
|  The features are always randomly permuted at each split. Therefore,
|  the best found split may vary, even with the same training data,
|  ``max_features=n_features`` and ``bootstrap=False``, if the improvement
|  of the criterion is identical for several splits enumerated during the
|  search of the best split. To obtain a deterministic behaviour during
|  fitting, ``random_state`` has to be fixed.
|
|  The default value ``max_features="auto"`` uses ``n_features``
|  rather than ``n_features / 3``. The latter was originally suggested in
|  [1], whereas the former was more recently justified empirically in [2].
|
|  References
|  ----------
|  .. [1] L. Breiman, "Random Forests", Machine Learning, 45(1), 5-32, 2001.
|
|  .. [2] P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized
|         trees", Machine Learning, 63(1), 3-42, 2006.
|
|  Examples
|  --------
|  >>> from sklearn.ensemble import RandomForestRegressor
|  >>> from sklearn.datasets import make_regression
|  >>> X, y = make_regression(n_features=4, n_informative=2,
|  …                         random_state=0, shuffle=False)
|  >>> regr = RandomForestRegressor(max_depth=2, random_state=0)
|  >>> regr.fit(X, y)
|  RandomForestRegressor(…)
|  >>> print(regr.predict([[0, 0, 0, 0]]))
|  [-8.32987858]
|
|  Method resolution order:
```

```
 |      RandomForestRegressor
 |      ForestRegressor
 |      sklearn.base.RegressorMixin
 |      BaseForest
 |      sklearn.base.MultiOutputMixin
 |      sklearn.ensemble._base.BaseEnsemble
 |      sklearn.base.MetaEstimatorMixin
 |      sklearn.base.BaseEstimator
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, n_estimators=100, *, criterion='mse', 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, ccp_alpha=0.0, max_samples=None)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __abstractmethods__ = frozenset()
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from ForestRegressor:
 |
 |  predict(self, X)
 |      Predict regression target for X.
 |
 |      The predicted regression target of an input sample is computed as the
 |      mean predicted regression targets of the trees in the forest.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features)
 |          The input samples. Internally, its dtype will be converted to
 |          ``dtype=np.float32``. If a sparse matrix is provided, it will be
 |          converted into a sparse ``csr_matrix``.
 |
 |      Returns
 |      -------
 |      y : ndarray of shape (n_samples,) or (n_samples, n_outputs)
 |          The predicted values.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.RegressorMixin:
 |
```

```
|  score(self, X, y, sample_weight=None)
|      Return the coefficient of determination R^2 of the prediction.
|
|      The coefficient R^2 is defined as (1 - u/v), where u is the residual
|      sum of squares ((y_true - y_pred) ** 2).sum() and v is the total
|      sum of squares ((y_true - y_true.mean()) ** 2).sum().
|      The best possible score is 1.0 and it can be negative (because the
|      model can be arbitrarily worse). A constant model that always
|      predicts the expected value of y, disregarding the input features,
|      would get a R^2 score of 0.0.
|
|      Parameters
|      ----------
|      X : array-like of shape (n_samples, n_features)
|          Test samples. For some estimators this may be a
|          precomputed kernel matrix or a list of generic objects instead,
|          shape = (n_samples, n_samples_fitted),
|          where n_samples_fitted is the number of
|          samples used in the fitting for the estimator.
|
|      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
|          True values for X.
|
|      sample_weight : array-like of shape (n_samples,), default=None
|          Sample weights.
|
|      Returns
|      -------
|      score : float
|          R^2 of self.predict(X) wrt. y.
|
|      Notes
|      -----
|      The R2 score used when calling ``score`` on a regressor uses
|      ``multioutput='uniform_average'`` from version 0.23 to keep consistent
|      with default value of :func:`~sklearn.metrics.r2_score`.
|      This influences the ``score`` method of all the multioutput
|      regressors (except for
|      :class:`~sklearn.multioutput.MultiOutputRegressor`).
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from sklearn.base.RegressorMixin:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
```

```
|
|  ----------------------------------------------------------------------
|  Methods inherited from BaseForest:
|
|  apply(self, X)
|      Apply trees in the forest to X, return leaf indices.
|
|      Parameters
|      ----------
|      X : {array-like, sparse matrix} of shape (n_samples, n_features)
|          The input samples. Internally, its dtype will be converted to
|          ``dtype=np.float32``. If a sparse matrix is provided, it will be
|          converted into a sparse ``csr_matrix``.
|
|      Returns
|      -------
|      X_leaves : ndarray of shape (n_samples, n_estimators)
|          For each datapoint x in X and for each tree in the forest,
|          return the index of the leaf x ends up in.
|
|  decision_path(self, X)
|      Return the decision path in the forest.
|
|      .. versionadded:: 0.18
|
|      Parameters
|      ----------
|      X : {array-like, sparse matrix} of shape (n_samples, n_features)
|          The input samples. Internally, its dtype will be converted to
|          ``dtype=np.float32``. If a sparse matrix is provided, it will be
|          converted into a sparse ``csr_matrix``.
|
|      Returns
|      -------
|      indicator : sparse matrix of shape (n_samples, n_nodes)
|          Return a node indicator matrix where non zero elements indicates
|          that the samples goes through the nodes. The matrix is of CSR
|          format.
|
|      n_nodes_ptr : ndarray of shape (n_estimators + 1,)
|          The columns from indicator[n_nodes_ptr[i]:n_nodes_ptr[i+1]]
|          gives the indicator value for the i-th estimator.
|
|  fit(self, X, y, sample_weight=None)
|      Build a forest of trees from the training set (X, y).
|
|      Parameters
|      ----------
```

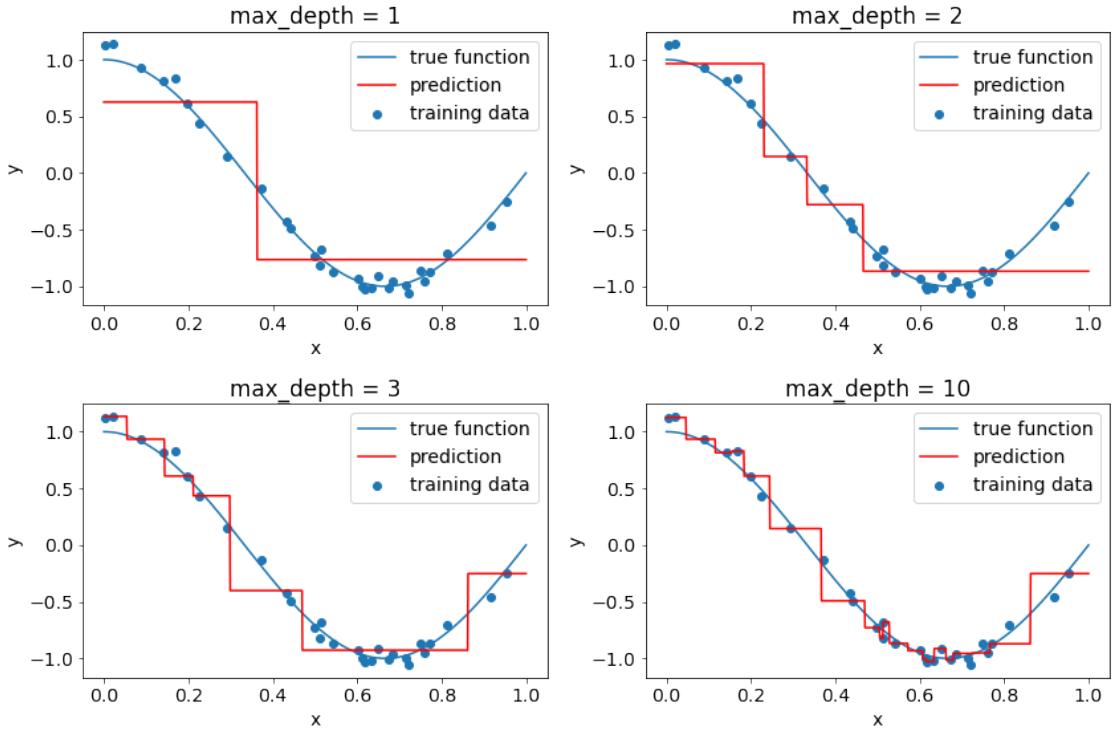
```
|      X : {array-like, sparse matrix} of shape (n_samples, n_features)
|          The training input samples. Internally, its dtype will be converted
|          to ``dtype=np.float32``. If a sparse matrix is provided, it will be
|          converted into a sparse ``csc_matrix``.
|
|      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
|          The target values (class labels in classification, real numbers in
|          regression).
|
|      sample_weight : array-like of shape (n_samples,), default=None
|          Sample weights. If None, then samples are equally weighted. Splits
|          that would create child nodes with net zero or negative weight are
|          ignored while searching for a split in each node. In the case of
|          classification, splits are also ignored if they would result in any
|          single class carrying a negative weight in either child node.
|
|      Returns
|      -------
|      self : object
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from BaseForest:
|
|  feature_importances_
|      The impurity-based feature importances.
|
|      The higher, the more important the feature.
|      The importance of a feature is computed as the (normalized)
|      total reduction of the criterion brought by that feature.  It is also
|      known as the Gini importance.
|
|      Warning: impurity-based feature importances can be misleading for
|      high cardinality features (many unique values). See
|      :func:`sklearn.inspection.permutation_importance` as an alternative.
|
|      Returns
|      -------
|      feature_importances_ : ndarray of shape (n_features,)
|          The values of this array sum to 1, unless all trees are single node
|          trees consisting of only the root node, in which case it will be an
|          array of zeros.
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.ensemble._base.BaseEnsemble:
|
|  __getitem__(self, index)
|      Return the index'th estimator in the ensemble.
|
```

```
 |  __iter__(self)
 |      Return iterator over estimators in the ensemble.
 |
 |  __len__(self)
 |      Return the number of estimators in the ensemble.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from
sklearn.ensemble._base.BaseEnsemble:
 |
 |  __annotations__ = {'_required_parameters': typing.List[str]}
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |
 |      Parameters
 |      ----------
 |      deep : bool, default=True
 |          If True, will return the parameters for this estimator and
 |          contained subobjects that are estimators.
 |
 |      Returns
 |      -------
 |      params : mapping of string to any
 |          Parameter names mapped to their values.
 |
 |  set_params(self, **params)
 |      Set the parameters of this estimator.
 |
 |      The method works on simple estimators as well as on nested objects
 |      (such as pipelines). The latter have parameters of the form
 |      ``<component>__<parameter>`` so that it's possible to update each
 |      component of a nested object.
 |
 |      Parameters
 |      ----------
 |      **params : dict
 |          Estimator parameters.
```

```
|
|       Returns
|       -------
|       self : object
|           Estimator instance.
```

[15]:
```
matplotlib.rcParams.update({'font.size': 14})


plt.figure(figsize=(12,8))

plt.subplot(2,2,1)
plt.scatter(X,y,label='training data')
plt.plot(np.linspace(0, 1, 100),true_fun(np.linspace(0, 1, 100)),label='true␣
 ↪function')
reg = RandomForestRegressor(n_estimators=1,max_depth=1)
reg.fit(X[:, np.newaxis],y)
y_new = reg.predict(X_new[:, np.newaxis])
plt.plot(X_new,y_new,'r',label='prediction')
plt.xlabel('x')
plt.ylabel('y')
plt.title('max_depth = 1')
plt.legend()

plt.subplot(2,2,2)
plt.scatter(X,y,label='training data')
plt.plot(np.linspace(0, 1, 100),true_fun(np.linspace(0, 1, 100)),label='true␣
 ↪function')
reg = RandomForestRegressor(n_estimators=1,max_depth=2)
reg.fit(X[:, np.newaxis],y)
y_new = reg.predict(X_new[:, np.newaxis])
plt.plot(X_new,y_new,'r',label='prediction')
plt.xlabel('x')
plt.ylabel('y')
plt.title('max_depth = 2')
plt.legend()

plt.subplot(2,2,3)
plt.scatter(X,y,label='training data')
plt.plot(np.linspace(0, 1, 100),true_fun(np.linspace(0, 1, 100)),label='true␣
 ↪function')
reg = RandomForestRegressor(n_estimators=1,max_depth=3)
reg.fit(X[:, np.newaxis],y)
y_new = reg.predict(X_new[:, np.newaxis])
plt.plot(X_new,y_new,'r',label='prediction')
plt.xlabel('x')
```

```
plt.ylabel('y')
plt.title('max_depth = 3')
plt.legend()


plt.subplot(2,2,4)
plt.scatter(X,y,label='training data')
plt.plot(np.linspace(0, 1, 100),true_fun(np.linspace(0, 1, 100)),label='true␣
↪function')
reg = RandomForestRegressor(n_estimators=1,max_depth=10)
reg.fit(X[:, np.newaxis],y)
y_new = reg.predict(X_new[:, np.newaxis])
plt.plot(X_new,y_new,'r',label='prediction')
plt.xlabel('x')
plt.ylabel('y')
plt.title('max_depth = 10')
plt.legend()

plt.tight_layout()
plt.savefig('figures/tree_reg.png',dpi=300)
plt.show()
```



3.2 How to avoid overfitting with random forests?

- tune some (or all) of following hyperparameters:
  - max_depth
  - min_samples_split
  - max_features
- With sklearn random forests, **do not tune n_estimators**!
  - the larger this value is, the better the forest will be
  - set n_estimators to maybe a 100 while tuning hyperparameters
  - increase it if necessary once the best hyperparameters are found

3.3 A random forest in classification

[16]:
```
from sklearn.datasets import make_moons

# create the data
X,y = make_moons(noise=0.2, random_state=1,n_samples=200)
```

[17]:
```
from sklearn.ensemble import RandomForestClassifier
help(RandomForestClassifier)
```

```
Help on class RandomForestClassifier in module sklearn.ensemble._forest:

class RandomForestClassifier(ForestClassifier)
 |  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)
 |
 |  A random forest classifier.
 |
 |  A random forest is a meta estimator that fits a number of decision tree
 |  classifiers on various sub-samples of the dataset and uses averaging to
 |  improve the predictive accuracy and control over-fitting.
 |  The sub-sample size is controlled with the `max_samples` parameter if
 |  `bootstrap=True` (default), otherwise the whole dataset is used to build
 |  each tree.
 |
 |  Read more in the :ref:`User Guide <forest>`.
 |
 |  Parameters
 |  ----------
 |  n_estimators : int, default=100
 |      The number of trees in the forest.
 |
 |      .. versionchanged:: 0.22
```

```
|         The default value of ``n_estimators`` changed from 10 to 100
|         in 0.22.
|
|   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.
|
|       .. versionchanged:: 0.18
|          Added float values for fractions.
|
|   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.
|
|       .. versionchanged:: 0.18
|          Added float values for fractions.
|
|   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
|        `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)` (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.
|
|  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.
|
|      .. versionadded:: 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.
|
|      .. deprecated:: 0.19
|         ``min_impurity_split`` has been deprecated in favor of
|         ``min_impurity_decrease`` in 0.19. The default value of
|         ``min_impurity_split`` has changed from 1e-7 to 0 in 0.23 and it
|         will be removed in 0.25. Use ``min_impurity_decrease`` instead.
|
|
|  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 accuracy.
 |
 |  n_jobs : int, default=None
 |      The number of jobs to run in parallel. :meth:`fit`, :meth:`predict`,
 |      :meth:`decision_path` and :meth:`apply` are all parallelized over the
 |      trees. ``None`` means 1 unless in a :obj:`joblib.parallel_backend`
 |      context. ``-1`` means using all processors. See :term:`Glossary
 |      <n_jobs>` for more details.
 |
 |  random_state : int or RandomState, 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 :term:`Glossary <random_state>` 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 :term:`the Glossary <warm_start>`.
 |
 |  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
|      :ref:`minimal_cost_complexity_pruning` for details.
|
|      .. versionadded:: 0.22
|
|  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)`.
|
|      .. versionadded:: 0.22
|
|  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
|      The number of features when ``fit`` is performed.
|
|  n_outputs_ : int
|      The number of outputs when ``fit`` is performed.
|
```

```
|  feature_importances_ : ndarray of shape (n_features,)
|      The impurity-based feature importances.
|      The higher, the more important the feature.
|      The importance of a feature is computed as the (normalized)
|      total reduction of the criterion brought by that feature.  It is also
|      known as the Gini importance.
|
|      Warning: impurity-based feature importances can be misleading for
|      high cardinality features (many unique values). See
|      :func:`sklearn.inspection.permutation_importance` as an alternative.
|
|  oob_score_ : float
|      Score of the training dataset obtained using an out-of-bag estimate.
|      This attribute exists only when ``oob_score`` is True.
|
|  oob_decision_function_ : ndarray of shape (n_samples, n_classes)
|      Decision function computed with out-of-bag estimate on the training
|      set. If n_estimators is small it might be possible that a data point
|      was never left out during the bootstrap. In this case,
|      `oob_decision_function_` might contain NaN. This attribute exists
|      only when ``oob_score`` is True.
|
|  See Also
|  --------
|  DecisionTreeClassifier, ExtraTreesClassifier
|
|  Notes
|  -----
|  The default values for the parameters controlling the size of the trees
|  (e.g. ``max_depth``, ``min_samples_leaf``, etc.) lead to fully grown and
|  unpruned trees which can potentially be very large on some data sets. To
|  reduce memory consumption, the complexity and size of the trees should be
|  controlled by setting those parameter values.
|
|  The features are always randomly permuted at each split. Therefore,
|  the best found split may vary, even with the same training data,
|  ``max_features=n_features`` and ``bootstrap=False``, if the improvement
|  of the criterion is identical for several splits enumerated during the
|  search of the best split. To obtain a deterministic behaviour during
|  fitting, ``random_state`` has to be fixed.
|
|  References
|  ----------
|  .. [1] L. Breiman, "Random Forests", Machine Learning, 45(1), 5-32, 2001.
|
|  Examples
|  --------
|  >>> from sklearn.ensemble import RandomForestClassifier
```

```
 |  >>> from sklearn.datasets import make_classification
 |  >>> X, y = make_classification(n_samples=1000, n_features=4,
 |  …                            n_informative=2, n_redundant=0,
 |  …                            random_state=0, shuffle=False)
 |  >>> clf = RandomForestClassifier(max_depth=2, random_state=0)
 |  >>> clf.fit(X, y)
 |  RandomForestClassifier(…)
 |  >>> print(clf.predict([[0, 0, 0, 0]]))
 |  [1]
 |
 |  Method resolution order:
 |      RandomForestClassifier
 |      ForestClassifier
 |      sklearn.base.ClassifierMixin
 |      BaseForest
 |      sklearn.base.MultiOutputMixin
 |      sklearn.ensemble._base.BaseEnsemble
 |      sklearn.base.MetaEstimatorMixin
 |      sklearn.base.BaseEstimator
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, 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)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __abstractmethods__ = frozenset()
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from ForestClassifier:
 |
 |  predict(self, X)
 |      Predict class for X.
 |
 |      The predicted class of an input sample is a vote by the trees in
 |      the forest, weighted by their probability estimates. That is,
 |      the predicted class is the one with highest mean probability
 |      estimate across the trees.
 |
 |      Parameters
```

```
|      ----------
|      X : {array-like, sparse matrix} of shape (n_samples, n_features)
|          The input samples. Internally, its dtype will be converted to
|          ``dtype=np.float32``. If a sparse matrix is provided, it will be
|          converted into a sparse ``csr_matrix``.
|
|      Returns
|      -------
|      y : ndarray of shape (n_samples,) or (n_samples, n_outputs)
|          The predicted classes.
|
|  predict_log_proba(self, X)
|      Predict class log-probabilities for X.
|
|      The predicted class log-probabilities of an input sample is computed as
|      the log of the mean predicted class probabilities of the trees in the
|      forest.
|
|      Parameters
|      ----------
|      X : {array-like, sparse matrix} of shape (n_samples, n_features)
|          The input samples. Internally, its dtype will be converted to
|          ``dtype=np.float32``. If a sparse matrix is provided, it will be
|          converted into a sparse ``csr_matrix``.
|
|      Returns
|      -------
|      p : ndarray of shape (n_samples, n_classes), or a list of n_outputs
|          such arrays if n_outputs > 1.
|          The class probabilities of the input samples. The order of the
|          classes corresponds to that in the attribute :term:`classes_`.
|
|  predict_proba(self, X)
|      Predict class probabilities for X.
|
|      The predicted class probabilities of an input sample are computed as
|      the mean predicted class probabilities of the trees in the forest.
|      The class probability of a single tree is the fraction of samples of
|      the same class in a leaf.
|
|      Parameters
|      ----------
|      X : {array-like, sparse matrix} of shape (n_samples, n_features)
|          The input samples. Internally, its dtype will be converted to
|          ``dtype=np.float32``. If a sparse matrix is provided, it will be
|          converted into a sparse ``csr_matrix``.
|
|      Returns
```

```
|      -------
|      p : ndarray of shape (n_samples, n_classes), or a list of n_outputs
|          such arrays if n_outputs > 1.
|          The class probabilities of the input samples. The order of the
|          classes corresponds to that in the attribute :term:`classes_`.
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.base.ClassifierMixin:
|
|  score(self, X, y, sample_weight=None)
|      Return the mean accuracy on the given test data and labels.
|
|      In multi-label classification, this is the subset accuracy
|      which is a harsh metric since you require for each sample that
|      each label set be correctly predicted.
|
|      Parameters
|      ----------
|      X : array-like of shape (n_samples, n_features)
|          Test samples.
|
|      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
|          True labels for X.
|
|      sample_weight : array-like of shape (n_samples,), default=None
|          Sample weights.
|
|      Returns
|      -------
|      score : float
|          Mean accuracy of self.predict(X) wrt. y.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from sklearn.base.ClassifierMixin:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Methods inherited from BaseForest:
|
|  apply(self, X)
|      Apply trees in the forest to X, return leaf indices.
|
|      Parameters
```

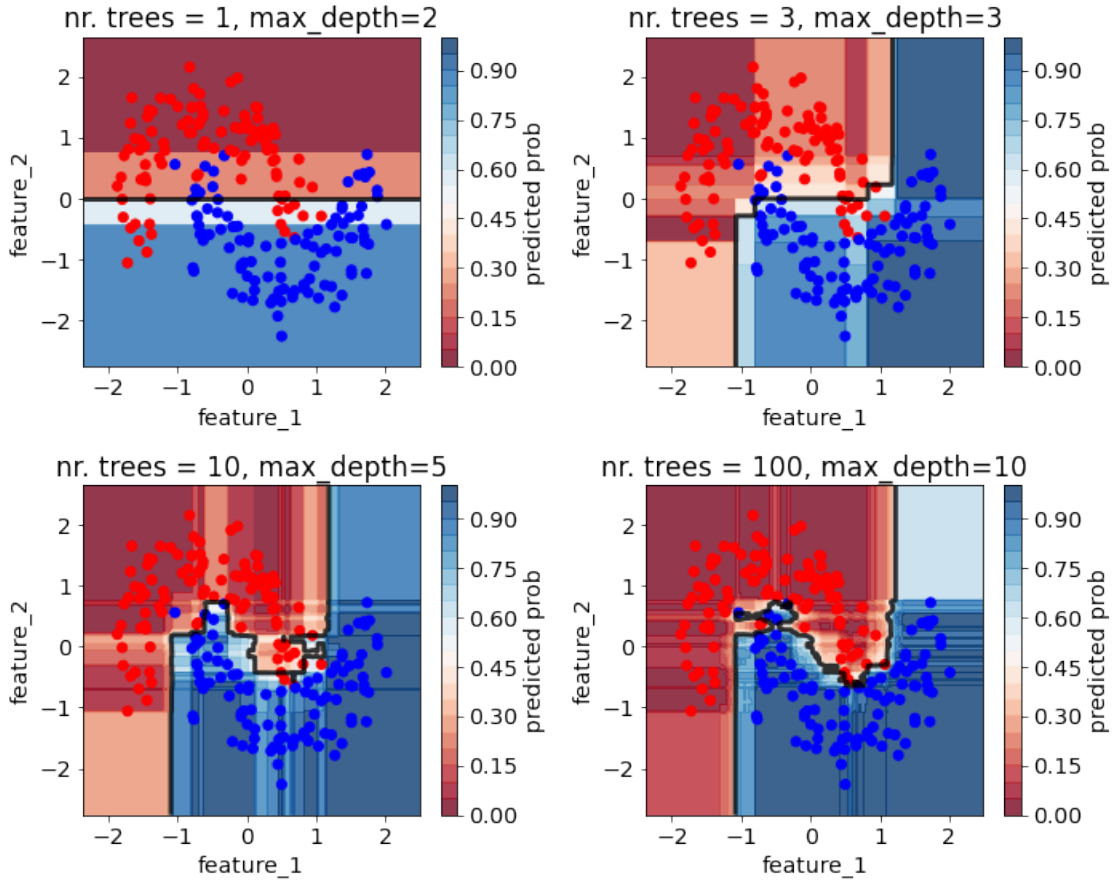
```
|      ----------
|      X : {array-like, sparse matrix} of shape (n_samples, n_features)
|          The input samples. Internally, its dtype will be converted to
|          ``dtype=np.float32``. If a sparse matrix is provided, it will be
|          converted into a sparse ``csr_matrix``.
|
|      Returns
|      -------
|      X_leaves : ndarray of shape (n_samples, n_estimators)
|          For each datapoint x in X and for each tree in the forest,
|          return the index of the leaf x ends up in.
|
|  decision_path(self, X)
|      Return the decision path in the forest.
|
|      .. versionadded:: 0.18
|
|      Parameters
|      ----------
|      X : {array-like, sparse matrix} of shape (n_samples, n_features)
|          The input samples. Internally, its dtype will be converted to
|          ``dtype=np.float32``. If a sparse matrix is provided, it will be
|          converted into a sparse ``csr_matrix``.
|
|      Returns
|      -------
|      indicator : sparse matrix of shape (n_samples, n_nodes)
|          Return a node indicator matrix where non zero elements indicates
|          that the samples goes through the nodes. The matrix is of CSR
|          format.
|
|      n_nodes_ptr : ndarray of shape (n_estimators + 1,)
|          The columns from indicator[n_nodes_ptr[i]:n_nodes_ptr[i+1]]
|          gives the indicator value for the i-th estimator.
|
|  fit(self, X, y, sample_weight=None)
|      Build a forest of trees from the training set (X, y).
|
|      Parameters
|      ----------
|      X : {array-like, sparse matrix} of shape (n_samples, n_features)
|          The training input samples. Internally, its dtype will be converted
|          to ``dtype=np.float32``. If a sparse matrix is provided, it will be
|          converted into a sparse ``csc_matrix``.
|
|      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
|          The target values (class labels in classification, real numbers in
|          regression).
```

```
|
|      sample_weight : array-like of shape (n_samples,), default=None
|          Sample weights. If None, then samples are equally weighted. Splits
|          that would create child nodes with net zero or negative weight are
|          ignored while searching for a split in each node. In the case of
|          classification, splits are also ignored if they would result in any
|          single class carrying a negative weight in either child node.
|
|      Returns
|      -------
|      self : object
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from BaseForest:
|
|  feature_importances_
|      The impurity-based feature importances.
|
|      The higher, the more important the feature.
|      The importance of a feature is computed as the (normalized)
|      total reduction of the criterion brought by that feature.  It is also
|      known as the Gini importance.
|
|      Warning: impurity-based feature importances can be misleading for
|      high cardinality features (many unique values). See
|      :func:`sklearn.inspection.permutation_importance` as an alternative.
|
|      Returns
|      -------
|      feature_importances_ : ndarray of shape (n_features,)
|          The values of this array sum to 1, unless all trees are single node
|          trees consisting of only the root node, in which case it will be an
|          array of zeros.
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.ensemble._base.BaseEnsemble:
|
|  __getitem__(self, index)
|      Return the index'th estimator in the ensemble.
|
|  __iter__(self)
|      Return iterator over estimators in the ensemble.
|
|  __len__(self)
|      Return the number of estimators in the ensemble.
|
|  ----------------------------------------------------------------------
|  Data and other attributes inherited from
```

```
sklearn.ensemble._base.BaseEnsemble:
 |
 |  __annotations__ = {'_required_parameters': typing.List[str]}
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |
 |      Parameters
 |      ----------
 |      deep : bool, default=True
 |          If True, will return the parameters for this estimator and
 |          contained subobjects that are estimators.
 |
 |      Returns
 |      -------
 |      params : mapping of string to any
 |          Parameter names mapped to their values.
 |
 |  set_params(self, **params)
 |      Set the parameters of this estimator.
 |
 |      The method works on simple estimators as well as on nested objects
 |      (such as pipelines). The latter have parameters of the form
 |      ``<component>__<parameter>`` so that it's possible to update each
 |      component of a nested object.
 |
 |      Parameters
 |      ----------
 |      **params : dict
 |          Estimator parameters.
 |
 |      Returns
 |      -------
 |      self : object
 |          Estimator instance.
```

[18]:
```
matplotlib.rcParams.update({'font.size': 14})

X = StandardScaler().fit_transform(X)

h = .02  # step size in the mesh

x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

plt.figure(figsize=(10,8))
cm_bright = ListedColormap(['#FF0000', '#0000FF'])
cm = plt.cm.RdBu

plt.subplot(2,2,1)
clf = RandomForestClassifier(n_estimators=1,max_depth=2,random_state=1)

clf.fit(X,y)
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cm, alpha=.8,vmin=0,vmax=1,levels=np.arange(0,1.
 ↪05,0.05))
plt.colorbar(label='predicted prob')
plt.contour(xx, yy, Z, alpha=.8,vmin=0,vmax=1,levels=[0.
 ↪5],colors=['k'],linewidths=3)
plt.scatter(X[:, 0], X[:, 1], c=y,cmap=cm_bright)
plt.xlabel('feature_1')
plt.ylabel('feature_2')
plt.title('nr. trees = 1, max_depth=2')

plt.subplot(2,2,2)
clf = RandomForestClassifier(n_estimators=3,max_depth=3,random_state=4)
clf.fit(X,y)
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cm, alpha=.8,vmin=0,vmax=1,levels=np.arange(0,1.
 ↪05,0.05))
plt.colorbar(label='predicted prob')
plt.contour(xx, yy, Z, alpha=.8,vmin=0,vmax=1,levels=[0.
 ↪5],colors=['k'],linewidths=3)
plt.scatter(X[:, 0], X[:, 1], c=y,cmap=cm_bright)
plt.xlabel('feature_1')
plt.ylabel('feature_2')
plt.title('nr. trees = 3, max_depth=3')
```

```
plt.subplot(2,2,3)
clf = RandomForestClassifier(n_estimators=10,max_depth=5,random_state=3)
clf.fit(X,y)
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cm, alpha=.8,vmin=0,vmax=1,levels=np.arange(0,1.
 ↪05,0.05))
plt.colorbar(label='predicted prob')
plt.contour(xx, yy, Z, alpha=.8,vmin=0,vmax=1,levels=[0.
 ↪5],colors=['k'],linewidths=3)
plt.scatter(X[:, 0], X[:, 1], c=y,cmap=cm_bright)
plt.xlabel('feature_1')
plt.ylabel('feature_2')
plt.title('nr. trees = 10, max_depth=5')

plt.subplot(2,2,4)
clf = RandomForestClassifier(n_estimators=100,max_depth=10,random_state=3)
clf.fit(X,y)
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cm, alpha=.8,vmin=0,vmax=1,levels=np.arange(0,1.
 ↪05,0.05))
plt.colorbar(label='predicted prob')
plt.contour(xx, yy, Z, alpha=.8,vmin=0,vmax=1,levels=[0.
 ↪5],colors=['k'],linewidths=3)
plt.scatter(X[:, 0], X[:, 1], c=y,cmap=cm_bright)
plt.xlabel('feature_1')
plt.ylabel('feature_2')
plt.title('nr. trees = 100, max_depth=10')


plt.tight_layout()

plt.savefig('figures/forest_clf.png',dpi=300)
plt.show()
```



3.4 RF notes

- RF is a very robust algorithm
  - decent performance on pretty much any dataset
- RF is great for any dataset size
  - it is very easy to parallelize the model training
  - check out the n_jobs argument of the functions
- it can capture non-linearities, feature correlations
- the prediction is not a smoothly varying function of the features
- behaves well wrt outliers

4 Module 4: XGBoost

4.0.1 Learning objectives of this module:

- Summarize how XGB works
- Describe which hyperparameters need to be tuned and what range the values should have
- Apply the algorithms in regression and classification
- Visualize the predictions of toy datasets

- Summarize under what circumstances a certain algorithm is expected to perform well or poorly and why

4.1 XGBoost

- eXtreme Gradient Boosting - a popular tree-based method
- blog post and paper
- more advanced than random forest
  - trees are not independent
    - the next tree is built to improve the previous tree
    - less trees are necessary to achieve same accuracy
    - but XGBoost trees can overfit - more on this later
  - handles missing values well
  - it has l1 and l2 regularization while random forest does not

4.2 XGB hyperparameters to tune

- reg_alpha, reg_lambda: the two regularization parameter
- max_depth: same as in a RF
- colsample_bytree:
- subsample:
- do not tune n_estimators, instead use early stopping

[ ]:
```
# this code is just illustration
import xgboost
from sklearn.model_selection import ParameterGrid
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score

param_grid = {"learning_rate": [0.03],
              "n_estimators": [10000],
              "seed": [0],
              #"reg_alpha": [0e0, 1e-2, 1e-1, 1e0, 1e1, 1e2],
              #"reg_lambda": [0e0, 1e-2, 1e-1, 1e0, 1e1, 1e2],
              "missing": [np.nan],
              #"max_depth": [1,3,10,30,100],
              "colsample_bytree": [0.9],
              "subsample": [0.66]}

XGB = xgboost.XGBRegressor()
XGB.set_params(**ParameterGrid(param_grid)[0])
XGB.fit(X_train,y_train,early_stopping_rounds=50,eval_set=[(X_CV, y_CV)],␣
 ↪verbose=False)
y_CV_pred = XGB.predict(X_CV)
print('the CV RMSE:',np.sqrt(mean_squared_error(y_CV,y_CV_pred)))
y_test_pred = XGB.predict(X_test)
print('the test RMSE:',np.sqrt(mean_squared_error(y_test,y_test_pred)))
print('the test R2:',r2_score(y_test,y_test_pred))
```

4.3 XGBRegressor

[19]:
```
import numpy as np
np.random.seed(10)
def true_fun(X):
    return np.cos(1.5 * np.pi * X)

n_samples = 30

X = np.random.rand(n_samples)
y = true_fun(X) + np.random.randn(n_samples) * 0.1

X_new = np.linspace(0, 1, 1000)
```

[25]:
```
from xgboost import XGBRegressor
help(XGBRegressor)
```

```
Help on class XGBRegressor in module xgboost.sklearn:

class XGBRegressor(XGBModel, sklearn.base.RegressorMixin)
 |  XGBRegressor(objective='reg:squarederror', **kwargs)
 |
 |  Implementation of the scikit-learn API for XGBoost regression.
 |
 |
 |  Parameters
 |  ----------
 |
 |      n_estimators : int
 |          Number of gradient boosted trees.  Equivalent to number of boosting
 |          rounds.
 |
 |      max_depth : int
 |          Maximum tree depth for base learners.
 |      learning_rate : float
 |          Boosting learning rate (xgb's "eta")
 |      verbosity : int
 |          The degree of verbosity. Valid values are 0 (silent) - 3 (debug).
 |      objective : string or callable
 |          Specify the learning task and the corresponding learning objective
or
 |          a custom objective function to be used (see note below).
 |      booster: string
 |          Specify which booster to use: gbtree, gblinear or dart.
 |      tree_method: string
 |          Specify which tree method to use.  Default to auto.  If this
parameter
 |          is set to default, XGBoost will choose the most conservative option
```

```
 |          available.  It's recommended to study this option from parameters
 |          document.
 |      n_jobs : int
 |          Number of parallel threads used to run xgboost.
 |      gamma : float
 |          Minimum loss reduction required to make a further partition on a
leaf
 |          node of the tree.
 |      min_child_weight : int
 |          Minimum sum of instance weight(hessian) needed in a child.
 |      max_delta_step : int
 |          Maximum delta step we allow each tree's weight estimation to be.
 |      subsample : float
 |          Subsample ratio of the training instance.
 |      colsample_bytree : float
 |          Subsample ratio of columns when constructing each tree.
 |      colsample_bylevel : float
 |          Subsample ratio of columns for each level.
 |      colsample_bynode : float
 |          Subsample ratio of columns for each split.
 |      reg_alpha : float (xgb's alpha)
 |          L1 regularization term on weights
 |      reg_lambda : float (xgb's lambda)
 |          L2 regularization term on weights
 |      scale_pos_weight : float
 |          Balancing of positive and negative weights.
 |      base_score:
 |          The initial prediction score of all instances, global bias.
 |      random_state : int
 |          Random number seed.
 |
 |          .. note::
 |
 |             Using gblinear booster with shotgun updater is nondeterministic
as
 |             it uses Hogwild algorithm.
 |
 |      missing : float, default np.nan
 |          Value in the data which needs to be present as a missing value.
 |      num_parallel_tree: int
 |          Used for boosting random forest.
 |      monotone_constraints : str
 |          Constraint of variable monotonicity.  See tutorial for more
 |          information.
 |      interaction_constraints : str
 |          Constraints for interaction representing permitted interactions.
The
 |          constraints must be specified in the form of a nest list, e.g. [[0,
```

```
1],
 |          [2, 3, 4]], where each inner list is a group of indices of features
 |          that are allowed to interact with each other.  See tutorial for more
 |          information
 |      importance_type: string, default "gain"
 |          The feature importance type for the feature_importances\_ property:
 |          either "gain", "weight", "cover", "total_gain" or "total_cover".
 |
 |      \*\*kwargs : dict, optional
 |          Keyword arguments for XGBoost Booster object.  Full documentation of
 |          parameters can be found here:
 |          https://github.com/dmlc/xgboost/blob/master/doc/parameter.rst.
 |          Attempting to set a parameter via the constructor args and
\*\*kwargs
 |          dict simultaneously will result in a TypeError.
 |
 |          .. note:: \*\*kwargs unsupported by scikit-learn
 |
 |              \*\*kwargs is unsupported by scikit-learn.  We do not guarantee
 |              that parameters passed via this argument will interact properly
 |              with scikit-learn.
 |
 |          .. note::  Custom objective function
 |
 |              A custom objective function can be provided for the
``objective``
 |              parameter. In this case, it should have the signature
 |              ``objective(y_true, y_pred) -> grad, hess``:
 |
 |              y_true: array_like of shape [n_samples]
 |                  The target values
 |              y_pred: array_like of shape [n_samples]
 |                  The predicted values
 |
 |              grad: array_like of shape [n_samples]
 |                  The value of the gradient for each sample point.
 |              hess: array_like of shape [n_samples]
 |                  The value of the second derivative for each sample point
 |
 |  Method resolution order:
 |      XGBRegressor
 |      XGBModel
 |      sklearn.base.BaseEstimator
 |      sklearn.base.RegressorMixin
 |      builtins.object
 |
 |  Methods defined here:
 |
```

```
 |  __init__(self, objective='reg:squarederror', **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from XGBModel:
 |
 |  apply(self, X, ntree_limit=0)
 |      Return the predicted leaf every tree for each sample.
 |
 |      Parameters
 |      ----------
 |      X : array_like, shape=[n_samples, n_features]
 |          Input features matrix.
 |
 |      ntree_limit : int
 |          Limit number of trees in the prediction; defaults to 0 (use all
trees).
 |
 |      Returns
 |      -------
 |      X_leaves : array_like, shape=[n_samples, n_trees]
 |          For each datapoint x in X and for each tree, return the index of the
 |          leaf x ends up in. Leaves are numbered within
 |          ``[0; 2**(self.max_depth+1))``, possibly with gaps in the numbering.
 |
 |  evals_result(self)
 |      Return the evaluation results.
 |
 |      If **eval_set** is passed to the `fit` function, you can call
 |      ``evals_result()`` to get evaluation results for all passed
**eval_sets**.
 |      When **eval_metric** is also passed to the `fit` function, the
 |      **evals_result** will contain the **eval_metrics** passed to the `fit`
function.
 |
 |      Returns
 |      -------
 |      evals_result : dictionary
 |
 |      Example
 |      -------
 |
 |      .. code-block:: python
 |
 |          param_dist = {'objective':'binary:logistic', 'n_estimators':2}
 |
 |          clf = xgb.XGBModel(**param_dist)
 |
```

```
 |          clf.fit(X_train, y_train,
 |                  eval_set=[(X_train, y_train), (X_test, y_test)],
 |                  eval_metric='logloss',
 |                  verbose=True)
 |
 |          evals_result = clf.evals_result()
 |
 |      The variable **evals_result** will contain:
 |
 |      .. code-block:: python
 |
 |          {'validation_0': {'logloss': ['0.604835', '0.531479']},
 |          'validation_1': {'logloss': ['0.41965', '0.17686']}}
 |
 |  fit(self, X, y, sample_weight=None, base_margin=None, eval_set=None,
eval_metric=None, early_stopping_rounds=None, verbose=True, xgb_model=None,
sample_weight_eval_set=None, callbacks=None)
 |      Fit gradient boosting model
 |
 |      Parameters
 |      ----------
 |      X : array_like
 |          Feature matrix
 |      y : array_like
 |          Labels
 |      sample_weight : array_like
 |          instance weights
 |      base_margin : array_like
 |          global bias for each instance.
 |      eval_set : list, optional
 |          A list of (X, y) tuple pairs to use as validation sets, for which
 |          metrics will be computed.
 |          Validation metrics will help us track the performance of the model.
 |      sample_weight_eval_set : list, optional
 |          A list of the form [L_1, L_2, …, L_n], where each L_i is a list of
 |          instance weights on the i-th validation set.
 |      eval_metric : str, list of str, or callable, optional
 |          If a str, should be a built-in evaluation metric to use. See
 |          doc/parameter.rst.
 |          If a list of str, should be the list of multiple built-in evaluation
metrics
 |          to use.
 |          If callable, a custom evaluation metric. The call
 |          signature is ``func(y_predicted, y_true)`` where ``y_true`` will be
a
 |          DMatrix object such that you may need to call the ``get_label``
 |          method. It must return a str, value pair where the str is a name
 |          for the evaluation and value is the value of the evaluation
```

```
 |          function. The callable custom objective is always minimized.
 |      early_stopping_rounds : int
 |          Activates early stopping. Validation metric needs to improve at
least once in
 |          every **early_stopping_rounds** round(s) to continue training.
 |          Requires at least one item in **eval_set**.
 |          The method returns the model from the last iteration (not the best
one).
 |          If there's more than one item in **eval_set**, the last entry will
be used
 |          for early stopping.
 |          If there's more than one metric in **eval_metric**, the last metric
will be
 |          used for early stopping.
 |          If early stopping occurs, the model will have three additional
fields:
 |          ``clf.best_score``, ``clf.best_iteration`` and
``clf.best_ntree_limit``.
 |      verbose : bool
 |          If `verbose` and an evaluation set is used, writes the evaluation
 |          metric measured on the validation set to stderr.
 |      xgb_model : str
 |          file name of stored XGBoost model or 'Booster' instance XGBoost
model to be
 |          loaded before training (allows training continuation).
 |      callbacks : list of callback functions
 |          List of callback functions that are applied at end of each
iteration.
 |          It is possible to use predefined callbacks by using
:ref:`callback_api`.
 |          Example:
 |
 |          .. code-block:: python
 |
 |              [xgb.callback.reset_learning_rate(custom_rates)]
 |
 |  get_booster(self)
 |      Get the underlying xgboost Booster of this model.
 |
 |      This will raise an exception when fit was not called
 |
 |      Returns
 |      -------
 |      booster : a xgboost booster of underlying model
 |
 |  get_num_boosting_rounds(self)
 |      Gets the number of xgboost boosting rounds.
 |
```

```
 |  get_params(self, deep=True)
 |      Get parameters.
 |
 |  get_xgb_params(self)
 |      Get xgboost specific parameters.
 |
 |  load_model(self, fname)
 |      Load the model from a file.
 |
 |      The model is loaded from an XGBoost internal format which is universal
 |      among the various XGBoost interfaces. Auxiliary attributes of the
 |      Python Booster object (such as feature names) will not be loaded.
 |
 |      Parameters
 |      ----------
 |      fname : string
 |          Input file name.
 |
 |  predict(self, data, output_margin=False, ntree_limit=None,
validate_features=True, base_margin=None)
 |      Predict with `data`.
 |
 |      .. note:: This function is not thread safe.
 |
 |        For each booster object, predict can only be called from one thread.
 |        If you want to run prediction using multiple thread, call
``xgb.copy()`` to make copies
 |        of model object and then call ``predict()``.
 |
 |        .. code-block:: python
 |
 |          preds = bst.predict(dtest, ntree_limit=num_round)
 |
 |      Parameters
 |      ----------
 |      data : numpy.array/scipy.sparse
 |          Data to predict with
 |      output_margin : bool
 |          Whether to output the raw untransformed margin value.
 |      ntree_limit : int
 |          Limit number of trees in the prediction; defaults to
best_ntree_limit if defined
 |          (i.e. it has been trained with early stopping), otherwise 0 (use all
trees).
 |      validate_features : bool
 |          When this is True, validate that the Booster's and data's
feature_names are identical.
 |          Otherwise, it is assumed that the feature_names are the same.
```

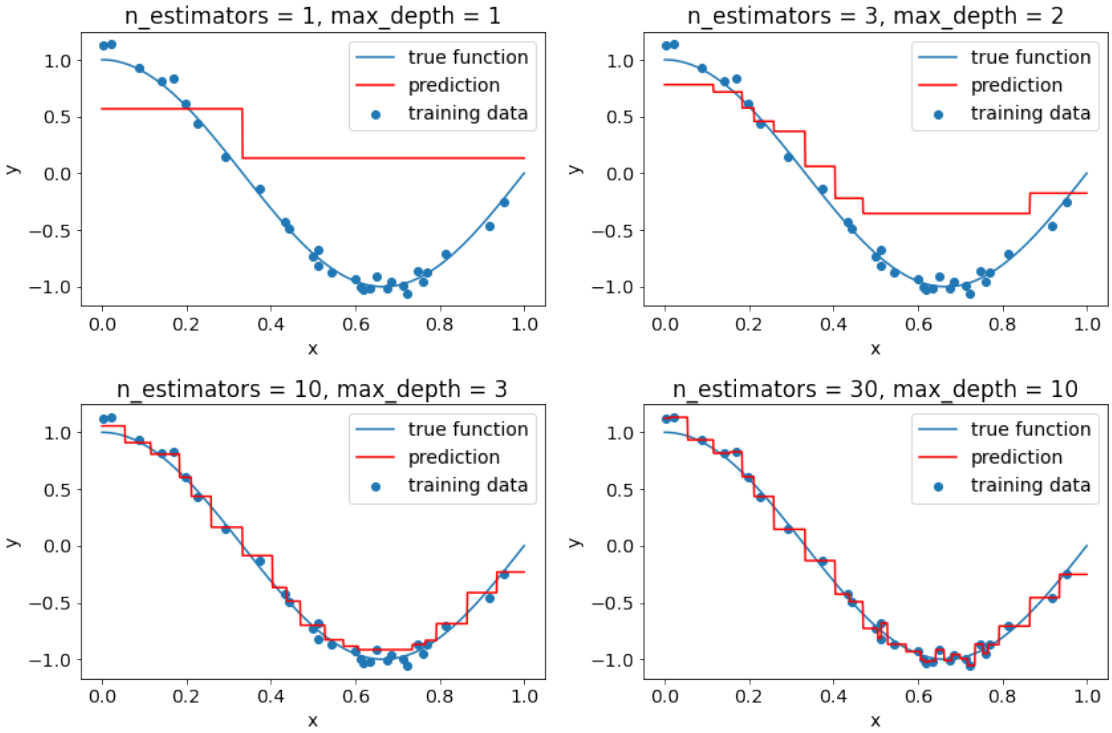
```
 |      Returns
 |      -------
 |      prediction : numpy array
 |
 |  save_model(self, fname: str)
 |      Save the model to a file.
 |
 |      The model is saved in an XGBoost internal format which is universal
 |      among the various XGBoost interfaces. Auxiliary attributes of the
 |      Python Booster object (such as feature names) will not be saved.
 |
 |        .. note::
 |
 |           See:
 |
 |           https://xgboost.readthedocs.io/en/latest/tutorials/saving_model.html
 |
 |      Parameters
 |      ----------
 |      fname : string
 |          Output file name
 |
 |  set_params(self, **params)
 |      Set the parameters of this estimator.  Modification of the sklearn
method to
 |      allow unknown kwargs. This allows using the full range of xgboost
 |      parameters that are not defined as member variables in sklearn grid
 |      search.
 |
 |      Returns
 |      -------
 |      self
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from XGBModel:
 |
 |  coef_
 |      Coefficients property
 |
 |      .. note:: Coefficients are defined only for linear learners
 |
 |          Coefficients are only defined when the linear model is chosen as
 |          base learner (`booster=gblinear`). It is not defined for other base
 |          learner types, such as tree learners (`booster=gbtree`).
 |
 |      Returns
 |      -------
 |      coef_ : array of shape ``[n_features]`` or ``[n_classes, n_features]``
```

```
 |
 |  feature_importances_
 |      Feature importances property
 |
 |      .. note:: Feature importance is defined only for tree boosters
 |
 |          Feature importance is only defined when the decision tree model is
chosen as base
 |          learner (`booster=gbtree`). It is not defined for other base learner
types, such
 |          as linear learners (`booster=gblinear`).
 |
 |      Returns
 |      -------
 |      feature_importances_ : array of shape ``[n_features]``
 |
 |  intercept_
 |      Intercept (bias) property
 |
 |      .. note:: Intercept is defined only for linear learners
 |
 |          Intercept (bias) is only defined when the linear model is chosen as
base
 |          learner (`booster=gblinear`). It is not defined for other base
learner types, such
 |          as tree learners (`booster=gbtree`).
 |
 |      Returns
 |      -------
 |      intercept_ : array of shape ``(1,)`` or ``[n_classes]``
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.BaseEstimator:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
```

```
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.base.RegressorMixin:
|
|  score(self, X, y, sample_weight=None)
|      Return the coefficient of determination R^2 of the prediction.
|
|      The coefficient R^2 is defined as (1 - u/v), where u is the residual
|      sum of squares ((y_true - y_pred) ** 2).sum() and v is the total
|      sum of squares ((y_true - y_true.mean()) ** 2).sum().
|      The best possible score is 1.0 and it can be negative (because the
|      model can be arbitrarily worse). A constant model that always
|      predicts the expected value of y, disregarding the input features,
|      would get a R^2 score of 0.0.
|
|      Parameters
|      ----------
|      X : array-like of shape (n_samples, n_features)
|          Test samples. For some estimators this may be a
|          precomputed kernel matrix or a list of generic objects instead,
|          shape = (n_samples, n_samples_fitted),
|          where n_samples_fitted is the number of
|          samples used in the fitting for the estimator.
|
|      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
|          True values for X.
|
|      sample_weight : array-like of shape (n_samples,), default=None
|          Sample weights.
|
|      Returns
|      -------
|      score : float
|          R^2 of self.predict(X) wrt. y.
|
|      Notes
|      -----
|      The R2 score used when calling ``score`` on a regressor uses
|      ``multioutput='uniform_average'`` from version 0.23 to keep consistent
|      with default value of :func:`~sklearn.metrics.r2_score`.
|      This influences the ``score`` method of all the multioutput
|      regressors (except for
|      :class:`~sklearn.multioutput.MultiOutputRegressor`).
```

[24]:
```
matplotlib.rcParams.update({'font.size': 14})


plt.figure(figsize=(12,8))

plt.subplot(2,2,1)
plt.scatter(X,y,label='training data')
plt.plot(np.linspace(0, 1, 100),true_fun(np.linspace(0, 1, 100)),label='true␣
 ↪function')
reg = XGBRegressor(n_estimators=1,max_depth=1)
reg.fit(X[:, np.newaxis],y)
y_new = reg.predict(X_new[:, np.newaxis])
plt.plot(X_new,y_new,'r',label='prediction')
plt.xlabel('x')
plt.ylabel('y')
plt.title('n_estimators = 1, max_depth = 1')
plt.legend()

plt.subplot(2,2,2)
plt.scatter(X,y,label='training data')
plt.plot(np.linspace(0, 1, 100),true_fun(np.linspace(0, 1, 100)),label='true␣
 ↪function')
reg = XGBRegressor(n_estimators=3,max_depth=2)
reg.fit(X[:, np.newaxis],y)
y_new = reg.predict(X_new[:, np.newaxis])
plt.plot(X_new,y_new,'r',label='prediction')
plt.xlabel('x')
plt.ylabel('y')
plt.title('n_estimators = 3, max_depth = 2')
plt.legend()

plt.subplot(2,2,3)
plt.scatter(X,y,label='training data')
plt.plot(np.linspace(0, 1, 100),true_fun(np.linspace(0, 1, 100)),label='true␣
 ↪function')
reg = XGBRegressor(n_estimators=10,max_depth=3)
reg.fit(X[:, np.newaxis],y)
y_new = reg.predict(X_new[:, np.newaxis])
plt.plot(X_new,y_new,'r',label='prediction')
plt.xlabel('x')
plt.ylabel('y')
plt.title('n_estimators = 10, max_depth = 3')
plt.legend()


plt.subplot(2,2,4)
plt.scatter(X,y,label='training data')
```

```
plt.plot(np.linspace(0, 1, 100),true_fun(np.linspace(0, 1, 100)),label='true␣
↪function')
reg = XGBRegressor(n_estimators=30,max_depth=10)
reg.fit(X[:, np.newaxis],y)
y_new = reg.predict(X_new[:, np.newaxis])
plt.plot(X_new,y_new,'r',label='prediction')
plt.xlabel('x')
plt.ylabel('y')
plt.title('n_estimators = 30, max_depth = 10')
plt.legend()

plt.tight_layout()
plt.savefig('figures/XGB_reg.png',dpi=300)
plt.show()
```



4.4 XGBClassifier

[26]:
```
from sklearn.datasets import make_moons

# create the data
X,y = make_moons(noise=0.2, random_state=1,n_samples=200)
```

[27]:
```
from xgboost import XGBClassifier
help(XGBClassifier)
```

```
Help on class XGBClassifier in module xgboost.sklearn:

class XGBClassifier(XGBModel, sklearn.base.ClassifierMixin)
 |  XGBClassifier(objective='binary:logistic', **kwargs)
 |
 |  Implementation of the scikit-learn API for XGBoost classification.
 |
 |
 |  Parameters
 |  ----------
 |
 |      max_depth : int
 |          Maximum tree depth for base learners.
 |      learning_rate : float
 |          Boosting learning rate (xgb's "eta")
 |      verbosity : int
 |          The degree of verbosity. Valid values are 0 (silent) - 3 (debug).
 |      objective : string or callable
 |          Specify the learning task and the corresponding learning objective
or
 |          a custom objective function to be used (see note below).
 |      booster: string
 |          Specify which booster to use: gbtree, gblinear or dart.
 |      tree_method: string
 |          Specify which tree method to use.  Default to auto.  If this
parameter
 |          is set to default, XGBoost will choose the most conservative option
 |          available.  It's recommended to study this option from parameters
 |          document.
 |      n_jobs : int
 |          Number of parallel threads used to run xgboost.
 |      gamma : float
 |          Minimum loss reduction required to make a further partition on a
leaf
 |          node of the tree.
 |      min_child_weight : int
 |          Minimum sum of instance weight(hessian) needed in a child.
 |      max_delta_step : int
 |          Maximum delta step we allow each tree's weight estimation to be.
 |      subsample : float
 |          Subsample ratio of the training instance.
 |      colsample_bytree : float
 |          Subsample ratio of columns when constructing each tree.
 |      colsample_bylevel : float
```

```
 |          Subsample ratio of columns for each level.
 |      colsample_bynode : float
 |          Subsample ratio of columns for each split.
 |      reg_alpha : float (xgb's alpha)
 |          L1 regularization term on weights
 |      reg_lambda : float (xgb's lambda)
 |          L2 regularization term on weights
 |      scale_pos_weight : float
 |          Balancing of positive and negative weights.
 |      base_score:
 |          The initial prediction score of all instances, global bias.
 |      random_state : int
 |          Random number seed.
 |
 |          .. note::
 |
 |             Using gblinear booster with shotgun updater is nondeterministic
as
 |             it uses Hogwild algorithm.
 |
 |      missing : float, default np.nan
 |          Value in the data which needs to be present as a missing value.
 |      num_parallel_tree: int
 |          Used for boosting random forest.
 |      monotone_constraints : str
 |          Constraint of variable monotonicity.  See tutorial for more
 |          information.
 |      interaction_constraints : str
 |          Constraints for interaction representing permitted interactions.
The
 |          constraints must be specified in the form of a nest list, e.g. [[0,
1],
 |          [2, 3, 4]], where each inner list is a group of indices of features
 |          that are allowed to interact with each other.  See tutorial for more
 |          information
 |      importance_type: string, default "gain"
 |          The feature importance type for the feature_importances\_ property:
 |          either "gain", "weight", "cover", "total_gain" or "total_cover".
 |
 |      \*\*kwargs : dict, optional
 |          Keyword arguments for XGBoost Booster object.  Full documentation of
 |          parameters can be found here:
 |          https://github.com/dmlc/xgboost/blob/master/doc/parameter.rst.
 |          Attempting to set a parameter via the constructor args and
\*\*kwargs
 |          dict simultaneously will result in a TypeError.
 |
 |          .. note:: \*\*kwargs unsupported by scikit-learn
```

```
 |
 |              \*\*kwargs is unsupported by scikit-learn.  We do not guarantee
 |              that parameters passed via this argument will interact properly
 |              with scikit-learn.
 |
 |          .. note::  Custom objective function
 |
 |              A custom objective function can be provided for the
``objective``
 |              parameter. In this case, it should have the signature
 |              ``objective(y_true, y_pred) -> grad, hess``:
 |
 |              y_true: array_like of shape [n_samples]
 |                  The target values
 |              y_pred: array_like of shape [n_samples]
 |                  The predicted values
 |
 |              grad: array_like of shape [n_samples]
 |                  The value of the gradient for each sample point.
 |              hess: array_like of shape [n_samples]
 |                  The value of the second derivative for each sample point
 |
 |  Method resolution order:
 |      XGBClassifier
 |      XGBModel
 |      sklearn.base.BaseEstimator
 |      sklearn.base.ClassifierMixin
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, objective='binary:logistic', **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  evals_result(self)
 |      Return the evaluation results.
 |
 |      If **eval_set** is passed to the `fit` function, you can call
 |      ``evals_result()`` to get evaluation results for all passed
**eval_sets**.
 |      When **eval_metric** is also passed to the `fit` function, the
 |      **evals_result** will contain the **eval_metrics** passed to the `fit`
function.
 |
 |      Returns
 |      -------
 |      evals_result : dictionary
 |
```

```
 |      Example
 |      -------
 |
 |      .. code-block:: python
 |
 |          param_dist = {'objective':'binary:logistic', 'n_estimators':2}
 |
 |          clf = xgb.XGBClassifier(**param_dist)
 |
 |          clf.fit(X_train, y_train,
 |                  eval_set=[(X_train, y_train), (X_test, y_test)],
 |                  eval_metric='logloss',
 |                  verbose=True)
 |
 |          evals_result = clf.evals_result()
 |
 |      The variable **evals_result** will contain
 |
 |      .. code-block:: python
 |
 |          {'validation_0': {'logloss': ['0.604835', '0.531479']},
 |          'validation_1': {'logloss': ['0.41965', '0.17686']}}
 |
 |  fit(self, X, y, sample_weight=None, base_margin=None, eval_set=None,
eval_metric=None, early_stopping_rounds=None, verbose=True, xgb_model=None,
sample_weight_eval_set=None, callbacks=None)
 |      Fit gradient boosting classifier
 |
 |      Parameters
 |      ----------
 |      X : array_like
 |          Feature matrix
 |      y : array_like
 |          Labels
 |      sample_weight : array_like
 |          instance weights
 |      base_margin : array_like
 |          global bias for each instance.
 |      eval_set : list, optional
 |          A list of (X, y) tuple pairs to use as validation sets, for which
 |          metrics will be computed.
 |          Validation metrics will help us track the performance of the model.
 |      sample_weight_eval_set : list, optional
 |          A list of the form [L_1, L_2, …, L_n], where each L_i is a list of
 |          instance weights on the i-th validation set.
 |      eval_metric : str, list of str, or callable, optional
 |          If a str, should be a built-in evaluation metric to use. See
 |          doc/parameter.rst.
```

```
 |          If a list of str, should be the list of multiple built-in evaluation
metrics
 |          to use.
 |          If callable, a custom evaluation metric. The call
 |          signature is ``func(y_predicted, y_true)`` where ``y_true`` will be
a
 |          DMatrix object such that you may need to call the ``get_label``
 |          method. It must return a str, value pair where the str is a name
 |          for the evaluation and value is the value of the evaluation
 |          function. The callable custom objective is always minimized.
 |      early_stopping_rounds : int
 |          Activates early stopping. Validation metric needs to improve at
least once in
 |          every **early_stopping_rounds** round(s) to continue training.
 |          Requires at least one item in **eval_set**.
 |          The method returns the model from the last iteration (not the best
one).
 |          If there's more than one item in **eval_set**, the last entry will
be used
 |          for early stopping.
 |          If there's more than one metric in **eval_metric**, the last metric
will be
 |          used for early stopping.
 |          If early stopping occurs, the model will have three additional
fields:
 |          ``clf.best_score``, ``clf.best_iteration`` and
``clf.best_ntree_limit``.
 |      verbose : bool
 |          If `verbose` and an evaluation set is used, writes the evaluation
 |          metric measured on the validation set to stderr.
 |      xgb_model : str
 |          file name of stored XGBoost model or 'Booster' instance XGBoost
model to be
 |          loaded before training (allows training continuation).
 |      callbacks : list of callback functions
 |          List of callback functions that are applied at end of each
iteration.
 |          It is possible to use predefined callbacks by using
:ref:`callback_api`.
 |          Example:
 |
 |          .. code-block:: python
 |
 |              [xgb.callback.reset_learning_rate(custom_rates)]
 |
 |  predict(self, data, output_margin=False, ntree_limit=None,
validate_features=True, base_margin=None)
 |      Predict with `data`.
```

```
 |
 |      .. note:: This function is not thread safe.
 |
 |        For each booster object, predict can only be called from one thread.
 |        If you want to run prediction using multiple thread, call
 |        ``xgb.copy()`` to make copies of model object and then call
 |        ``predict()``.
 |
 |        .. code-block:: python
 |
 |          preds = bst.predict(dtest, ntree_limit=num_round)
 |
 |      Parameters
 |      ----------
 |      data : array_like
 |          The dmatrix storing the input.
 |      output_margin : bool
 |          Whether to output the raw untransformed margin value.
 |      ntree_limit : int
 |          Limit number of trees in the prediction; defaults to
 |          best_ntree_limit if defined (i.e. it has been trained with early
 |          stopping), otherwise 0 (use all trees).
 |      validate_features : bool
 |          When this is True, validate that the Booster's and data's
 |          feature_names are identical.  Otherwise, it is assumed that the
 |          feature_names are the same.
 |
 |      Returns
 |      -------
 |      prediction : numpy array
 |
 |  predict_proba(self, data, ntree_limit=None, validate_features=True,
base_margin=None)
 |      Predict the probability of each `data` example being of a given class.
 |
 |      .. note:: This function is not thread safe
 |
 |          For each booster object, predict can only be called from one
 |          thread.  If you want to run prediction using multiple thread, call
 |          ``xgb.copy()`` to make copies of model object and then call predict
 |
 |      Parameters
 |      ----------
 |      data : DMatrix
 |          The dmatrix storing the input.
 |      ntree_limit : int
 |          Limit number of trees in the prediction; defaults to
best_ntree_limit if defined
```

```
 |          (i.e. it has been trained with early stopping), otherwise 0 (use all
trees).
 |      validate_features : bool
 |          When this is True, validate that the Booster's and data's
feature_names are identical.
 |          Otherwise, it is assumed that the feature_names are the same.
 |
 |      Returns
 |      -------
 |      prediction : numpy array
 |          a numpy array with the probability of each data example being of a
given class.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from XGBModel:
 |
 |  apply(self, X, ntree_limit=0)
 |      Return the predicted leaf every tree for each sample.
 |
 |      Parameters
 |      ----------
 |      X : array_like, shape=[n_samples, n_features]
 |          Input features matrix.
 |
 |      ntree_limit : int
 |          Limit number of trees in the prediction; defaults to 0 (use all
trees).
 |
 |      Returns
 |      -------
 |      X_leaves : array_like, shape=[n_samples, n_trees]
 |          For each datapoint x in X and for each tree, return the index of the
 |          leaf x ends up in. Leaves are numbered within
 |          ``[0; 2**(self.max_depth+1))``, possibly with gaps in the numbering.
 |
 |  get_booster(self)
 |      Get the underlying xgboost Booster of this model.
 |
 |      This will raise an exception when fit was not called
 |
 |      Returns
 |      -------
 |      booster : a xgboost booster of underlying model
 |
 |  get_num_boosting_rounds(self)
 |      Gets the number of xgboost boosting rounds.
 |
 |  get_params(self, deep=True)
```

```
 |      Get parameters.
 |
 |  get_xgb_params(self)
 |      Get xgboost specific parameters.
 |
 |  load_model(self, fname)
 |      Load the model from a file.
 |
 |      The model is loaded from an XGBoost internal format which is universal
 |      among the various XGBoost interfaces. Auxiliary attributes of the
 |      Python Booster object (such as feature names) will not be loaded.
 |
 |      Parameters
 |      ----------
 |      fname : string
 |          Input file name.
 |
 |  save_model(self, fname: str)
 |      Save the model to a file.
 |
 |      The model is saved in an XGBoost internal format which is universal
 |      among the various XGBoost interfaces. Auxiliary attributes of the
 |      Python Booster object (such as feature names) will not be saved.
 |
 |        .. note::
 |
 |          See:
 |
 |          https://xgboost.readthedocs.io/en/latest/tutorials/saving_model.html
 |
 |      Parameters
 |      ----------
 |      fname : string
 |          Output file name
 |
 |  set_params(self, **params)
 |      Set the parameters of this estimator.  Modification of the sklearn
method to
 |      allow unknown kwargs. This allows using the full range of xgboost
 |      parameters that are not defined as member variables in sklearn grid
 |      search.
 |
 |      Returns
 |      -------
 |      self
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from XGBModel:
```

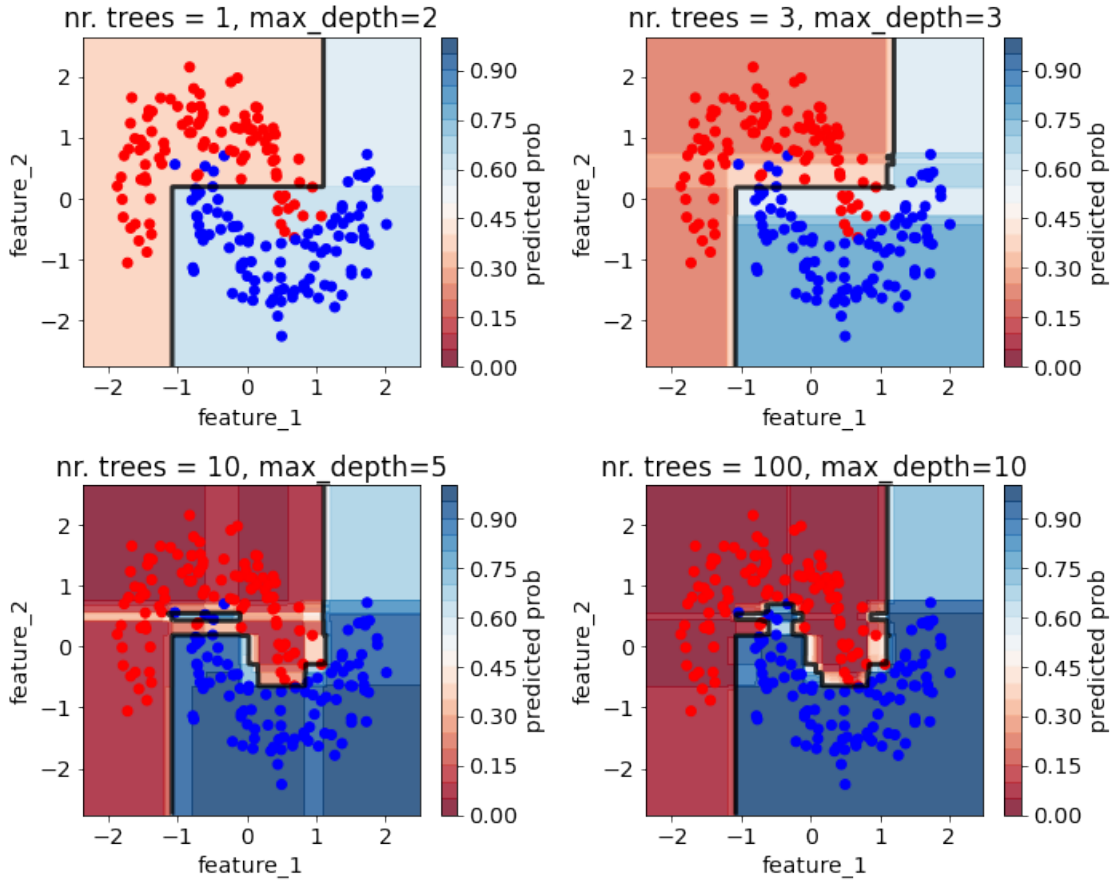
```
 |
 |  coef_
 |      Coefficients property
 |
 |      .. note:: Coefficients are defined only for linear learners
 |
 |          Coefficients are only defined when the linear model is chosen as
 |          base learner (`booster=gblinear`). It is not defined for other base
 |          learner types, such as tree learners (`booster=gbtree`).
 |
 |      Returns
 |      -------
 |      coef_ : array of shape ``[n_features]`` or ``[n_classes, n_features]``
 |
 |  feature_importances_
 |      Feature importances property
 |
 |      .. note:: Feature importance is defined only for tree boosters
 |
 |          Feature importance is only defined when the decision tree model is
chosen as base
 |          learner (`booster=gbtree`). It is not defined for other base learner
types, such
 |          as linear learners (`booster=gblinear`).
 |
 |      Returns
 |      -------
 |      feature_importances_ : array of shape ``[n_features]``
 |
 |  intercept_
 |      Intercept (bias) property
 |
 |      .. note:: Intercept is defined only for linear learners
 |
 |          Intercept (bias) is only defined when the linear model is chosen as
base
 |          learner (`booster=gblinear`). It is not defined for other base
learner types, such
 |          as tree learners (`booster=gbtree`).
 |
 |      Returns
 |      -------
 |      intercept_ : array of shape ``(1,)`` or ``[n_classes]``
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
```

```
|
|  __repr__(self, N_CHAR_MAX=700)
|      Return repr(self).
|
|  __setstate__(self, state)
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from sklearn.base.BaseEstimator:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Methods inherited from sklearn.base.ClassifierMixin:
|
|  score(self, X, y, sample_weight=None)
|      Return the mean accuracy on the given test data and labels.
|
|      In multi-label classification, this is the subset accuracy
|      which is a harsh metric since you require for each sample that
|      each label set be correctly predicted.
|
|      Parameters
|      ----------
|      X : array-like of shape (n_samples, n_features)
|          Test samples.
|
|      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
|          True labels for X.
|
|      sample_weight : array-like of shape (n_samples,), default=None
|          Sample weights.
|
|      Returns
|      -------
|      score : float
|          Mean accuracy of self.predict(X) wrt. y.
```

[28]:
```
matplotlib.rcParams.update({'font.size': 14})

X = StandardScaler().fit_transform(X)

h = .02  # step size in the mesh
```

```
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

plt.figure(figsize=(10,8))
cm_bright = ListedColormap(['#FF0000', '#0000FF'])
cm = plt.cm.RdBu

plt.subplot(2,2,1)
clf = XGBClassifier(n_estimators=1,max_depth=2,random_state=1)

clf.fit(X,y)
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cm, alpha=.8,vmin=0,vmax=1,levels=np.arange(0,1.
 ↪05,0.05))
plt.colorbar(label='predicted prob')
plt.contour(xx, yy, Z, alpha=.8,vmin=0,vmax=1,levels=[0.
 ↪5],colors=['k'],linewidths=3)
plt.scatter(X[:, 0], X[:, 1], c=y,cmap=cm_bright)
plt.xlabel('feature_1')
plt.ylabel('feature_2')
plt.title('nr. trees = 1, max_depth=2')

plt.subplot(2,2,2)
clf = XGBClassifier(n_estimators=3,max_depth=3,random_state=4)
clf.fit(X,y)
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cm, alpha=.8,vmin=0,vmax=1,levels=np.arange(0,1.
 ↪05,0.05))
plt.colorbar(label='predicted prob')
plt.contour(xx, yy, Z, alpha=.8,vmin=0,vmax=1,levels=[0.
 ↪5],colors=['k'],linewidths=3)
plt.scatter(X[:, 0], X[:, 1], c=y,cmap=cm_bright)
plt.xlabel('feature_1')
plt.ylabel('feature_2')
plt.title('nr. trees = 3, max_depth=3')

plt.subplot(2,2,3)
clf = XGBClassifier(n_estimators=10,max_depth=5,random_state=3)
clf.fit(X,y)
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
```

```
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cm, alpha=.8,vmin=0,vmax=1,levels=np.arange(0,1.
 ↪05,0.05))
plt.colorbar(label='predicted prob')
plt.contour(xx, yy, Z, alpha=.8,vmin=0,vmax=1,levels=[0.
 ↪5],colors=['k'],linewidths=3)
plt.scatter(X[:, 0], X[:, 1], c=y,cmap=cm_bright)
plt.xlabel('feature_1')
plt.ylabel('feature_2')
plt.title('nr. trees = 10, max_depth=5')

plt.subplot(2,2,4)
clf = XGBClassifier(n_estimators=100,max_depth=10,random_state=3)
clf.fit(X,y)
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cm, alpha=.8,vmin=0,vmax=1,levels=np.arange(0,1.
 ↪05,0.05))
plt.colorbar(label='predicted prob')
plt.contour(xx, yy, Z, alpha=.8,vmin=0,vmax=1,levels=[0.
 ↪5],colors=['k'],linewidths=3)
plt.scatter(X[:, 0], X[:, 1], c=y,cmap=cm_bright)
plt.xlabel('feature_1')
plt.ylabel('feature_2')
plt.title('nr. trees = 100, max_depth=10')


plt.tight_layout()

plt.savefig('figures/XGB_clf.png',dpi=300)
plt.show()
```



4.5 XGB notes

- XGB is not easy to use, it has many hyper-parameters, but it is a powerful technique
- good on any dataset size, it can train a model on multiple cores
- it works if you have missing values! See part 5 of the course series
- it can capture non-linearities, feature correlations
- the prediction is not a smoothly varying function of the features
- behaves well wrt outliers