

EXAMPLES

5.1 Miscellaneous examples

Miscellaneous and introductory examples for scikit-learn.

Note: Click [here](#) to download the full example code

5.1.1 Compact estimator representations

This example illustrates the use of the print_changed_only global parameter.

Setting print_changed_only to True will alterate the representation of estimators to only show the parameters that have been set to non-default values. This can be used to have more compact representations.

Out:

```
Default representation:  
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,  
                    intercept_scaling=1, l1_ratio=None, max_iter=100,  
                    multi_class='warn', n_jobs=None, penalty='l1',  
                    random_state=None, solver='warn', tol=0.0001, verbose=0,  
                    warm_start=False)  
  
With changed_only option:  
LogisticRegression(penalty='l1')
```

```
print(__doc__)  
  
from sklearn.linear_model import LogisticRegression  
from sklearn import set_config  
  
lr = LogisticRegression(penalty='l1')  
print('Default representation:')  
print(lr)  
# LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,  
#                   intercept_scaling=1, l1_ratio=None, max_iter=100,
```

```
#           multi_class='warn', n_jobs=None, penalty='l1',
#           random_state=None, solver='warn', tol=0.0001, verbose=0,
#           warm_start=False)

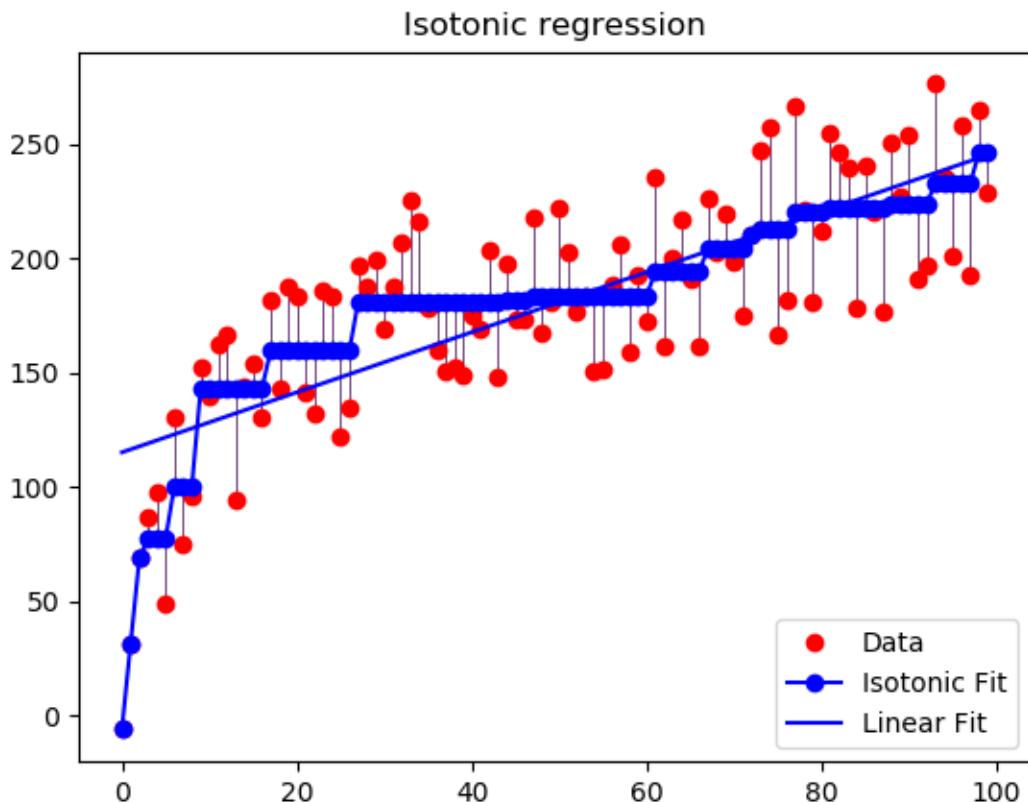
set_config(print_changed_only=True)
print ('\nWith changed_only option:')
print (lr)
# LogisticRegression(penalty='l1')
```

Total running time of the script: (0 minutes 0.003 seconds)

Note: Click [here](#) to download the full example code

5.1.2 Isotonic Regression

An illustration of the isotonic regression on generated data. The isotonic regression finds a non-decreasing approximation of a function while minimizing the mean squared error on the training data. The benefit of such a model is that it does not assume any form for the target function such as linearity. For comparison a linear regression is also presented.



```
print (__doc__)
```

```

# Author: Nelle Varoquaux <nelle.varoquaux@gmail.com>
#          Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection

from sklearn.linear_model import LinearRegression
from sklearn.isotonic import IsotonicRegression
from sklearn.utils import check_random_state

n = 100
x = np.arange(n)
rs = check_random_state(0)
y = rs.randint(-50, 50, size=(n,)) + 50. * np.log1p(np.arange(n))

# ##### Fit IsotonicRegression and LinearRegression models

ir = IsotonicRegression()

y_ = ir.fit_transform(x, y)

lr = LinearRegression()
lr.fit(x[:, np.newaxis], y) # x needs to be 2d for LinearRegression

# #### Plot result

segments = [[[i, y[i]], [i, y_[i]]] for i in range(n)]
lc = LineCollection(segments, zorder=0)
lc.set_array(np.ones(len(y)))
lc.set_linewidths(np.full(n, 0.5))

fig = plt.figure()
plt.plot(x, y, 'r.', markersize=12)
plt.plot(x, y_, 'b.-', markersize=12)
plt.plot(x, lr.predict(x[:, np.newaxis]), 'b-')
plt.gca().add_collection(lc)
plt.legend(['Data', 'Isotonic Fit', 'Linear Fit'], loc='lower right')
plt.title('Isotonic regression')
plt.show()

```

Total running time of the script: (0 minutes 0.056 seconds)

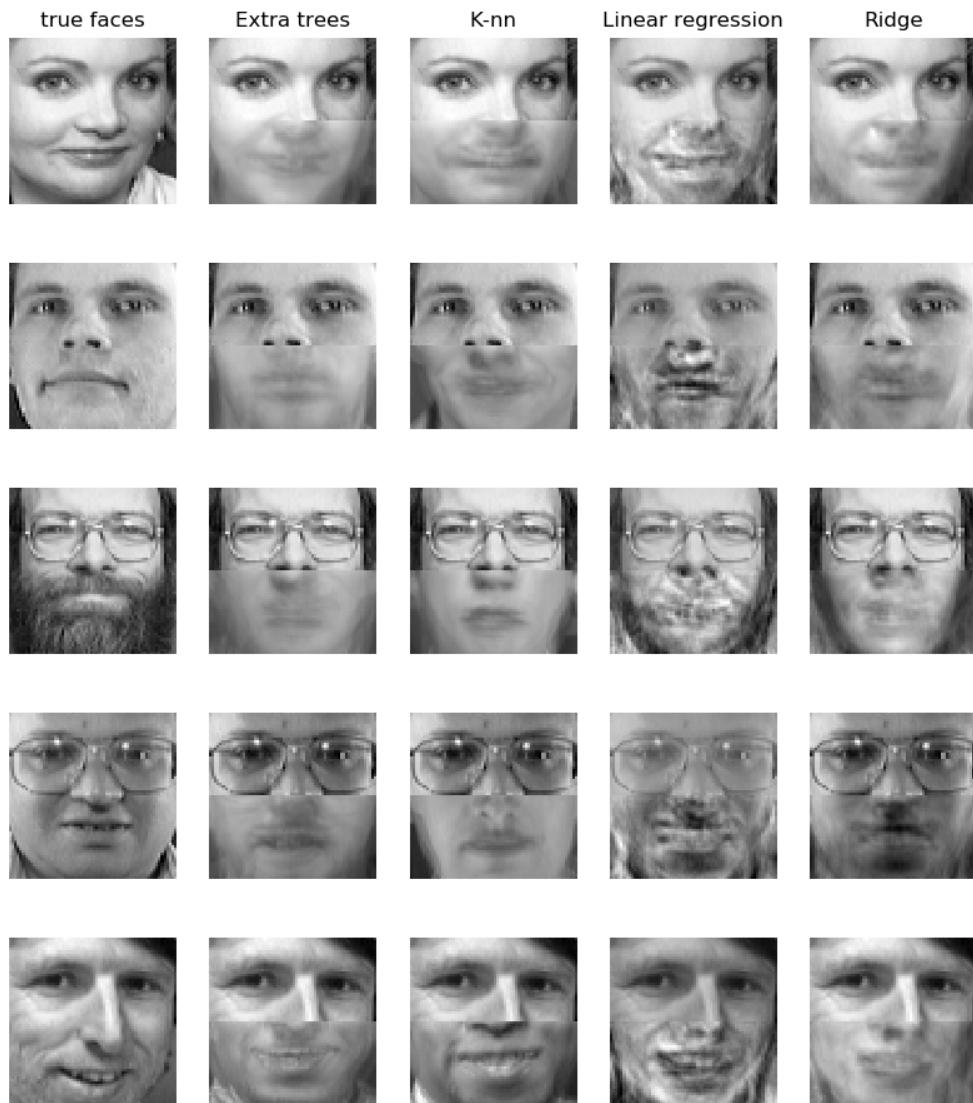
Note: Click [here](#) to download the full example code

5.1.3 Face completion with a multi-output estimators

This example shows the use of multi-output estimator to complete images. The goal is to predict the lower half of a face given its upper half.

The first column of images shows true faces. The next columns illustrate how extremely randomized trees, k nearest neighbors, linear regression and ridge regression complete the lower half of those faces.

Face completion with multi-output estimators



Out:

```
downloading Olivetti faces from https://ndownloader.figshare.com/files/5976027 to /  
→home/circleci/scikit_learn_data
```

```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_olivetti_faces
from sklearn.utils.validation import check_random_state

from sklearn.ensemble import ExtraTreesRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import RidgeCV

# Load the faces datasets
data = fetch_olivetti_faces()
targets = data.target

data = data.images.reshape((len(data.images), -1))
train = data[targets < 30]
test = data[targets >= 30] # Test on independent people

# Test on a subset of people
n_faces = 5
rng = check_random_state(4)
face_ids = rng.randint(test.shape[0], size=(n_faces, ))
test = test[face_ids, :]

n_pixels = data.shape[1]
# Upper half of the faces
X_train = train[:, :(n_pixels + 1) // 2]
# Lower half of the faces
y_train = train[:, n_pixels // 2:]
X_test = test[:, :(n_pixels + 1) // 2]
y_test = test[:, n_pixels // 2:]

# Fit estimators
ESTIMATORS = {
    "Extra trees": ExtraTreesRegressor(n_estimators=10, max_features=32,
                                        random_state=0),
    "K-nn": KNeighborsRegressor(),
    "Linear regression": LinearRegression(),
    "Ridge": RidgeCV(),
}

y_test_predict = dict()
for name, estimator in ESTIMATORS.items():
    estimator.fit(X_train, y_train)
    y_test_predict[name] = estimator.predict(X_test)

# Plot the completed faces
image_shape = (64, 64)

n_cols = 1 + len(ESTIMATORS)
plt.figure(figsize=(2. * n_cols, 2.26 * n_faces))
plt.suptitle("Face completion with multi-output estimators", size=16)

for i in range(n_faces):
    true_face = np.hstack((X_test[i], y_test_predict[i]))

```

```
if i:
    sub = plt.subplot(n_faces, n_cols, i * n_cols + 1)
else:
    sub = plt.subplot(n_faces, n_cols, i * n_cols + 1,
                      title="true faces")

sub.axis("off")
sub.imshow(true_face.reshape(image_shape),
           cmap=plt.cm.gray,
           interpolation="nearest")

for j, est in enumerate(sorted(ESTIMATORS)):
    completed_face = np.hstack((X_test[i], y_test_predict[est][i]))

    if i:
        sub = plt.subplot(n_faces, n_cols, i * n_cols + 2 + j)

    else:
        sub = plt.subplot(n_faces, n_cols, i * n_cols + 2 + j,
                          title=est)

    sub.axis("off")
    sub.imshow(completed_face.reshape(image_shape),
               cmap=plt.cm.gray,
               interpolation="nearest")

plt.show()
```

Total running time of the script: (0 minutes 3.749 seconds)

Note: Click [here](#) to download the full example code

5.1.4 Multilabel classification

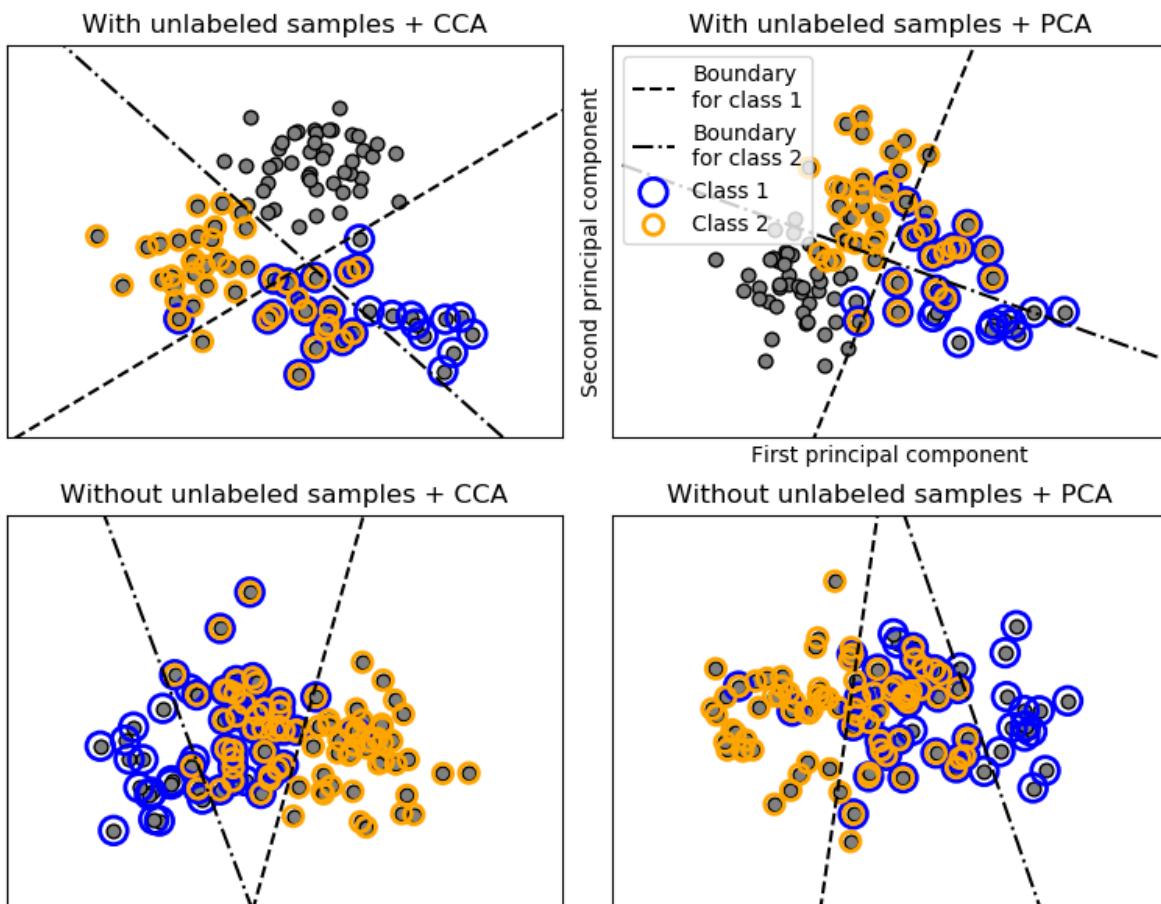
This example simulates a multi-label document classification problem. The dataset is generated randomly based on the following process:

- pick the number of labels: $n \sim \text{Poisson}(n_labels)$
- n times, choose a class c : $c \sim \text{Multinomial}(\theta)$
- pick the document length: $k \sim \text{Poisson}(\text{length})$
- k times, choose a word: $w \sim \text{Multinomial}(\theta_c)$

In the above process, rejection sampling is used to make sure that n is more than 2, and that the document length is never zero. Likewise, we reject classes which have already been chosen. The documents that are assigned to both classes are plotted surrounded by two colored circles.

The classification is performed by projecting to the first two principal components found by PCA and CCA for visualisation purposes, followed by using the `sklearn.multiclass.OneVsRestClassifier` metaclassifier using two SVCs with linear kernels to learn a discriminative model for each class. Note that PCA is used to perform an unsupervised dimensionality reduction, while CCA is used to perform a supervised one.

Note: in the plot, “unlabeled samples” does not mean that we don’t know the labels (as in semi-supervised learning) but that the samples simply do *not* have a label.



```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_multilabel_classification
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import SVC
from sklearn.decomposition import PCA
from sklearn.cross_decomposition import CCA


def plot_hyperplane(clf, min_x, max_x, linestyle, label):
    # get the separating hyperplane
    w = clf.coef_[0]
    a = -w[0] / w[1]
    xx = np.linspace(min_x - 5, max_x + 5) # make sure the line is long enough
    yy = a * xx - (clf.intercept_[0]) / w[1]
    plt.plot(xx, yy, linestyle, label=label)

def plot_subfigure(X, Y, subplot, title, transform):
    if transform == "pca":
        X = PCA(n_components=2).fit_transform(X)
    elif transform == "cca":
        X = CCA(n_components=2).fit(X, Y).transform(X)

    plt.subplot(subplot)
    plt.title(title)
    plt.scatter(X[:, 0], X[:, 1], c=Y)

```

```

else:
    raise ValueError

min_x = np.min(X[:, 0])
max_x = np.max(X[:, 0])

min_y = np.min(X[:, 1])
max_y = np.max(X[:, 1])

classif = OneVsRestClassifier(SVC(kernel='linear'))
classif.fit(X, Y)

plt.subplot(2, 2, subplot)
plt.title(title)

zero_class = np.where(Y[:, 0])
one_class = np.where(Y[:, 1])
plt.scatter(X[:, 0], X[:, 1], s=40, c='gray', edgecolors=(0, 0, 0))
plt.scatter(X[zero_class, 0], X[zero_class, 1], s=160, edgecolors='b',
            facecolors='none', linewidths=2, label='Class 1')
plt.scatter(X[one_class, 0], X[one_class, 1], s=80, edgecolors='orange',
            facecolors='none', linewidths=2, label='Class 2')

plot_hyperplane(classif.estimators_[0], min_x, max_x, 'k--',
                 'Boundary\nfor class 1')
plot_hyperplane(classif.estimators_[1], min_x, max_x, 'k-.',
                 'Boundary\nfor class 2')
plt.xticks(())
plt.yticks(())

plt.xlim(min_x - .5 * max_x, max_x + .5 * max_x)
plt.ylim(min_y - .5 * max_y, max_y + .5 * max_y)
if subplot == 2:
    plt.xlabel('First principal component')
    plt.ylabel('Second principal component')
    plt.legend(loc="upper left")

plt.figure(figsize=(8, 6))

X, Y = make_multilabel_classification(n_classes=2, n_labels=1,
                                       allow_unlabeled=True,
                                       random_state=1)

plot_subfigure(X, Y, 1, "With unlabeled samples + CCA", "cca")
plot_subfigure(X, Y, 2, "With unlabeled samples + PCA", "pca")

X, Y = make_multilabel_classification(n_classes=2, n_labels=1,
                                       allow_unlabeled=False,
                                       random_state=1)

plot_subfigure(X, Y, 3, "Without unlabeled samples + CCA", "cca")
plot_subfigure(X, Y, 4, "Without unlabeled samples + PCA", "pca")

plt.subplots_adjust(.04, .02, .97, .94, .09, .2)
plt.show()

```

Total running time of the script: (0 minutes 0.093 seconds)

Note: Click [here](#) to download the full example code

5.1.5 Comparing anomaly detection algorithms for outlier detection on toy datasets

This example shows characteristics of different anomaly detection algorithms on 2D datasets. Datasets contain one or two modes (regions of high density) to illustrate the ability of algorithms to cope with multimodal data.

For each dataset, 15% of samples are generated as random uniform noise. This proportion is the value given to the `n_u` parameter of the `OneClassSVM` and the contamination parameter of the other outlier detection algorithms. Decision boundaries between inliers and outliers are displayed in black except for Local Outlier Factor (LOF) as it has no predict method to be applied on new data when it is used for outlier detection.

The `sklearn.svm.OneClassSVM` is known to be sensitive to outliers and thus does not perform very well for outlier detection. This estimator is best suited for novelty detection when the training set is not contaminated by outliers. That said, outlier detection in high-dimension, or without any assumptions on the distribution of the inlying data is very challenging, and a One-class SVM might give useful results in these situations depending on the value of its hyperparameters.

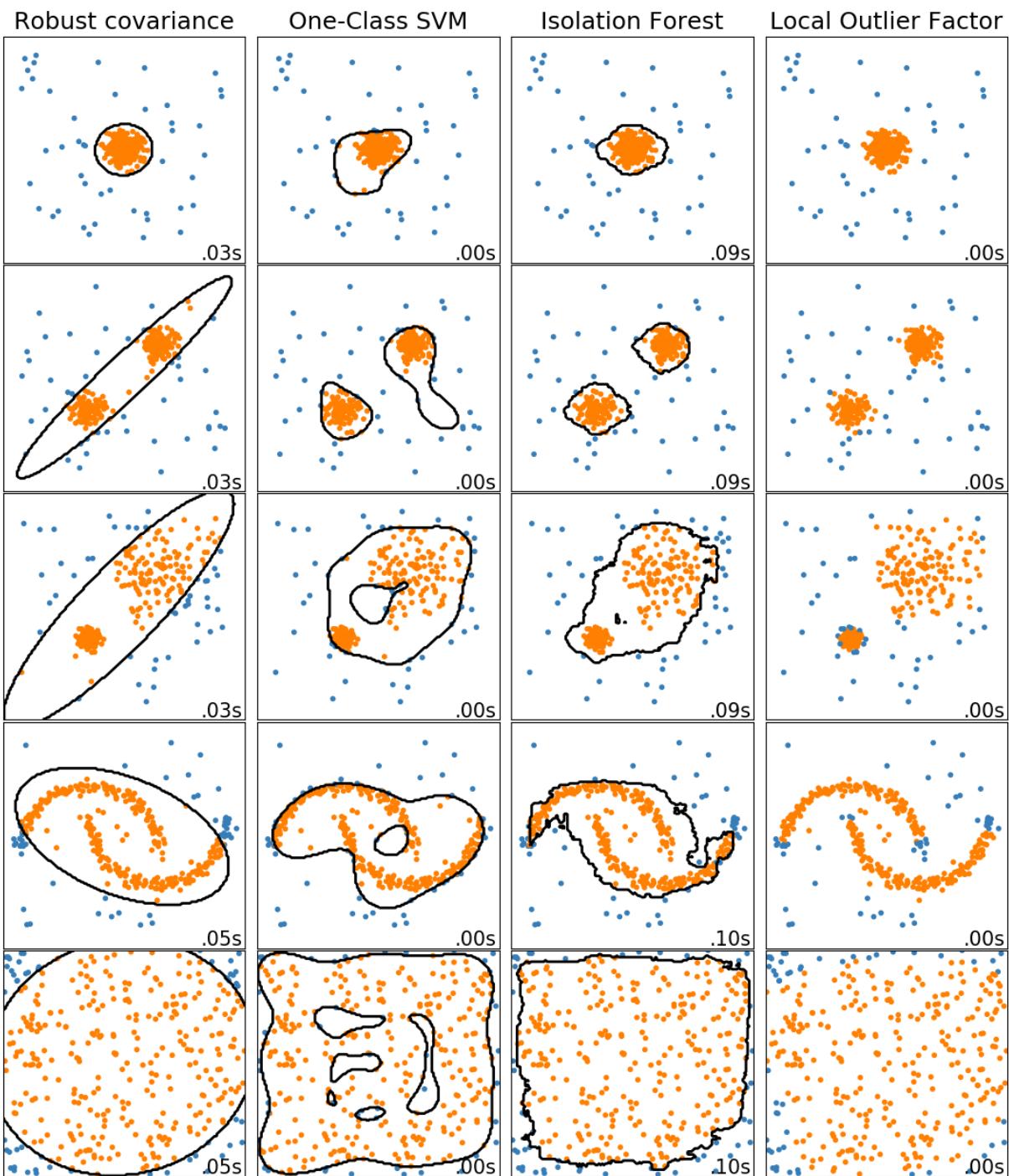
`sklearn.covariance.EllipticEnvelope` assumes the data is Gaussian and learns an ellipse. It thus degrades when the data is not unimodal. Notice however that this estimator is robust to outliers.

`sklearn.ensemble.IsolationForest` and `sklearn.neighbors.LocalOutlierFactor` seem to perform reasonably well for multi-modal data sets. The advantage of `sklearn.neighbors.LocalOutlierFactor` over the other estimators is shown for the third data set, where the two modes have different densities. This advantage is explained by the local aspect of LOF, meaning that it only compares the score of abnormality of one sample with the scores of its neighbors.

Finally, for the last data set, it is hard to say that one sample is more abnormal than another sample as they are uniformly distributed in a hypercube. Except for the `sklearn.svm.OneClassSVM` which overfits a little, all estimators present decent solutions for this situation. In such a case, it would be wise to look more closely at the scores of abnormality of the samples as a good estimator should assign similar scores to all the samples.

While these examples give some intuition about the algorithms, this intuition might not apply to very high dimensional data.

Finally, note that parameters of the models have been here handpicked but that in practice they need to be adjusted. In the absence of labelled data, the problem is completely unsupervised so model selection can be a challenge.



```
# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#         Albert Thomas <albert.thomas@telecom-paristech.fr>
# License: BSD 3 clause
```

```
import time

import numpy as np
import matplotlib
import matplotlib.pyplot as plt
```

```

from sklearn import svm
from sklearn.datasets import make_moons, make_blobs
from sklearn.covariance import EllipticEnvelope
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor

print(__doc__)

matplotlib.rcParams['contour.negative_linestyle'] = 'solid'

# Example settings
n_samples = 300
outliers_fraction = 0.15
n_outliers = int(outliers_fraction * n_samples)
n_inliers = n_samples - n_outliers

# Define outlier/anomaly detection methods to be compared
anomaly_algorithms = [
    ("Robust covariance", EllipticEnvelope(contamination=outliers_fraction)),
    ("One-Class SVM", svm.OneClassSVM(nu=outliers_fraction, kernel="rbf",
                                       gamma=0.1)),
    ("Isolation Forest", IsolationForest(behaviour='new',
                                          contamination=outliers_fraction,
                                          random_state=42)),
    ("Local Outlier Factor", LocalOutlierFactor(
        n_neighbors=35, contamination=outliers_fraction))]

# Define datasets
blobs_params = dict(random_state=0, n_samples=n_inliers, n_features=2)
datasets = [
    make_blobs(centers=[[0, 0], [0, 0]], cluster_std=0.5,
               **blobs_params)[0],
    make_blobs(centers=[[2, 2], [-2, -2]], cluster_std=[0.5, 0.5],
               **blobs_params)[0],
    make_blobs(centers=[[2, 2], [-2, -2]], cluster_std=[1.5, .3],
               **blobs_params)[0],
    4. * (make_moons(n_samples=n_samples, noise=.05, random_state=0)[0] -
           np.array([0.5, 0.25])),
    14. * (np.random.RandomState(42).rand(n_samples, 2) - 0.5)]

# Compare given classifiers under given settings
xx, yy = np.meshgrid(np.linspace(-7, 7, 150),
                     np.linspace(-7, 7, 150))

plt.figure(figsize=(len(anomaly_algorithms) * 2 + 3, 12.5))
plt.subplots_adjust(left=.02, right=.98, bottom=.001, top=.96, wspace=.05,
                   hspace=.01)

plot_num = 1
rng = np.random.RandomState(42)

for i_dataset, X in enumerate(datasets):
    # Add outliers
    X = np.concatenate([X, rng.uniform(low=-6, high=6,
                                       size=(n_outliers, 2))], axis=0)

    for name, algorithm in anomaly_algorithms:

```

```

t0 = time.time()
algorithm.fit(X)
t1 = time.time()
plt.subplot(len(datasets), len(anomaly_algorithms), plot_num)
if i_dataset == 0:
    plt.title(name, size=18)

    # fit the data and tag outliers
if name == "Local Outlier Factor":
    y_pred = algorithm.fit_predict(X)
else:
    y_pred = algorithm.fit(X).predict(X)

    # plot the levels lines and the points
if name != "Local Outlier Factor": # LOF does not implement predict
    Z = algorithm.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.contour(xx, yy, Z, levels=[0], linewidths=2, colors='black')

colors = np.array(['#377eb8', '#ff7f00'])
plt.scatter(X[:, 0], X[:, 1], s=10, color=colors[(y_pred + 1) // 2])

plt.xlim(-7, 7)
plt.ylim(-7, 7)
plt.xticks(())
plt.yticks(())
plt.text(.99, .01, ('%.2fs' % (t1 - t0)).lstrip('0'),
         transform=plt.gca().transAxes, size=15,
         horizontalalignment='right')
plot_num += 1

plt.show()

```

Total running time of the script: (0 minutes 3.491 seconds)

Note: Click [here](#) to download the full example code

5.1.6 The Johnson-Lindenstrauss bound for embedding with random projections

The Johnson-Lindenstrauss lemma states that any high dimensional dataset can be randomly projected into a lower dimensional Euclidean space while controlling the distortion in the pairwise distances.

Theoretical bounds

The distortion introduced by a random projection p is asserted by the fact that p is defining an ϵ -embedding with good probability as defined by:

$$(1 - \epsilon)\|u - v\|^2 < \|p(u) - p(v)\|^2 < (1 + \epsilon)\|u - v\|^2$$

Where u and v are any rows taken from a dataset of shape [n_samples, n_features] and p is a projection by a random Gaussian $N(0, 1)$ matrix with shape [n_components, n_features] (or a sparse Achlioptas matrix).

The minimum number of components to guarantees the ϵ -embedding is given by:

$$n_components \geq 4\log(n_samples)/(\epsilon^2/2 - \epsilon^3/3)$$

The first plot shows that with an increasing number of samples `n_samples`, the minimal number of dimensions `n_components` increased logarithmically in order to guarantee an `eps`-embedding.

The second plot shows that an increase of the admissible distortion `eps` allows to reduce drastically the minimal number of dimensions `n_components` for a given number of samples `n_samples`

Empirical validation

We validate the above bounds on the digits dataset or on the 20 newsgroups text document (TF-IDF word frequencies) dataset:

- for the digits dataset, some 8x8 gray level pixels data for 500 handwritten digits pictures are randomly projected to spaces for various larger number of dimensions `n_components`.
- for the 20 newsgroups dataset some 500 documents with 100k features in total are projected using a sparse random matrix to smaller euclidean spaces with various values for the target number of dimensions `n_components`.

The default dataset is the digits dataset. To run the example on the twenty newsgroups dataset, pass the `--twenty-newsgroups` command line argument to this script.

For each value of `n_components`, we plot:

- 2D distribution of sample pairs with pairwise distances in original and projected spaces as x and y axis respectively.
- 1D histogram of the ratio of those distances (projected / original).

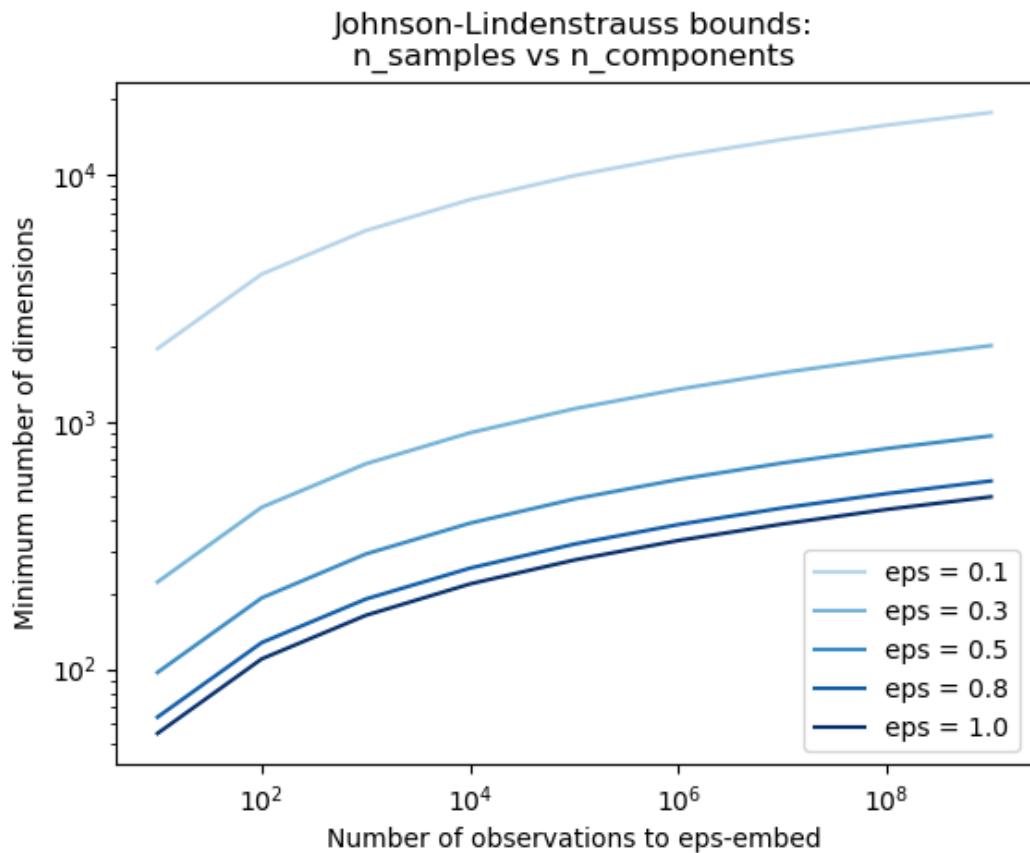
We can see that for low values of `n_components` the distribution is wide with many distorted pairs and a skewed distribution (due to the hard limit of zero ratio on the left as distances are always positives) while for larger values of `n_components` the distortion is controlled and the distances are well preserved by the random projection.

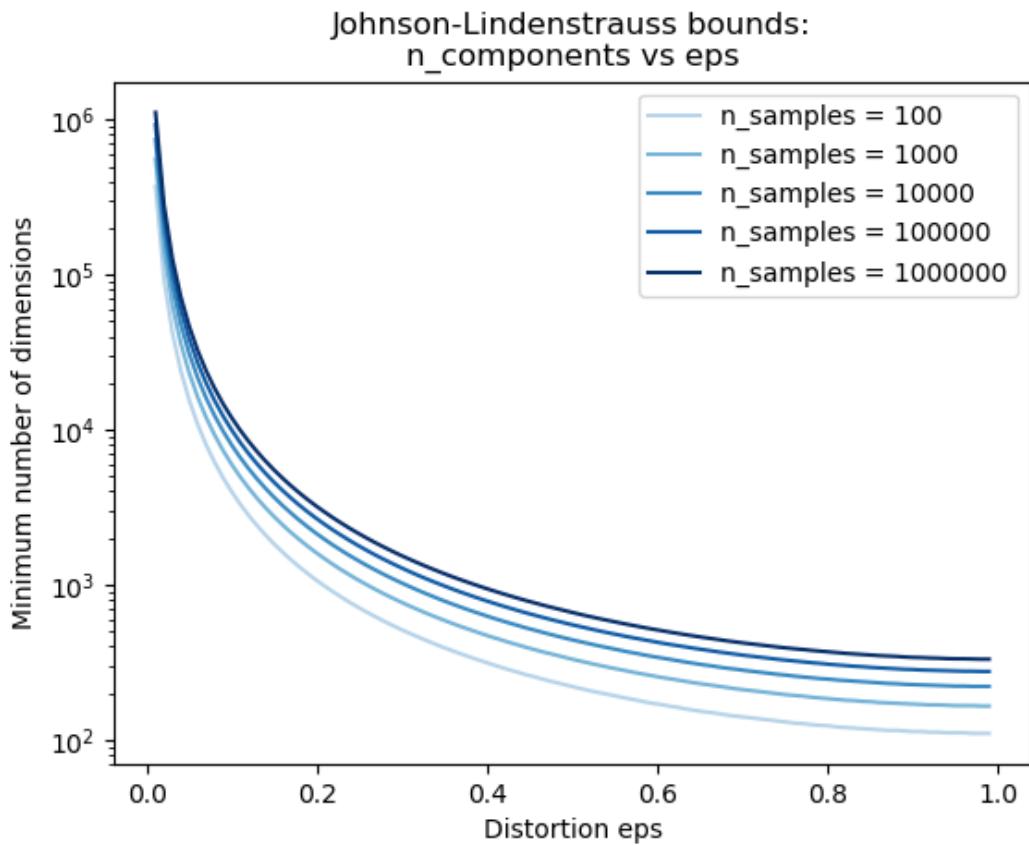
Remarks

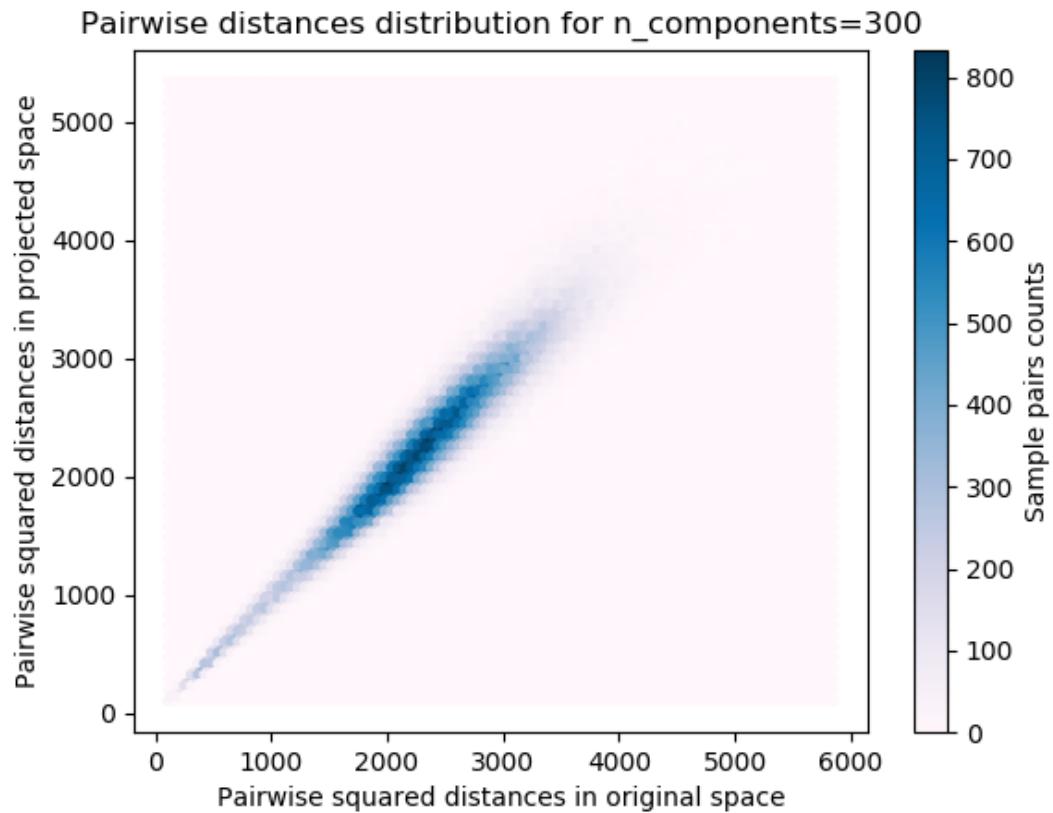
According to the JL lemma, projecting 500 samples without too much distortion will require at least several thousands dimensions, irrespective of the number of features of the original dataset.

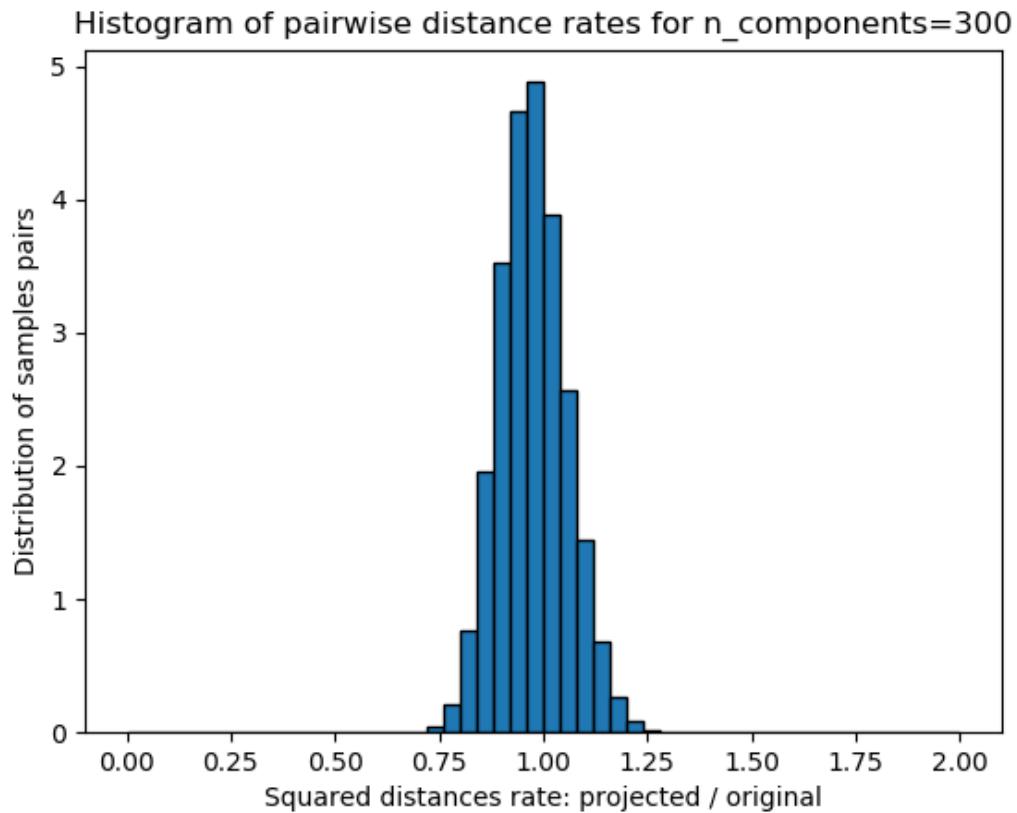
Hence using random projections on the digits dataset which only has 64 features in the input space does not make sense: it does not allow for dimensionality reduction in this case.

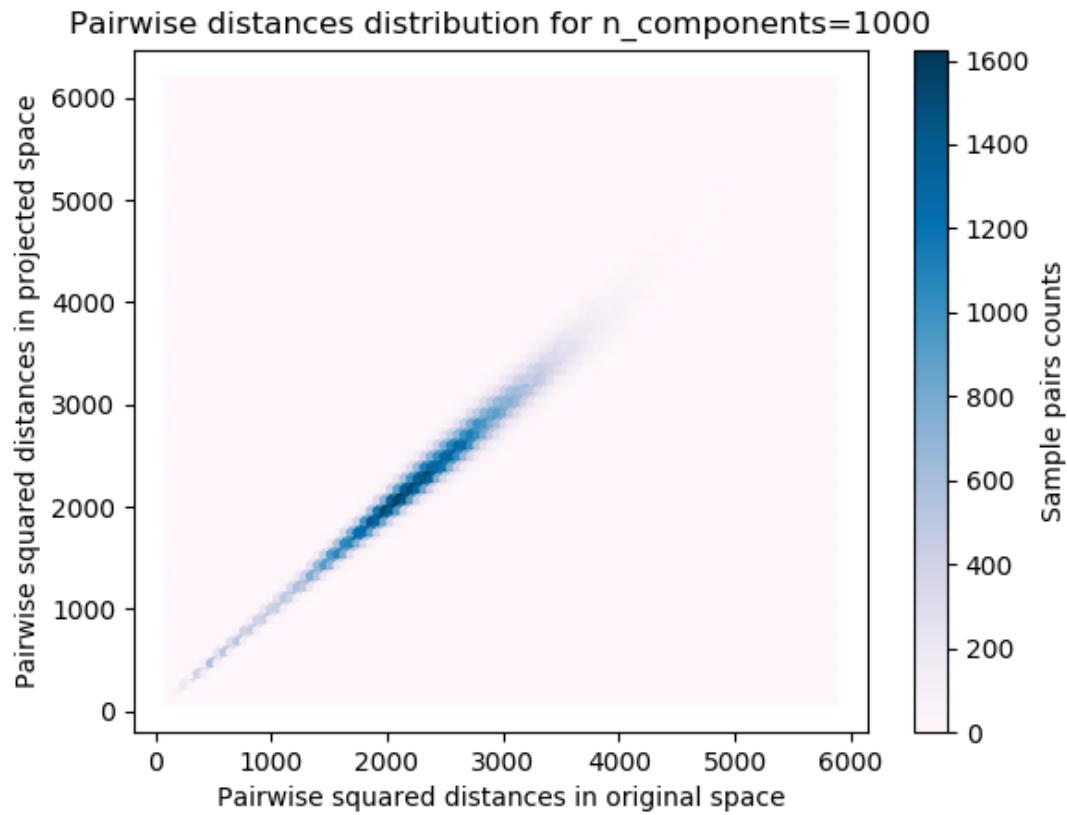
On the twenty newsgroups on the other hand the dimensionality can be decreased from 56436 down to 10000 while reasonably preserving pairwise distances.

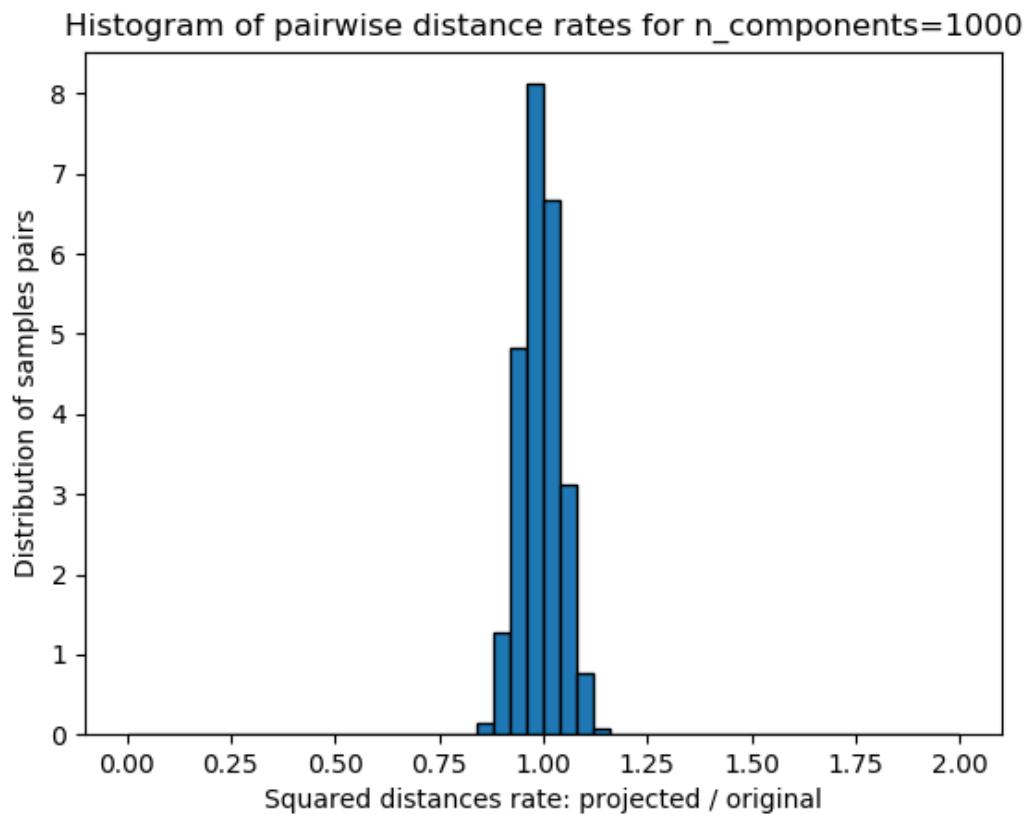


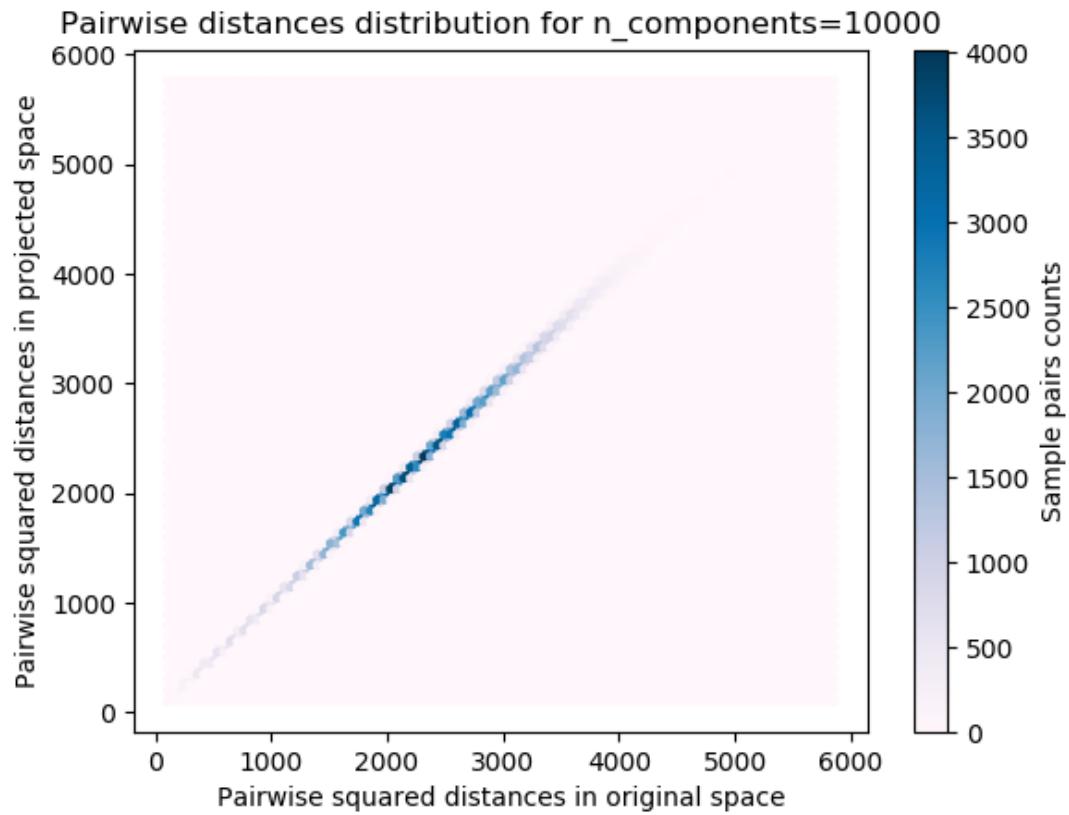


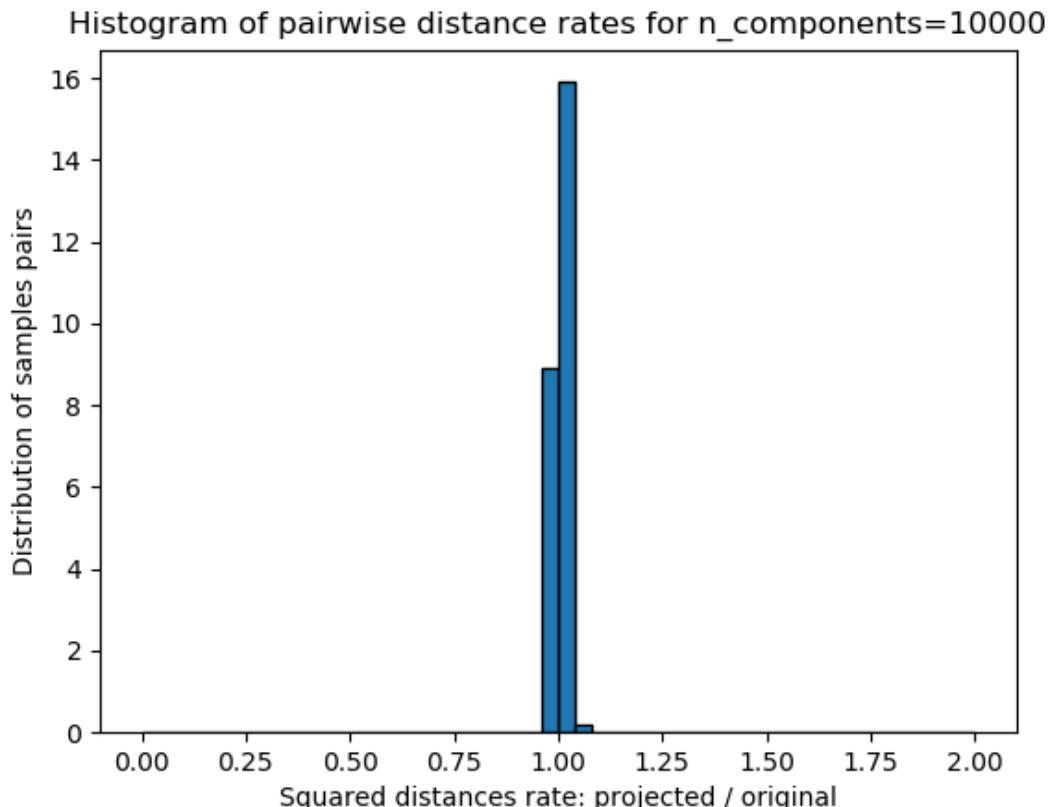












Out:

```
Embedding 500 samples with dim 64 using various random projections
Projected 500 samples from 64 to 300 in 0.016s
Random matrix with size: 0.028MB
Mean distances rate: 0.97 (0.08)
Projected 500 samples from 64 to 1000 in 0.048s
Random matrix with size: 0.096MB
Mean distances rate: 0.99 (0.05)
Projected 500 samples from 64 to 10000 in 0.594s
Random matrix with size: 0.964MB
Mean distances rate: 1.01 (0.01)
```

```
print(__doc__)

import sys
from time import time
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from distutils.version import LooseVersion
from sklearn.random_projection import johnson_lindenstrauss_min_dim
```

```

from sklearn.random_projection import SparseRandomProjection
from sklearn.datasets import fetch_20newsgroups_vectorized
from sklearn.datasets import load_digits
from sklearn.metrics.pairwise import euclidean_distances

# `normed` is being deprecated in favor of `density` in histograms
if LooseVersion(matplotlib.__version__) >= '2.1':
    density_param = {'density': True}
else:
    density_param = {'normed': True}

# Part 1: plot the theoretical dependency between n_components_min and
# n_samples

# range of admissible distortions
eps_range = np.linspace(0.1, 0.99, 5)
colors = plt.cm.Blues(np.linspace(0.3, 1.0, len(eps_range)))

# range of number of samples (observation) to embed
n_samples_range = np.logspace(1, 9, 9)

plt.figure()
for eps, color in zip(eps_range, colors):
    min_n_components = johnson_lindenstrauss_min_dim(n_samples_range, eps=eps)
    plt.loglog(n_samples_range, min_n_components, color=color)

plt.legend(["eps = %0.1f" % eps for eps in eps_range], loc="lower right")
plt.xlabel("Number of observations to eps-embed")
plt.ylabel("Minimum number of dimensions")
plt.title("Johnson-Lindenstrauss bounds:\nn_samples vs n_components")

# range of admissible distortions
eps_range = np.linspace(0.01, 0.99, 100)

# range of number of samples (observation) to embed
n_samples_range = np.logspace(2, 6, 5)
colors = plt.cm.Blues(np.linspace(0.3, 1.0, len(n_samples_range)))

plt.figure()
for n_samples, color in zip(n_samples_range, colors):
    min_n_components = johnson_lindenstrauss_min_dim(n_samples, eps=eps_range)
    plt.semilogy(eps_range, min_n_components, color=color)

plt.legend(["n_samples = %d" % n for n in n_samples_range], loc="upper right")
plt.xlabel("Distortion eps")
plt.ylabel("Minimum number of dimensions")
plt.title("Johnson-Lindenstrauss bounds:\nn_components vs eps")

# Part 2: perform sparse random projection of some digits images which are
# quite low dimensional and dense or documents of the 20 newsgroups dataset
# which is both high dimensional and sparse

if '--twenty-newsgroups' in sys.argv:
    # Need an internet connection hence not enabled by default
    data = fetch_20newsgroups_vectorized().data[:500]
else:
    data = load_digits().data[:500]

```

```

n_samples, n_features = data.shape
print("Embedding %d samples with dim %d using various random projections"
      % (n_samples, n_features))

n_components_range = np.array([300, 1000, 10000])
dists = euclidean_distances(data, squared=True).ravel()

# select only non-identical samples pairs
nonzero = dists != 0
dists = dists[nonzero]

for n_components in n_components_range:
    t0 = time()
    rp = SparseRandomProjection(n_components=n_components)
    projected_data = rp.fit_transform(data)
    print("Projected %d samples from %d to %d in %.3fs"
          % (n_samples, n_features, n_components, time() - t0))
    if hasattr(rp, 'components_'):
        n_bytes = rp.components_.data.nbytes
        n_bytes += rp.components_.indices.nbytes
        print("Random matrix with size: %.3fMB" % (n_bytes / 1e6))

    projected_dists = euclidean_distances(
        projected_data, squared=True).ravel()[nonzero]

    plt.figure()
    plt.hexbin(dists, projected_dists, gridsize=100, cmap=plt.cm.PuBu)
    plt.xlabel("Pairwise squared distances in original space")
    plt.ylabel("Pairwise squared distances in projected space")
    plt.title("Pairwise distances distribution for n_components=%d" %
              n_components)
    cb = plt.colorbar()
    cb.set_label('Sample pairs counts')

    rates = projected_dists / dists
    print("Mean distances rate: %.2f (%.2f)"
          % (np.mean(rates), np.std(rates)))

    plt.figure()
    plt.hist(rates, bins=50, range=(0., 2.), edgecolor='k', **density_param)
    plt.xlabel("Squared distances rate: projected / original")
    plt.ylabel("Distribution of samples pairs")
    plt.title("Histogram of pairwise distance rates for n_components=%d" %
              n_components)

    # TODO: compute the expected value of eps and add them to the previous plot
    # as vertical lines / region

plt.show()

```

Total running time of the script: (0 minutes 1.837 seconds)

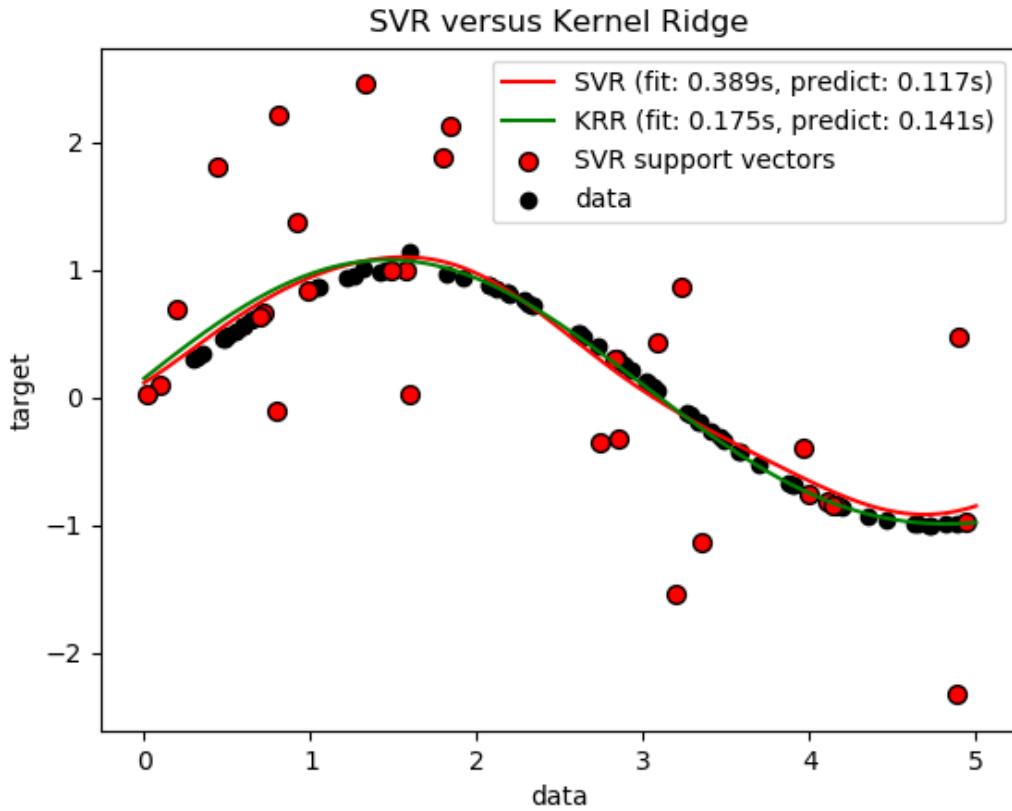
Note: Click [here](#) to download the full example code

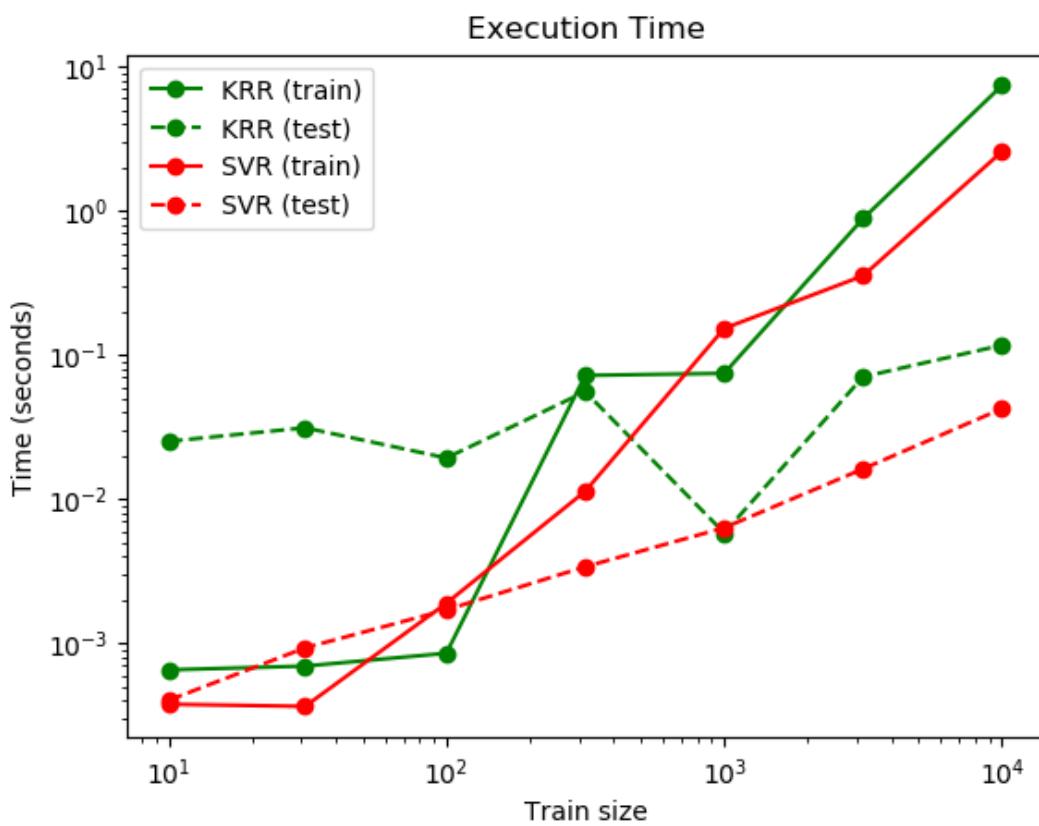
5.1.7 Comparison of kernel ridge regression and SVR

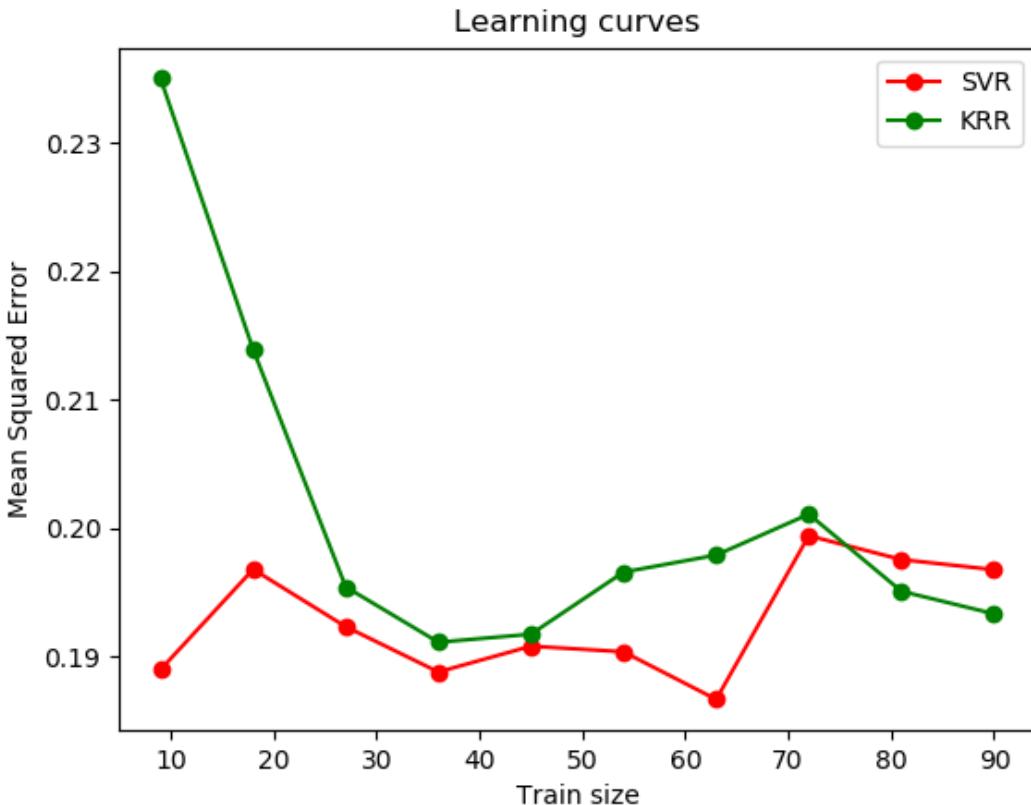
Both kernel ridge regression (KRR) and SVR learn a non-linear function by employing the kernel trick, i.e., they learn a linear function in the space induced by the respective kernel which corresponds to a non-linear function in the original space. They differ in the loss functions (ridge versus epsilon-insensitive loss). In contrast to SVR, fitting a KRR can be done in closed-form and is typically faster for medium-sized datasets. On the other hand, the learned model is non-sparse and thus slower than SVR at prediction-time.

This example illustrates both methods on an artificial dataset, which consists of a sinusoidal target function and strong noise added to every fifth datapoint. The first figure compares the learned model of KRR and SVR when both complexity/regularization and bandwidth of the RBF kernel are optimized using grid-search. The learned functions are very similar; however, fitting KRR is approx. seven times faster than fitting SVR (both with grid-search). However, prediction of 100000 target values is more than three times faster with SVR since it has learned a sparse model using only approx. 1/3 of the 100 training datapoints as support vectors.

The next figure compares the time for fitting and prediction of KRR and SVR for different sizes of the training set. Fitting KRR is faster than SVR for medium-sized training sets (less than 1000 samples); however, for larger training sets SVR scales better. With regard to prediction time, SVR is faster than KRR for all sizes of the training set because of the learned sparse solution. Note that the degree of sparsity and thus the prediction time depends on the parameters epsilon and C of the SVR.







Out:

```
SVR complexity and bandwidth selected and model fitted in 0.389 s
KRR complexity and bandwidth selected and model fitted in 0.175 s
Support vector ratio: 0.320
SVR prediction for 100000 inputs in 0.117 s
KRR prediction for 100000 inputs in 0.141 s
```

```
# Authors: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
# License: BSD 3 clause

import time

import numpy as np

from sklearn.svm import SVR
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import learning_curve
from sklearn.kernel_ridge import KernelRidge
import matplotlib.pyplot as plt
```

```

rng = np.random.RandomState(0)

# ##########
# Generate sample data
X = 5 * rng.rand(10000, 1)
y = np.sin(X).ravel()

# Add noise to targets
y[::5] += 3 * (0.5 - rng.rand(X.shape[0] // 5))

X_plot = np.linspace(0, 5, 100000)[:, None]

# #####
# Fit regression model
train_size = 100
svr = GridSearchCV(SVR(kernel='rbf', gamma=0.1), cv=5,
                    param_grid={"C": [1e0, 1e1, 1e2, 1e3],
                                "gamma": np.logspace(-2, 2, 5)})

kr = GridSearchCV(KernelRidge(kernel='rbf', gamma=0.1), cv=5,
                    param_grid={"alpha": [1e0, 0.1, 1e-2, 1e-3],
                                "gamma": np.logspace(-2, 2, 5)})

t0 = time.time()
svr.fit(X[:train_size], y[:train_size])
svr_fit = time.time() - t0
print("SVR complexity and bandwidth selected and model fitted in %.3f s"
      % svr_fit)

t0 = time.time()
kr.fit(X[:train_size], y[:train_size])
kr_fit = time.time() - t0
print("KRR complexity and bandwidth selected and model fitted in %.3f s"
      % kr_fit)

sv_ratio = svr.best_estimator_.support_.shape[0] / train_size
print("Support vector ratio: %.3f" % sv_ratio)

t0 = time.time()
y_svr = svr.predict(X_plot)
svr_predict = time.time() - t0
print("SVR prediction for %d inputs in %.3f s"
      % (X_plot.shape[0], svr_predict))

t0 = time.time()
y_kr = kr.predict(X_plot)
kr_predict = time.time() - t0
print("KRR prediction for %d inputs in %.3f s"
      % (X_plot.shape[0], kr_predict))

# #####
# Look at the results
sv_ind = svr.best_estimator_.support_
plt.scatter(X[sv_ind], y[sv_ind], c='r', s=50, label='SVR support vectors',
            zorder=2, edgecolors=(0, 0, 0))
plt.scatter(X[:100], y[:100], c='k', label='data', zorder=1,
            edgecolors=(0, 0, 0))

```

```

plt.plot(X_plot, y_svr, c='r',
         label='SVR (fit: %.3fs, predict: %.3fs)' % (svr_fit, svr_predict))
plt.plot(X_plot, y_kr, c='g',
         label='KRR (fit: %.3fs, predict: %.3fs)' % (kr_fit, kr_predict))
plt.xlabel('data')
plt.ylabel('target')
plt.title('SVR versus Kernel Ridge')
plt.legend()

# Visualize training and prediction time
plt.figure()

# Generate sample data
X = 5 * rng.rand(10000, 1)
y = np.sin(X).ravel()
y[::5] += 3 * (0.5 - rng.rand(X.shape[0] // 5))
sizes = np.logspace(1, 4, 7).astype(np.int)
for name, estimator in {"KRR": KernelRidge(kernel='rbf', alpha=0.1,
                                             gamma=10),
                        "SVR": SVR(kernel='rbf', C=1e1, gamma=10)}.items():
    train_time = []
    test_time = []
    for train_test_size in sizes:
        t0 = time.time()
        estimator.fit(X[:train_test_size], y[:train_test_size])
        train_time.append(time.time() - t0)

        t0 = time.time()
        estimator.predict(X_plot[:1000])
        test_time.append(time.time() - t0)

    plt.plot(sizes, train_time, 'o-', color="r" if name == "SVR" else "g",
             label="%s (train)" % name)
    plt.plot(sizes, test_time, 'o--', color="r" if name == "SVR" else "g",
             label="%s (test)" % name)

plt.xscale("log")
plt.yscale("log")
plt.xlabel("Train size")
plt.ylabel("Time (seconds)")
plt.title('Execution Time')
plt.legend(loc="best")

# Visualize learning curves
plt.figure()

svr = SVR(kernel='rbf', C=1e1, gamma=0.1)
kr = KernelRidge(kernel='rbf', alpha=0.1, gamma=0.1)
train_sizes, train_scores_svr, test_scores_svr = \
    learning_curve(svr, X[:100], y[:100], train_sizes=np.linspace(0.1, 1, 10),
                   scoring="neg_mean_squared_error", cv=10)
train_sizes_abs, train_scores_kr, test_scores_kr = \
    learning_curve(kr, X[:100], y[:100], train_sizes=np.linspace(0.1, 1, 10),
                   scoring="neg_mean_squared_error", cv=10)

plt.plot(train_sizes, -test_scores_svr.mean(1), 'o-', color="r",
         label="SVR")
plt.plot(train_sizes, -test_scores_kr.mean(1), 'o-', color="g",
         label="KRR")

```

```

        label="KRR")
plt.xlabel("Train size")
plt.ylabel("Mean Squared Error")
plt.title('Learning curves')
plt.legend(loc="best")

plt.show()

```

Total running time of the script: (0 minutes 13.067 seconds)

Note: Click [here](#) to download the full example code

5.1.8 Explicit feature map approximation for RBF kernels

An example illustrating the approximation of the feature map of an RBF kernel.

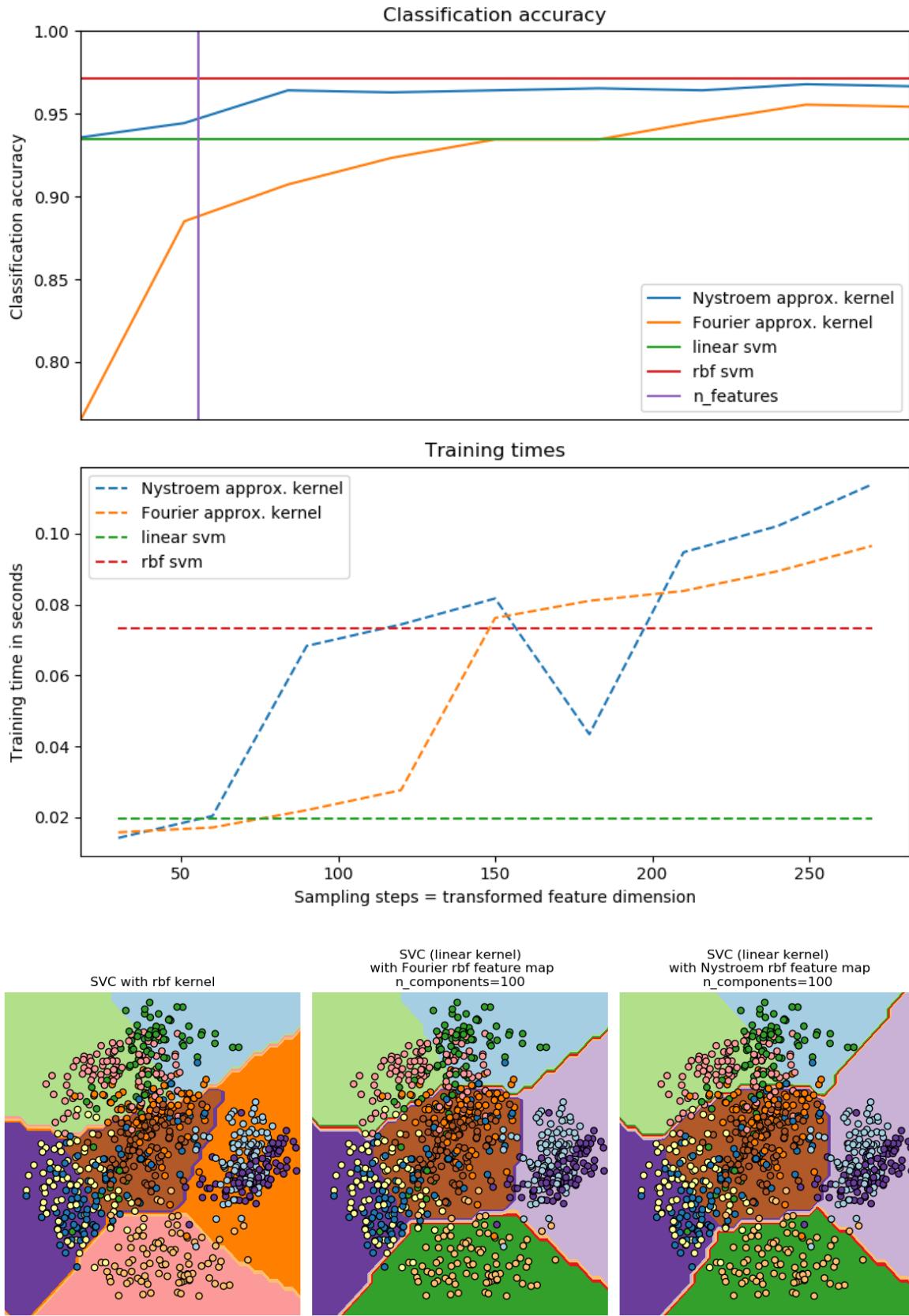
It shows how to use `RBFSampler` and `Nystroem` to approximate the feature map of an RBF kernel for classification with an SVM on the digits dataset. Results using a linear SVM in the original space, a linear SVM using the approximate mappings and using a kernelized SVM are compared. Timings and accuracy for varying amounts of Monte Carlo samplings (in the case of `RBFSampler`, which uses random Fourier features) and different sized subsets of the training set (for `Nystroem`) for the approximate mapping are shown.

Please note that the dataset here is not large enough to show the benefits of kernel approximation, as the exact SVM is still reasonably fast.

Sampling more dimensions clearly leads to better classification results, but comes at a greater cost. This means there is a tradeoff between runtime and accuracy, given by the parameter `n_components`. Note that solving the Linear SVM and also the approximate kernel SVM could be greatly accelerated by using stochastic gradient descent via `sklearn.linear_model.SGDClassifier`. This is not easily possible for the case of the kernelized SVM.

The second plot visualized the decision surfaces of the RBF kernel SVM and the linear SVM with approximate kernel maps. The plot shows decision surfaces of the classifiers projected onto the first two principal components of the data. This visualization should be taken with a grain of salt since it is just an interesting slice through the decision surface in 64 dimensions. In particular note that a datapoint (represented as a dot) does not necessarily be classified into the region it is lying in, since it will not lie on the plane that the first two principal components span.

The usage of `RBFSampler` and `Nystroem` is described in detail in [Kernel Approximation](#).



```

print(__doc__)

# Author: Gael Varoquaux <gael dot varoquaux at normalesup dot org>
#         Andreas Mueller <amueller@ais.uni-bonn.de>
# License: BSD 3 clause

# Standard scientific Python imports
import matplotlib.pyplot as plt
import numpy as np
from time import time

# Import datasets, classifiers and performance metrics
from sklearn import datasets, svm, pipeline
from sklearn.kernel_approximation import (RBFSampler,
                                            Nystroem)
from sklearn.decomposition import PCA

# The digits dataset
digits = datasets.load_digits(n_class=9)

# To apply an classifier on this data, we need to flatten the image, to
# turn the data in a (samples, feature) matrix:
n_samples = len(digits.data)
data = digits.data / 16.
data -= data.mean(axis=0)

# We learn the digits on the first half of the digits
data_train, targets_train = (data[:n_samples // 2],
                             digits.target[:n_samples // 2])

# Now predict the value of the digit on the second half:
data_test, targets_test = (data[n_samples // 2:],
                           digits.target[n_samples // 2:])
# data_test = scaler.transform(data_test)

# Create a classifier: a support vector classifier
kernel_svm = svm.SVC(gamma=.2)
linear_svm = svm.LinearSVC()

# create pipeline from kernel approximation
# and linear svm
feature_map_fourier = RBFSampler(gamma=.2, random_state=1)
feature_map_nystroem = Nystroem(gamma=.2, random_state=1)
fourier_approx_svm = pipeline.Pipeline([('feature_map', feature_map_fourier),
                                         ("svm", svm.LinearSVC())])

nystroem_approx_svm = pipeline.Pipeline([('feature_map', feature_map_nystroem),
                                         ("svm", svm.LinearSVC())])

# fit and predict using linear and kernel svm:

kernel_svm_time = time()
kernel_svm.fit(data_train, targets_train)
kernel_svm_score = kernel_svm.score(data_test, targets_test)
kernel_svm_time = time() - kernel_svm_time

linear_svm_time = time()

```

```

linear_svm.fit(data_train, targets_train)
linear_svm_score = linear_svm.score(data_test, targets_test)
linear_svm_time = time() - linear_svm_time

sample_sizes = 30 * np.arange(1, 10)
fourier_scores = []
nystroem_scores = []
fourier_times = []
nystroem_times = []

for D in sample_sizes:
    fourier_approx_svm.set_params(feature_map__n_components=D)
    nystroem_approx_svm.set_params(feature_map__n_components=D)
    start = time()
    nystroem_approx_svm.fit(data_train, targets_train)
    nystroem_times.append(time() - start)

    start = time()
    fourier_approx_svm.fit(data_train, targets_train)
    fourier_times.append(time() - start)

    fourier_score = fourier_approx_svm.score(data_test, targets_test)
    nystroem_score = nystroem_approx_svm.score(data_test, targets_test)
    nystroem_scores.append(nystroem_score)
    fourier_scores.append(fourier_score)

# plot the results:
plt.figure(figsize=(8, 8))
accuracy = plt.subplot(211)
# second y axis for timeings
timescale = plt.subplot(212)

accuracy.plot(sample_sizes, nystroem_scores, label="Nystroem approx. kernel")
timescale.plot(sample_sizes, nystroem_times, '--',
              label='Nystroem approx. kernel')

accuracy.plot(sample_sizes, fourier_scores, label="Fourier approx. kernel")
timescale.plot(sample_sizes, fourier_times, '--',
              label='Fourier approx. kernel')

# horizontal lines for exact rbf and linear kernels:
accuracy.plot([sample_sizes[0], sample_sizes[-1]],
              [linear_svm_score, linear_svm_score], label="linear svm")
timescale.plot([sample_sizes[0], sample_sizes[-1]],
              [linear_svm_time, linear_svm_time], '--', label='linear svm')

accuracy.plot([sample_sizes[0], sample_sizes[-1]],
              [kernel_svm_score, kernel_svm_score], label="rbf svm")
timescale.plot([sample_sizes[0], sample_sizes[-1]],
              [kernel_svm_time, kernel_svm_time], '--', label='rbf svm')

# vertical line for dataset dimensionality = 64
accuracy.plot([64, 64], [0.7, 1], label="n_features")

# legends and labels
accuracy.set_title("Classification accuracy")
timescale.set_title("Training times")
accuracy.set_xlim(sample_sizes[0], sample_sizes[-1])

```

```

accuracy.set_xticks(())
accuracy.set_yticks(np.min(fourier_scores), 1)
timescale.set_xlabel("Sampling steps = transformed feature dimension")
accuracy.set_ylabel("Classification accuracy")
timescale.set_ylabel("Training time in seconds")
accuracy.legend(loc='best')
timescale.legend(loc='best')

# visualize the decision surface, projected down to the first
# two principal components of the dataset
pca = PCA(n_components=8).fit(data_train)

X = pca.transform(data_train)

# Generate grid along first two principal components
multiples = np.arange(-2, 2, 0.1)
# steps along first component
first = multiples[:, np.newaxis] * pca.components_[0, :]
# steps along second component
second = multiples[:, np.newaxis] * pca.components_[1, :]
# combine
grid = first[np.newaxis, :, :] + second[:, np.newaxis, :]
flat_grid = grid.reshape(-1, data.shape[1])

# title for the plots
titles = ['SVC with rbf kernel',
          'SVC (linear kernel)\n with Fourier rbf feature map\n'
          'n_components=100',
          'SVC (linear kernel)\n with Nystroem rbf feature map\n'
          'n_components=100']

plt.tight_layout()
plt.figure(figsize=(12, 5))

# predict and plot
for i, clf in enumerate((kernel_svm, nystroem_approx_svm,
                        fourier_approx_svm)):
    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    plt.subplot(1, 3, i + 1)
    Z = clf.predict(flat_grid)

    # Put the result into a color plot
    Z = Z.reshape(grid.shape[:-1])
    plt.contourf(multiples, multiples, Z, cmap=plt.cm.Paired)
    plt.axis('off')

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=targets_train, cmap=plt.cm.Paired,
                edgecolors=(0, 0, 0))

    plt.title(titles[i])
plt.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 2.188 seconds)

5.2 Examples based on real world datasets

Applications to real world problems with some medium sized datasets or interactive user interface.

Note: Click [here](#) to download the full example code

5.2.1 Outlier detection on a real data set

This example illustrates the need for robust covariance estimation on a real data set. It is useful both for outlier detection and for a better understanding of the data structure.

We selected two sets of two variables from the Boston housing data set as an illustration of what kind of analysis can be done with several outlier detection tools. For the purpose of visualization, we are working with two-dimensional examples, but one should be aware that things are not so trivial in high-dimension, as it will be pointed out.

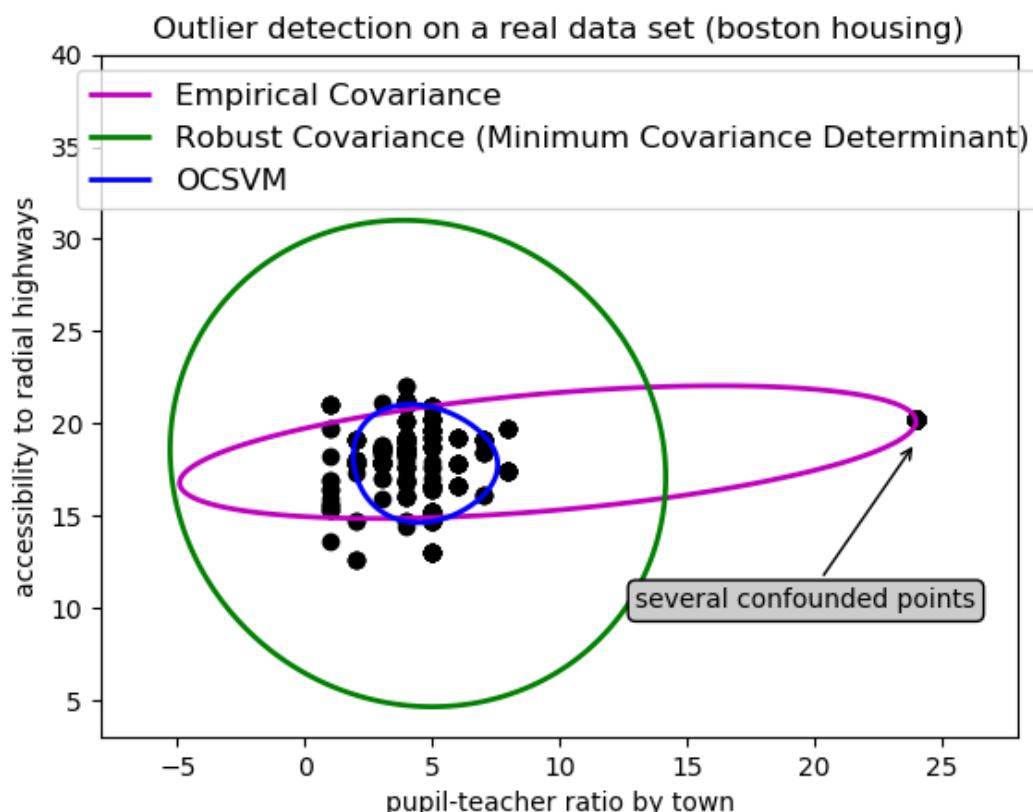
In both examples below, the main result is that the empirical covariance estimate, as a non-robust one, is highly influenced by the heterogeneous structure of the observations. Although the robust covariance estimate is able to focus on the main mode of the data distribution, it sticks to the assumption that the data should be Gaussian distributed, yielding some biased estimation of the data structure, but yet accurate to some extent. The One-Class SVM does not assume any parametric form of the data distribution and can therefore model the complex shape of the data much better.

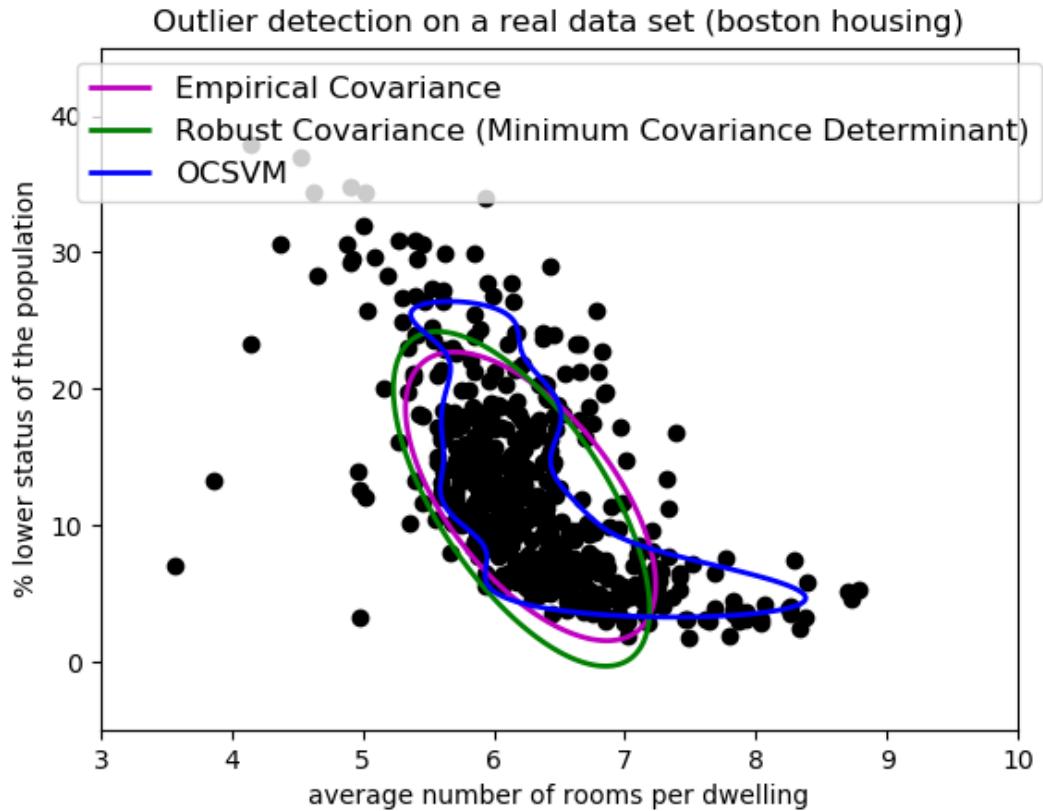
First example

The first example illustrates how robust covariance estimation can help concentrating on a relevant cluster when another one exists. Here, many observations are confounded into one and break down the empirical covariance estimation. Of course, some screening tools would have pointed out the presence of two clusters (Support Vector Machines, Gaussian Mixture Models, univariate outlier detection, ...). But had it been a high-dimensional example, none of these could be applied that easily.

Second example

The second example shows the ability of the Minimum Covariance Determinant robust estimator of covariance to concentrate on the main mode of the data distribution: the location seems to be well estimated, although the covariance is hard to estimate due to the banana-shaped distribution. Anyway, we can get rid of some outlying observations. The One-Class SVM is able to capture the real data structure, but the difficulty is to adjust its kernel bandwidth parameter so as to obtain a good compromise between the shape of the data scatter matrix and the risk of over-fitting the data.





```

print(__doc__)

# Author: Virgile Fritsch <virgile.fritsch@inria.fr>
# License: BSD 3 clause

import numpy as np
from sklearn.covariance import EllipticEnvelope
from sklearn.svm import OneClassSVM
import matplotlib.pyplot as plt
import matplotlib.font_manager
from sklearn.datasets import load_boston

# Get data
X1 = load_boston()['data'][:, [8, 10]] # two clusters
X2 = load_boston()['data'][:, [5, 12]] # "banana"-shaped

# Define "classifiers" to be used
classifiers = {
    "Empirical Covariance": EllipticEnvelope(support_fraction=1.,
                                              contamination=0.261),
    "Robust Covariance (Minimum Covariance Determinant)": EllipticEnvelope(contamination=0.261),
    "OCSVM": OneClassSVM(nu=0.261, gamma=0.05)}
colors = ['m', 'g', 'b']
legend1 = {}
legend2 = {}

```

```

# Learn a frontier for outlier detection with several classifiers
xx1, yy1 = np.meshgrid(np.linspace(-8, 28, 500), np.linspace(3, 40, 500))
xx2, yy2 = np.meshgrid(np.linspace(3, 10, 500), np.linspace(-5, 45, 500))
for i, (clf_name, clf) in enumerate(classifiers.items()):
    plt.figure(1)
    clf.fit(X1)
    Z1 = clf.decision_function(np.c_[xx1.ravel(), yy1.ravel()])
    Z1 = Z1.reshape(xx1.shape)
    legend1[clf_name] = plt.contour(
        xx1, yy1, Z1, levels=[0], linewidths=2, colors=colors[i])
    plt.figure(2)
    clf.fit(X2)
    Z2 = clf.decision_function(np.c_[xx2.ravel(), yy2.ravel()])
    Z2 = Z2.reshape(xx2.shape)
    legend2[clf_name] = plt.contour(
        xx2, yy2, Z2, levels=[0], linewidths=2, colors=colors[i])

legend1_values_list = list(legend1.values())
legend1_keys_list = list(legend1.keys())

# Plot the results (= shape of the data points cloud)
plt.figure(1) # two clusters
plt.title("Outlier detection on a real data set (boston housing)")
plt.scatter(X1[:, 0], X1[:, 1], color='black')
bbox_args = dict(boxstyle="round", fc="0.8")
arrow_args = dict(arrowstyle="->")
plt.annotate("several confounded points", xy=(24, 19),
            xycoords="data", textcoords="data",
            xytext=(13, 10), bbox=bbox_args, arrowprops=arrow_args)
plt.xlim((xx1.min(), xx1.max()))
plt.ylim((yy1.min(), yy1.max()))
plt.legend((legend1_values_list[0].collections[0],
            legend1_values_list[1].collections[0],
            legend1_values_list[2].collections[0]),
            (legend1_keys_list[0], legend1_keys_list[1], legend1_keys_list[2]),
            loc="upper center",
            prop=matplotlib.font_manager.FontProperties(size=12))
plt.ylabel("accessibility to radial highways")
plt.xlabel("pupil-teacher ratio by town")

legend2_values_list = list(legend2.values())
legend2_keys_list = list(legend2.keys())

plt.figure(2) # "banana" shape
plt.title("Outlier detection on a real data set (boston housing)")
plt.scatter(X2[:, 0], X2[:, 1], color='black')
plt.xlim((xx2.min(), xx2.max()))
plt.ylim((yy2.min(), yy2.max()))
plt.legend((legend2_values_list[0].collections[0],
            legend2_values_list[1].collections[0],
            legend2_values_list[2].collections[0]),
            (legend2_keys_list[0], legend2_keys_list[1], legend2_keys_list[2]),
            loc="upper center",
            prop=matplotlib.font_manager.FontProperties(size=12))
plt.ylabel("% lower status of the population")
plt.xlabel("average number of rooms per dwelling")

plt.show()

```

Total running time of the script: (0 minutes 3.436 seconds)

Note: Click [here](#) to download the full example code

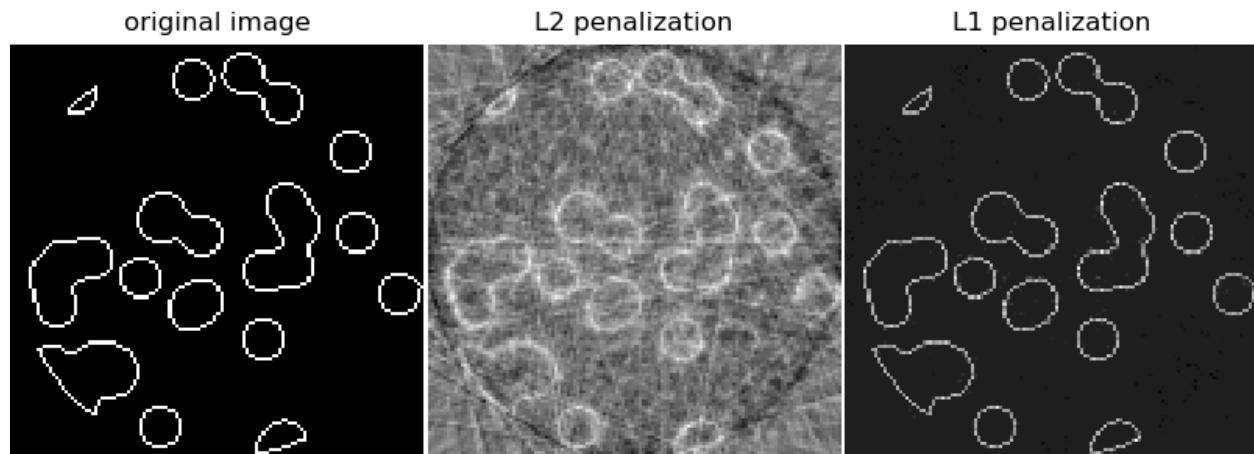
5.2.2 Compressive sensing: tomography reconstruction with L1 prior (Lasso)

This example shows the reconstruction of an image from a set of parallel projections, acquired along different angles. Such a dataset is acquired in **computed tomography** (CT).

Without any prior information on the sample, the number of projections required to reconstruct the image is of the order of the linear size l of the image (in pixels). For simplicity we consider here a sparse image, where only pixels on the boundary of objects have a non-zero value. Such data could correspond for example to a cellular material. Note however that most images are sparse in a different basis, such as the Haar wavelets. Only $1/7$ projections are acquired, therefore it is necessary to use prior information available on the sample (its sparsity): this is an example of **compressive sensing**.

The tomography projection operation is a linear transformation. In addition to the data-fidelity term corresponding to a linear regression, we penalize the L1 norm of the image to account for its sparsity. The resulting optimization problem is called the **Lasso**. We use the class `sklearn.linear_model.Lasso`, that uses the coordinate descent algorithm. Importantly, this implementation is more computationally efficient on a sparse matrix, than the projection operator used here.

The reconstruction with L1 penalization gives a result with zero error (all pixels are successfully labeled with 0 or 1), even if noise was added to the projections. In comparison, an L2 penalization (`sklearn.linear_model.Ridge`) produces a large number of labeling errors for the pixels. Important artifacts are observed on the reconstructed image, contrary to the L1 penalization. Note in particular the circular artifact separating the pixels in the corners, that have contributed to fewer projections than the central disk.



```
print(__doc__)

# Author: Emmanuelle Gouillart <emmanuelle.gouillart@nsup.org>
# License: BSD 3 clause

import numpy as np
from scipy import sparse
from scipy import ndimage
from sklearn.linear_model import Lasso
```

```

from sklearn.linear_model import Ridge
import matplotlib.pyplot as plt

def _weights(x, dx=1, orig=0):
    x = np.ravel(x)
    floor_x = np.floor((x - orig) / dx).astype(np.int64)
    alpha = (x - orig - floor_x * dx) / dx
    return np.hstack((floor_x, floor_x + 1)), np.hstack((1 - alpha, alpha))

def _generate_center_coordinates(l_x):
    X, Y = np.mgrid[:l_x, :l_x].astype(np.float64)
    center = l_x / 2.
    X += 0.5 - center
    Y += 0.5 - center
    return X, Y

def build_projection_operator(l_x, n_dir):
    """ Compute the tomography design matrix.

    Parameters
    -----
    l_x : int
        linear size of image array

    n_dir : int
        number of angles at which projections are acquired.

    Returns
    -----
    p : sparse matrix of shape (n_dir l_x, l_x**2)
    """
    X, Y = _generate_center_coordinates(l_x)
    angles = np.linspace(0, np.pi, n_dir, endpoint=False)
    data_inds, weights, camera_inds = [], [], []
    data_unravel_indices = np.arange(l_x ** 2)
    data_unravel_indices = np.hstack((data_unravel_indices,
                                      data_unravel_indices))
    for i, angle in enumerate(angles):
        Xrot = np.cos(angle) * X - np.sin(angle) * Y
        inds, w = _weights(Xrot, dx=1, orig=X.min())
        mask = np.logical_and(inds >= 0, inds < l_x)
        weights += list(w[mask])
        camera_inds += list(inds[mask] + i * l_x)
        data_inds += list(data_unravel_indices[mask])
    proj_operator = sparse.coo_matrix((weights, (camera_inds, data_inds)))
    return proj_operator

def generate_synthetic_data():
    """ Synthetic binary data """
    rs = np.random.RandomState(0)
    n_pts = 36
    x, y = np.ogrid[0:1, 0:1]
    mask_outer = (x - 1 / 2.) ** 2 + (y - 1 / 2.) ** 2 < (1 / 2.) ** 2

```

```
mask = np.zeros((l, l))
points = l * rs.rand(2, n_pts)
mask[(points[0]).astype(np.int), (points[1]).astype(np.int)] = 1
mask = ndimage.gaussian_filter(mask, sigma=l / n_pts)
res = np.logical_and(mask > mask.mean(), mask_outer)
return np.logical_xor(res, ndimage.binary_erosion(res))

# Generate synthetic images, and projections
l = 128
proj_operator = build_projection_operator(l, l // 7)
data = generate_synthetic_data()
proj = proj_operator * data.ravel()[:, np.newaxis]
proj += 0.15 * np.random.randn(*proj.shape)

# Reconstruction with L2 (Ridge) penalization
rgr_ridge = Ridge(alpha=0.2)
rgr_ridge.fit(proj_operator, proj.ravel())
rec_l2 = rgr_ridge.coef_.reshape(l, l)

# Reconstruction with L1 (Lasso) penalization
# the best value of alpha was determined using cross validation
# with LassoCV
rgr_lasso = Lasso(alpha=0.001)
rgr_lasso.fit(proj_operator, proj.ravel())
rec_l1 = rgr_lasso.coef_.reshape(l, l)

plt.figure(figsize=(8, 3.3))
plt.subplot(131)
plt.imshow(data, cmap=plt.cm.gray, interpolation='nearest')
plt.axis('off')
plt.title('original image')
plt.subplot(132)
plt.imshow(rec_l2, cmap=plt.cm.gray, interpolation='nearest')
plt.title('L2 penalization')
plt.axis('off')
plt.subplot(133)
plt.imshow(rec_l1, cmap=plt.cm.gray, interpolation='nearest')
plt.title('L1 penalization')
plt.axis('off')

plt.subplots_adjust(hspace=0.01, wspace=0.01, top=1, bottom=0, left=0,
                   right=1)

plt.show()
```

Total running time of the script: (0 minutes 9.761 seconds)

Note: Click [here](#) to download the full example code

5.2.3 Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation

This is an example of applying `sklearn.decomposition.NMF` and `sklearn.decomposition.LatentDirichletAllocation` on a corpus of documents and extract additive models of the topic structure

of the corpus. The output is a list of topics, each represented as a list of terms (weights are not shown).

Non-negative Matrix Factorization is applied with two different objective functions: the Frobenius norm, and the generalized Kullback-Leibler divergence. The latter is equivalent to Probabilistic Latent Semantic Indexing.

The default parameters (n_samples / n_features / n_components) should make the example runnable in a couple of tens of seconds. You can try to increase the dimensions of the problem, but be aware that the time complexity is polynomial in NMF. In LDA, the time complexity is proportional to (n_samples * iterations).

Out:

```

Loading dataset...
done in 7.911s.
Extracting tf-idf features for NMF...
done in 0.268s.
Extracting tf features for LDA...
done in 0.254s.

Fitting the NMF model (Frobenius norm) with tf-idf features, n_samples=2000 and n_
↳features=1000...
done in 0.406s.

Topics in NMF model (Frobenius norm):
Topic #0: just people don think like know time good make way really say right ve want_
↳did ll new use years
Topic #1: windows use dos using window program os drivers application help software_
↳pc running ms screen files version card code work
Topic #2: god jesus bible faith christian christ christians does heaven sin believe_
↳lord life church mary atheism belief human love religion
Topic #3: thanks know does mail advance hi info interested email anybody looking card_
↳help like appreciated information send list video need
Topic #4: car cars tires miles 00 new engine insurance price condition oil power_
↳speed good 000 brake year models used bought
Topic #5: edu soon com send university internet mit ftp mail cc pub article_
↳information hope program mac email home contact blood
Topic #6: file problem files format win sound ftp pub read save site help image_
↳available create copy running memory self version
Topic #7: game team games year win play season players nhl runs goal hockey toronto_
↳division flyers player defense leafs bad teams
Topic #8: drive drives hard disk floppy software card mac computer power scsi_
↳controller apple mb 00 pc rom sale problem internal
Topic #9: key chip clipper keys encryption government public use secure enforcement_
↳phone nsa communications law encrypted security clinton used legal standard

Fitting the NMF model (generalized Kullback-Leibler divergence) with tf-idf features,_
↳n_samples=2000 and n_features=1000...
done in 1.769s.

Topics in NMF model (generalized Kullback-Leibler divergence):
Topic #0: just people don like did know make really right think say things time look_
↳way didn ve course probably good
Topic #1: help thanks windows know hi need using does looking anybody appreciated_
↳card mail software use info email ftp available pc
Topic #2: does god believe know mean true christians read point jesus christian_
↳church come people fact says religion say agree bible
Topic #3: know thanks mail interested like new just bike email edu advance want_
↳contact really list heard com post hear information
Topic #4: 10 new 30 12 20 50 11 sale 16 15 time 14 old power ago good 100 great offer_
↳cost

```

```

Topic #5: number 1993 data subject government new numbers provide information space
↳following com research include large note group major time talk
Topic #6: edu problem file com remember try soon article mike files code program sun
↳free send think cases manager little called
Topic #7: game year team games world fact second case won said win division play best
↳clearly claim allow example used doesn
Topic #8: think don drive hard need bit mac make sure read apple going comes disk
↳computer case pretty drives software ve
Topic #9: good just use like doesn got way don 11 going does chip better doing bad
↳key want sure bit car

```

Fitting LDA models with tf features, n_samples=2000 and n_features=1000...
done in 3.167s.

Topics in LDA model:

```

Topic #0: edu com mail send graphics ftp pub available contact university list faq ca
↳information cs 1993 program sun uk mit
Topic #1: don like just know think ve way use right good going make sure 11 point got
↳need really time doesn
Topic #2: christian think atheism faith pittsburgh new bible radio games alt lot just
↳religion like book read play time subject believe
Topic #3: drive disk windows thanks use card drives hard version pc software file
↳using scsi help does new dos controller 16
Topic #4: hiv health aids disease april medical care research 1993 light information
↳study national service test led 10 page new drug
Topic #5: god people does just good don jesus say israel way life know true fact time
↳law want believe make think
Topic #6: 55 10 11 18 15 team game 19 period play 23 12 13 flyers 20 25 22 17 24 16
Topic #7: car year just cars new engine like bike good oil insurance better tires 000
↳thing speed model brake driving performance
Topic #8: people said did just didn know time like went think children came come don
↳took years say dead told started
Topic #9: key space law government public use encryption earth section security moon
↳probe enforcement keys states lunar military crime surface technology

```

```

# Author: Olivier Grisel <olivier.grisel@ensta.org>
#         Lars Buitinck
#         Chyi-Kwei Yau <chyikwei.yau@gmail.com>
# License: BSD 3 clause

from time import time

from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.decomposition import NMF, LatentDirichletAllocation
from sklearn.datasets import fetch_20newsgroups

n_samples = 2000
n_features = 1000
n_components = 10
n_top_words = 20

def print_top_words(model, feature_names, n_top_words):

```

```

for topic_idx, topic in enumerate(model.components_):
    message = "Topic #%d: " % topic_idx
    message += " ".join([feature_names[i]
                         for i in topic.argsort()[:-n_top_words - 1:-1]])
    print(message)
print()

# Load the 20 newsgroups dataset and vectorize it. We use a few heuristics
# to filter out useless terms early on: the posts are stripped of headers,
# footers and quoted replies, and common English words, words occurring in
# only one document or in at least 95% of the documents are removed.

print("Loading dataset...")
t0 = time()
dataset = fetch_20newsgroups(shuffle=True, random_state=1,
                             remove=('headers', 'footers', 'quotes'))
data_samples = dataset.data[:n_samples]
print("done in %0.3fs." % (time() - t0))

# Use tf-idf features for NMF.
print("Extracting tf-idf features for NMF...")
tfidf_vectorizer = TfidfVectorizer(max_df=0.95, min_df=2,
                                    max_features=n_features,
                                    stop_words='english')
t0 = time()
tfidf = tfidf_vectorizer.fit_transform(data_samples)
print("done in %0.3fs." % (time() - t0))

# Use tf (raw term count) features for LDA.
print("Extracting tf features for LDA...")
tf_vectorizer = CountVectorizer(max_df=0.95, min_df=2,
                                max_features=n_features,
                                stop_words='english')
t0 = time()
tf = tf_vectorizer.fit_transform(data_samples)
print("done in %0.3fs." % (time() - t0))
print()

# Fit the NMF model
print("Fitting the NMF model (Frobenius norm) with tf-idf features, "
      "n_samples=%d and n_features=%d..." %
      (n_samples, n_features))
t0 = time()
nmf = NMF(n_components=n_components, random_state=1,
          alpha=.1, l1_ratio=.5).fit(tfidf)
print("done in %0.3fs." % (time() - t0))

print("\nTopics in NMF model (Frobenius norm):")
tfidf_feature_names = tfidf_vectorizer.get_feature_names()
print_top_words(nmf, tfidf_feature_names, n_top_words)

# Fit the NMF model
print("Fitting the NMF model (generalized Kullback-Leibler divergence) with "
      "tf-idf features, n_samples=%d and n_features=%d..." %
      (n_samples, n_features))
t0 = time()
nmf = NMF(n_components=n_components, random_state=1,

```

```
    beta_loss='kullback-leibler', solver='mu', max_iter=1000, alpha=.1,
    l1_ratio=.5).fit(tfidf)

print("done in %0.3fs." % (time() - t0))

print("\nTopics in NMF model (generalized Kullback-Leibler divergence):")
tfidf_feature_names = tfidf_vectorizer.get_feature_names()
print_top_words(nmf, tfidf_feature_names, n_top_words)

print("Fitting LDA models with tf features, "
      "n_samples=%d and n_features=%d..." %
      (n_samples, n_features))
lda = LatentDirichletAllocation(n_components=n_components, max_iter=5,
                                 learning_method='online',
                                 learning_offset=50.,
                                 random_state=0)
t0 = time()
lda.fit(tf)
print("done in %0.3fs." % (time() - t0))

print("\nTopics in LDA model:")
tf_feature_names = tf_vectorizer.get_feature_names()
print_top_words(lda, tf_feature_names, n_top_words)
```

Total running time of the script: (0 minutes 13.781 seconds)

Note: Click [here](#) to download the full example code

5.2.4 Faces recognition example using eigenfaces and SVMs

The dataset used in this example is a preprocessed excerpt of the “Labeled Faces in the Wild”, aka LFW:

<http://vis-www.cs.umass.edu/lfw/lfw-funneled.tgz> (233MB)

Expected results for the top 5 most represented people in the dataset:

Ariel Sharon	0.67	0.92	0.77	13
Colin Powell	0.75	0.78	0.76	60
Donald Rumsfeld	0.78	0.67	0.72	27
George W Bush	0.86	0.86	0.86	146
Gerhard Schroeder	0.76	0.76	0.76	25
Hugo Chavez	0.67	0.67	0.67	15
Tony Blair	0.81	0.69	0.75	36
avg / total	0.80	0.80	0.80	322

predicted: Bush
true: Bush



predicted: Bush
true: Bush



predicted: Blair
true: Blair



predicted: Bush
true: Bush



predicted: Bush
true: Bush



predicted: Bush
true: Bush



predicted: Schroeder
true: Schroeder



predicted: Powell
true: Powell



predicted: Bush
true: Bush



predicted: Bush
true: Bush



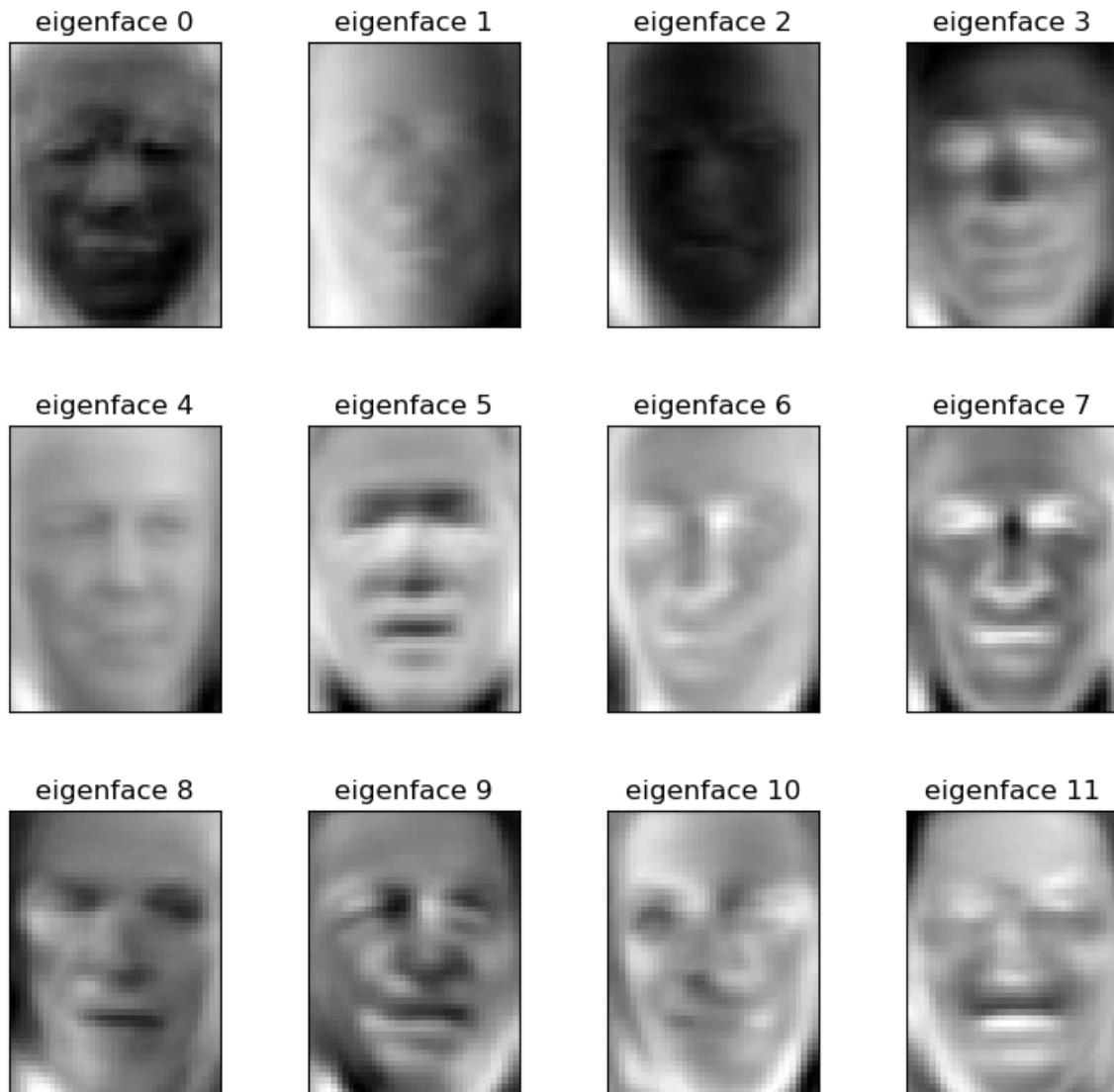
predicted: Bush
true: Bush



predicted: Bush
true: Bush



.



Out:

```
Total dataset size:  
n_samples: 1288  
n_features: 1850  
n_classes: 7  
Extracting the top 150 eigenfaces from 966 faces  
done in 0.118s  
Projecting the input data on the eigenfaces orthonormal basis  
done in 0.005s  
Fitting the classifier to the training set  
done in 36.078s  
Best estimator found by grid search:  
SVC(C=1000.0, class_weight='balanced', gamma=0.005)  
Predicting people's names on the test set  
done in 0.061s  
precision      recall    f1-score   support
```

Ariel Sharon	0.75	0.46	0.57	13
Colin Powell	0.79	0.87	0.83	60
Donald Rumsfeld	0.94	0.63	0.76	27
George W Bush	0.83	0.98	0.90	146
Gerhard Schroeder	0.91	0.80	0.85	25
Hugo Chavez	1.00	0.53	0.70	15
Tony Blair	0.96	0.75	0.84	36
accuracy			0.85	322
macro avg	0.88	0.72	0.78	322
weighted avg	0.86	0.85	0.84	322
[[6 2 0 5 0 0 0]				
[1 52 0 7 0 0 0]				
[1 3 17 6 0 0 0]				
[0 3 0 143 0 0 0]				
[0 1 0 3 20 0 1]				
[0 4 0 2 1 8 0]				
[0 1 1 6 1 0 27]]				

```

from time import time
import logging
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.datasets import fetch_lfw_people
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.decomposition import PCA
from sklearn.svm import SVC

print(__doc__)

# Display progress logs on stdout
logging.basicConfig(level=logging.INFO, format='%(asctime)s %(message)s')

# ##########
# Download the data, if not already on disk and load it as numpy arrays

lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)

# introspect the images arrays to find the shapes (for plotting)
n_samples, h, w = lfw_people.images.shape

# for machine learning we use the 2 data directly (as relative pixel
# positions info is ignored by this model)
X = lfw_people.data
n_features = X.shape[1]

# the label to predict is the id of the person

```

```

y = lfw_people.target
target_names = lfw_people.target_names
n_classes = target_names.shape[0]

print("Total dataset size:")
print("n_samples: %d" % n_samples)
print("n_features: %d" % n_features)
print("n_classes: %d" % n_classes)

# ######
# Split into a training set and a test set using a stratified k fold

# split into a training and testing set
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=42)

# ######
# Compute a PCA (eigenfaces) on the face dataset (treated as unlabeled
# dataset): unsupervised feature extraction / dimensionality reduction
n_components = 150

print("Extracting the top %d eigenfaces from %d faces"
      % (n_components, X_train.shape[0]))
t0 = time()
pca = PCA(n_components=n_components, svd_solver='randomized',
           whiten=True).fit(X_train)
print("done in %0.3fs" % (time() - t0))

eigenfaces = pca.components_.reshape((n_components, h, w))

print("Projecting the input data on the eigenfaces orthonormal basis")
t0 = time()
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)
print("done in %0.3fs" % (time() - t0))

# ######
# Train a SVM classification model

print("Fitting the classifier to the training set")
t0 = time()
param_grid = {'C': [1e3, 5e3, 1e4, 5e4, 1e5],
              'gamma': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1], }
clf = GridSearchCV(SVC(kernel='rbf', class_weight='balanced'),
                    param_grid, cv=5, iid=False)
clf = clf.fit(X_train_pca, y_train)
print("done in %0.3fs" % (time() - t0))
print("Best estimator found by grid search:")
print(clf.best_estimator_)

# ######
# Quantitative evaluation of the model quality on the test set

print("Predicting people's names on the test set")

```

```

t0 = time()
y_pred = clf.predict(X_test_pca)
print("done in %0.3fs" % (time() - t0))

print(classification_report(y_test, y_pred, target_names=target_names))
print(confusion_matrix(y_test, y_pred, labels=range(n_classes)))

# ##### Qualitative evaluation of the predictions using matplotlib

def plot_gallery(images, titles, h, w, n_row=3, n_col=4):
    """Helper function to plot a gallery of portraits"""
    plt.figure(figsize=(1.8 * n_col, 2.4 * n_row))
    plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
    for i in range(n_row * n_col):
        plt.subplot(n_row, n_col, i + 1)
        plt.imshow(images[i].reshape((h, w)), cmap=plt.cm.gray)
        plt.title(titles[i], size=12)
        plt.xticks(())
        plt.yticks(())

# plot the result of the prediction on a portion of the test set

def title(y_pred, y_test, target_names, i):
    pred_name = target_names[y_pred[i]].rsplit(' ', 1)[-1]
    true_name = target_names[y_test[i]].rsplit(' ', 1)[-1]
    return 'predicted: %s\ntrue: %s' % (pred_name, true_name)

prediction_titles = [title(y_pred, y_test, target_names, i)
                     for i in range(y_pred.shape[0])]

plot_gallery(X_test, prediction_titles, h, w)

# plot the gallery of the most significative eigenfaces

eigenface_titles = ["eigenface %d" % i for i in range(eigenfaces.shape[0])]
plot_gallery(eigenfaces, eigenface_titles, h, w)

plt.show()

```

Total running time of the script: (0 minutes 59.514 seconds)

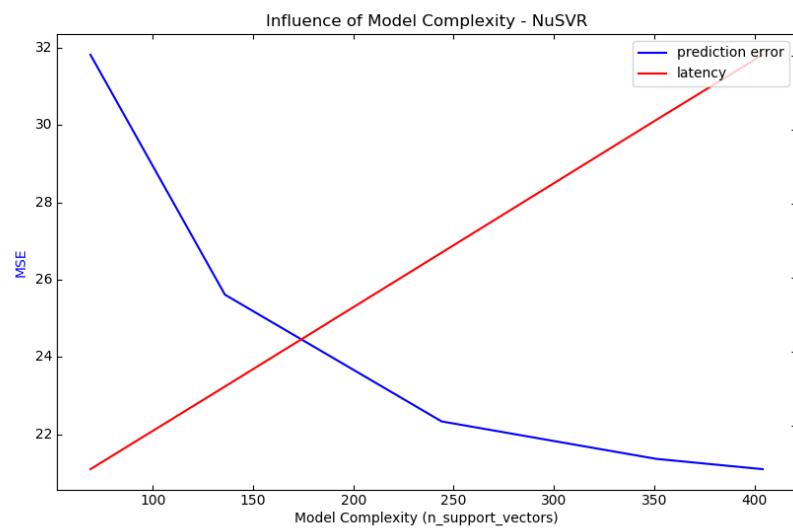
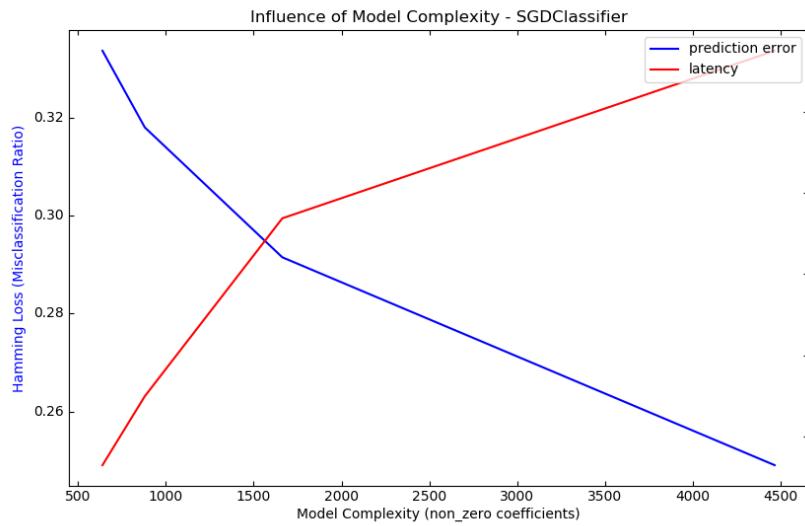
Note: Click [here](#) to download the full example code

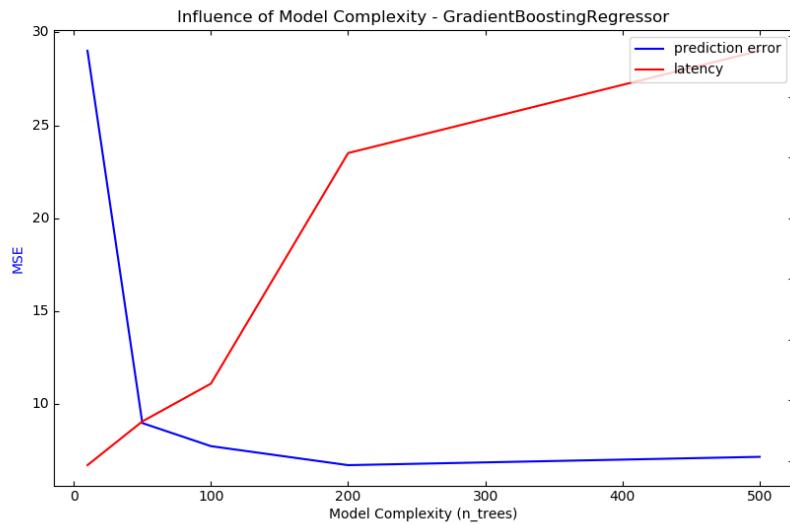
5.2.5 Model Complexity Influence

Demonstrate how model complexity influences both prediction accuracy and computational performance.

The dataset is the Boston Housing dataset (resp. 20 Newsgroups) for regression (resp. classification).

For each class of models we make the model complexity vary through the choice of relevant model parameters and measure the influence on both computational performance (latency) and predictive power (MSE or Hamming Loss).





Out:

```
Benchmarking SGDClassifier(alpha=0.001, l1_ratio=0.25, loss='modified_huber',
                           penalty='elasticnet')
Complexity: 4466 | Hamming Loss (Misclassification Ratio): 0.2491 | Pred. Time: 0.
↪021496s

Benchmarking SGDClassifier(alpha=0.001, l1_ratio=0.5, loss='modified_huber',
                           penalty='elasticnet')
Complexity: 1663 | Hamming Loss (Misclassification Ratio): 0.2915 | Pred. Time: 0.
↪017370s

Benchmarking SGDClassifier(alpha=0.001, l1_ratio=0.75, loss='modified_huber',
                           penalty='elasticnet')
Complexity: 880 | Hamming Loss (Misclassification Ratio): 0.3180 | Pred. Time: 0.
↪012989s

Benchmarking SGDClassifier(alpha=0.001, l1_ratio=0.9, loss='modified_huber',
                           penalty='elasticnet')
Complexity: 639 | Hamming Loss (Misclassification Ratio): 0.3337 | Pred. Time: 0.
↪011292s

Benchmarking NuSVR(C=1000.0, gamma=3.0517578125e-05, nu=0.1)
Complexity: 69 | MSE: 31.8139 | Pred. Time: 0.000283s

Benchmarking NuSVR(C=1000.0, gamma=3.0517578125e-05, nu=0.25)
Complexity: 136 | MSE: 25.6140 | Pred. Time: 0.000506s

Benchmarking NuSVR(C=1000.0, gamma=3.0517578125e-05)
Complexity: 244 | MSE: 22.3375 | Pred. Time: 0.000868s

Benchmarking NuSVR(C=1000.0, gamma=3.0517578125e-05, nu=0.75)
Complexity: 351 | MSE: 21.3688 | Pred. Time: 0.001226s

Benchmarking NuSVR(C=1000.0, gamma=3.0517578125e-05, nu=0.9)
Complexity: 404 | MSE: 21.1033 | Pred. Time: 0.001402s

Benchmarking GradientBoostingRegressor(n_estimators=10)
```

```
Complexity: 10 | MSE: 29.0148 | Pred. Time: 0.000093s

Benchmarking GradientBoostingRegressor(n_estimators=50)
Complexity: 50 | MSE: 8.9630 | Pred. Time: 0.000165s

Benchmarking GradientBoostingRegressor()
Complexity: 100 | MSE: 7.7187 | Pred. Time: 0.000227s

Benchmarking GradientBoostingRegressor(n_estimators=200)
Complexity: 200 | MSE: 6.6955 | Pred. Time: 0.000608s

Benchmarking GradientBoostingRegressor(n_estimators=500)
Complexity: 500 | MSE: 7.1437 | Pred. Time: 0.000776s
```

```
print(__doc__)

# Author: Eustache Diemert <eustache@diemert.fr>
# License: BSD 3 clause

import time
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.parasite_axes import host_subplot
from mpl_toolkits.axisartist.axislines import Axes
from scipy.sparse.csr import csr_matrix

from sklearn import datasets
from sklearn.utils import shuffle
from sklearn.metrics import mean_squared_error
from sklearn.svm.classes import NuSVR
from sklearn.ensemble.gradient_boosting import GradientBoostingRegressor
from sklearn.linear_model.stochastic_gradient import SGDClassifier
from sklearn.metrics import hamming_loss

# ##### Routines #####
# Routines

# Initialize random generator
np.random.seed(0)

def generate_data(case, sparse=False):
    """Generate regression/classification data."""
    bunch = None
    if case == 'regression':
        bunch = datasets.load_boston()
    elif case == 'classification':
        bunch = datasets.fetch_20newsgroups_vectorized(subset='all')
    X, y = shuffle(bunch.data, bunch.target)
    offset = int(X.shape[0] * 0.8)
    X_train, y_train = X[:offset], y[:offset]
    X_test, y_test = X[offset:], y[offset:]
```

```

if sparse:
    X_train = csr_matrix(X_train)
    X_test = csr_matrix(X_test)
else:
    X_train = np.array(X_train)
    X_test = np.array(X_test)
y_test = np.array(y_test)
y_train = np.array(y_train)
data = {'X_train': X_train, 'X_test': X_test, 'y_train': y_train,
        'y_test': y_test}
return data

def benchmark_influence(conf):
    """
    Benchmark influence of :changing_param: on both MSE and latency.
    """
    prediction_times = []
    prediction_powers = []
    complexities = []
    for param_value in conf['changing_param_values']:
        conf['tuned_params'][conf['changing_param']] = param_value
        estimator = conf['estimator'](***conf['tuned_params'])
        print("Benchmarking %s" % estimator)
        estimator.fit(conf['data']['X_train'], conf['data']['y_train'])
        conf['postfit_hook'](estimator)
        complexity = conf['complexity_computer'](estimator)
        complexities.append(complexity)
        start_time = time.time()
        for _ in range(conf['n_samples']):
            y_pred = estimator.predict(conf['data']['X_test'])
            elapsed_time = (time.time() - start_time) / float(conf['n_samples'])
            prediction_times.append(elapsed_time)
            pred_score = conf['prediction_performance_computer'](
                conf['data']['y_test'], y_pred)
            prediction_powers.append(pred_score)
            print("Complexity: %d | %s: %.4f | Pred. Time: %fs\n" %
                  (complexity, conf['prediction_performance_label'], pred_score,
                   elapsed_time))
    return prediction_powers, prediction_times, complexities

def plot_influence(conf, mse_values, prediction_times, complexities):
    """
    Plot influence of model complexity on both accuracy and latency.
    """
    plt.figure(figsize=(12, 6))
    host = host_subplot(111, axes_class=Axes)
    plt.subplots_adjust(right=0.75)
    par1 = host.twinx()
    host.set_xlabel('Model Complexity (%s)' % conf['complexity_label'])
    y1_label = conf['prediction_performance_label']
    y2_label = "Time (s)"
    host.set_ylabel(y1_label)
    par1.set_ylabel(y2_label)
    p1, = host.plot(complexities, mse_values, 'b-', label="prediction error")
    p2, = par1.plot(complexities, prediction_times, 'r-',
                    label="latency")

```

```

host.legend(loc='upper right')
host.axis["left"].label.set_color(p1.get_color())
par1.axis["right"].label.set_color(p2.get_color())
plt.title('Influence of Model Complexity - %s' % conf['estimator'].__name__)
plt.show()

def _count_nonzero_coefficients(estimator):
    a = estimator.coef_.toarray()
    return np.count_nonzero(a)

# ##########
# Main code
regression_data = generate_data('regression')
classification_data = generate_data('classification', sparse=True)
configurations = [
    {'estimator': SGDClassifier,
     'tuned_params': {'penalty': 'elasticnet', 'alpha': 0.001, 'loss':
                     'modified_huber', 'fit_intercept': True, 'tol': 1e-3},
     'changing_param': 'l1_ratio',
     'changing_param_values': [0.25, 0.5, 0.75, 0.9],
     'complexity_label': 'non_zero_coefficients',
     'complexity_computer': _count_nonzero_coefficients,
     'prediction_performance_computer': hamming_loss,
     'prediction_performance_label': 'Hamming Loss (Misclassification Ratio)',
     'postfit_hook': lambda x: x.sparsify(),
     'data': classification_data,
     'n_samples': 30},
    {'estimator': NuSVR,
     'tuned_params': {'C': 1e3, 'gamma': 2 ** -15},
     'changing_param': 'nu',
     'changing_param_values': [0.1, 0.25, 0.5, 0.75, 0.9],
     'complexity_label': 'n_support_vectors',
     'complexity_computer': lambda x: len(x.support_vectors_),
     'data': regression_data,
     'postfit_hook': lambda x: x,
     'prediction_performance_computer': mean_squared_error,
     'prediction_performance_label': 'MSE',
     'n_samples': 30},
    {'estimator': GradientBoostingRegressor,
     'tuned_params': {'loss': 'ls'},
     'changing_param': 'n_estimators',
     'changing_param_values': [10, 50, 100, 200, 500],
     'complexity_label': 'n_trees',
     'complexity_computer': lambda x: x.n_estimators,
     'data': regression_data,
     'postfit_hook': lambda x: x,
     'prediction_performance_computer': mean_squared_error,
     'prediction_performance_label': 'MSE',
     'n_samples': 30},
]
for conf in configurations:
    prediction_performances, prediction_times, complexities = \
        benchmark_influence(conf)
    plot_influence(conf, prediction_performances, prediction_times,
                  complexities)

```

Total running time of the script: (0 minutes 35.625 seconds)

Note: Click [here](#) to download the full example code

5.2.6 Species distribution modeling

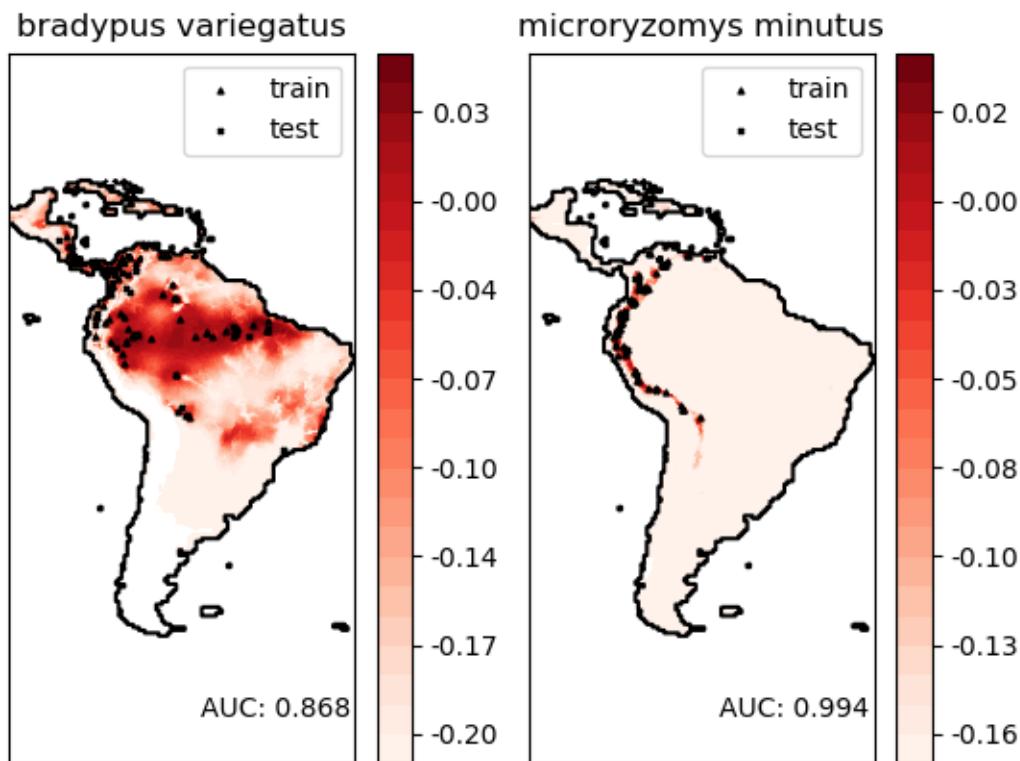
Modeling species' geographic distributions is an important problem in conservation biology. In this example we model the geographic distribution of two south american mammals given past observations and 14 environmental variables. Since we have only positive examples (there are no unsuccessful observations), we cast this problem as a density estimation problem and use the `OneClassSVM` provided by the package `sklearn.svm` as our modeling tool. The dataset is provided by Phillips et. al. (2006). If available, the example uses `basemap` to plot the coast lines and national boundaries of South America.

The two species are:

- “*Bradypus variegatus*”, the Brown-throated Sloth.
- “*Microryzomys minutus*”, also known as the Forest Small Rice Rat, a rodent that lives in Peru, Colombia, Ecuador, Peru, and Venezuela.

References

- “Maximum entropy modeling of species geographic distributions” S. J. Phillips, R. P. Anderson, R. E. Schapire - Ecological Modelling, 190:231-259, 2006.



Out:

```
Modeling distribution of species 'bradypus variegatus'  
- fit OneClassSVM ... done.  
- plot coastlines from coverage  
- predict species distribution  
  
Area under the ROC curve : 0.868443  
  
Modeling distribution of species 'microryzomys minutus'  
- fit OneClassSVM ... done.  
- plot coastlines from coverage  
- predict species distribution  
  
Area under the ROC curve : 0.993919  
  
time elapsed: 20.18s
```

```
# Authors: Peter Prettenhofer <peter.prettenhofer@gmail.com>  
#          Jake Vanderplas <vanderplas@astro.washington.edu>  
#  
# License: BSD 3 clause  
  
from time import time  
  
import numpy as np  
import matplotlib.pyplot as plt  
  
from sklearn.datasets.base import Bunch  
from sklearn.datasets import fetch_species_distributions  
from sklearn.datasets.species_distributions import construct_grids  
from sklearn import svm, metrics  
  
# if basemap is available, we'll use it.  
# otherwise, we'll improvise later...  
try:  
    from mpl_toolkits.basemap import Basemap  
    basemap = True  
except ImportError:  
    basemap = False  
  
print(__doc__)  
  
  
def create_species_bunch(species_name, train, test, coverages, xgrid, ygrid):  
    """Create a bunch with information about a particular organism  
  
    This will use the test/train record arrays to extract the  
    data specific to the given species name.  
    """  
    bunch = Bunch(name=' '.join(species_name.split('_')[:2]))  
    species_name = species_name.encode('ascii')
```

```

points = dict(test=test, train=train)

for label, pts in points.items():
    # choose points associated with the desired species
    pts = pts[pts['species'] == species_name]
    bunch['pts_%s' % label] = pts

    # determine coverage values for each of the training & testing points
    ix = np.searchsorted(xgrid, pts['dd long'])
    iy = np.searchsorted(ygrid, pts['dd lat'])
    bunch['cov_%s' % label] = coverages[:, -iy, ix].T

return bunch


def plot_species_distribution(species=("bradypus_variegatus_0",
                                         "microryzomys_minutus_0")):
    """
    Plot the species distribution.
    """
    if len(species) > 2:
        print("Note: when more than two species are provided,"
              " only the first two will be used")

t0 = time()

# Load the compressed data
data = fetch_species_distributions()

# Set up the data grid
xgrid, ygrid = construct_grids(data)

# The grid in x,y coordinates
X, Y = np.meshgrid(xgrid, ygrid[::-1])

# create a bunch for each species
BV_bunch = create_species_bunch(species[0],
                                 data.train, data.test,
                                 data.coverages, xgrid, ygrid)
MM_bunch = create_species_bunch(species[1],
                                 data.train, data.test,
                                 data.coverages, xgrid, ygrid)

# background points (grid coordinates) for evaluation
np.random.seed(13)
background_points = np.c_[np.random.randint(low=0, high=data.Ny,
                                             size=10000),
                           np.random.randint(low=0, high=data.Nx,
                                             size=10000)].T

# We'll make use of the fact that coverages[6] has measurements at all
# land points. This will help us decide between land and water.
land_reference = data.coverages[6]

# Fit, predict, and plot for each species.
for i, species in enumerate([BV_bunch, MM_bunch]):
    print("_" * 80)
    print("Modeling distribution of species '%s'" % species.name)

```

```

# Standardize features
mean = species.cov_train.mean(axis=0)
std = species.cov_train.std(axis=0)
train_cover_std = (species.cov_train - mean) / std

# Fit OneClassSVM
print(" - fit OneClassSVM ... ", end='')
clf = svm.OneClassSVM(nu=0.1, kernel="rbf", gamma=0.5)
clf.fit(train_cover_std)
print("done.")

# Plot map of South America
plt.subplot(1, 2, i + 1)
if basemap:
    print(" - plot coastlines using basemap")
    m = Basemap(projection='cyl', llcrnrlat=Y.min(),
                 urcrnrlat=Y.max(), llcrnrlon=X.min(),
                 urcrnrlon=X.max(), resolution='c')
    m.drawcoastlines()
    m.drawcountries()
else:
    print(" - plot coastlines from coverage")
    plt.contour(X, Y, land_reference,
                levels=[-9998], colors="k",
                linestyles="solid")
    plt.xticks([])
    plt.yticks([])

print(" - predict species distribution")

# Predict species distribution using the training data
Z = np.ones((data.Ny, data.Nx), dtype=np.float64)

# We'll predict only for the land points.
idx = np.where(land_reference > -9999)
coverages_land = data.coverages[:, idx[0], idx[1]].T

pred = clf.decision_function((coverages_land - mean) / std)
Z *= pred.min()
Z[idx[0], idx[1]] = pred

levels = np.linspace(Z.min(), Z.max(), 25)
Z[land_reference == -9999] = -9999

# plot contours of the prediction
plt.contourf(X, Y, Z, levels=levels, cmap=plt.cm.Reds)
plt.colorbar(format='%.2f')

# scatter training/testing points
plt.scatter(species.pts_train['dd long'], species.pts_train['dd lat'],
            s=2 ** 2, c='black',
            marker='^', label='train')
plt.scatter(species.pts_test['dd long'], species.pts_test['dd lat'],
            s=2 ** 2, c='black',
            marker='x', label='test')
plt.legend()
plt.title(species.name)

```

```

plt.axis('equal')

# Compute AUC with regards to background points
pred_background = Z[background_points[0], background_points[1]]
pred_test = clf.decision_function((species.cov_test - mean) / std)
scores = np.r_[pred_test, pred_background]
y = np.r_[np.ones(pred_test.shape), np.zeros(pred_background.shape)]
fpr, tpr, thresholds = metrics.roc_curve(y, scores)
roc_auc = metrics.auc(fpr, tpr)
plt.text(-35, -70, "AUC: %.3f" % roc_auc, ha="right")
print("\n Area under the ROC curve : %f" % roc_auc)

print("\ntime elapsed: %.2fs" % (time() - t0))

plot_species_distribution()
plt.show()

```

Total running time of the script: (0 minutes 20.185 seconds)

Note: Click [here](#) to download the full example code

5.2.7 Visualizing the stock market structure

This example employs several unsupervised learning techniques to extract the stock market structure from variations in historical quotes.

The quantity that we use is the daily variation in quote price: quotes that are linked tend to cofluctuate during a day.

Learning a graph structure

We use sparse inverse covariance estimation to find which quotes are correlated conditionally on the others. Specifically, sparse inverse covariance gives us a graph, that is a list of connection. For each symbol, the symbols that it is connected too are those useful to explain its fluctuations.

Clustering

We use clustering to group together quotes that behave similarly. Here, amongst the *various clustering techniques* available in the scikit-learn, we use *Affinity Propagation* as it does not enforce equal-size clusters, and it can choose automatically the number of clusters from the data.

Note that this gives us a different indication than the graph, as the graph reflects conditional relations between variables, while the clustering reflects marginal properties: variables clustered together can be considered as having a similar impact at the level of the full stock market.

Embedding in 2D space

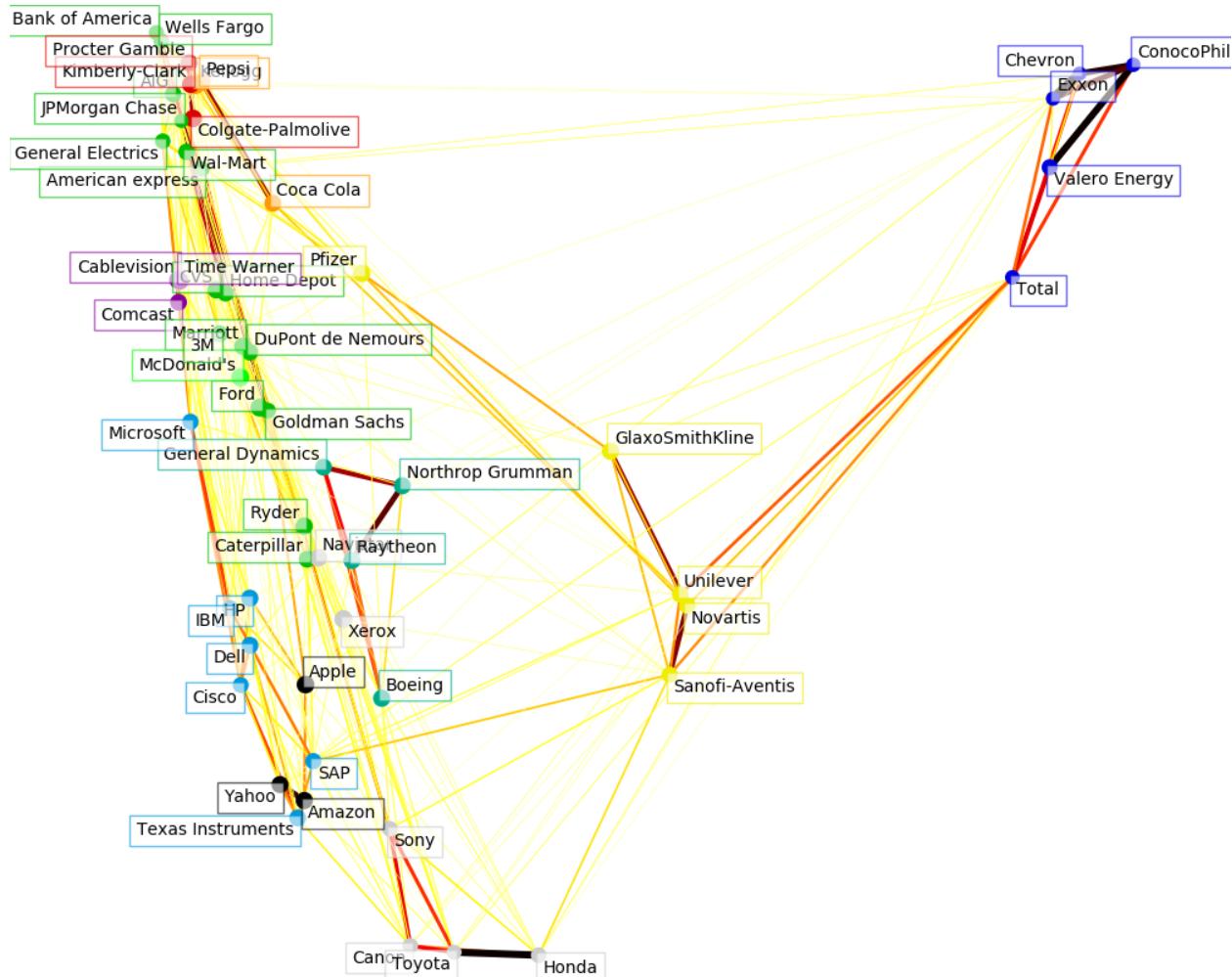
For visualization purposes, we need to lay out the different symbols on a 2D canvas. For this we use *Manifold learning* techniques to retrieve 2D embedding.

Visualization

The output of the 3 models are combined in a 2D graph where nodes represents the stocks and edges the:

- cluster labels are used to define the color of the nodes
- the sparse covariance model is used to display the strength of the edges
- the 2D embedding is used to position the nodes in the plan

This example has a fair amount of visualization-related code, as visualization is crucial here to display the graph. One of the challenge is to position the labels minimizing overlap. For this we use an heuristic based on the direction of the nearest neighbor along each axis.



Out:

```

Cluster 1: Apple, Amazon, Yahoo
Cluster 2: Comcast, Cablevision, Time Warner
Cluster 3: ConocoPhillips, Chevron, Total, Valero Energy, Exxon
Cluster 4: Cisco, Dell, HP, IBM, Microsoft, SAP, Texas Instruments
Cluster 5: Boeing, General Dynamics, Northrop Grumman, Raytheon
Cluster 6: AIG, American express, Bank of America, Caterpillar, CVS, DuPont de Nemours, Ford, General Electric, Goldman Sachs, Home Depot, JPMorgan Chase, Marriott, 3M, Ryder, Wells Fargo, Wal-Mart
Cluster 7: McDonald's

```

```
Cluster 8: GlaxoSmithKline, Novartis, Pfizer, Sanofi-Aventis, Unilever
Cluster 9: Kellogg, Coca Cola, Pepsi
Cluster 10: Colgate-Palmolive, Kimberly-Clark, Procter Gamble
Cluster 11: Canon, Honda, Navistar, Sony, Toyota, Xerox
```

```
# Author: Gael Varoquaux gael.varoquaux@normalesup.org
# License: BSD 3 clause

import sys

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection

import pandas as pd

from sklearn import cluster, covariance, manifold

print(__doc__)

# ##########
# Retrieve the data from Internet

# The data is from 2003 - 2008. This is reasonably calm: (not too long ago so
# that we get high-tech firms, and before the 2008 crash). This kind of
# historical data can be obtained for from APIs like the quandl.com and
# alphavantage.co ones.

symbol_dict = {
    'TOT': 'Total',
    'XOM': 'Exxon',
    'CVX': 'Chevron',
    'COP': 'ConocoPhillips',
    'VLO': 'Valero Energy',
    'MSFT': 'Microsoft',
    'IBM': 'IBM',
    'TWX': 'Time Warner',
    'CMCSA': 'Comcast',
    'CVC': 'Cablevision',
    'YHOO': 'Yahoo',
    'DELL': 'Dell',
    'HPQ': 'HP',
    'AMZN': 'Amazon',
    'TM': 'Toyota',
    'CAJ': 'Canon',
    'SNE': 'Sony',
    'F': 'Ford',
    'HMC': 'Honda',
    'NAV': 'Navistar',
    'NOC': 'Northrop Grumman',
    'BA': 'Boeing',
    'KO': 'Coca Cola',
```

```

'MMM': '3M',
'MCD': 'McDonald\\'s',
'PEP': 'Pepsi',
'K': 'Kellogg',
'UN': 'Unilever',
'MAR': 'Marriott',
'PG': 'Procter Gamble',
'CL': 'Colgate-Palmolive',
'GE': 'General Electrics',
'WFC': 'Wells Fargo',
'JPM': 'JPMorgan Chase',
'AIG': 'AIG',
'AXP': 'American express',
'BAC': 'Bank of America',
'GS': 'Goldman Sachs',
'AAPL': 'Apple',
'SAP': 'SAP',
'CSCO': 'Cisco',
'TXN': 'Texas Instruments',
'XRX': 'Xerox',
'WMT': 'Wal-Mart',
'HD': 'Home Depot',
'GSK': 'GlaxoSmithKline',
'PFE': 'Pfizer',
'SNY': 'Sanofi-Aventis',
'NVS': 'Novartis',
'KMB': 'Kimberly-Clark',
'R': 'Ryder',
'GD': 'General Dynamics',
'RTN': 'Raytheon',
'CVS': 'CVS',
'CAT': 'Caterpillar',
'DD': 'DuPont de Nemours'}

```

symbols, names = np.array(sorted(symbol_dict.items())) .T

quotes = []

```

for symbol in symbols:
    print('Fetching quote history for %r' % symbol, file=sys.stderr)
    url = ('https://raw.githubusercontent.com/scikit-learn/examples-data/'
           'master/financial-data/{}.csv')
    quotes.append(pd.read_csv(url.format(symbol)))

```

close_prices = np.vstack([q['close'] for q in quotes])
open_prices = np.vstack([q['open'] for q in quotes])

```

# The daily variations of the quotes are what carry most information
variation = close_prices - open_prices

```

```

#####
# Learn a graphical structure from the correlations
edge_model = covariance.GraphicalLassoCV(cv=5)

# standardize the time series: using correlations rather than covariance
# is more efficient for structure recovery

```

```

X = variation.copy().T
X /= X.std(axis=0)
edge_model.fit(X)

# ######
# Cluster using affinity propagation

_, labels = cluster.affinity_propagation(edge_model.covariance_)
n_labels = labels.max()

for i in range(n_labels + 1):
    print('Cluster %i: %s' % ((i + 1), ', '.join(names[labels == i])))

# #####
# Find a low-dimension embedding for visualization: find the best position of
# the nodes (the stocks) on a 2D plane

# We use a dense eigen_solver to achieve reproducibility (arpack is
# initiated with random vectors that we don't control). In addition, we
# use a large number of neighbors to capture the large-scale structure.
node_position_model = manifold.LocallyLinearEmbedding(
    n_components=2, eigen_solver='dense', n_neighbors=6)

embedding = node_position_model.fit_transform(X.T).T

# #####
# Visualization
plt.figure(1, facecolor='w', figsize=(10, 8))
plt.clf()
ax = plt.axes([0., 0., 1., 1.])
plt.axis('off')

# Display a graph of the partial correlations
partial_correlations = edge_model.precision_.copy()
d = 1 / np.sqrt(np.diag(partial_correlations))
partial_correlations *= d
partial_correlations *= d[:, np.newaxis]
non_zero = (np.abs(np.triu(partial_correlations, k=1)) > 0.02)

# Plot the nodes using the coordinates of our embedding
plt.scatter(embedding[0], embedding[1], s=100 * d ** 2, c=labels,
            cmap=plt.cm.nipy_spectral)

# Plot the edges
start_idx, end_idx = np.where(non_zero)
# a sequence of (*line0*, *line1*, *line2*), where::
#      linen = (x0, y0), (x1, y1), ... (xm, ym)
segments = [[embedding[:, start], embedding[:, stop]]
            for start, stop in zip(start_idx, end_idx)]
values = np.abs(partial_correlations[non_zero])
lc = LineCollection(segments,
                     zorder=0, cmap=plt.cm.hot_r,
                     norm=plt.Normalize(0, .7 * values.max()))
lc.set_array(values)
lc.set_linewidths(15 * values)
ax.add_collection(lc)

# Add a label to each node. The challenge here is that we want to

```

```
# position the labels to avoid overlap with other labels
for index, (name, label, (x, y)) in enumerate(
    zip(names, labels, embedding.T)):

    dx = x - embedding[0]
    dx[index] = 1
    dy = y - embedding[1]
    dy[index] = 1
    this_dx = dx[np.argmin(np.abs(dy))]
    this_dy = dy[np.argmin(np.abs(dx))]
    if this_dx > 0:
        horizontalalignment = 'left'
        x = x + .002
    else:
        horizontalalignment = 'right'
        x = x - .002
    if this_dy > 0:
        verticalalignment = 'bottom'
        y = y + .002
    else:
        verticalalignment = 'top'
        y = y - .002
    plt.text(x, y, name, size=10,
              horizontalalignment=horizontalalignment,
              verticalalignment=verticalalignment,
              bbox=dict(facecolor='w',
                        edgecolor=plt.cm.nipy_spectral(label / float(n_labels)),
                        alpha=.6))

plt.xlim(embedding[0].min() - .15 * embedding[0].ptp(),
          embedding[0].max() + .10 * embedding[0].ptp(),)
plt.ylim(embedding[1].min() - .03 * embedding[1].ptp(),
          embedding[1].max() + .03 * embedding[1].ptp(),)

plt.show()
```

Total running time of the script: (0 minutes 4.857 seconds)

Note: Click [here](#) to download the full example code

5.2.8 Wikipedia principal eigenvector

A classical way to assert the relative importance of vertices in a graph is to compute the principal eigenvector of the adjacency matrix so as to assign to each vertex the values of the components of the first eigenvector as a centrality score:

https://en.wikipedia.org/wiki/Eigenvector_centrality

On the graph of webpages and links those values are called the PageRank scores by Google.

The goal of this example is to analyze the graph of links inside wikipedia articles to rank articles by relative importance according to this eigenvector centrality.

The traditional way to compute the principal eigenvector is to use the power iteration method:

https://en.wikipedia.org/wiki/Power_iteration

Here the computation is achieved thanks to Martinsson's Randomized SVD algorithm implemented in scikit-learn.

The graph data is fetched from the DBpedia dumps. DBpedia is an extraction of the latent structured data of the Wikipedia content.

```
# Author: Olivier Grisel <olivier.grisel@ensta.org>
# License: BSD 3 clause

from bz2 import BZ2File
import os
from datetime import datetime
from pprint import pprint
from time import time

import numpy as np

from scipy import sparse

from joblib import Memory

from sklearn.decomposition import randomized_svd
from urllib.request import urlopen

print(__doc__)

# ######
# Where to download the data, if not already on disk
redirects_url = "http://downloads.dbpedia.org/3.5.1/en/redirects_en.nt.bz2"
redirects_filename = redirects_url.rsplit("/", 1)[1]

page_links_url = "http://downloads.dbpedia.org/3.5.1/en/page_links_en.nt.bz2"
page_links_filename = page_links_url.rsplit("/", 1)[1]

resources = [
    (redirects_url, redirects_filename),
    (page_links_url, page_links_filename),
]

for url, filename in resources:
    if not os.path.exists(filename):
        print("Downloading data from '%s', please wait..." % url)
        opener = urlopen(url)
        open(filename, 'wb').write(opener.read())
        print()

# ######
# Loading the redirect files

memory = Memory(cachedir=".")

def index(redirects, index_map, k):
    """Find the index of an article name after redirect resolution"""
    k = redirects.get(k, k)
    return index_map.setdefault(k, len(index_map))
```

```

DBPEDIA_RESOURCE_PREFIX_LEN = len("http://dbpedia.org/resource/")
SHORTNAME_SLICE = slice(DBPEDIA_RESOURCE_PREFIX_LEN + 1, -1)

def short_name(nt_uri):
    """Remove the < and > URI markers and the common URI prefix"""
    return nt_uri[SHORTNAME_SLICE]

def get_redirects(redirects_filename):
    """Parse the redirections and build a transitively closed map out of it"""
    redirects = {}
    print("Parsing the NT redirect file")
    for l, line in enumerate(BZ2File(redirects_filename)):
        split = line.split()
        if len(split) != 4:
            print("ignoring malformed line: " + line)
            continue
        redirects[short_name(split[0])] = short_name(split[2])
        if l % 1000000 == 0:
            print("[{}] line: {}08d" % (datetime.now().isoformat(), l))

    # compute the transitive closure
    print("Computing the transitive closure of the redirect relation")
    for l, source in enumerate(redirects.keys()):
        transitive_target = None
        target = redirects[source]
        seen = {source}
        while True:
            transitive_target = target
            target = redirects.get(target)
            if target is None or target in seen:
                break
            seen.add(target)
        redirects[source] = transitive_target
        if l % 1000000 == 0:
            print("[{}] line: {}08d" % (datetime.now().isoformat(), l))

    return redirects

# disabling joblib as the pickling of large dicts seems much too slow
#@memory.cache
def get_adjacency_matrix(redirects_filename, page_links_filename, limit=None):
    """Extract the adjacency graph as a scipy sparse matrix

    Redirects are resolved first.

    Returns X, the scipy sparse adjacency matrix, redirects as python dict
    from article names to article names and index_map a python dict
    from article names to python int (article indexes).
    """

    print("Computing the redirect map")
    redirects = get_redirects(redirects_filename)

    print("Computing the integer index map")
    index_map = dict()

```

```

links = list()
for l, line in enumerate(BZ2File(page_links_filename)):
    split = line.split()
    if len(split) != 4:
        print("ignoring malformed line: " + line)
        continue
    i = index(redirections, index_map, short_name(split[0]))
    j = index(redirections, index_map, short_name(split[2]))
    links.append((i, j))
    if l % 1000000 == 0:
        print("[%s] line: %08d" % (datetime.now().isoformat(), l))

    if limit is not None and l >= limit - 1:
        break

print("Computing the adjacency matrix")
X = sparse.lil_matrix((len(index_map), len(index_map)), dtype=np.float32)
for i, j in links:
    X[i, j] = 1.0
del links
print("Converting to CSR representation")
X = X.tocsr()
print("CSR conversion done")
return X, redirections, index_map

# stop after 5M links to make it possible to work in RAM
X, redirections, index_map = get_adjacency_matrix(
    redirections_filename, page_links_filename, limit=5000000)
names = {i: name for name, i in index_map.items()}

print("Computing the principal singular vectors using randomized_svd")
t0 = time()
U, s, V = randomized_svd(X, 5, n_iter=3)
print("done in %0.3fs" % (time() - t0))

# print the names of the wikipedia related strongest components of the
# principal singular vector which should be similar to the highest eigenvector
print("Top wikipedia pages according to principal singular vectors")
pprint([names[i] for i in np.abs(U.T[0]).argsort()[-10:]])
pprint([names[i] for i in np.abs(V[0]).argsort()[-10:]])


def centrality_scores(X, alpha=0.85, max_iter=100, tol=1e-10):
    """Power iteration computation of the principal eigenvector

    This method is also known as Google PageRank and the implementation
    is based on the one from the NetworkX project (BSD licensed too)
    with copyrights by:

    Aric Hagberg <hagberg@lanl.gov>
    Dan Schult <dschult@colgate.edu>
    Pieter Swart <swart@lanl.gov>
    """
    n = X.shape[0]
    X = X.copy()
    incoming_counts = np.asarray(X.sum(axis=1)).ravel()

```

```
print("Normalizing the graph")
for i in incoming_counts.nonzero()[0]:
    X.data[X.indptr[i]:X.indptr[i + 1]] *= 1.0 / incoming_counts[i]
dangle = np.asarray(np.where(np.isclose(X.sum(axis=1), 0),
                            1.0 / n, 0)).ravel()

scores = np.full(n, 1. / n, dtype=np.float32) # initial guess
for i in range(max_iter):
    print("power iteration # %d" % i)
    prev_scores = scores
    scores = (alpha * (scores * X + np.dot(dangle, prev_scores)))
        + (1 - alpha) * prev_scores.sum() / n
    # check convergence: normalized l_inf norm
    scores_max = np.abs(scores).max()
    if scores_max == 0.0:
        scores_max = 1.0
    err = np.abs(scores - prev_scores).max() / scores_max
    print("error: %0.6f" % err)
    if err < n * tol:
        return scores

return scores

print("Computing principal eigenvector score using a power iteration method")
t0 = time()
scores = centrality_scores(X, max_iter=100, tol=1e-10)
print("done in %0.3fs" % (time() - t0))
pprint([names[i] for i in np.abs(scores).argsort()[-10:]])
```

Total running time of the script: (0 minutes 0.000 seconds)

Note: Click [here](#) to download the full example code

5.2.9 Libsvm GUI

A simple graphical frontend for Libsvm mainly intended for didactic purposes. You can create data points by point and click and visualize the decision region induced by different kernels and parameter settings.

To create positive examples click the left mouse button; to create negative examples click the right button.

If all examples are from the same class, it uses a one-class SVM.

```
print(__doc__)

# Author: Peter Prettenhoer <peter.prettenhofer@gmail.com>
#
# License: BSD 3 clause

import matplotlib
matplotlib.use('TkAgg')

from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.backends.backend_tkagg import NavigationToolbar2TkAgg
from matplotlib.figure import Figure
from matplotlib.contour import ContourSet
```

```

import sys
import numpy as np
import tkinter as Tk

from sklearn import svm
from sklearn.datasets import dump_svmlight_file

y_min, y_max = -50, 50
x_min, x_max = -50, 50

class Model:
    """The Model which hold the data. It implements the
    observable in the observer pattern and notifies the
    registered observers on change event.
    """

    def __init__(self):
        self.observers = []
        self.surface = None
        self.data = []
        self.cls = None
        self.surface_type = 0

    def changed(self, event):
        """Notify the observers."""
        for observer in self.observers:
            observer.update(event, self)

    def add_observer(self, observer):
        """Register an observer."""
        self.observers.append(observer)

    def set_surface(self, surface):
        self.surface = surface

    def dump_svmlight_file(self, file):
        data = np.array(self.data)
        X = data[:, 0:2]
        y = data[:, 2]
        dump_svmlight_file(X, y, file)

class Controller:
    def __init__(self, model):
        self.model = model
        self.kernel = Tk.IntVar()
        self.surface_type = Tk.IntVar()
        # Whether or not a model has been fitted
        self.fitted = False

    def fit(self):
        print("fit the model")
        train = np.array(self.model.data)
        X = train[:, 0:2]
        y = train[:, 2]

```

```

C = float(self.complexity.get())
gamma = float(self.gamma.get())
coef0 = float(self.coef0.get())
degree = int(self.degree.get())
kernel_map = {0: "linear", 1: "rbf", 2: "poly"}
if len(np.unique(y)) == 1:
    clf = svm.OneClassSVM(kernel=kernel_map[self.kernel.get()],
                           gamma=gamma, coef0=coef0, degree=degree)
    clf.fit(X)
else:
    clf = svm.SVC(kernel=kernel_map[self.kernel.get()], C=C,
                   gamma=gamma, coef0=coef0, degree=degree)
    clf.fit(X, y)
if hasattr(clf, 'score'):
    print("Accuracy:", clf.score(X, y) * 100)
X1, X2, Z = self.decision_surface(clf)
self.model.clf = clf
self.model.set_surface((X1, X2, Z))
self.model.surface_type = self.surface_type.get()
self.fitted = True
self.model.changed("surface")

def decision_surface(self, cls):
    delta = 1
    x = np.arange(x_min, x_max + delta, delta)
    y = np.arange(y_min, y_max + delta, delta)
    X1, X2 = np.meshgrid(x, y)
    Z = cls.decision_function(np.c_[X1.ravel(), X2.ravel()])
    Z = Z.reshape(X1.shape)
    return X1, X2, Z

def clear_data(self):
    self.model.data = []
    self.fitted = False
    self.model.changed("clear")

def add_example(self, x, y, label):
    self.model.data.append((x, y, label))
    self.model.changed("example_added")

    # update decision surface if already fitted.
    self.refit()

def refit(self):
    """Refit the model if already fitted. """
    if self.fitted:
        self.fit()

class View:
    """Test docstring. """
    def __init__(self, root, controller):
        f = Figure()
        ax = f.add_subplot(111)
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_xlim((x_min, x_max))
        ax.set_ylim((y_min, y_max))

```

```

canvas = FigureCanvasTkAgg(f, master=root)
canvas.show()
canvas.get_tk_widget().pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)
canvas._tkcanvas.pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)
canvas.mpl_connect('button_press_event', self.onclick)
toolbar = NavigationToolbar2TkAgg(canvas, root)
toolbar.update()
self.controlbar = ControllBar(root, controller)
self.f = f
self.ax = ax
self.canvas = canvas
self.controller = controller
self.contours = []
self.c_labels = None
self.plot_kernels()

def plot_kernels(self):
    self.ax.text(-50, -60, "Linear: $u^T v$")
    self.ax.text(-20, -60, r"RBF: $\exp(-\gamma | u-v |^2)$")
    self.ax.text(10, -60, r"Poly: $(\gamma, u^T v + r)^d$")

def onclick(self, event):
    if event.xdata and event.ydata:
        if event.button == 1:
            self.controller.add_example(event.xdata, event.ydata, 1)
        elif event.button == 3:
            self.controller.add_example(event.xdata, event.ydata, -1)

def update_example(self, model, idx):
    x, y, l = model.data[idx]
    if l == 1:
        color = 'w'
    elif l == -1:
        color = 'k'
    self.ax.plot([x], [y], "%s" % color, scalex=0.0, scaley=0.0)

def update(self, event, model):
    if event == "examples_loaded":
        for i in range(len(model.data)):
            self.update_example(model, i)

    if event == "example_added":
        self.update_example(model, -1)

    if event == "clear":
        self.ax.clear()
        self.ax.set_xticks([])
        self.ax.set_yticks([])
        self.contours = []
        self.c_labels = None
        self.plot_kernels()

    if event == "surface":
        self.remove_surface()
        self.plot_support_vectors(model.clf.support_vectors_)
        self.plot_decision_surface(model.surface, model.surface_type)

    self.canvas.draw()

```

```

def remove_surface(self):
    """Remove old decision surface."""
    if len(self.contours) > 0:
        for contour in self.contours:
            if isinstance(contour, ContourSet):
                for lineset in contour.collections:
                    lineset.remove()
            else:
                contour.remove()
        self.contours = []

def plot_support_vectors(self, support_vectors):
    """Plot the support vectors by placing circles over the
    corresponding data points and adds the circle collection
    to the contours list."""
    cs = self.ax.scatter(support_vectors[:, 0], support_vectors[:, 1],
                         s=80, edgecolors="k", facecolors="none")
    self.contours.append(cs)

def plot_decision_surface(self, surface, type):
    X1, X2, Z = surface
    if type == 0:
        levels = [-1.0, 0.0, 1.0]
        linestyles = ['dashed', 'solid', 'dashed']
        colors = 'k'
        self.contours.append(self.ax.contour(X1, X2, Z, levels,
                                              colors=colors,
                                              linestyles=linestyles))
    elif type == 1:
        self.contours.append(self.ax.contourf(X1, X2, Z, 10,
                                              cmap=matplotlib.cm.bone,
                                              origin='lower', alpha=0.85))
        self.contours.append(self.ax.contour(X1, X2, Z, [0.0], colors='k',
                                             linestyles=['solid']))
    else:
        raise ValueError("surface type unknown")

class ContrackBar:
    def __init__(self, root, controller):
        fm = Tk.Frame(root)
        kernel_group = Tk.Frame(fm)
        Tk.Radiobutton(kernel_group, text="Linear", variable=controller.kernel,
                       value=0, command=controller.refit).pack(anchor=Tk.W)
        Tk.Radiobutton(kernel_group, text="RBF", variable=controller.kernel,
                       value=1, command=controller.refit).pack(anchor=Tk.W)
        Tk.Radiobutton(kernel_group, text="Poly", variable=controller.kernel,
                       value=2, command=controller.refit).pack(anchor=Tk.W)
        kernel_group.pack(side=Tk.LEFT)

        valbox = Tk.Frame(fm)
        controller.complexity = Tk.StringVar()
        controller.complexity.set("1.0")
        c = Tk.Frame(valbox)
        Tk.Label(c, text="C:", anchor="e", width=7).pack(side=Tk.LEFT)
        Tk.Entry(c, width=6, textvariable=controller.complexity).pack(
            side=Tk.LEFT)

```

```

c.pack()

controller.gamma = Tk.StringVar()
controller.gamma.set("0.01")
g = Tk.Frame(valbox)
Tk.Label(g, text="gamma:", anchor="e", width=7).pack(side=Tk.LEFT)
Tk.Entry(g, width=6, textvariable=controller.gamma).pack(side=Tk.LEFT)
g.pack()

controller.degree = Tk.StringVar()
controller.degree.set("3")
d = Tk.Frame(valbox)
Tk.Label(d, text="degree:", anchor="e", width=7).pack(side=Tk.LEFT)
Tk.Entry(d, width=6, textvariable=controller.degree).pack(side=Tk.LEFT)
d.pack()

controller.coef0 = Tk.StringVar()
controller.coef0.set("0")
r = Tk.Frame(valbox)
Tk.Label(r, text="coef0:", anchor="e", width=7).pack(side=Tk.LEFT)
Tk.Entry(r, width=6, textvariable=controller.coef0).pack(side=Tk.LEFT)
r.pack()
valbox.pack(side=Tk.LEFT)

cmap_group = Tk.Frame(fm)
Tk.Radiobutton(cmap_group, text="Hyperplanes",
                variable=controller.surface_type, value=0,
                command=controller.refit).pack(anchor=Tk.W)
Tk.Radiobutton(cmap_group, text="Surface",
                variable=controller.surface_type, value=1,
                command=controller.refit).pack(anchor=Tk.W)

cmap_group.pack(side=Tk.LEFT)

train_button = Tk.Button(fm, text='Fit', width=5,
                        command=controller.fit)
train_button.pack()
fm.pack(side=Tk.LEFT)
Tk.Button(fm, text='Clear', width=5,
          command=controller.clear_data).pack(side=Tk.LEFT)

def get_parser():
    from optparse import OptionParser
    op = OptionParser()
    op.add_option("--output",
                  action="store", type="str", dest="output",
                  help="Path where to dump data.")
    return op

def main(argv):
    op = get_parser()
    opts, args = op.parse_args(argv[1:])
    root = Tk.Tk()
    model = Model()
    controller = Controller(model)
    root.wm_title("Scikit-learn Libsvm GUI")

```

```

view = View(root, controller)
model.add_observer(view)
Tk.mainloop()

if opts.output:
    model.dump_svmlight_file(opts.output)

if __name__ == "__main__":
    main(sys.argv)

```

Total running time of the script: (0 minutes 0.000 seconds)

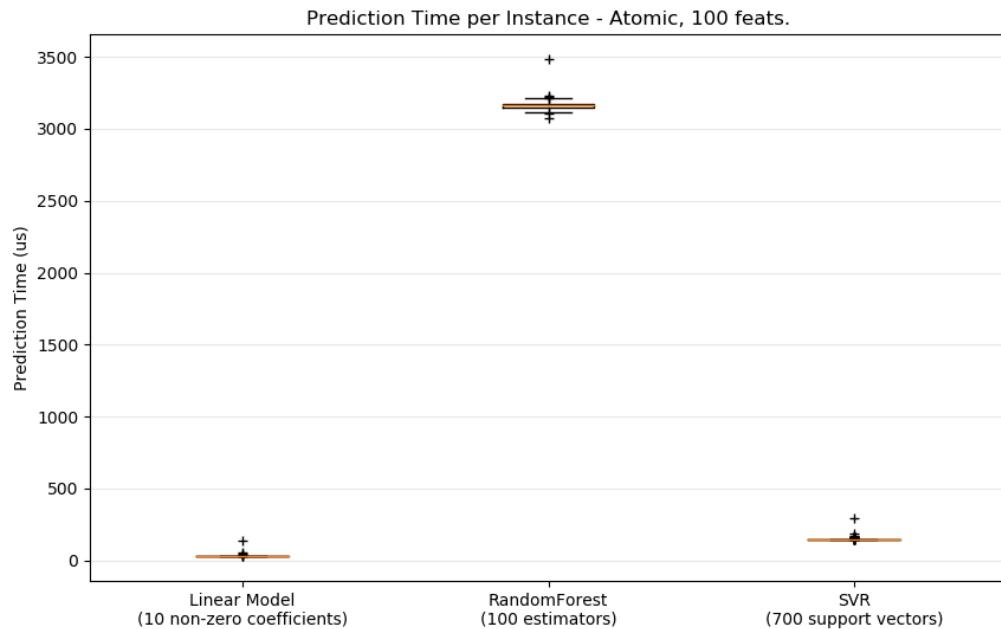
Note: Click [here](#) to download the full example code

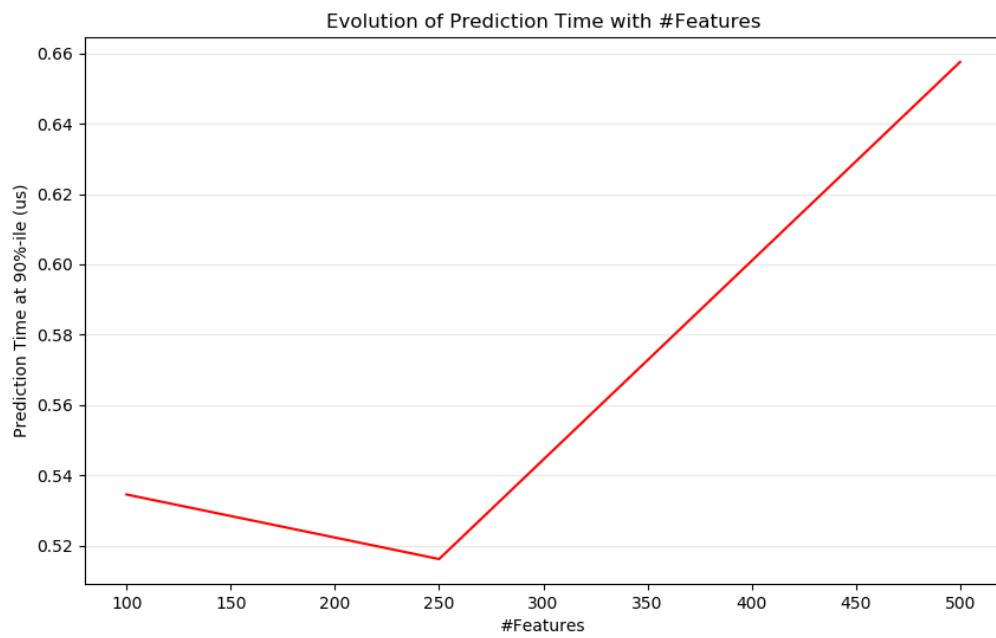
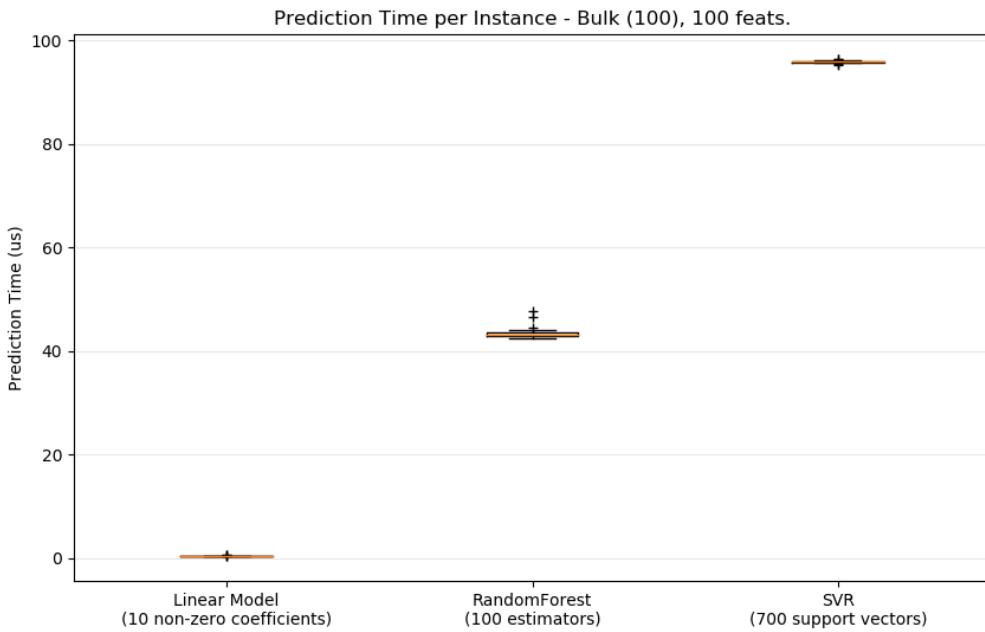
5.2.10 Prediction Latency

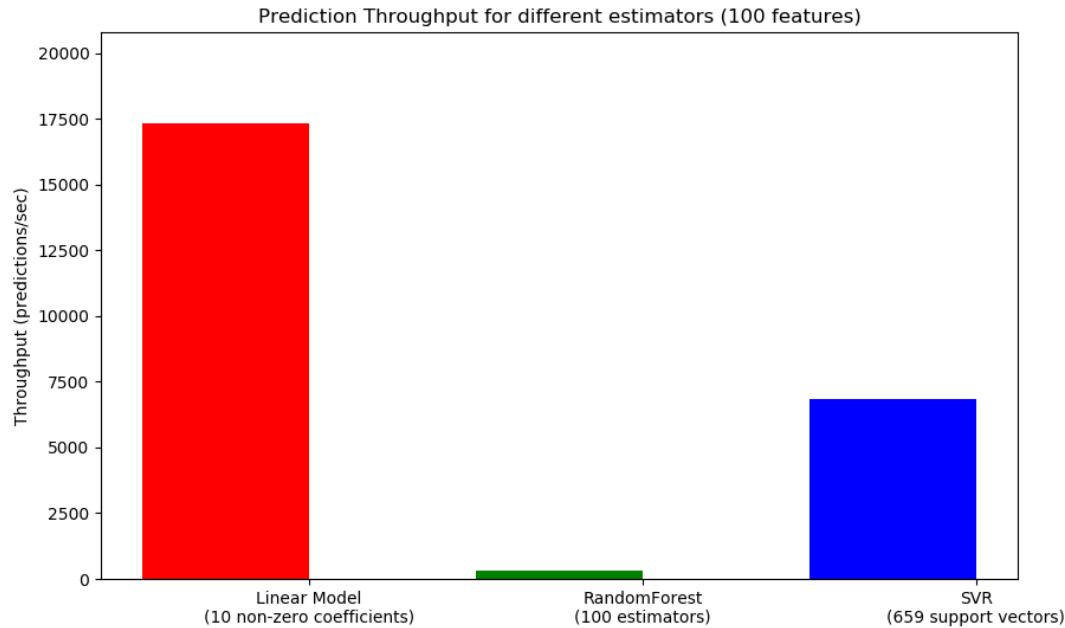
This is an example showing the prediction latency of various scikit-learn estimators.

The goal is to measure the latency one can expect when doing predictions either in bulk or atomic (i.e. one by one) mode.

The plots represent the distribution of the prediction latency as a boxplot.







Out:

```
Benchmarking SGDRegressor(alpha=0.01, l1_ratio=0.25, penalty='elasticnet', tol=0.0001)
Benchmarking RandomForestRegressor(n_estimators=100)
Benchmarking SVR()
benchmarking with 100 features
benchmarking with 250 features
benchmarking with 500 features
example run in 9.60s
```

```
# Authors: Eustache Diemert <eustache@diemert.fr>
# License: BSD 3 clause

from collections import defaultdict

import time
import gc
import numpy as np
import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.datasets.samples_generator import make_regression
from sklearn.ensemble.forest import RandomForestRegressor
from sklearn.linear_model.ridge import Ridge
from sklearn.linear_model.stochastic_gradient import SGDRegressor
from sklearn.svm.classes import SVR
from sklearn.utils import shuffle
```

```

def _not_in_sphinx():
    # Hack to detect whether we are running by the sphinx builder
    return '__file__' in globals()

def atomic_benchmark_estimator(estimator, X_test, verbose=False):
    """Measure runtime prediction of each instance."""
    n_instances = X_test.shape[0]
    runtimes = np.zeros(n_instances, dtype=np.float)
    for i in range(n_instances):
        instance = X_test[[i], :]
        start = time.time()
        estimator.predict(instance)
        runtimes[i] = time.time() - start
    if verbose:
        print("atomic_benchmark runtimes:", min(runtimes), np.percentile(
            runtimes, 50), max(runtimes))
    return runtimes

def bulk_benchmark_estimator(estimator, X_test, n_bulk_repeats, verbose):
    """Measure runtime prediction of the whole input."""
    n_instances = X_test.shape[0]
    runtimes = np.zeros(n_bulk_repeats, dtype=np.float)
    for i in range(n_bulk_repeats):
        start = time.time()
        estimator.predict(X_test)
        runtimes[i] = time.time() - start
    runtimes = np.array(list(map(lambda x: x / float(n_instances), runtimes)))
    if verbose:
        print("bulk_benchmark runtimes:", min(runtimes), np.percentile(
            runtimes, 50), max(runtimes))
    return runtimes

def benchmark_estimator(estimator, X_test, n_bulk_repeats=30, verbose=False):
    """
    Measure runtimes of prediction in both atomic and bulk mode.

    Parameters
    -----
    estimator : already trained estimator supporting `predict()`
    X_test : test input
    n_bulk_repeats : how many times to repeat when evaluating bulk mode

    Returns
    -----
    atomic_runtimes, bulk_runtimes : a pair of `np.array` which contain the
    runtimes in seconds.

    """
    atomic_runtimes = atomic_benchmark_estimator(estimator, X_test, verbose)
    bulk_runtimes = bulk_benchmark_estimator(estimator, X_test, n_bulk_repeats,
                                             verbose)
    return atomic_runtimes, bulk_runtimes

```

```

def generate_dataset(n_train, n_test, n_features, noise=0.1, verbose=False):
    """Generate a regression dataset with the given parameters."""
    if verbose:
        print("generating dataset...")

    X, y, coef = make_regression(n_samples=n_train + n_test,
                                  n_features=n_features, noise=noise, coef=True)

    random_seed = 13
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, train_size=n_train, test_size=n_test, random_state=random_seed)
    X_train, y_train = shuffle(X_train, y_train, random_state=random_seed)

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

    y_scaler = StandardScaler()
    y_train = y_scaler.fit_transform(y_train[:, None])[:, 0]
    y_test = y_scaler.transform(y_test[:, None])[:, 0]

    gc.collect()
    if verbose:
        print("ok")
    return X_train, y_train, X_test, y_test

def boxplot_runtimes(runtimes, pred_type, configuration):
    """
    Plot a new `Figure` with boxplots of prediction runtimes.

    Parameters
    -----
    runtimes : list of `np.array` of latencies in micro-seconds
    cls_names : list of estimator class names that generated the runtimes
    pred_type : 'bulk' or 'atomic'

    """
    fig, ax1 = plt.subplots(figsize=(10, 6))
    bp = plt.boxplot(runtimes, )

    cls_infos = ['%s\n(%d %s)' % (estimator_conf['name'],
                                    estimator_conf['complexity_computer'](
                                        estimator_conf['instance']),
                                    estimator_conf['complexity_label']) for
                 estimator_conf in configuration['estimators']]
    plt.setp(ax1, xticklabels=cls_infos)
    plt.setp(bp['boxes'], color='black')
    plt.setp(bp['whiskers'], color='black')
    plt.setp(bp['fliers'], color='red', marker='+')

    ax1.yaxis.grid(True, linestyle='-', which='major', color='lightgrey',
                  alpha=0.5)

    ax1.set_axisbelow(True)
    ax1.set_title('Prediction Time per Instance - %s, %d feats.' % (

```

```

pred_type.capitalize(),
configuration['n_features']))
ax1.set_ylabel('Prediction Time (us)')

plt.show()

def benchmark(configuration):
    """Run the whole benchmark."""
    X_train, y_train, X_test, y_test = generate_dataset(
        configuration['n_train'], configuration['n_test'],
        configuration['n_features'])

    stats = {}
    for estimator_conf in configuration['estimators']:
        print("Benchmarking", estimator_conf['instance'])
        estimator_conf['instance'].fit(X_train, y_train)
        gc.collect()
        a, b = benchmark_estimator(estimator_conf['instance'], X_test)
        stats[estimator_conf['name']] = {'atomic': a, 'bulk': b}

    cls_names = [estimator_conf['name'] for estimator_conf in configuration[
        'estimators']]
    runtimes = [1e6 * stats[clf_name]['atomic'] for clf_name in cls_names]
    boxplot_runtimes(runtimes, 'atomic', configuration)
    runtimes = [1e6 * stats[clf_name]['bulk'] for clf_name in cls_names]
    boxplot_runtimes(runtimes, 'bulk (%d)' % configuration['n_test'],
                      configuration)

def n_feature_influence(estimators, n_train, n_test, n_features, percentile):
    """
    Estimate influence of the number of features on prediction time.

    Parameters
    -------

    estimators : dict of (name (str), estimator) to benchmark
    n_train : nber of training instances (int)
    n_test : nber of testing instances (int)
    n_features : list of feature-space dimensionality to test (int)
    percentile : percentile at which to measure the speed (int [0-100])

    Returns:
    -------

    percentiles : dict (estimator_name,
                         dict (n_features, percentile_perf_in_us))
    """

    percentiles = defaultdict(defaultdict)
    for n in n_features:
        print("benchmarking with %d features" % n)
        X_train, y_train, X_test, y_test = generate_dataset(n_train, n_test, n)
        for cls_name, estimator in estimators.items():
            estimator.fit(X_train, y_train)
            gc.collect()
            runtimes = bulk_benchmark_estimator(estimator, X_test, 30, False)

```

```

percentiles[cls_name][n] = 1e6 * np.percentile(runtimes,
                                              percentile)
return percentiles

def plot_n_features_influence(percentiles, percentile):
    fig, ax1 = plt.subplots(figsize=(10, 6))
    colors = ['r', 'g', 'b']
    for i, cls_name in enumerate(percentiles.keys()):
        x = np.array(sorted([n for n in percentiles[cls_name].keys()]))
        y = np.array([percentiles[cls_name][n] for n in x])
        plt.plot(x, y, color=colors[i], )
    ax1.yaxis.grid(True, linestyle='-', which='major', color='lightgrey',
                  alpha=0.5)
    ax1.set_axisbelow(True)
    ax1.set_title('Evolution of Prediction Time with #Features')
    ax1.set_xlabel('#Features')
    ax1.set_ylabel('Prediction Time at %d%-ile (us)' % percentile)
    plt.show()

def benchmark_throughputs(configuration, duration_secs=0.1):
    """benchmark throughput for different estimators."""
    X_train, y_train, X_test, y_test = generate_dataset(
        configuration['n_train'], configuration['n_test'],
        configuration['n_features'])
    throughputs = dict()
    for estimator_config in configuration['estimators']:
        estimator_config['instance'].fit(X_train, y_train)
        start_time = time.time()
        n_predictions = 0
        while (time.time() - start_time) < duration_secs:
            estimator_config['instance'].predict(X_test[[0]])
            n_predictions += 1
        throughputs[estimator_config['name']] = n_predictions / duration_secs
    return throughputs

def plot_benchmark_throughput(throughputs, configuration):
    fig, ax = plt.subplots(figsize=(10, 6))
    colors = ['r', 'g', 'b']
    cls_infos = ['%s\n(%d %s)' % (estimator_conf['name'],
                                    estimator_conf['complexity_computer'],
                                    estimator_conf['instance']),
                 estimator_conf['complexity_label']) for
                 estimator_conf in configuration['estimators']]
    cls_values = [throughputs[estimator_conf['name']] for estimator_conf in
                 configuration['estimators']]
    plt.bar(range(len(throughputs)), cls_values, width=0.5, color=colors)
    ax.set_xticks(np.linspace(0.25, len(throughputs) - 0.75, len(throughputs)))
    ax.set_xticklabels(cls_infos, fontsize=10)
    ymax = max(cls_values) * 1.2
    ax.set_xlim((0, ymax))
    ax.set_ylabel('Throughput (predictions/sec)')
    ax.set_title('Prediction Throughput for different estimators (%d '
                 'features)' % configuration['n_features'])
    plt.show()

```

```

# ##### Main code #####
# Main code

start_time = time.time()

# ##### Benchmark bulk/atomic prediction speed for various regressors #####
# Benchmark bulk/atomic prediction speed for various regressors
configuration = {
    'n_train': int(1e3),
    'n_test': int(1e2),
    'n_features': int(1e2),
    'estimators': [
        {'name': 'Linear Model',
         'instance': SGDRegressor(penalty='elasticnet', alpha=0.01,
                                   l1_ratio=0.25, fit_intercept=True,
                                   tol=1e-4),
         'complexity_label': 'non-zero coefficients',
         'complexity_computer': lambda clf: np.count_nonzero(clf.coef_)},
        {'name': 'RandomForest',
         'instance': RandomForestRegressor(n_estimators=100),
         'complexity_label': 'estimators',
         'complexity_computer': lambda clf: clf.n_estimators},
        {'name': 'SVR',
         'instance': SVR(kernel='rbf'),
         'complexity_label': 'support vectors',
         'complexity_computer': lambda clf: len(clf.support_vectors_)},
    ]
}
benchmark(configuration)

# benchmark n_features influence on prediction speed
percentile = 90
percentiles = n_feature_influence({'ridge': Ridge()},
                                    configuration['n_train'],
                                    configuration['n_test'],
                                    [100, 250, 500], percentile)
plot_n_features_influence(percentiles, percentile)

# benchmark throughput
throughputs = benchmark_throughputs(configuration)
plot_benchmark_throughput(throughputs, configuration)

stop_time = time.time()
print("example run in %.2fs" % (stop_time - start_time))

```

Total running time of the script: (0 minutes 9.597 seconds)

Note: Click [here](#) to download the full example code

5.2.11 Out-of-core classification of text documents

This is an example showing how scikit-learn can be used for classification using an out-of-core approach: learning from data that doesn't fit into main memory. We make use of an online classifier, i.e., one that supports the `partial_fit` method, that will be fed with batches of examples. To guarantee that the features space remains the same over time

we leverage a HashingVectorizer that will project each example into the same feature space. This is especially useful in the case of text classification where new features (words) may appear in each batch.

The dataset used in this example is Reuters-21578 as provided by the UCI ML repository. It will be automatically downloaded and uncompressed on first run.

The plot represents the learning curve of the classifier: the evolution of classification accuracy over the course of the mini-batches. Accuracy is measured on the first 1000 samples, held out as a validation set.

To limit the memory consumption, we queue examples up to a fixed amount before feeding them to the learner.

```
# Authors: Eustache Diemert <eustache@diemert.fr>
#          @FedericoV <https://github.com/FedericoV/>
# License: BSD 3 clause

from glob import glob
import itertools
import os.path
import re
import tarfile
import time
import sys

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams

from html.parser import HTMLParser
from urllib.request import urlretrieve
from sklearn.datasets import get_data_home
from sklearn.feature_extraction.text import HashingVectorizer
from sklearn.linear_model import SGDClassifier
from sklearn.linear_model import PassiveAggressiveClassifier
from sklearn.linear_model import Perceptron
from sklearn.naive_bayes import MultinomialNB

def _not_in_sphinx():
    # Hack to detect whether we are running by the sphinx builder
    return '__file__' in globals()
```

Reuters Dataset related routines

```
class ReutersParser(HTMLParser):
    """Utility class to parse a SGML file and yield documents one at a time."""

    def __init__(self, encoding='latin-1'):
        HTMLParser.__init__(self)
        self._reset()
        self.encoding = encoding

    def handle_starttag(self, tag, attrs):
        method = 'start_' + tag
        getattr(self, method, lambda x: None)(attrs)

    def handle_endtag(self, tag):
        method = 'end_' + tag
        getattr(self, method, lambda: None)()
```

```

def _reset(self):
    self.in_title = 0
    self.in_body = 0
    self.in_topics = 0
    self.in_topic_d = 0
    self.title = ""
    self.body = ""
    self.topics = []
    self.topic_d = ""

def parse(self, fd):
    self.docs = []
    for chunk in fd:
        self.feed(chunk.decode(self.encoding))
        for doc in self.docs:
            yield doc
        self.docs = []
    self.close()

def handle_data(self, data):
    if self.in_body:
        self.body += data
    elif self.in_title:
        self.title += data
    elif self.in_topic_d:
        self.topic_d += data

def start_reuters(self, attributes):
    pass

def end_reuters(self):
    self.body = re.sub(r'\s+', r' ', self.body)
    self.docs.append({'title': self.title,
                      'body': self.body,
                      'topics': self.topics})
    self._reset()

def start_title(self, attributes):
    self.in_title = 1

def end_title(self):
    self.in_title = 0

def start_body(self, attributes):
    self.in_body = 1

def end_body(self):
    self.in_body = 0

def start_topics(self, attributes):
    self.in_topics = 1

def end_topics(self):
    self.in_topics = 0

def start_d(self, attributes):
    self.in_topic_d = 1

```

```

def end_d(self):
    self.in_topic_d = 0
    self.topics.append(self.topic_d)
    self.topic_d = ""

def stream_reuters_documents(data_path=None):
    """Iterate over documents of the Reuters dataset.

    The Reuters archive will automatically be downloaded and uncompressed if
    the `data_path` directory does not exist.

    Documents are represented as dictionaries with 'body' (str),
    'title' (str), 'topics' (list(str)) keys.

    """
    DOWNLOAD_URL = ('http://archive.ics.uci.edu/ml/machine-learning-databases/'
                    'reuters21578-mld/reuters21578.tar.gz')
    ARCHIVE_FILENAME = 'reuters21578.tar.gz'

    if data_path is None:
        data_path = os.path.join(get_data_home(), "reuters")
    if not os.path.exists(data_path):
        """Download the dataset."""
        print("downloading dataset (once and for all) into %s" %
              data_path)
        os.mkdir(data_path)

    def progress(blocknum, bs, size):
        total_sz_mb = '%.2f MB' % (size / 1e6)
        current_sz_mb = '%.2f MB' % ((blocknum * bs) / 1e6)
        if _not_in_sphinx():
            sys.stdout.write(
                '\rdownloaded %s / %s' % (current_sz_mb, total_sz_mb))

    archive_path = os.path.join(data_path, ARCHIVE_FILENAME)
    urlretrieve(DOWNLOAD_URL, filename=archive_path,
                reporthook=progress)
    if _not_in_sphinx():
        sys.stdout.write('\r')
    print("untarring Reuters dataset...")
    tarfile.open(archive_path, 'r:gz').extractall(data_path)
    print("done.")

    parser = ReutersParser()
    for filename in glob(os.path.join(data_path, "*.sgm")):
        for doc in parser.parse(open(filename, 'rb')):
            yield doc

```

Main

Create the vectorizer and limit the number of features to a reasonable maximum

```
vectorizer = HashingVectorizer(decode_error='ignore', n_features=2 ** 18,
                             alternate_sign=False)
```

```

# Iterator over parsed Reuters SGML files.
data_stream = stream_reuters_documents()

# We learn a binary classification between the "acq" class and all the others.
# "acq" was chosen as it is more or less evenly distributed in the Reuters
# files. For other datasets, one should take care of creating a test set with
# a realistic portion of positive instances.
all_classes = np.array([0, 1])
positive_class = 'acq'

# Here are some classifiers that support the `partial_fit` method
partial_fit_classifiers = {
    'SGD': SGDClassifier(max_iter=5, tol=1e-3),
    'Perceptron': Perceptron(tol=1e-3),
    'NB Multinomial': MultinomialNB(alpha=0.01),
    'Passive-Aggressive': PassiveAggressiveClassifier(tol=1e-3),
}

def get_minibatch(doc_iter, size, pos_class=positive_class):
    """Extract a minibatch of examples, return a tuple X_text, y.

    Note: size is before excluding invalid docs with no topics assigned.

    """
    data = [('{title}\n\n{body}'.format(**doc), pos_class in doc['topics'])
            for doc in itertools.islice(doc_iter, size)
            if doc['topics']]
    if not len(data):
        return np.asarray([], dtype=int), np.asarray([], dtype=int)
    X_text, y = zip(*data)
    return X_text, np.asarray(y, dtype=int)

def iter_minibatches(doc_iter, minibatch_size):
    """Generator of minibatches."""
    X_text, y = get_minibatch(doc_iter, minibatch_size)
    while len(X_text):
        yield X_text, y
        X_text, y = get_minibatch(doc_iter, minibatch_size)

# test data statistics
test_stats = {'n_test': 0, 'n_test_pos': 0}

# First we hold out a number of examples to estimate accuracy
n_test_documents = 1000
tick = time.time()
X_test_text, y_test = get_minibatch(data_stream, 1000)
parsing_time = time.time() - tick
tick = time.time()
X_test = vectorizer.transform(X_test_text)
vectorizing_time = time.time() - tick
test_stats['n_test'] += len(y_test)
test_stats['n_test_pos'] += sum(y_test)
print("Test set is %d documents (%d positive) % (len(y_test), sum(y_test)))
```

```

def progress(cls_name, stats):
    """Report progress information, return a string."""
    duration = time.time() - stats['t0']
    s = "%20s classifier : \t" % cls_name
    s += "%(n_train)6d train docs (%(n_train_pos)6d positive) " % stats
    s += "%(n_test)6d test docs (%(n_test_pos)6d positive) " % test_stats
    s += "accuracy: %(accuracy).3f " % stats
    s += "in %.2fs (%5d docs/s)" % (duration, stats['n_train'] / duration)
    return s

cls_stats = {}

for cls_name in partial_fit_classifiers:
    stats = {'n_train': 0, 'n_train_pos': 0,
              'accuracy': 0.0, 'accuracy_history': [(0, 0)], 't0': time.time(),
              'runtime_history': [(0, 0)], 'total_fit_time': 0.0}
    cls_stats[cls_name] = stats

get_minibatch(data_stream, n_test_documents)
# Discard test set

# We will feed the classifier with mini-batches of 1000 documents; this means
# we have at most 1000 docs in memory at any time. The smaller the document
# batch, the bigger the relative overhead of the partial fit methods.
minibatch_size = 1000

# Create the data_stream that parses Reuters SGML files and iterates on
# documents as a stream.
minibatch_iterators = iter_minibatches(data_stream, minibatch_size)
total_vect_time = 0.0

# Main loop : iterate on mini-batches of examples
for i, (X_train_text, y_train) in enumerate(minibatch_iterators):

    tick = time.time()
    X_train = vectorizer.transform(X_train_text)
    total_vect_time += time.time() - tick

    for cls_name, cls in partial_fit_classifiers.items():
        tick = time.time()
        # update estimator with examples in the current mini-batch
        cls.partial_fit(X_train, y_train, classes=all_classes)

        # accumulate test accuracy stats
        cls_stats[cls_name]['total_fit_time'] += time.time() - tick
        cls_stats[cls_name]['n_train'] += X_train.shape[0]
        cls_stats[cls_name]['n_train_pos'] += sum(y_train)
        tick = time.time()
        cls_stats[cls_name]['accuracy'] = cls.score(X_test, y_test)
        cls_stats[cls_name]['prediction_time'] = time.time() - tick
        acc_history = (cls_stats[cls_name]['accuracy'],
                      cls_stats[cls_name]['n_train'])
        cls_stats[cls_name]['accuracy_history'].append(acc_history)
        run_history = (cls_stats[cls_name]['accuracy'],
                      total_vect_time + cls_stats[cls_name]['total_fit_time'])


```

```

cls_stats[cls_name]['runtime_history'].append(run_history)

if i % 3 == 0:
    print(progress(cls_name, cls_stats[cls_name]))
if i % 3 == 0:
    print('\n')

```

Out:

```

downloading dataset (once and for all) into /home/circleci/scikit_learn_data/reuters
untarring Reuters dataset...
done.

Test set is 966 documents (88 positive)
      SGD classifier : 994 train docs ( 121 positive) 966_
→ test docs ( 88 positive) accuracy: 0.907 in 0.74s ( 1342 docs/s)
      Perceptron classifier : 994 train docs ( 121 positive) 966_
→ test docs ( 88 positive) accuracy: 0.922 in 0.74s ( 1338 docs/s)
      NB Multinomial classifier : 994 train docs ( 121 positive) 966_
→ test docs ( 88 positive) accuracy: 0.909 in 0.75s ( 1327 docs/s)
      Passive-Aggressive classifier : 994 train docs ( 121 positive) 966_
→ test docs ( 88 positive) accuracy: 0.944 in 0.75s ( 1323 docs/s)

      SGD classifier : 3924 train docs ( 496 positive) 966_
→ test docs ( 88 positive) accuracy: 0.957 in 2.20s ( 1784 docs/s)
      Perceptron classifier : 3924 train docs ( 496 positive) 966_
→ test docs ( 88 positive) accuracy: 0.952 in 2.20s ( 1782 docs/s)
      NB Multinomial classifier : 3924 train docs ( 496 positive) 966_
→ test docs ( 88 positive) accuracy: 0.917 in 2.21s ( 1778 docs/s)
      Passive-Aggressive classifier : 3924 train docs ( 496 positive) 966_
→ test docs ( 88 positive) accuracy: 0.967 in 2.21s ( 1776 docs/s)

      SGD classifier : 6836 train docs ( 793 positive) 966_
→ test docs ( 88 positive) accuracy: 0.967 in 3.82s ( 1791 docs/s)
      Perceptron classifier : 6836 train docs ( 793 positive) 966_
→ test docs ( 88 positive) accuracy: 0.950 in 3.82s ( 1790 docs/s)
      NB Multinomial classifier : 6836 train docs ( 793 positive) 966_
→ test docs ( 88 positive) accuracy: 0.921 in 3.82s ( 1787 docs/s)
      Passive-Aggressive classifier : 6836 train docs ( 793 positive) 966_
→ test docs ( 88 positive) accuracy: 0.967 in 3.83s ( 1786 docs/s)

      SGD classifier : 9597 train docs ( 1139 positive) 966_
→ test docs ( 88 positive) accuracy: 0.963 in 5.39s ( 1779 docs/s)
      Perceptron classifier : 9597 train docs ( 1139 positive) 966_
→ test docs ( 88 positive) accuracy: 0.955 in 5.40s ( 1778 docs/s)
      NB Multinomial classifier : 9597 train docs ( 1139 positive) 966_
→ test docs ( 88 positive) accuracy: 0.930 in 5.40s ( 1776 docs/s)
      Passive-Aggressive classifier : 9597 train docs ( 1139 positive) 966_
→ test docs ( 88 positive) accuracy: 0.965 in 5.41s ( 1775 docs/s)

      SGD classifier : 12513 train docs ( 1558 positive) 966_
→ test docs ( 88 positive) accuracy: 0.969 in 7.02s ( 1783 docs/s)
      Perceptron classifier : 12513 train docs ( 1558 positive) 966_
→ test docs ( 88 positive) accuracy: 0.843 in 7.02s ( 1783 docs/s)
      NB Multinomial classifier : 12513 train docs ( 1558 positive) 966_
→ test docs ( 88 positive) accuracy: 0.936 in 7.02s ( 1781 docs/s)

```

```

Passive-Aggressive classifier :      12513 train docs ( 1558 positive) 966_
→test docs ( 88 positive) accuracy: 0.965 in 7.03s ( 1781 docs/s)

SGD classifier :      14935 train docs ( 1863 positive) 966_
→test docs ( 88 positive) accuracy: 0.966 in 8.52s ( 1752 docs/s)
Perceptron classifier :      14935 train docs ( 1863 positive) 966_
→test docs ( 88 positive) accuracy: 0.958 in 8.53s ( 1751 docs/s)
NB Multinomial classifier :      14935 train docs ( 1863 positive) 966_
→test docs ( 88 positive) accuracy: 0.938 in 8.53s ( 1750 docs/s)
Passive-Aggressive classifier :      14935 train docs ( 1863 positive) 966_
→test docs ( 88 positive) accuracy: 0.955 in 8.54s ( 1749 docs/s)

SGD classifier :      17417 train docs ( 2171 positive) 966_
→test docs ( 88 positive) accuracy: 0.960 in 9.98s ( 1745 docs/s)
Perceptron classifier :      17417 train docs ( 2171 positive) 966_
→test docs ( 88 positive) accuracy: 0.964 in 9.98s ( 1744 docs/s)
NB Multinomial classifier :      17417 train docs ( 2171 positive) 966_
→test docs ( 88 positive) accuracy: 0.943 in 9.99s ( 1743 docs/s)
Passive-Aggressive classifier :      17417 train docs ( 2171 positive) 966_
→test docs ( 88 positive) accuracy: 0.960 in 9.99s ( 1743 docs/s)

```

Plot results

```

def plot_accuracy(x, y, x_legend):
    """Plot accuracy as a function of x."""
    x = np.array(x)
    y = np.array(y)
    plt.title('Classification accuracy as a function of %s' % x_legend)
    plt.xlabel('%s' % x_legend)
    plt.ylabel('Accuracy')
    plt.grid(True)
    plt.plot(x, y)

rcParams['legend.fontsize'] = 10
cls_names = list(sorted(cls_stats.keys()))

# Plot accuracy evolution
plt.figure()
for _, stats in sorted(cls_stats.items()):
    # Plot accuracy evolution with #examples
    accuracy, n_examples = zip(*stats['accuracy_history'])
    plot_accuracy(n_examples, accuracy, "training examples (#)")
    ax = plt.gca()
    ax.set_ylim((0.8, 1))
plt.legend(cls_names, loc='best')

plt.figure()
for _, stats in sorted(cls_stats.items()):
    # Plot accuracy evolution with runtime
    accuracy, runtime = zip(*stats['runtime_history'])
    plot_accuracy(runtime, accuracy, 'runtime (s)')
    ax = plt.gca()
    ax.set_ylim((0.8, 1))

```

```

plt.legend(cls_names, loc='best')

# Plot fitting times
plt.figure()
fig = plt.gcf()
cls_runtime = [stats['total_fit_time']]
    for cls_name, stats in sorted(cls_stats.items())]

cls_runtime.append(total_vect_time)
cls_names.append('Vectorization')
bar_colors = ['b', 'g', 'r', 'c', 'm', 'y']

ax = plt.subplot(111)
rectangles = plt.bar(range(len(cls_names)), cls_runtime, width=0.5,
                     color=bar_colors)

ax.set_xticks(np.linspace(0, len(cls_names) - 1, len(cls_names)))
ax.set_xticklabels(cls_names, fontsize=10)
ymax = max(cls_runtime) * 1.2
ax.set_xlim((0, ymax))
ax.set_ylabel('runtime (s)')
ax.set_title('Training Times')

def autolabel(rectangles):
    """attach some text vi autolabel on rectangles."""
    for rect in rectangles:
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width() / 2.,
                1.05 * height, '%.4f' % height,
                ha='center', va='bottom')
    plt.setp(plt.xticks()[1], rotation=30)

autolabel(rectangles)
plt.tight_layout()
plt.show()

# Plot prediction times
plt.figure()
cls_runtime = []
cls_names = list(sorted(cls_stats.keys()))
for cls_name, stats in sorted(cls_stats.items()):
    cls_runtime.append(stats['prediction_time'])
cls_runtime.append(parsing_time)
cls_names.append('Read/Parse\nFeat.Extr.')
cls_runtime.append(vectorizing_time)
cls_names.append('Hashing\nVect.')

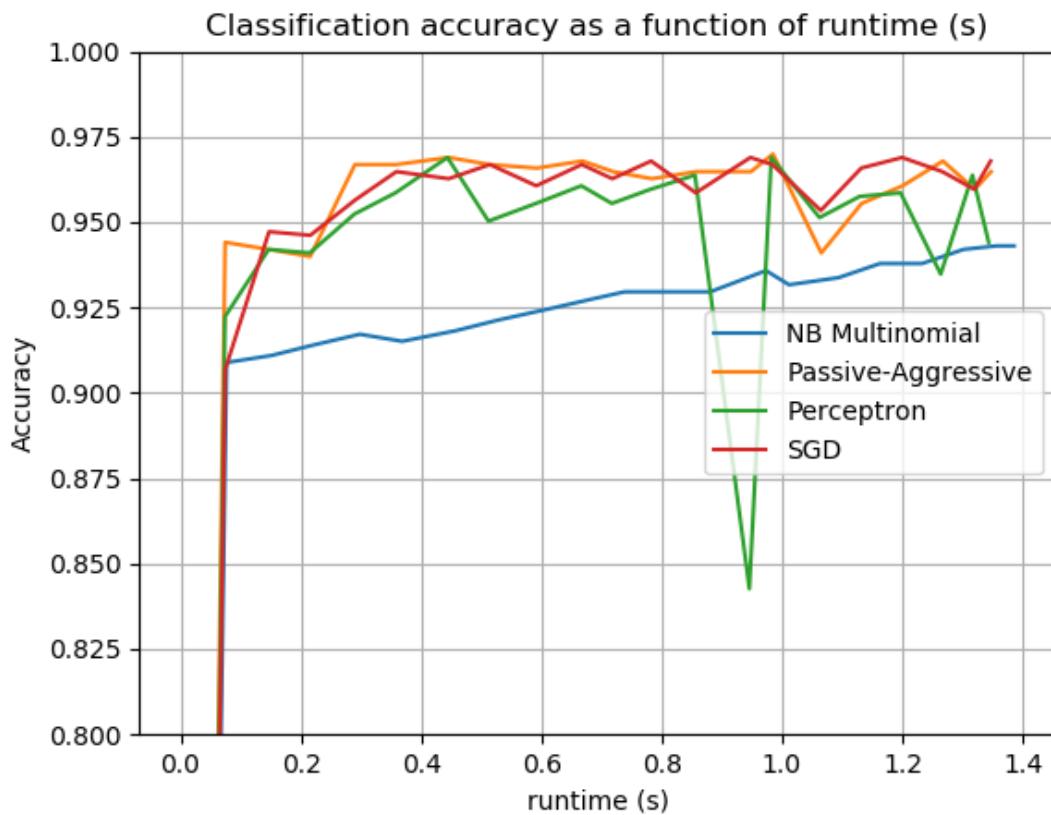
ax = plt.subplot(111)
rectangles = plt.bar(range(len(cls_names)), cls_runtime, width=0.5,
                     color=bar_colors)

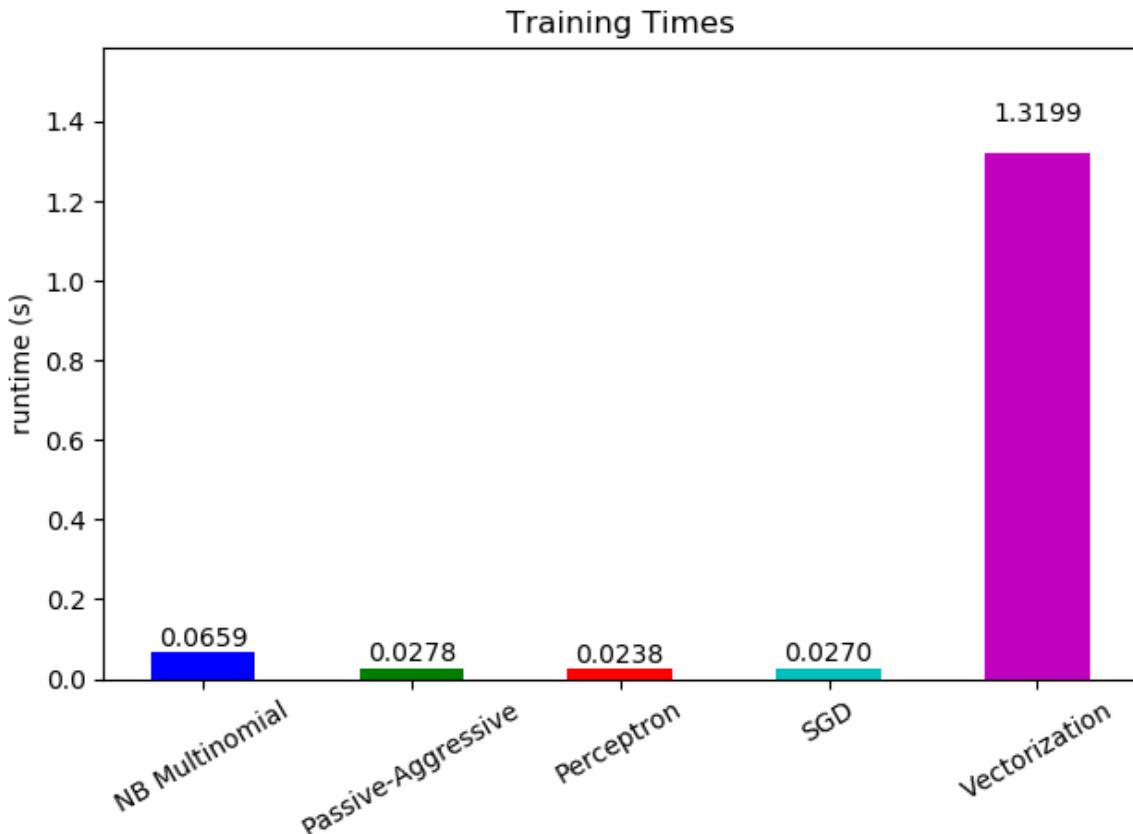
ax.set_xticks(np.linspace(0, len(cls_names) - 1, len(cls_names)))
ax.set_xticklabels(cls_names, fontsize=8)
plt.setp(plt.xticks()[1], rotation=30)
ymax = max(cls_runtime) * 1.2
ax.set_xlim((0, ymax))

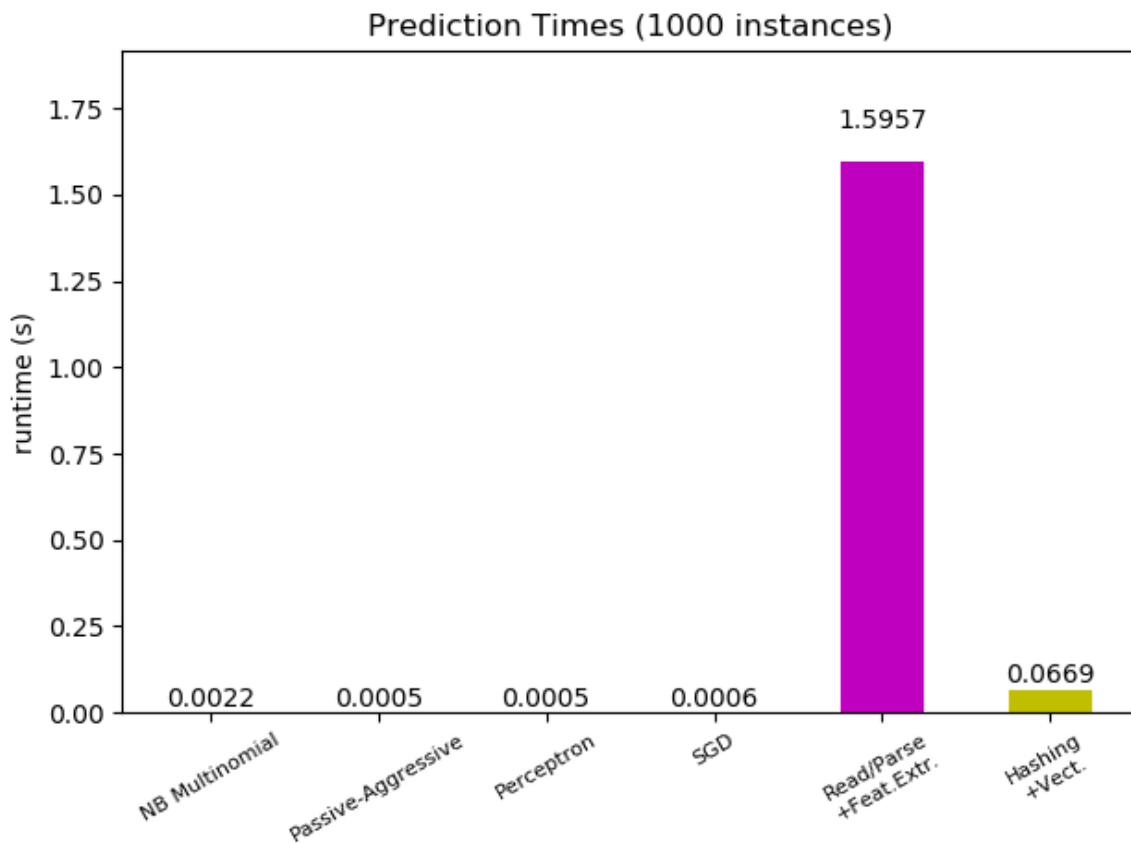
```

```
ax.set_ylabel('runtime (s)')
ax.set_title('Prediction Times (%d instances)' % n_test_documents)
autolabel(rectangles)
plt.tight_layout()
plt.show()
```









Total running time of the script: (0 minutes 12.190 seconds)

5.3 Biclustering

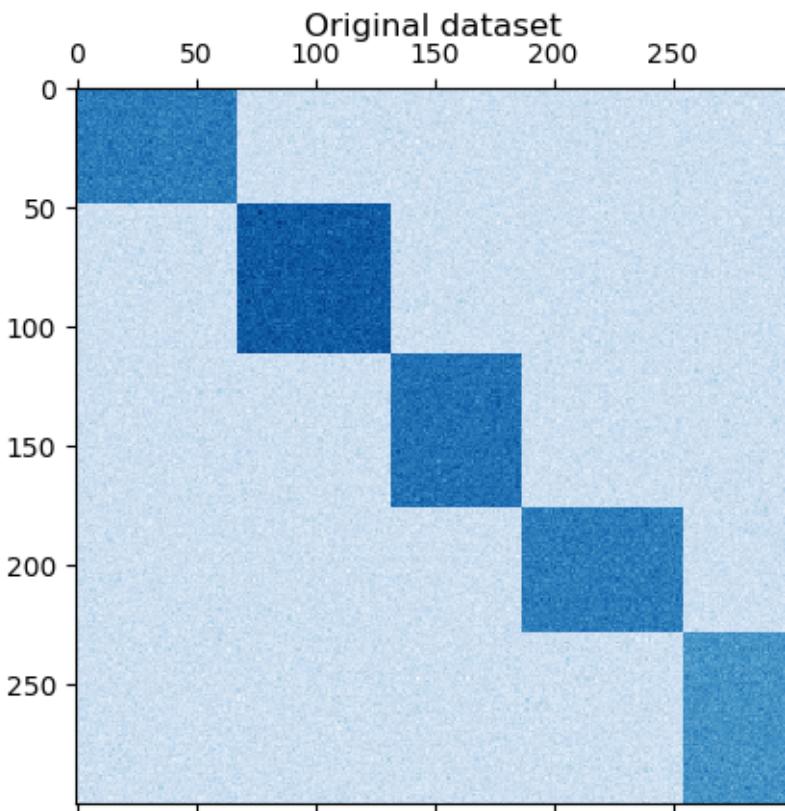
Examples concerning the `sklearn.cluster.bicluster` module.

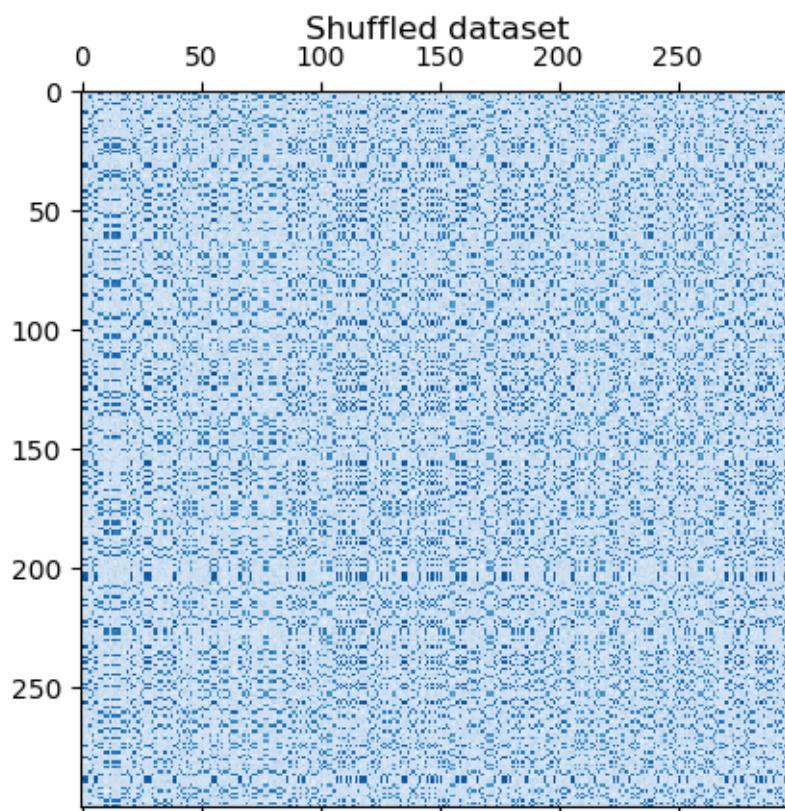
Note: Click [here](#) to download the full example code

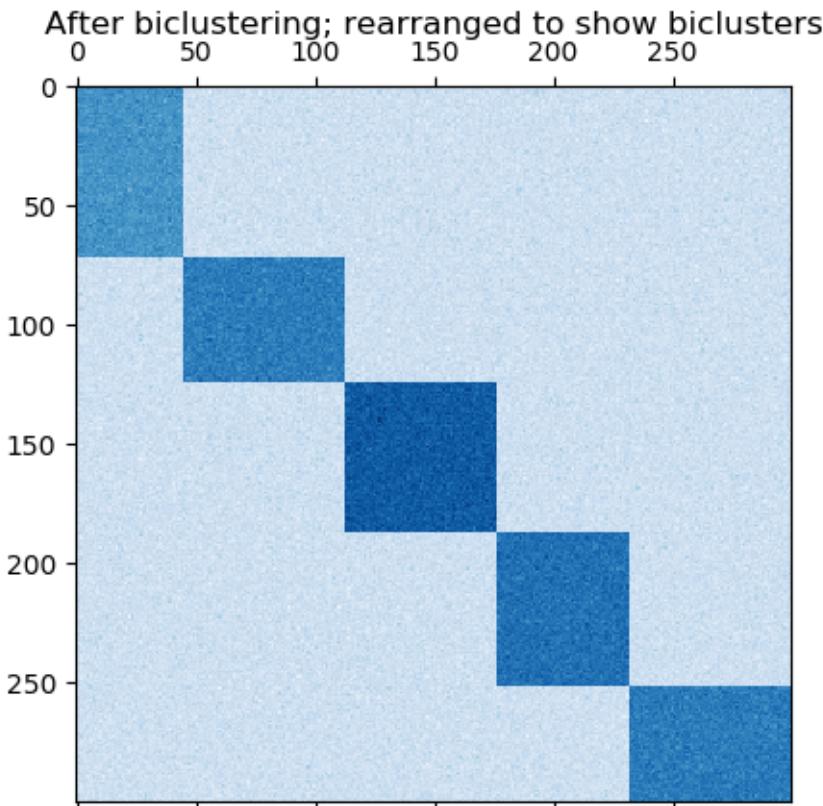
5.3.1 A demo of the Spectral Co-Clustering algorithm

This example demonstrates how to generate a dataset and bicluster it using the Spectral Co-Clustering algorithm.

The dataset is generated using the `make_biclusters` function, which creates a matrix of small values and implants bicluster with large values. The rows and columns are then shuffled and passed to the Spectral Co-Clustering algorithm. Rearranging the shuffled matrix to make biclusters contiguous shows how accurately the algorithm found the biclusters.







Out:

```
consensus score: 1.000
```

```
print(__doc__)

# Author: Kemal Eren <kemal@kemaleren.com>
# License: BSD 3 clause

import numpy as np
from matplotlib import pyplot as plt

from sklearn.datasets import make_biclusters
from sklearn.datasets import samples_generator as sg
from sklearn.cluster.bicluster import SpectralCoclustering
from sklearn.metrics import consensus_score

data, rows, columns = make_biclusters(
    shape=(300, 300), n_clusters=5, noise=5,
    shuffle=False, random_state=0)
```

```

plt.matshow(data, cmap=plt.cm.Blues)
plt.title("Original dataset")

data, row_idx, col_idx = sg._shuffle(data, random_state=0)
plt.matshow(data, cmap=plt.cm.Blues)
plt.title("Shuffled dataset")

model = SpectralCoclustering(n_clusters=5, random_state=0)
model.fit(data)
score = consensus_score(model.biclusters_,
                        (rows[:, row_idx], columns[:, col_idx]))

print("consensus score: {:.3f}".format(score))

fit_data = data[np.argsort(model.row_labels_)]
fit_data = fit_data[:, np.argsort(model.column_labels_)]

plt.matshow(fit_data, cmap=plt.cm.Blues)
plt.title("After biclustering; rearranged to show biclusters")

plt.show()

```

Total running time of the script: (0 minutes 0.061 seconds)

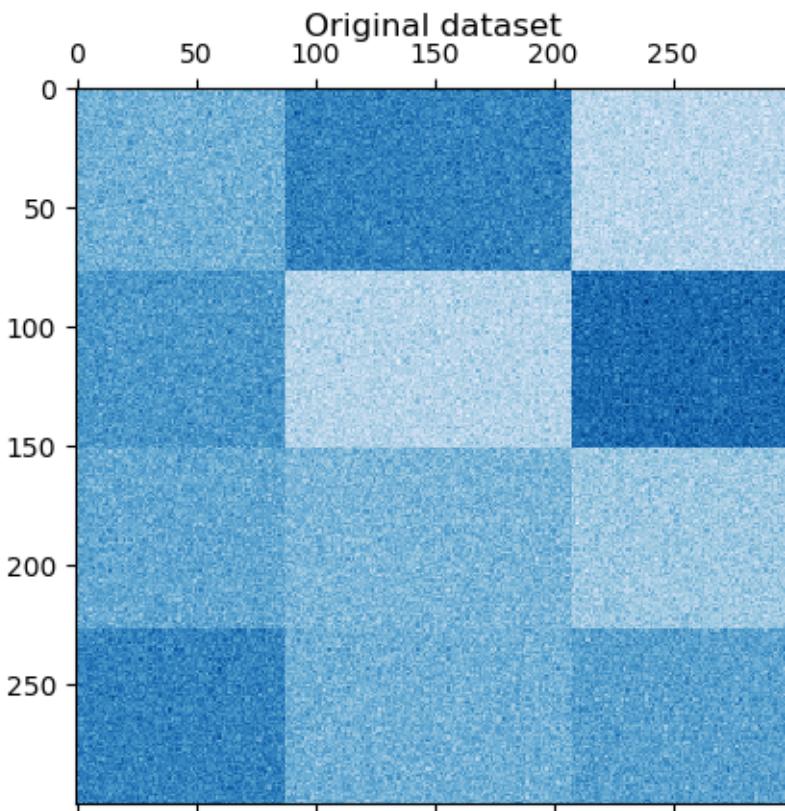
Note: Click [here](#) to download the full example code

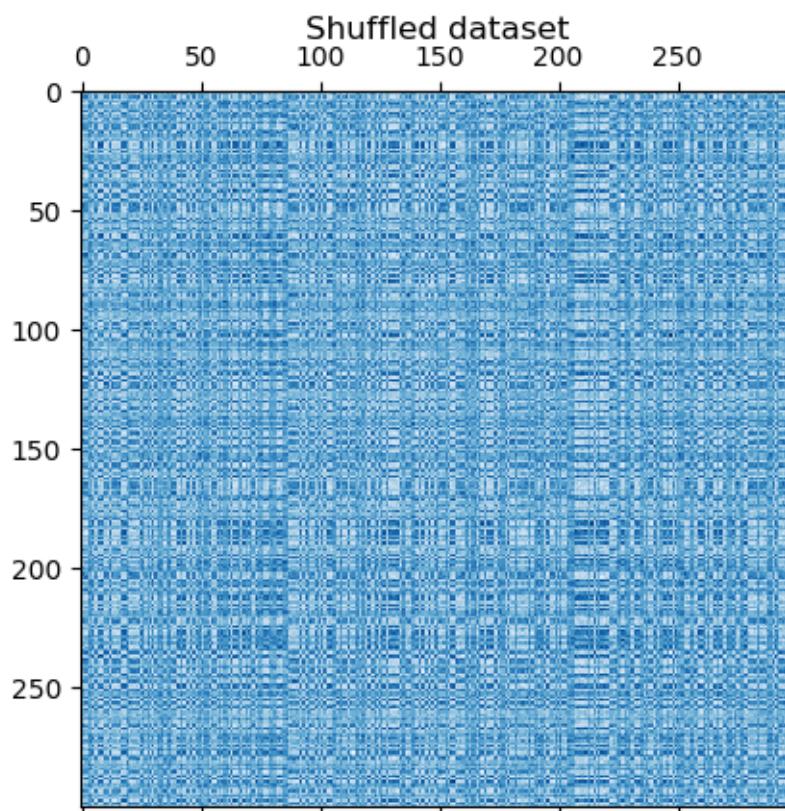
5.3.2 A demo of the Spectral Biclustering algorithm

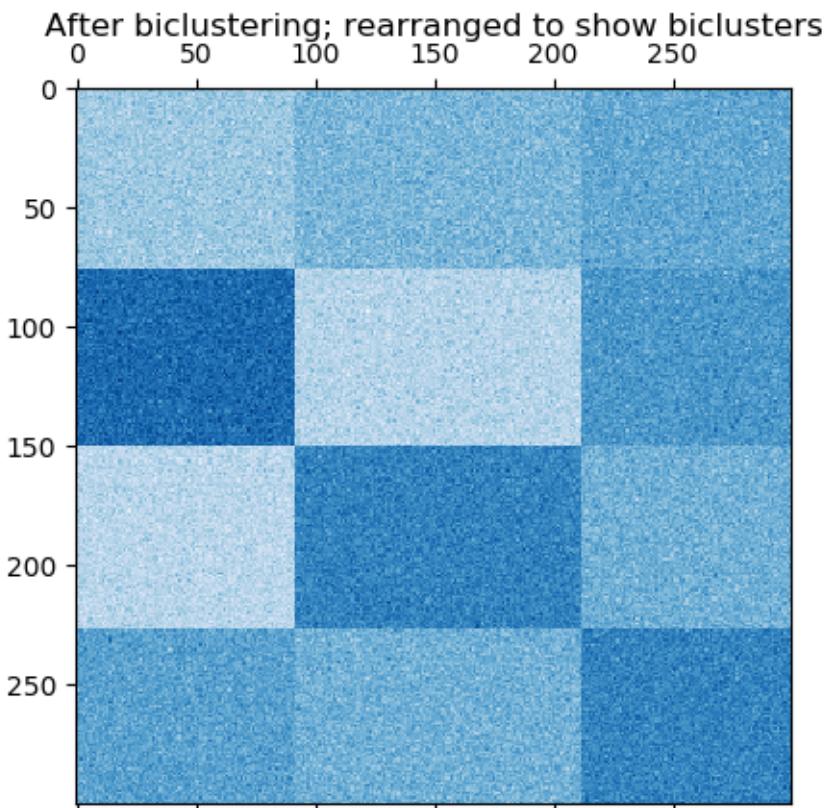
This example demonstrates how to generate a checkerboard dataset and bicluster it using the Spectral Biclustering algorithm.

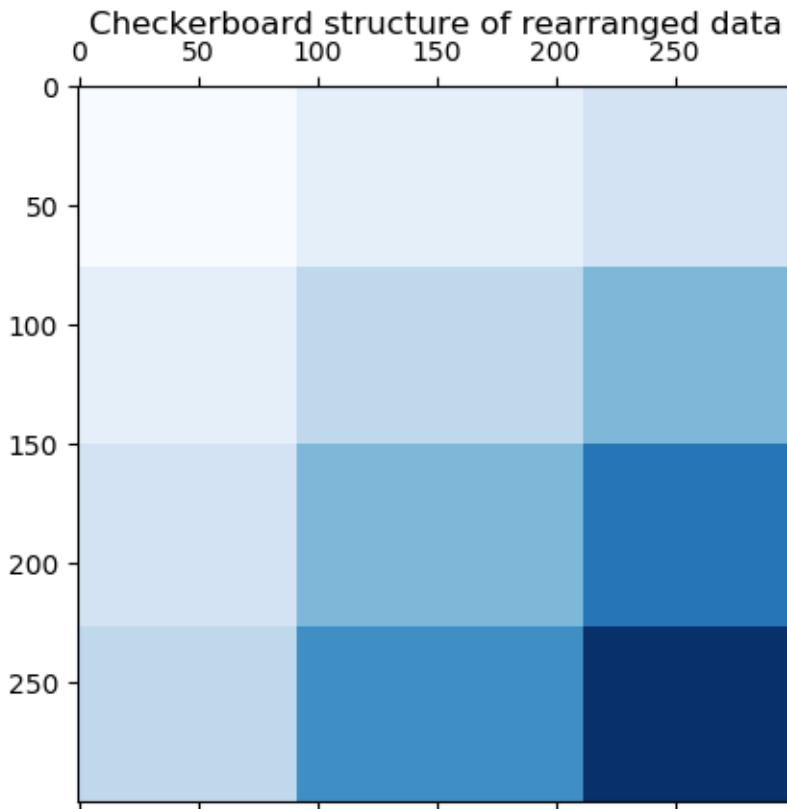
The data is generated with the `make_checkerboard` function, then shuffled and passed to the Spectral Biclustering algorithm. The rows and columns of the shuffled matrix are rearranged to show the biclusters found by the algorithm.

The outer product of the row and column label vectors shows a representation of the checkerboard structure.









Out:

```
consensus score: 1.0
```

```
print(__doc__)

# Author: Kemal Eren <kemal@kemaleren.com>
# License: BSD 3 clause

import numpy as np
from matplotlib import pyplot as plt

from sklearn.datasets import make_checkerboard
from sklearn.datasets import samples_generator as sg
from sklearn.cluster.bicluster import SpectralBiclustering
from sklearn.metrics import consensus_score

n_clusters = (4, 3)
data, rows, columns = make_checkerboard(
    shape=(300, 300), n_clusters=n_clusters, noise=10,
    shuffle=False, random_state=0)
```

```
plt.matshow(data, cmap=plt.cm.Blues)
plt.title("Original dataset")

data, row_idx, col_idx = sg._shuffle(data, random_state=0)
plt.matshow(data, cmap=plt.cm.Blues)
plt.title("Shuffled dataset")

model = SpectralBiclustering(n_clusters=n_clusters, method='log',
                             random_state=0)
model.fit(data)
score = consensus_score(model.biclusters_,
                        (rows[:, row_idx], columns[:, col_idx]))

print("consensus score: {:.1f}".format(score))

fit_data = data[np.argsort(model.row_labels_)]
fit_data = fit_data[:, np.argsort(model.column_labels_)]

plt.matshow(fit_data, cmap=plt.cm.Blues)
plt.title("After biclustering; rearranged to show biclusters")

plt.matshow(np.outer(np.sort(model.row_labels_) + 1,
                    np.sort(model.column_labels_) + 1),
            cmap=plt.cm.Blues)
plt.title("Checkerboard structure of rearranged data")

plt.show()
```

Total running time of the script: (0 minutes 0.447 seconds)

Note: Click [here](#) to download the full example code

5.3.3 Biclustering documents with the Spectral Co-clustering algorithm

This example demonstrates the Spectral Co-clustering algorithm on the twenty newsgroups dataset. The ‘comp.os.ms-windows.misc’ category is excluded because it contains many posts containing nothing but data.

The TF-IDF vectorized posts form a word frequency matrix, which is then biclustered using Dhillon’s Spectral Co-Clustering algorithm. The resulting document-word biclusters indicate subsets words used more often in those subsets documents.

For a few of the best biclusters, its most common document categories and its ten most important words get printed. The best biclusters are determined by their normalized cut. The best words are determined by comparing their sums inside and outside the bicluster.

For comparison, the documents are also clustered using MiniBatchKMeans. The document clusters derived from the biclusters achieve a better V-measure than clusters found by MiniBatchKMeans.

Out:

```
Vectorizing...
Coclustering...
Done in 4.10s. V-measure: 0.4435
MiniBatchKMeans...
Done in 6.36s. V-measure: 0.3344
```

```
Best biclusters:
-----
bicluster 0 : 1957 documents, 4363 words
categories   : 23% talk.politics.guns, 18% talk.politics.misc, 17% sci.med
words       : gun, guns, geb, banks, gordon, clinton, pitt, cdt, surrender, veal

bicluster 1 : 1263 documents, 3551 words
categories   : 27% soc.religion.christian, 25% talk.politics.mideast, 24% alt.atheism
words       : god, jesus, christians, sin, objective, kent, belief, christ, faith, ↵
               ↵moral

bicluster 2 : 2212 documents, 2774 words
categories   : 18% comp.sys.mac.hardware, 17% comp.sys.ibm.pc.hardware, 15% comp.
               ↵graphics
words       : voltage, board, dsp, stereo, receiver, packages, shipping, circuit, ↵
               ↵package, compression

bicluster 3 : 1774 documents, 2629 words
categories   : 27% rec.motorcycles, 23% rec.autos, 13% misc.forsale
words       : bike, car, dod, engine, motorcycle, ride, honda, bikes, helmet, bmw

bicluster 4 : 200 documents, 1167 words
categories   : 81% talk.politics.mideast, 10% alt.atheism, 8% soc.religion.christian
words       : turkish, armenia, armenian, armenians, turks, petch, sera, zuma, argic,
               ↵ gvg47
```

```
from collections import defaultdict
import operator
from time import time

import numpy as np

from sklearn.cluster.bicluster import SpectralCoclustering
from sklearn.cluster import MiniBatchKMeans
from sklearn.datasets.twenty_newsgroups import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.cluster import v_measure_score

print(__doc__)

def number_normalizer(tokens):
    """ Map all numeric tokens to a placeholder.

    For many applications, tokens that begin with a number are not directly
    useful, but the fact that such a token exists can be relevant. By applying
    this form of dimensionality reduction, some methods may perform better.
    """
    return ("#NUMBER" if token[0].isdigit() else token for token in tokens)

class NumberNormalizingVectorizer(TfidfVectorizer):
```

```

def build_tokenizer(self):
    tokenize = super().build_tokenizer()
    return lambda doc: list(number_normalizer(tokenize(doc)))

# exclude 'comp.os.ms-windows.misc'
categories = ['alt.atheism', 'comp.graphics',
              'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware',
              'comp.windows.x', 'misc.forsale', 'rec.autos',
              'rec.motorcycles', 'rec.sport.baseball',
              'rec.sport.hockey', 'sci.crypt', 'sci.electronics',
              'sci.med', 'sci.space', 'soc.religion.christian',
              'talk.politics.guns', 'talk.politics.mideast',
              'talk.politics.misc', 'talk.religion.misc']
newsgroups = fetch_20newsgroups(categories=categories)
y_true = newsgroups.target

vectorizer = NumberNormalizingVectorizer(stop_words='english', min_df=5)
cocluster = SpectralCoclustering(n_clusters=len(categories),
                                  svd_method='arpack', random_state=0)
kmeans = MiniBatchKMeans(n_clusters=len(categories), batch_size=20000,
                         random_state=0)

print("Vectorizing...")
X = vectorizer.fit_transform(newsgroups.data)

print("Coclustering...")
start_time = time()
cocluster.fit(X)
y_cocluster = cocluster.row_labels_
print("Done in {:.2f}s. V-measure: {:.4f}".format(
    time() - start_time,
    v_measure_score(y_cocluster, y_true)))

print("MiniBatchKMeans...")
start_time = time()
y_kmeans = kmeans.fit_predict(X)
print("Done in {:.2f}s. V-measure: {:.4f}".format(
    time() - start_time,
    v_measure_score(y_kmeans, y_true)))

feature_names = vectorizer.get_feature_names()
document_names = list(newsgroups.target_names[i] for i in newsgroups.target)

def bicluster_ncut(i):
    rows, cols = cocluster.get_indices(i)
    if not (np.any(rows) and np.any(cols)):
        import sys
        return sys.float_info.max
    row_complement = np.nonzero(np.logical_not(cocluster.rows_[i]))[0]
    col_complement = np.nonzero(np.logical_not(cocluster.columns_[i]))[0]
    # Note: the following is identical to X[rows[:, np.newaxis],
    # cols].sum() but much faster in scipy <= 0.16
    weight = X[rows][:, cols].sum()
    cut = (X[row_complement][:, cols].sum() +
           X[rows][:, col_complement].sum())
    return cut / weight

```

```

def most_common(d):
    """Items of a defaultdict(int) with the highest values.

    Like Counter.most_common in Python >=2.7.
    """
    return sorted(d.items(), key=operator.itemgetter(1), reverse=True)

bicluster_ncuts = list(bicluster_ncut(i)
                      for i in range(len(newsgroups.target_names)))
best_idx = np.argsort(bicluster_ncuts)[:5]

print()
print("Best biclusters:")
print("-----")
for idx, cluster in enumerate(best_idx):
    n_rows, n_cols = cocluster.get_shape(cluster)
    cluster_docs, cluster_words = cocluster.get_indices(cluster)
    if not len(cluster_docs) or not len(cluster_words):
        continue

    # categories
    counter = defaultdict(int)
    for i in cluster_docs:
        counter[document_names[i]] += 1
    cat_string = ", ".join("{:.0f}% {}".format(float(c) / n_rows * 100, name)
                           for name, c in most_common(counter)[:3])

    # words
    out_of_cluster_docs = cocluster.row_labels_ != cluster
    out_of_cluster_docs = np.where(out_of_cluster_docs)[0]
    word_col = X[:, cluster_words]
    word_scores = np.array(word_col[cluster_docs, :].sum(axis=0) -
                           word_col[out_of_cluster_docs, :].sum(axis=0))
    word_scores = word_scores.ravel()
    important_words = list(feature_names[cluster_words[i]])
    for i in word_scores.argsort()[:-11:-1])

    print("bicluster {} : {} documents, {} words".format(
        idx, n_rows, n_cols))
    print("categories : {}".format(cat_string))
    print("words : {}\n".format(','.join(important_words)))

```

Total running time of the script: (0 minutes 13.291 seconds)

5.4 Calibration

Examples illustrating the calibration of predicted probabilities of classifiers.

Note: Click [here](#) to download the full example code

5.4.1 Comparison of Calibration of Classifiers

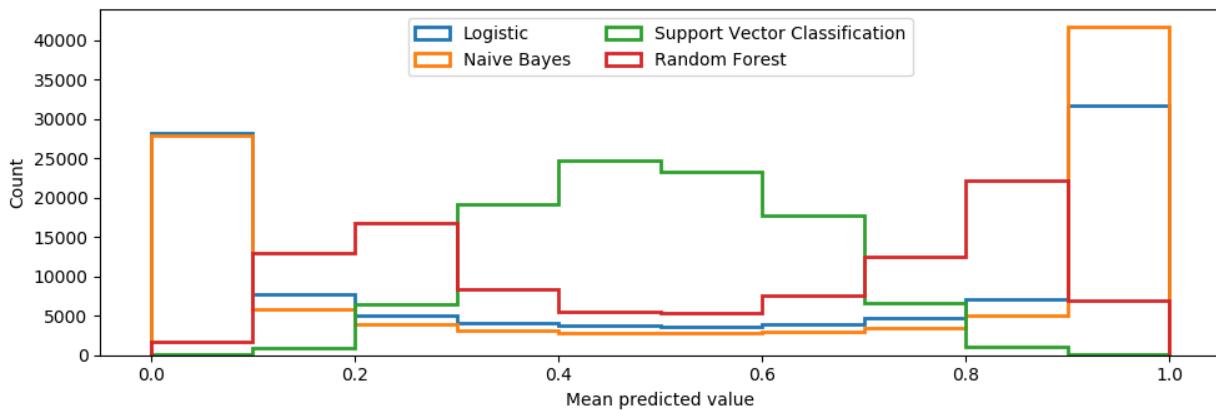
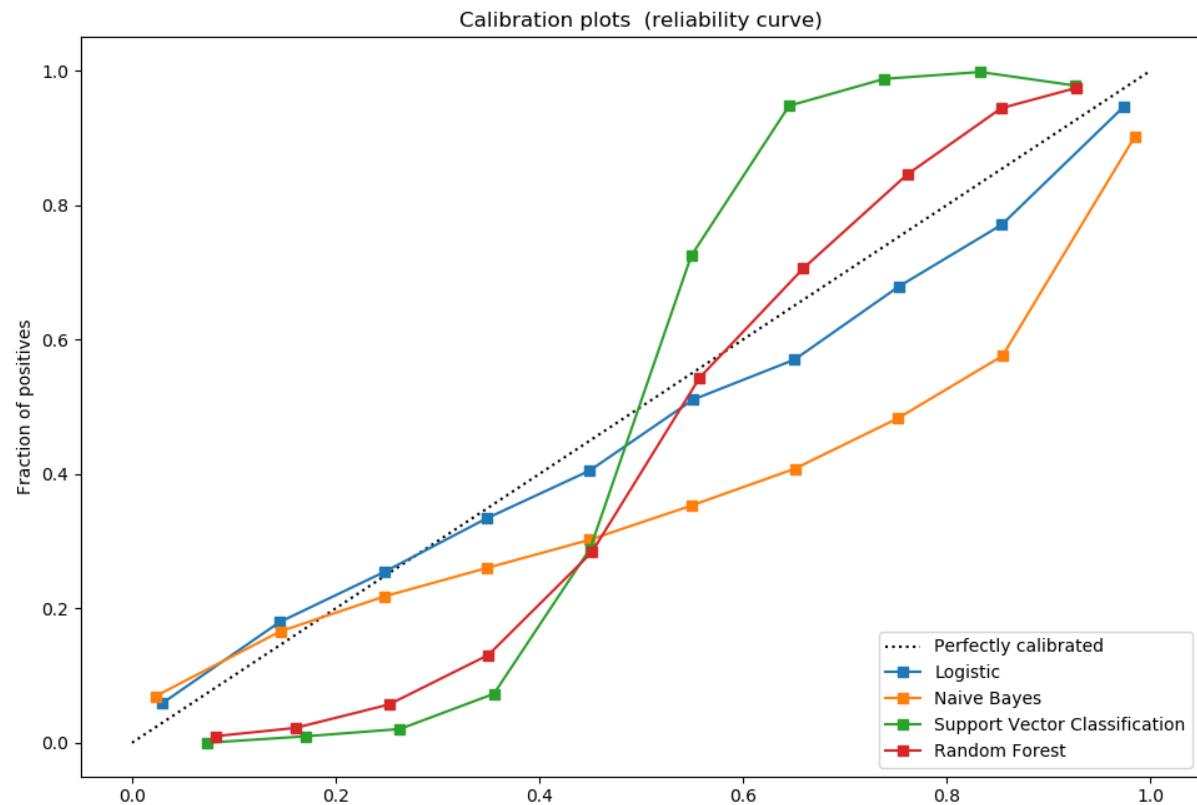
Well calibrated classifiers are probabilistic classifiers for which the output of the predict_proba method can be directly interpreted as a confidence level. For instance a well calibrated (binary) classifier should classify the samples such that among the samples to which it gave a predict_proba value close to 0.8, approx. 80% actually belong to the positive class.

LogisticRegression returns well calibrated predictions as it directly optimizes log-loss. In contrast, the other methods return biased probabilities, with different biases per method:

- GaussianNaiveBayes tends to push probabilities to 0 or 1 (note the counts in the histograms). This is mainly because it makes the assumption that features are conditionally independent given the class, which is not the case in this dataset which contains 2 redundant features.
- RandomForestClassifier shows the opposite behavior: the histograms show peaks at approx. 0.2 and 0.9 probability, while probabilities close to 0 or 1 are very rare. An explanation for this is given by Niculescu-Mizil and Caruana¹: “Methods such as bagging and random forests that average predictions from a base set of models can have difficulty making predictions near 0 and 1 because variance in the underlying base models will bias predictions that should be near zero or one away from these values. Because predictions are restricted to the interval [0,1], errors caused by variance tend to be one-sided near zero and one. For example, if a model should predict $p = 0$ for a case, the only way bagging can achieve this is if all bagged trees predict zero. If we add noise to the trees that bagging is averaging over, this noise will cause some trees to predict values larger than 0 for this case, thus moving the average prediction of the bagged ensemble away from 0. We observe this effect most strongly with random forests because the base-level trees trained with random forests have relatively high variance due to feature subsetting.” As a result, the calibration curve shows a characteristic sigmoid shape, indicating that the classifier could trust its “intuition” more and return probabilities closer to 0 or 1 typically.
- Support Vector Classification (SVC) shows an even more sigmoid curve as the RandomForestClassifier, which is typical for maximum-margin methods (compare Niculescu-Mizil and Caruana¹), which focus on hard samples that are close to the decision boundary (the support vectors).

References:

¹ Predicting Good Probabilities with Supervised Learning, A. Niculescu-Mizil & R. Caruana, ICML 2005



```

print(__doc__)

# Author: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
# License: BSD Style.

import numpy as np
np.random.seed(0)

import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import LinearSVC

```

```

from sklearn.calibration import calibration_curve

X, y = datasets.make_classification(n_samples=100000, n_features=20,
                                    n_informative=2, n_redundant=2)

train_samples = 100 # Samples used for training the models

X_train = X[:train_samples]
X_test = X[train_samples:]
y_train = y[:train_samples]
y_test = y[train_samples:]

# Create classifiers
lr = LogisticRegression(solver='lbfgs')
gnb = GaussianNB()
svc = LinearSVC(C=1.0)
rfc = RandomForestClassifier(n_estimators=100)

# ##########
# Plot calibration plots

plt.figure(figsize=(10, 10))
ax1 = plt.subplot2grid((3, 1), (0, 0), rowspan=2)
ax2 = plt.subplot2grid((3, 1), (2, 0))

ax1.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated")
for clf, name in [(lr, 'Logistic'),
                   (gnb, 'Naive Bayes'),
                   (svc, 'Support Vector Classification'),
                   (rfc, 'Random Forest')]:
    clf.fit(X_train, y_train)
    if hasattr(clf, "predict_proba"):
        prob_pos = clf.predict_proba(X_test)[:, 1]
    else: # use decision function
        prob_pos = clf.decision_function(X_test)
        prob_pos = \
            (prob_pos - prob_pos.min()) / (prob_pos.max() - prob_pos.min())
    fraction_of_positives, mean_predicted_value = \
        calibration_curve(y_test, prob_pos, n_bins=10)

    ax1.plot(mean_predicted_value, fraction_of_positives, "s-",
             label="%s" % (name, ))
    ax2.hist(prob_pos, range=(0, 1), bins=10, label=name,
             histtype="step", lw=2)

ax1.set_ylabel("Fraction of positives")
ax1.set_ylim([-0.05, 1.05])
ax1.legend(loc="lower right")
ax1.set_title('Calibration plots (reliability curve)')

ax2.set_xlabel("Mean predicted value")
ax2.set_ylabel("Count")
ax2.legend(loc="upper center", ncol=2)

plt.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 1.028 seconds)

Note: Click [here](#) to download the full example code

5.4.2 Probability Calibration curves

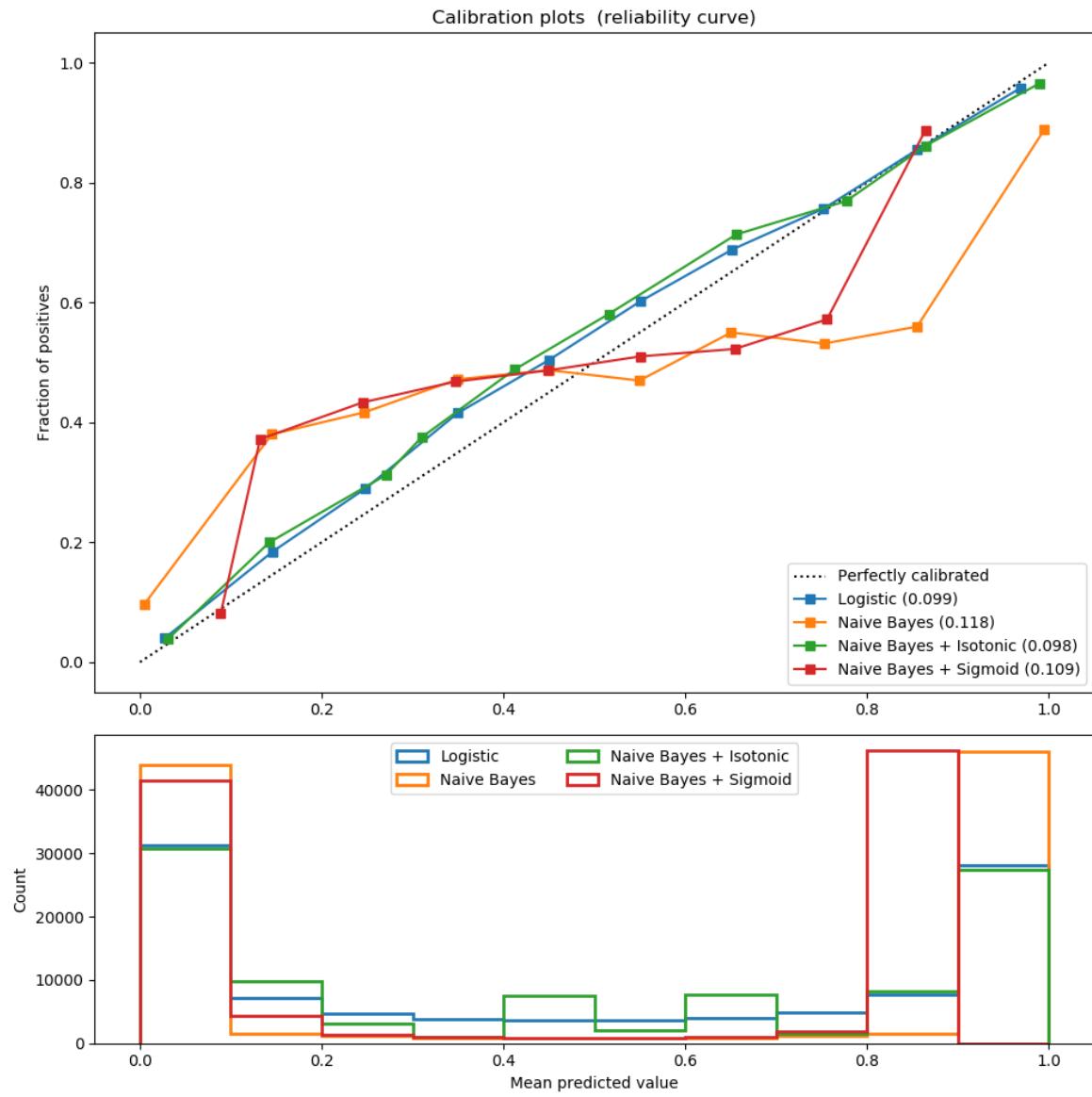
When performing classification one often wants to predict not only the class label, but also the associated probability. This probability gives some kind of confidence on the prediction. This example demonstrates how to display how well calibrated the predicted probabilities are and how to calibrate an uncalibrated classifier.

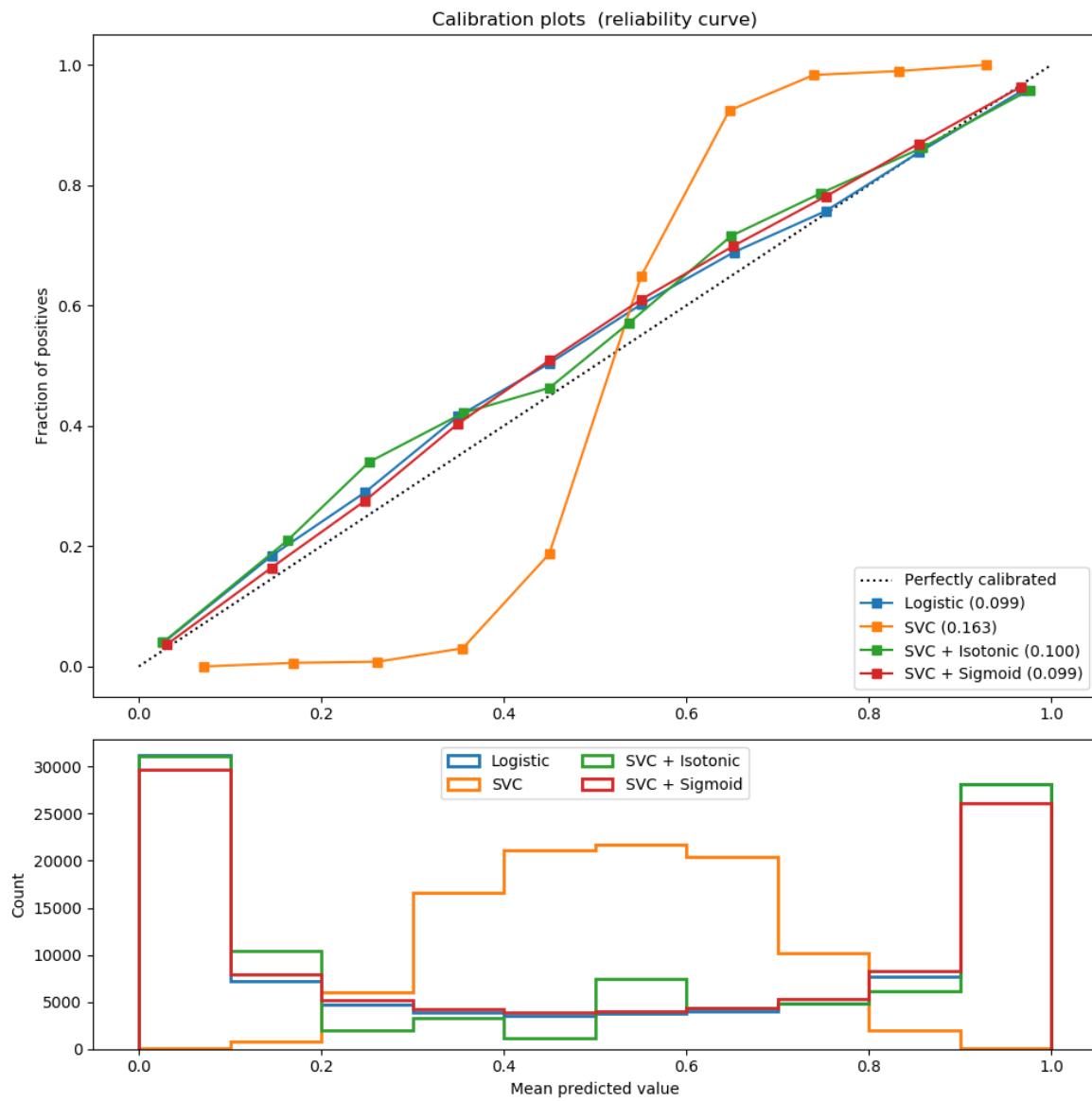
The experiment is performed on an artificial dataset for binary classification with 100,000 samples (1,000 of them are used for model fitting) with 20 features. Of the 20 features, only 2 are informative and 10 are redundant. The first figure shows the estimated probabilities obtained with logistic regression, Gaussian naive Bayes, and Gaussian naive Bayes with both isotonic calibration and sigmoid calibration. The calibration performance is evaluated with Brier score, reported in the legend (the smaller the better). One can observe here that logistic regression is well calibrated while raw Gaussian naive Bayes performs very badly. This is because of the redundant features which violate the assumption of feature-independence and result in an overly confident classifier, which is indicated by the typical transposed-sigmoid curve.

Calibration of the probabilities of Gaussian naive Bayes with isotonic regression can fix this issue as can be seen from the nearly diagonal calibration curve. Sigmoid calibration also improves the brier score slightly, albeit not as strongly as the non-parametric isotonic regression. This can be attributed to the fact that we have plenty of calibration data such that the greater flexibility of the non-parametric model can be exploited.

The second figure shows the calibration curve of a linear support-vector classifier (LinearSVC). LinearSVC shows the opposite behavior as Gaussian naive Bayes: the calibration curve has a sigmoid curve, which is typical for an under-confident classifier. In the case of LinearSVC, this is caused by the margin property of the hinge loss, which lets the model focus on hard samples that are close to the decision boundary (the support vectors).

Both kinds of calibration can fix this issue and yield nearly identical results. This shows that sigmoid calibration can deal with situations where the calibration curve of the base classifier is sigmoid (e.g., for LinearSVC) but not where it is transposed-sigmoid (e.g., Gaussian naive Bayes).





Out:

```
Logistic:
    Brier: 0.099
    Precision: 0.872
    Recall: 0.851
    F1: 0.862
```

```
Naive Bayes:
    Brier: 0.118
    Precision: 0.857
    Recall: 0.876
    F1: 0.867
```

```
Naive Bayes + Isotonic:
    Brier: 0.098
    Precision: 0.883
```

```
Recall: 0.836
F1: 0.859

Naive Bayes + Sigmoid:
    Brier: 0.109
    Precision: 0.861
    Recall: 0.871
    F1: 0.866

Logistic:
    Brier: 0.099
    Precision: 0.872
    Recall: 0.851
    F1: 0.862

SVC:
    Brier: 0.163
    Precision: 0.872
    Recall: 0.852
    F1: 0.862

SVC + Isotonic:
    Brier: 0.100
    Precision: 0.853
    Recall: 0.878
    F1: 0.865

SVC + Sigmoid:
    Brier: 0.099
    Precision: 0.874
    Recall: 0.849
    F1: 0.861
```

```
print(__doc__)

# Author: Alexandre Gramfort <alexandre.gramfort@telecom-paristech.fr>
#         Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
# License: BSD Style.

import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import LinearSVC
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (brier_score_loss, precision_score, recall_score,
                             f1_score)
from sklearn.calibration import CalibratedClassifierCV, calibration_curve
from sklearn.model_selection import train_test_split

# Create dataset of classification task with many redundant and few
# informative features
```

```

x, y = datasets.make_classification(n_samples=100000, n_features=20,
                                    n_informative=2, n_redundant=10,
                                    random_state=42)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.99,
                                                    random_state=42)

def plot_calibration_curve(est, name, fig_index):
    """Plot calibration curve for est w/o and with calibration. """
    # Calibrated with isotonic calibration
    isotonic = CalibratedClassifierCV(est, cv=2, method='isotonic')

    # Calibrated with sigmoid calibration
    sigmoid = CalibratedClassifierCV(est, cv=2, method='sigmoid')

    # Logistic regression with no calibration as baseline
    lr = LogisticRegression(C=1., solver='lbfgs')

    fig = plt.figure(fig_index, figsize=(10, 10))
    ax1 = plt.subplot2grid((3, 1), (0, 0), rowspan=2)
    ax2 = plt.subplot2grid((3, 1), (2, 0))

    ax1.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated")
    for clf, name in [(lr, 'Logistic'),
                       (est, name),
                       (isotonic, name + ' + Isotonic'),
                       (sigmoid, name + ' + Sigmoid')]:
        clf.fit(X_train, y_train)
        y_pred = clf.predict(X_test)
        if hasattr(clf, "predict_proba"):
            prob_pos = clf.predict_proba(X_test)[:, 1]
        else: # use decision function
            prob_pos = clf.decision_function(X_test)
            prob_pos = \
                (prob_pos - prob_pos.min()) / (prob_pos.max() - prob_pos.min())

        clf_score = brier_score_loss(y_test, prob_pos, pos_label=y.max())
        print("%s:\n\tBrier: %.3f\n\tPrecision: %.3f\n\tRecall: %.3f\n\tF1: %.3f\n" % (name, clf_score))
        print("\tPrecision: %.3f\n\tRecall: %.3f\n\tF1: %.3f\n" % precision_score(y_test, y_pred),
              recall_score(y_test, y_pred),
              f1_score(y_test, y_pred))

        fraction_of_positives, mean_predicted_value = \
            calibration_curve(y_test, prob_pos, n_bins=10)

        ax1.plot(mean_predicted_value, fraction_of_positives, "s-",
                 label="%s (%.3f)" % (name, clf_score))

        ax2.hist(prob_pos, range=(0, 1), bins=10, label=name,
                 histtype="step", lw=2)

    ax1.set_ylabel("Fraction of positives")
    ax1.set_ylim([-0.05, 1.05])
    ax1.legend(loc="lower right")
    ax1.set_title('Calibration plots (reliability curve)')

```

```
ax2.set_xlabel("Mean predicted value")
ax2.set_ylabel("Count")
ax2.legend(loc="upper center", ncol=2)

plt.tight_layout()

# Plot calibration curve for Gaussian Naive Bayes
plot_calibration_curve(GaussianNB(), "Naive Bayes", 1)

# Plot calibration curve for Linear SVC
plot_calibration_curve(LinearSVC(max_iter=10000), "SVC", 2)

plt.show()
```

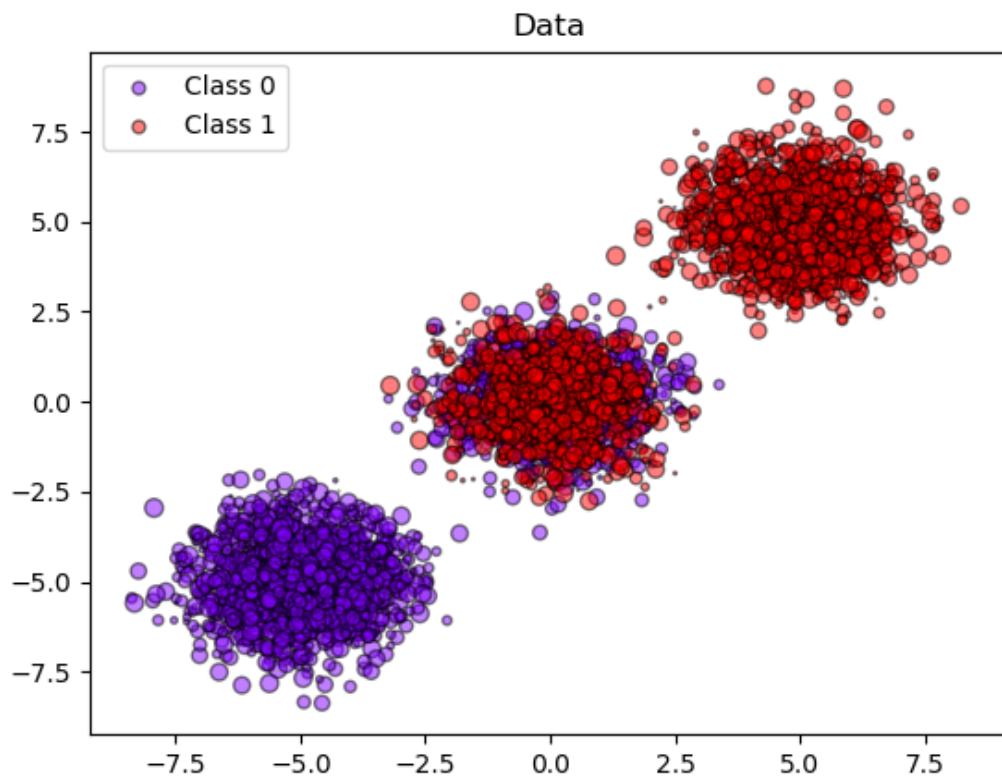
Total running time of the script: (0 minutes 1.993 seconds)

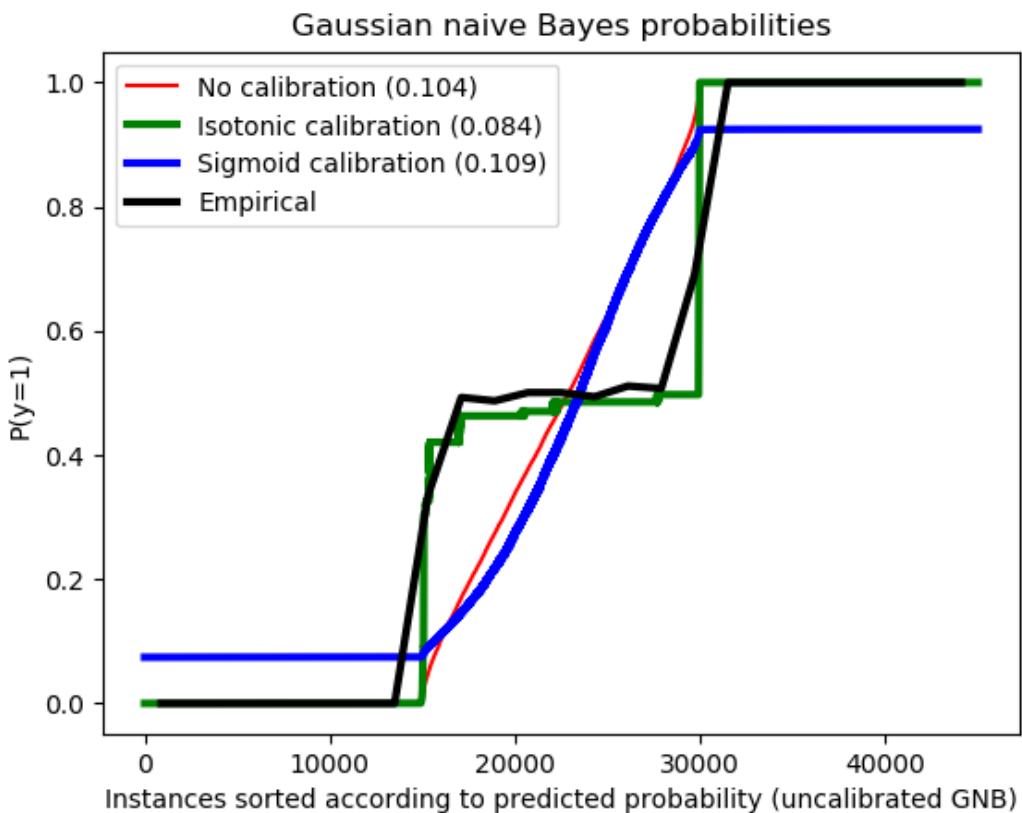
Note: Click [here](#) to download the full example code

5.4.3 Probability calibration of classifiers

When performing classification you often want to predict not only the class label, but also the associated probability. This probability gives you some kind of confidence on the prediction. However, not all classifiers provide well-calibrated probabilities, some being over-confident while others being under-confident. Thus, a separate calibration of predicted probabilities is often desirable as a postprocessing. This example illustrates two different methods for this calibration and evaluates the quality of the returned probabilities using Brier's score (see https://en.wikipedia.org/wiki/Brier_score).

Compared are the estimated probability using a Gaussian naive Bayes classifier without calibration, with a sigmoid calibration, and with a non-parametric isotonic calibration. One can observe that only the non-parametric model is able to provide a probability calibration that returns probabilities close to the expected 0.5 for most of the samples belonging to the middle cluster with heterogeneous labels. This results in a significantly improved Brier score.





Out:

```
Brier scores: (the smaller the better)
No calibration: 0.104
With isotonic calibration: 0.084
With sigmoid calibration: 0.109
```

```
print(__doc__)

# Author: Mathieu Blondel <mathieu@mblondel.org>
#         Alexandre Gramfort <alexandre.gramfort@telecom-paristech.fr>
#         Balazs Kegl <balazs.kegl@gmail.com>
#         Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
# License: BSD Style.

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm

from sklearn.datasets import make_blobs
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import brier_score_loss
```

```

from sklearn.calibration import CalibratedClassifierCV
from sklearn.model_selection import train_test_split

n_samples = 50000
n_bins = 3 # use 3 bins for calibration_curve as we have 3 clusters here

# Generate 3 blobs with 2 classes where the second blob contains
# half positive samples and half negative samples. Probability in this
# blob is therefore 0.5.
centers = [(-5, -5), (0, 0), (5, 5)]
X, y = make_blobs(n_samples=n_samples, n_features=2, cluster_std=1.0,
                  centers=centers, shuffle=False, random_state=42)

y[:n_samples // 2] = 0
y[n_samples // 2:] = 1
sample_weight = np.random.RandomState(42).rand(y.shape[0])

# split train, test for calibration
X_train, X_test, y_train, y_test, sw_train, sw_test = \
    train_test_split(X, y, sample_weight, test_size=0.9, random_state=42)

# Gaussian Naive-Bayes with no calibration
clf = GaussianNB()
clf.fit(X_train, y_train) # GaussianNB itself does not support sample-weights
prob_pos_clf = clf.predict_proba(X_test)[:, 1]

# Gaussian Naive-Bayes with isotonic calibration
clf_isotonic = CalibratedClassifierCV(clf, cv=2, method='isotonic')
clf_isotonic.fit(X_train, y_train, sw_train)
prob_pos_isotonic = clf_isotonic.predict_proba(X_test)[:, 1]

# Gaussian Naive-Bayes with sigmoid calibration
clf_sigmoid = CalibratedClassifierCV(clf, cv=2, method='sigmoid')
clf_sigmoid.fit(X_train, y_train, sw_train)
prob_pos_sigmoid = clf_sigmoid.predict_proba(X_test)[:, 1]

print("Brier scores: (the smaller the better)")

clf_score = brier_score_loss(y_test, prob_pos_clf, sw_test)
print("No calibration: %1.3f" % clf_score)

clf_isotonic_score = brier_score_loss(y_test, prob_pos_isotonic, sw_test)
print("With isotonic calibration: %1.3f" % clf_isotonic_score)

clf_sigmoid_score = brier_score_loss(y_test, prob_pos_sigmoid, sw_test)
print("With sigmoid calibration: %1.3f" % clf_sigmoid_score)

# ##########
# Plot the data and the predicted probabilities
plt.figure()
y_unique = np.unique(y)
colors = cm.rainbow(np.linspace(0.0, 1.0, y_unique.size))
for this_y, color in zip(y_unique, colors):
    this_X = X_train[y_train == this_y]
    this_sw = sw_train[y_train == this_y]
    plt.scatter(this_X[:, 0], this_X[:, 1], s=this_sw * 50,
                c=color[newaxis, :],

```

```
alpha=0.5, edgecolor='k',
label="Class %s" % this_y)
plt.legend(loc="best")
plt.title("Data")

plt.figure()
order = np.lexsort((prob_pos_clf, ))
plt.plot(prob_pos_clf[order], 'r', label='No calibration (%1.3f)' % clf_score)
plt.plot(prob_pos_isotonic[order], 'g', linewidth=3,
         label='Isotonic calibration (%1.3f)' % clf_isotonic_score)
plt.plot(prob_pos_sigmoid[order], 'b', linewidth=3,
         label='Sigmoid calibration (%1.3f)' % clf_sigmoid_score)
plt.plot(np.linspace(0, y_test.size, 51)[1::2],
         y_test[order].reshape(25, -1).mean(1),
         'k', linewidth=3, label=r'Empirical')
plt.ylim([-0.05, 1.05])
plt.xlabel("Instances sorted according to predicted probability "
           "(uncalibrated GNB)")
plt.ylabel("P(y=1)")
plt.legend(loc="upper left")
plt.title("Gaussian naive Bayes probabilities")

plt.show()
```

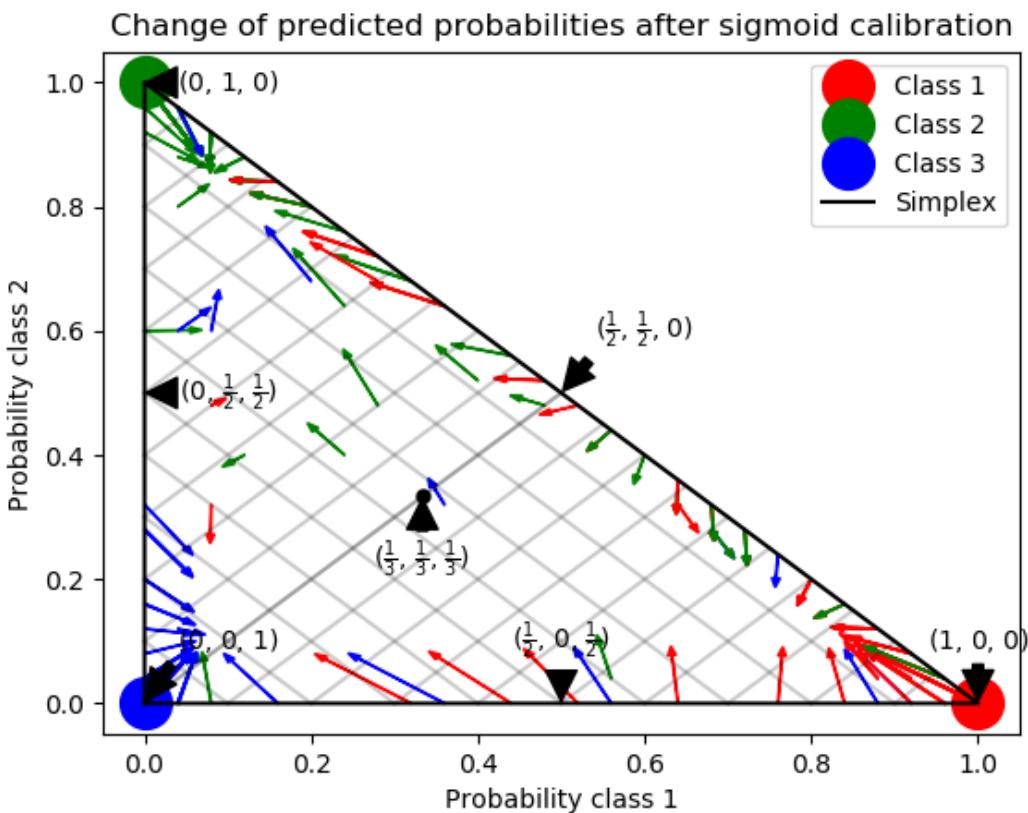
Total running time of the script: (0 minutes 0.108 seconds)

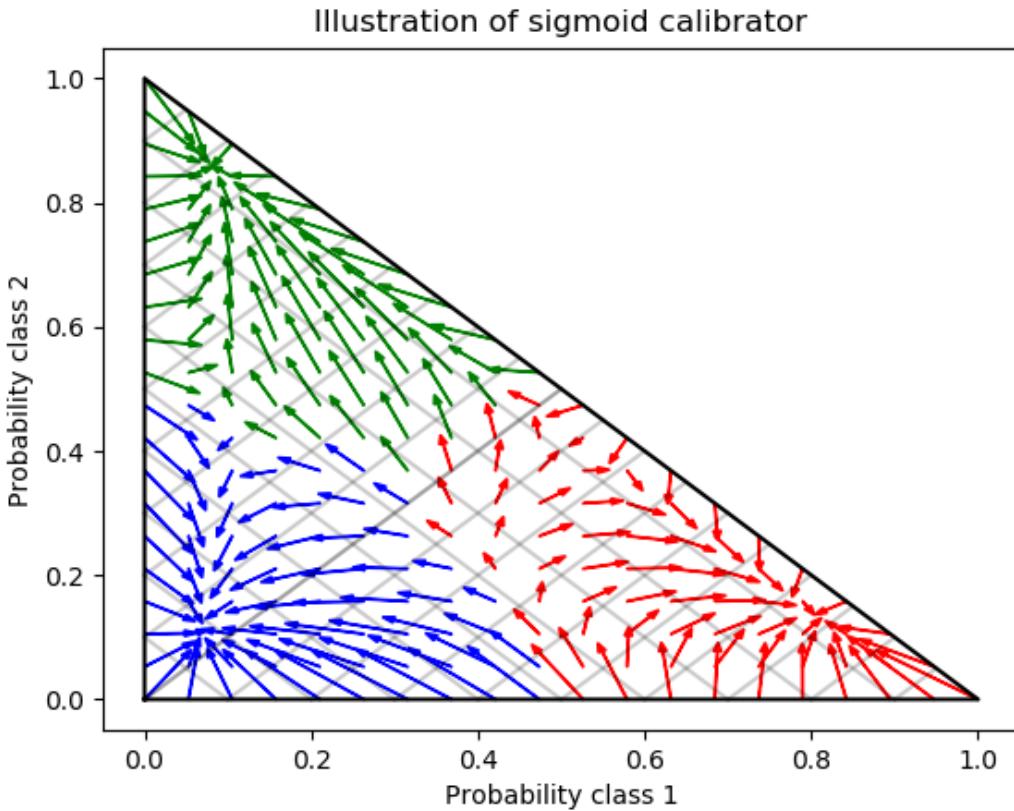
Note: Click [here](#) to download the full example code

5.4.4 Probability Calibration for 3-class classification

This example illustrates how sigmoid calibration changes predicted probabilities for a 3-class classification problem. Illustrated is the standard 2-simplex, where the three corners correspond to the three classes. Arrows point from the probability vectors predicted by an uncalibrated classifier to the probability vectors predicted by the same classifier after sigmoid calibration on a hold-out validation set. Colors indicate the true class of an instance (red: class 1, green: class 2, blue: class 3).

The base classifier is a random forest classifier with 25 base estimators (trees). If this classifier is trained on all 800 training datapoints, it is overly confident in its predictions and thus incurs a large log-loss. Calibrating an identical classifier, which was trained on 600 datapoints, with method='sigmoid' on the remaining 200 datapoints reduces the confidence of the predictions, i.e., moves the probability vectors from the edges of the simplex towards the center. This calibration results in a lower log-loss. Note that an alternative would have been to increase the number of base estimators which would have resulted in a similar decrease in log-loss.





Out:

```
Log-loss of
* uncalibrated classifier trained on 800 datapoints: 1.280
* classifier trained on 600 datapoints and calibrated on 200 datapoint: 0.534
```

```
print(__doc__)

# Author: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
# License: BSD Style.

import matplotlib.pyplot as plt

import numpy as np

from sklearn.datasets import make_blobs
from sklearn.ensemble import RandomForestClassifier
from sklearn.calibration import CalibratedClassifierCV
from sklearn.metrics import log_loss

np.random.seed(0)
```

```

# Generate data
X, y = make_blobs(n_samples=1000, n_features=2, random_state=42,
                   cluster_std=5.0)
X_train, y_train = X[:600], y[:600]
X_valid, y_valid = X[600:800], y[600:800]
X_train_valid, y_train_valid = X[:800], y[:800]
X_test, y_test = X[800:], y[800:]

# Train uncalibrated random forest classifier on whole train and validation
# data and evaluate on test data
clf = RandomForestClassifier(n_estimators=25)
clf.fit(X_train_valid, y_train_valid)
clf_probs = clf.predict_proba(X_test)
score = log_loss(y_test, clf_probs)

# Train random forest classifier, calibrate on validation data and evaluate
# on test data
clf = RandomForestClassifier(n_estimators=25)
clf.fit(X_train, y_train)
clf_probs = clf.predict_proba(X_test)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid", cv="prefit")
sig_clf.fit(X_valid, y_valid)
sig_clf_probs = sig_clf.predict_proba(X_test)
sig_score = log_loss(y_test, sig_clf_probs)

# Plot changes in predicted probabilities via arrows
plt.figure()
colors = ["r", "g", "b"]
for i in range(clf_probs.shape[0]):
    plt.arrow(clf_probs[i, 0], clf_probs[i, 1],
              sig_clf_probs[i, 0] - clf_probs[i, 0],
              sig_clf_probs[i, 1] - clf_probs[i, 1],
              color=colors[y_test[i]], head_width=1e-2)

# Plot perfect predictions
plt.plot([1.0], [0.0], 'ro', ms=20, label="Class 1")
plt.plot([0.0], [1.0], 'go', ms=20, label="Class 2")
plt.plot([0.0], [0.0], 'bo', ms=20, label="Class 3")

# Plot boundaries of unit simplex
plt.plot([0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0], 'k', label="Simplex")

# Annotate points on the simplex
plt.annotate(r'($\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$)', xy=(1.0/3, 1.0/3), xytext=(1.0/3, .23), xycoords='data',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='center', verticalalignment='center')
plt.plot([1.0/3], [1.0/3], 'ko', ms=5)
plt.annotate(r'($\frac{1}{2}, 0, \frac{1}{2}$)', xy=(.5, 0), xytext=(.5, .1), xycoords='data',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='center', verticalalignment='center')
plt.annotate(r'($0, \frac{1}{2}, \frac{1}{2}$)', xy=(0, .5), xytext=(-.1, .5), xycoords='data',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='center', verticalalignment='center')
plt.annotate(r'($\frac{1}{2}, \frac{1}{2}, 0$)', xy=(.5, .5), xytext=(.5, .5), xycoords='data',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='center', verticalalignment='center')

```

```

        xy=(.5, .5), xytext=(.6, .6), xycoords='data',
        arrowprops=dict(facecolor='black', shrink=0.05),
        horizontalalignment='center', verticalalignment='center')
plt.annotate(r'($0$, $0$, $1$)',
            xy=(0, 0), xytext=(.1, .1), xycoords='data',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='center', verticalalignment='center')
plt.annotate(r'($1$, $0$, $0$)',
            xy=(1, 0), xytext=(.1, .1), xycoords='data',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='center', verticalalignment='center')
plt.annotate(r'($1$, $1$, $0$)',
            xy=(0, 1), xytext=(.1, .1), xycoords='data',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='center', verticalalignment='center')
plt.annotate(r'($0$, $1$, $0$)',
            xy=(0, 1), xytext=(.1, .1), xycoords='data',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='center', verticalalignment='center')

# Add grid
plt.grid(False)
for x in [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]:
    plt.plot([0, x], [x, 0], 'k', alpha=0.2)
    plt.plot([0, 0 + (1-x)/2], [x, x + (1-x)/2], 'k', alpha=0.2)
    plt.plot([x, x + (1-x)/2], [0, 0 + (1-x)/2], 'k', alpha=0.2)

plt.title("Change of predicted probabilities after sigmoid calibration")
plt.xlabel("Probability class 1")
plt.ylabel("Probability class 2")
plt.xlim(-0.05, 1.05)
plt.ylim(-0.05, 1.05)
plt.legend(loc="best")

print("Log-loss of")
print(" * uncalibrated classifier trained on 800 datapoints: %.3f "
      % score)
print(" * classifier trained on 600 datapoints and calibrated on "
      "200 datapoint: %.3f" % sig_score)

# Illustrate calibrator
plt.figure()
# generate grid over 2-simplex
p1d = np.linspace(0, 1, 20)
p0, p1 = np.meshgrid(p1d, p1d)
p2 = 1 - p0 - p1
p = np.c_[p0.ravel(), p1.ravel(), p2.ravel()]
p = p[p[:, 2] >= 0]

calibrated_classifier = sig_clf.calibrated_classifiers_[0]
prediction = np.vstack([calibrator.predict(this_p)
                        for calibrator, this_p in
                        zip(calibrated_classifier.calibrators_, p.T)]).T
prediction /= prediction.sum(axis=1)[:, None]

# Plot modifications of calibrator
for i in range(prediction.shape[0]):
    plt.arrow(p[i, 0], p[i, 1],
              prediction[i, 0] - p[i, 0], prediction[i, 1] - p[i, 1],
              head_width=1e-2, color=colors[np.argmax(p[i])])
# Plot boundaries of unit simplex
plt.plot([0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0], 'k', label="Simplex")

```

```
plt.grid(False)
for x in [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]:
    plt.plot([0, x], [x, 0], 'k', alpha=0.2)
    plt.plot([0, 0 + (1-x)/2], [x, x + (1-x)/2], 'k', alpha=0.2)
    plt.plot([x, x + (1-x)/2], [0, 0 + (1-x)/2], 'k', alpha=0.2)

plt.title("Illustration of sigmoid calibrator")
plt.xlabel("Probability class 1")
plt.ylabel("Probability class 2")
plt.xlim(-0.05, 1.05)
plt.ylim(-0.05, 1.05)

plt.show()
```

Total running time of the script: (0 minutes 0.318 seconds)

5.5 Classification

General examples about classification algorithms.

Note: Click [here](#) to download the full example code

5.5.1 Recognizing hand-written digits

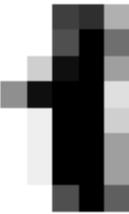
An example showing how the scikit-learn can be used to recognize images of hand-written digits.

This example is commented in the *tutorial section of the user manual*.

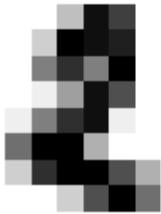
Training: 0



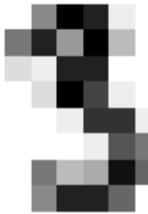
Training: 1



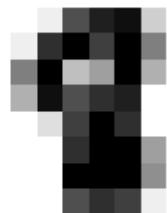
Training: 2



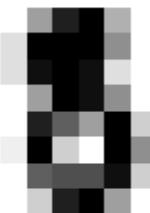
Training: 3



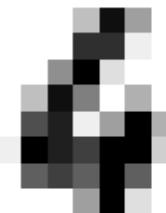
Prediction: 8



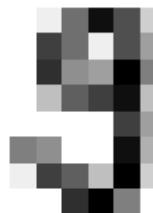
Prediction: 8



Prediction: 4



Prediction: 9



Out:

```
Classification report for classifier SVC(gamma=0.001):  
precision    recall    f1-score   support
```

0	1.00	0.99	0.99	88
1	0.99	0.97	0.98	91
2	0.99	0.99	0.99	86
3	0.98	0.87	0.92	91
4	0.99	0.96	0.97	92
5	0.95	0.97	0.96	91
6	0.99	0.99	0.99	91
7	0.96	0.99	0.97	89
8	0.94	1.00	0.97	88
9	0.93	0.98	0.95	92

```
accuracy           0.97
```

```
macro avg       0.97    0.97    0.97    899  
weighted avg    0.97    0.97    0.97    899
```

```
Confusion matrix:
```

```
[[87  0  0  0  1  0  0  0  0  0]  
 [ 0 88  1  0  0  0  0  0  1  1]  
 [ 0  0 85  1  0  0  0  0  0  0]  
 [ 0  0  0 79  0  3  0  4  5  0]]
```

```
[ 0  0  0  0  88  0  0  0  0  4]
[ 0  0  0  0  88  1  0  0  0  2]
[ 0  1  0  0  0  0  90  0  0  0]
[ 0  0  0  0  0  1  0  88  0  0]
[ 0  0  0  0  0  0  0  0  88  0]
[ 0  0  0  1  0  1  0  0  0  90]]
```

```
print(__doc__)

# Author: Gael Varoquaux <gael dot varoquaux at normalesup dot org>
# License: BSD 3 clause

# Standard scientific Python imports
import matplotlib.pyplot as plt

# Import datasets, classifiers and performance metrics
from sklearn import datasets, svm, metrics

# The digits dataset
digits = datasets.load_digits()

# The data that we are interested in is made of 8x8 images of digits, let's
# have a look at the first 4 images, stored in the `images` attribute of the
# dataset. If we were working from image files, we could load them using
# matplotlib.pyplot.imread. Note that each image must have the same size. For these
# images, we know which digit they represent: it is given in the 'target' of
# the dataset.
images_and_labels = list(zip(digits.images, digits.target))
for index, (image, label) in enumerate(images_and_labels[:4]):
    plt.subplot(2, 4, index + 1)
    plt.axis('off')
    plt.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    plt.title('Training: %i' % label)

# To apply a classifier on this data, we need to flatten the image, to
# turn the data in a (samples, feature) matrix:
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))

# Create a classifier: a support vector classifier
classifier = svm.SVC(gamma=0.001)

# We learn the digits on the first half of the digits
classifier.fit(data[:n_samples // 2], digits.target[:n_samples // 2])

# Now predict the value of the digit on the second half:
expected = digits.target[n_samples // 2:]
predicted = classifier.predict(data[n_samples // 2:])

print("Classification report for classifier %s:\n%s\n"
      % (classifier, metrics.classification_report(expected, predicted)))
print("Confusion matrix:\n%s" % metrics.confusion_matrix(expected, predicted))
```

```

images_and_predictions = list(zip(digits.images[n_samples // 2:], predicted))
for index, (image, prediction) in enumerate(images_and_predictions[:4]):
    plt.subplot(2, 4, index + 5)
    plt.axis('off')
    plt.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    plt.title('Prediction: %i' % prediction)

plt.show()

```

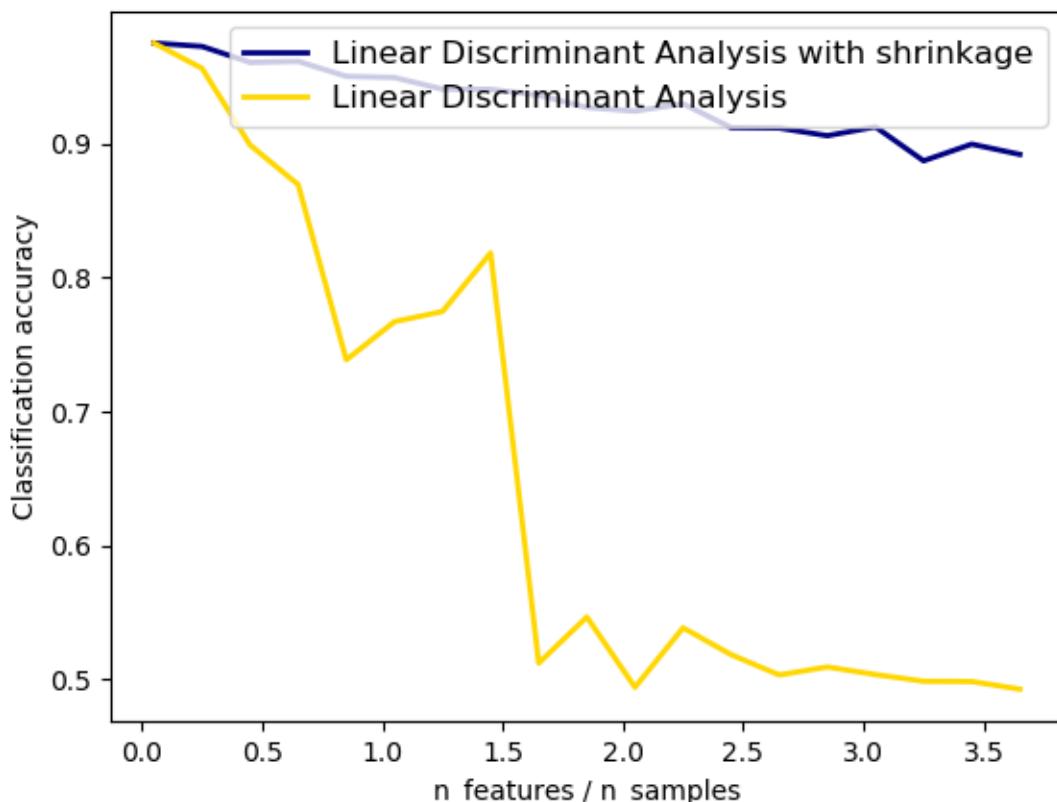
Total running time of the script: (0 minutes 0.237 seconds)

Note: Click [here](#) to download the full example code

5.5.2 Normal and Shrinkage Linear Discriminant Analysis for classification

Shows how shrinkage improves classification.

Linear Discriminant Analysis vs. shrinkage Linear Discriminant Analysis (1 discriminant)



```

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_blobs
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

```

```

n_train = 20 # samples for training
n_test = 200 # samples for testing
n_averages = 50 # how often to repeat classification
n_features_max = 75 # maximum number of features
step = 4 # step size for the calculation

def generate_data(n_samples, n_features):
    """Generate random blob-ish data with noisy features.

    This returns an array of input data with shape `(n_samples, n_features)`
    and an array of `n_samples` target labels.

    Only one feature contains discriminative information, the other features
    contain only noise.
    """
    X, y = make_blobs(n_samples=n_samples, n_features=1, centers=[[2], [2]])

    # add non-discriminative features
    if n_features > 1:
        X = np.hstack([X, np.random.randn(n_samples, n_features - 1)])
    return X, y

acc_clf1, acc_clf2 = [], []
n_features_range = range(1, n_features_max + 1, step)
for n_features in n_features_range:
    score_clf1, score_clf2 = 0, 0
    for _ in range(n_averages):
        X, y = generate_data(n_train, n_features)

        clf1 = LinearDiscriminantAnalysis(solver='lsqr', shrinkage='auto').fit(X, y)
        clf2 = LinearDiscriminantAnalysis(solver='lsqr', shrinkage=None).fit(X, y)

        X, y = generate_data(n_test, n_features)
        score_clf1 += clf1.score(X, y)
        score_clf2 += clf2.score(X, y)

    acc_clf1.append(score_clf1 / n_averages)
    acc_clf2.append(score_clf2 / n_averages)

features_samples_ratio = np.array(n_features_range) / n_train

plt.plot(features_samples_ratio, acc_clf1, linewidth=2,
         label="Linear Discriminant Analysis with shrinkage", color='navy')
plt.plot(features_samples_ratio, acc_clf2, linewidth=2,
         label="Linear Discriminant Analysis", color='gold')

plt.xlabel('n_features / n_samples')
plt.ylabel('Classification accuracy')

plt.legend(loc=1, prop={'size': 12})
plt.suptitle('Linear Discriminant Analysis vs. \
shrinkage Linear Discriminant Analysis (1 discriminative feature)')
plt.show()

```

Total running time of the script: (0 minutes 6.160 seconds)

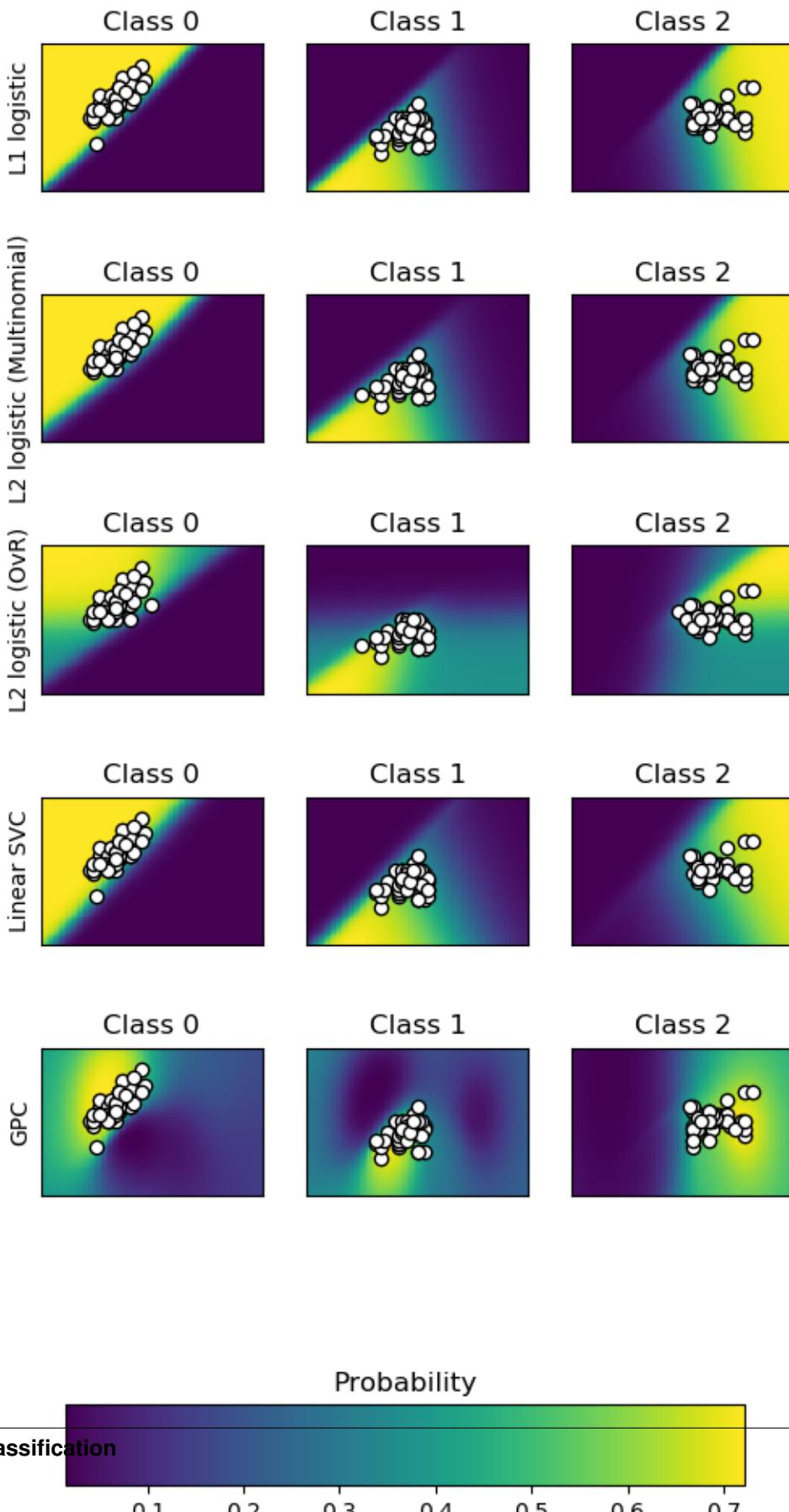
Note: Click [here](#) to download the full example code

5.5.3 Plot classification probability

Plot the classification probability for different classifiers. We use a 3 class dataset, and we classify it with a Support Vector classifier, L1 and L2 penalized logistic regression with either a One-Vs-Rest or multinomial setting, and Gaussian process classification.

Linear SVC is not a probabilistic classifier by default but it has a built-in calibration option enabled in this example (`probability=True`).

The logistic regression with One-Vs-Rest is not a multiclass classifier out of the box. As a result it has more trouble in separating class 2 and 3 than the other estimators.



Out:

```
Accuracy (train) for L1 logistic: 83.3%
Accuracy (train) for L2 logistic (Multinomial): 82.7%
Accuracy (train) for L2 logistic (OvR): 79.3%
Accuracy (train) for Linear SVC: 82.0%
Accuracy (train) for GPC: 82.7%
```

```
print(__doc__)

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause

import matplotlib.pyplot as plt
import numpy as np

from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn import datasets

iris = datasets.load_iris()
X = iris.data[:, 0:2] # we only take the first two features for visualization
y = iris.target

n_features = X.shape[1]

C = 10
kernel = 1.0 * RBF([1.0, 1.0]) # for GPC

# Create different classifiers.
classifiers = {
    'L1 logistic': LogisticRegression(C=C, penalty='l1',
                                       solver='saga',
                                       multi_class='multinomial',
                                       max_iter=10000),
    'L2 logistic (Multinomial)': LogisticRegression(C=C, penalty='l2',
                                                    solver='saga',
                                                    multi_class='multinomial',
                                                    max_iter=10000),
    'L2 logistic (OvR)': LogisticRegression(C=C, penalty='l2',
                                             solver='saga',
                                             multi_class='ovr',
                                             max_iter=10000),
    'Linear SVC': SVC(kernel='linear', C=C, probability=True,
                       random_state=0),
    'GPC': GaussianProcessClassifier(kernel)
}

n_classifiers = len(classifiers)
```

```

plt.figure(figsize=(3 * 2, n_classifiers * 2))
plt.subplots_adjust(bottom=.2, top=.95)

xx = np.linspace(3, 9, 100)
yy = np.linspace(1, 5, 100).T
xx, yy = np.meshgrid(xx, yy)
Xfull = np.c_[xx.ravel(), yy.ravel()]

for index, (name, classifier) in enumerate(classifiers.items()):
    classifier.fit(X, y)

    y_pred = classifier.predict(X)
    accuracy = accuracy_score(y, y_pred)
    print("Accuracy (train) for %s: %0.1f%%" % (name, accuracy * 100))

    # View probabilities:
    probas = classifier.predict_proba(Xfull)
    n_classes = np.unique(y_pred).size
    for k in range(n_classes):
        plt.subplot(n_classifiers, n_classes, index * n_classes + k + 1)
        plt.title("Class %d" % k)
        if k == 0:
            plt.ylabel(name)
        imshow_handle = plt.imshow(probas[:, k].reshape((100, 100)),
                                   extent=(3, 9, 1, 5), origin='lower')
        plt.xticks(())
        plt.yticks(())
        idx = (y_pred == k)
        if idx.any():
            plt.scatter(X[idx, 0], X[idx, 1], marker='o', c='w', edgecolor='k')

ax = plt.axes([0.15, 0.04, 0.7, 0.05])
plt.title("Probability")
plt.colorbar(imshow_handle, cax=ax, orientation='horizontal')

plt.show()

```

Total running time of the script: (0 minutes 1.409 seconds)

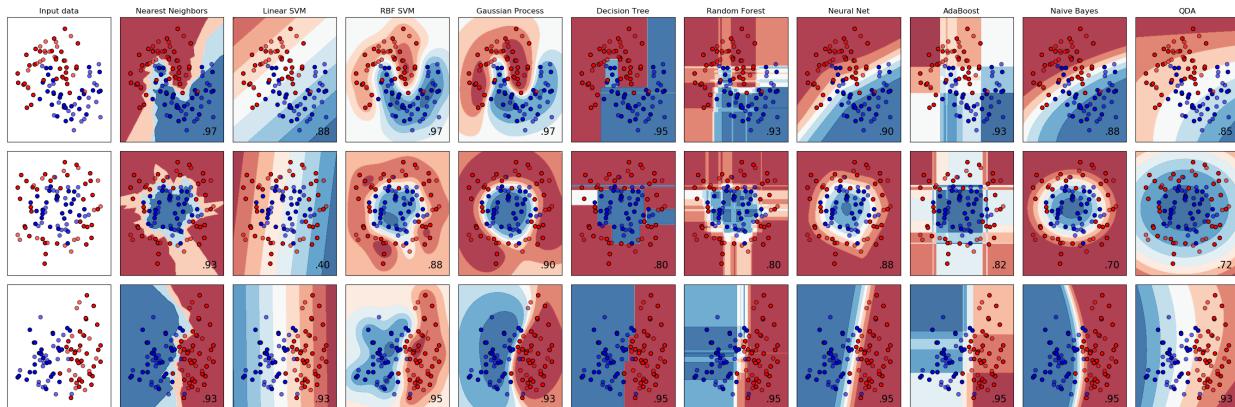
Note: Click [here](#) to download the full example code

5.5.4 Classifier comparison

A comparison of a several classifiers in scikit-learn on synthetic datasets. The point of this example is to illustrate the nature of decision boundaries of different classifiers. This should be taken with a grain of salt, as the intuition conveyed by these examples does not necessarily carry over to real datasets.

Particularly in high-dimensional spaces, data can more easily be separated linearly and the simplicity of classifiers such as naive Bayes and linear SVMs might lead to better generalization than is achieved by other classifiers.

The plots show training points in solid colors and testing points semi-transparent. The lower right shows the classification accuracy on the test set.



```

print(__doc__)

# Code source: Gaël Varoquaux
#              Andreas Müller
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

h = .02 # step size in the mesh

names = ["Nearest Neighbors", "Linear SVM", "RBF SVM", "Gaussian Process",
         "Decision Tree", "Random Forest", "Neural Net", "AdaBoost",
         "Naive Bayes", "QDA"]

classifiers = [
    KNeighborsClassifier(3),
    SVC(kernel="linear", C=0.025),
    SVC(gamma=2, C=1),
    GaussianProcessClassifier(1.0 * RBF(1.0)),
    DecisionTreeClassifier(max_depth=5),
    RandomForestClassifier(max_depth=5, n_estimators=10, max_features=1),
    MLPClassifier(alpha=1, max_iter=1000),
    AdaBoostClassifier(),
    GaussianNB(),
    QuadraticDiscriminantAnalysis()]

X, y = make_classification(n_features=2, n_redundant=0, n_informative=2,
                           random_state=1, shuffle=False)

```

```

        random_state=1, n_clusters_per_class=1)
rng = np.random.RandomState(2)
X += 2 * rng.uniform(size=X.shape)
linearly_separable = (X, y)

datasets = [make_moons(noise=0.3, random_state=0),
            make_circles(noise=0.2, factor=0.5, random_state=1),
            linearly_separable
            ]

figure = plt.figure(figsize=(27, 9))
i = 1
# iterate over datasets
for ds_cnt, ds in enumerate(datasets):
    # preprocess dataset, split into training and test part
    X, y = ds
    X = StandardScaler().fit_transform(X)
    X_train, X_test, y_train, y_test = \
        train_test_split(X, y, test_size=.4, random_state=42)

    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))

    # just plot the dataset first
    cm = plt.cm.RdBu
    cm_bright = ListedColormap(['#FF0000', '#0000FF'])
    ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
    if ds_cnt == 0:
        ax.set_title("Input data")
    # Plot the training points
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
               edgecolors='k')
    # Plot the testing points
    ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, alpha=0.6,
               edgecolors='k')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xticks(())
    ax.set_yticks(())
    i += 1

    # iterate over classifiers
    for name, clf in zip(names, classifiers):
        ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
        clf.fit(X_train, y_train)
        score = clf.score(X_test, y_test)

        # Plot the decision boundary. For that, we will assign a color to each
        # point in the mesh [x_min, x_max]x[y_min, y_max].
        if hasattr(clf, "decision_function"):
            Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
        else:
            Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

        # Put the result into a color plot
        Z = Z.reshape(xx.shape)

```

```
ax.contourf(xx, yy, z, cmap=cm, alpha=.8)

# Plot the training points
ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
           edgecolors='k')
# Plot the testing points
ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright,
           edgecolors='k', alpha=0.6)

ax.set_xlim(xx.min(), xx.max())
ax.set_ylim(yy.min(), yy.max())
ax.set_xticks(())
ax.set_yticks(())
if ds_cnt == 0:
    ax.set_title(name)
    ax.text(xx.max() - .3, yy.min() + .3, ('%.2f' % score).lstrip('0'),
            size=15, horizontalalignment='right')
    i += 1

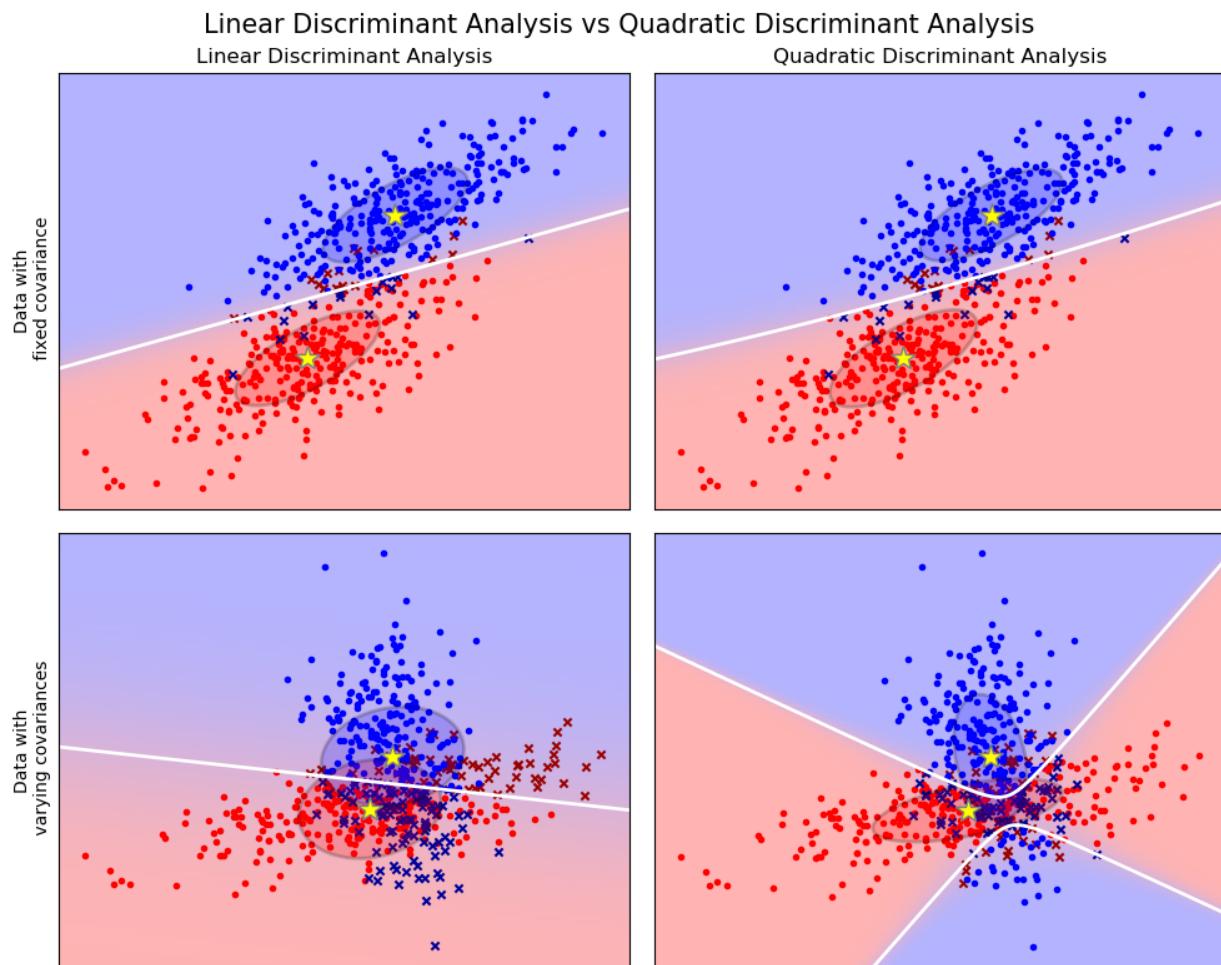
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 5.178 seconds)

Note: Click [here](#) to download the full example code

5.5.5 Linear and Quadratic Discriminant Analysis with covariance ellipsoid

This example plots the covariance ellipsoids of each class and decision boundary learned by LDA and QDA. The ellipsoids display the double standard deviation for each class. With LDA, the standard deviation is the same for all the classes, while each class has its own standard deviation with QDA.



```

print(__doc__)

from scipy import linalg
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib import colors

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

# ##########
# Colormap
cmap = colors.LinearSegmentedColormap(
    'red_blue_classes',
    {'red': [(0, 1, 1), (1, 0.7, 0.7)],
     'green': [(0, 0.7, 0.7), (1, 0.7, 0.7)],
     'blue': [(0, 0.7, 0.7), (1, 1, 1)]})
plt.cm.register_cmap(cmap=cmap)

# #####
# Generate datasets

```

```

def dataset_fixed_cov():
    '''Generate 2 Gaussians samples with the same covariance matrix'''
    n, dim = 300, 2
    np.random.seed(0)
    C = np.array([[0., -0.23], [0.83, .23]])
    X = np.r_[np.dot(np.random.randn(n, dim), C),
              np.dot(np.random.randn(n, dim), C) + np.array([1, 1])]
    y = np.hstack((np.zeros(n), np.ones(n)))
    return X, y

def dataset_cov():
    '''Generate 2 Gaussians samples with different covariance matrices'''
    n, dim = 300, 2
    np.random.seed(0)
    C = np.array([[0., -1.], [2.5, .7]]) * 2.
    X = np.r_[np.dot(np.random.randn(n, dim), C),
              np.dot(np.random.randn(n, dim), C.T) + np.array([1, 4])]
    y = np.hstack((np.zeros(n), np.ones(n)))
    return X, y

# ##### Plot functions #####
# Plot functions
def plot_data(lda, X, y, y_pred, fig_index):
    splot = plt.subplot(2, 2, fig_index)
    if fig_index == 1:
        plt.title('Linear Discriminant Analysis')
        plt.ylabel('Data with\nfixed covariance')
    elif fig_index == 2:
        plt.title('Quadratic Discriminant Analysis')
    elif fig_index == 3:
        plt.ylabel('Data with\nvarying covariances')

    tp = (y == y_pred) # True Positive
    tp0, tp1 = tp[y == 0], tp[y == 1]
    X0, X1 = X[y == 0], X[y == 1]
    X0_tp, X0_fp = X0[tp0], X0[~tp0]
    X1_tp, X1_fp = X1[tp1], X1[~tp1]

    # class 0: dots
    plt.scatter(X0_tp[:, 0], X0_tp[:, 1], marker='.', color='red')
    plt.scatter(X0_fp[:, 0], X0_fp[:, 1], marker='x',
                s=20, color='#990000') # dark red

    # class 1: dots
    plt.scatter(X1_tp[:, 0], X1_tp[:, 1], marker='.', color='blue')
    plt.scatter(X1_fp[:, 0], X1_fp[:, 1], marker='x',
                s=20, color='#000099') # dark blue

    # class 0 and 1 : areas
    nx, ny = 200, 100
    x_min, x_max = plt.xlim()
    y_min, y_max = plt.ylim()
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, nx),
                         np.linspace(y_min, y_max, ny))
    Z = lda.predict_proba(np.c_[xx.ravel(), yy.ravel()])
    Z = Z[:, 1].reshape(xx.shape)

```

```

plt.pcolormesh(xx, yy, Z, cmap='red_blue_classes',
               norm=colors.Normalize(0., 1.), zorder=0)
plt.contour(xx, yy, Z, [0.5], linewidths=2., colors='white')

# means
plt.plot(lda.means_[0][0], lda.means_[0][1],
         '*', color='yellow', markersize=15, markeredgewidth=2, markeredgecolor='grey')
plt.plot(lda.means_[1][0], lda.means_[1][1],
         '*', color='yellow', markersize=15, markeredgewidth=2, markeredgecolor='grey')

return splot

def plot_ellipse(splot, mean, cov, color):
    v, w = linalg.eigh(cov)
    u = w[0] / linalg.norm(w[0])
    angle = np.arctan(u[1] / u[0])
    angle = 180 * angle / np.pi # convert to degrees
    # filled Gaussian at 2 standard deviation
    ell = mpl.patches.Ellipse(mean, 2 * v[0] ** 0.5, 2 * v[1] ** 0.5,
                               180 + angle, facecolor=color,
                               edgecolor='black', linewidth=2)
    ell.set_clip_box(splot.bbox)
    ell.set_alpha(0.2)
    splot.add_artist(ell)
    splot.set_xticks(())
    splot.set_yticks(())

def plot_lda_cov(lda, splot):
    plot_ellipse(splot, lda.means_[0], lda.covariance_, 'red')
    plot_ellipse(splot, lda.means_[1], lda.covariance_, 'blue')

def plot_qda_cov(qda, splot):
    plot_ellipse(splot, qda.means_[0], qda.covariance_[0], 'red')
    plot_ellipse(splot, qda.means_[1], qda.covariance_[1], 'blue')

plt.figure(figsize=(10, 8), facecolor='white')
plt.suptitle('Linear Discriminant Analysis vs Quadratic Discriminant Analysis',
             y=0.98, fontsize=15)
for i, (X, y) in enumerate([dataset_fixed_cov(), dataset_cov()]):
    # Linear Discriminant Analysis
    lda = LinearDiscriminantAnalysis(solver="svd", store_covariance=True)
    y_pred = lda.fit(X, y).predict(X)
    splot = plot_data(lda, X, y, y_pred, fig_index=2 * i + 1)
    plot_lda_cov(lda, splot)
    plt.axis('tight')

    # Quadratic Discriminant Analysis
    qda = QuadraticDiscriminantAnalysis(store_covariance=True)
    y_pred = qda.fit(X, y).predict(X)
    splot = plot_data(qda, X, y, y_pred, fig_index=2 * i + 2)
    plot_qda_cov(qda, splot)
    plt.axis('tight')
plt.tight_layout()
plt.subplots_adjust(top=0.92)

```

```
plt.show()
```

Total running time of the script: (0 minutes 0.231 seconds)

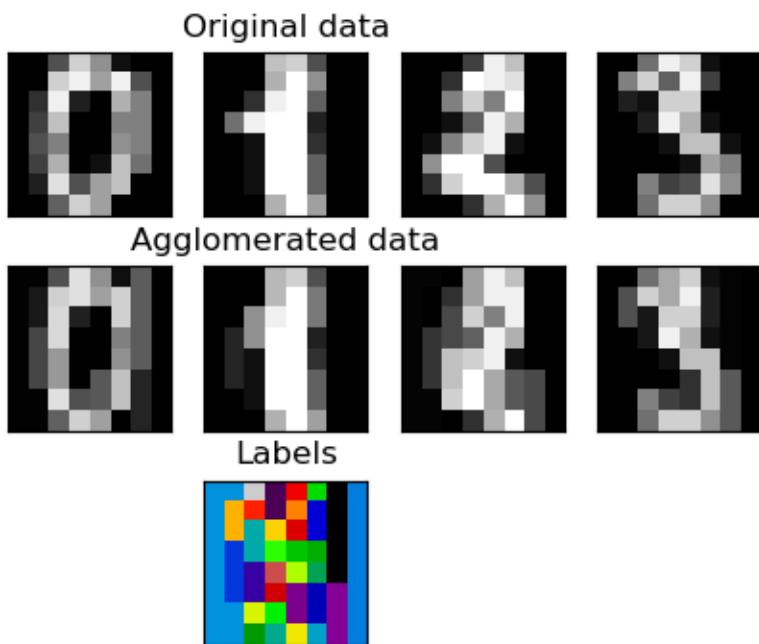
5.6 Clustering

Examples concerning the `sklearn.cluster` module.

Note: Click [here](#) to download the full example code

5.6.1 Feature agglomeration

These images show how similar features are merged together using feature agglomeration.



```
print(__doc__)

# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn import datasets, cluster
from sklearn.feature_extraction.image import grid_to_graph

digits = datasets.load_digits()
images = digits.images
X = np.reshape(images, (len(images), -1))
```

```

connectivity = grid_to_graph(*images[0].shape)

agglo = cluster.FeatureAgglomeration(connectivity=connectivity,
                                       n_clusters=32)

agglo.fit(X)
X_reduced = agglo.transform(X)

X_restored = agglo.inverse_transform(X_reduced)
images_restored = np.reshape(X_restored, images.shape)
plt.figure(1, figsize=(4, 3.5))
plt.clf()
plt.subplots_adjust(left=.01, right=.99, bottom=.01, top=.91)
for i in range(4):
    plt.subplot(3, 4, i + 1)
    plt.imshow(images[i], cmap=plt.cm.gray, vmax=16, interpolation='nearest')
    plt.xticks(())
    plt.yticks(())
    if i == 1:
        plt.title('Original data')
    plt.subplot(3, 4, 4 + i + 1)
    plt.imshow(images_restored[i], cmap=plt.cm.gray, vmax=16,
               interpolation='nearest')
    if i == 1:
        plt.title('Agglomerated data')
    plt.xticks(())
    plt.yticks(())

plt.subplot(3, 4, 10)
plt.imshow(np.reshape(agglo.labels_, images[0].shape),
           interpolation='nearest', cmap=plt.cm.nipy_spectral)
plt.xticks(())
plt.yticks(())
plt.title('Labels')
plt.show()

```

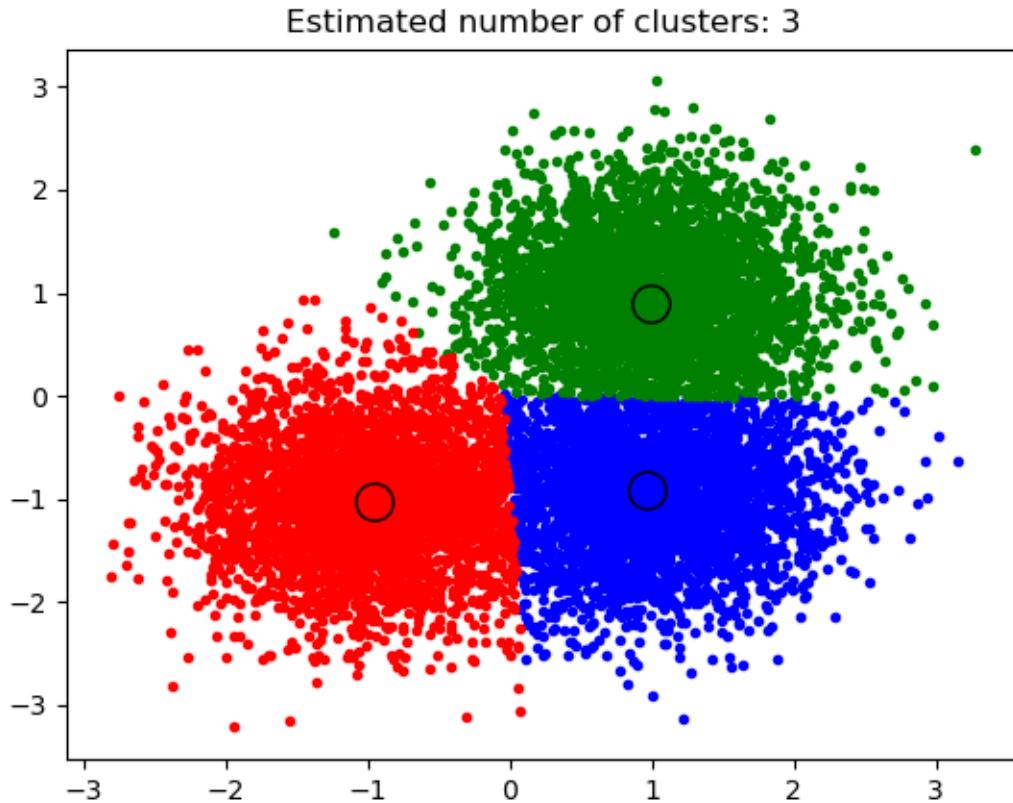
Total running time of the script: (0 minutes 0.174 seconds)

Note: Click [here](#) to download the full example code

5.6.2 A demo of the mean-shift clustering algorithm

Reference:

Dorin Comaniciu and Peter Meer, “Mean Shift: A robust approach toward feature space analysis”. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2002. pp. 603-619.



Out:

```
number of estimated clusters : 3
```

```
print(__doc__)

import numpy as np
from sklearn.cluster import MeanShift, estimate_bandwidth
from sklearn.datasets.samples_generator import make_blobs

# ##########
# Generate sample data
centers = [[1, 1], [-1, -1], [1, -1]]
X, _ = make_blobs(n_samples=10000, centers=centers, cluster_std=0.6)

# #####
# Compute clustering with MeanShift

# The following bandwidth can be automatically detected using
bandwidth = estimate_bandwidth(X, quantile=0.2, n_samples=500)
```

```

ms = MeanShift(bandwidth=bandwidth, bin_seeding=True)
ms.fit(X)
labels = ms.labels_
cluster_centers = ms.cluster_centers_

labels_unique = np.unique(labels)
n_clusters_ = len(labels_unique)

print("number of estimated clusters : %d" % n_clusters_)

# ##### Plot result
# Import plotting modules
import matplotlib.pyplot as plt
from itertools import cycle

plt.figure(1)
plt.clf()

colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
for k, col in zip(range(n_clusters_), colors):
    my_members = labels == k
    cluster_center = cluster_centers[k]
    plt.plot(X[my_members, 0], X[my_members, 1], col + '.')
    plt.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
             markeredgecolor='k', markersize=14)
plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()

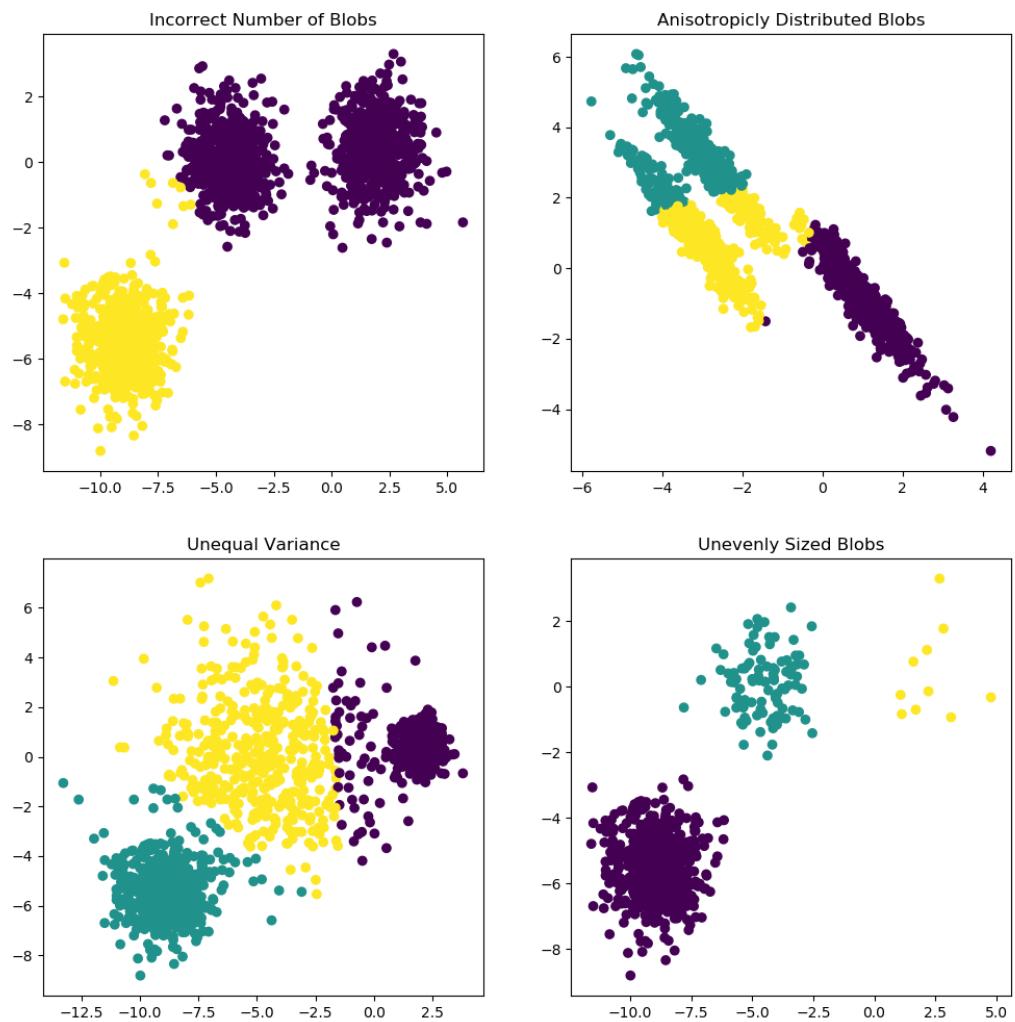
```

Total running time of the script: (0 minutes 0.397 seconds)

Note: Click [here](#) to download the full example code

5.6.3 Demonstration of k-means assumptions

This example is meant to illustrate situations where k-means will produce unintuitive and possibly unexpected clusters. In the first three plots, the input data does not conform to some implicit assumption that k-means makes and undesirable clusters are produced as a result. In the last plot, k-means returns intuitive clusters despite unevenly sized blobs.



```
print(__doc__)

# Author: Phil Roth <mr.phil.roth@gmail.com>
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

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

n_samples = 1500
random_state = 170
```

```

x, y = make_blobs(n_samples=n_samples, random_state=random_state)

# Incorrect number of clusters
y_pred = KMeans(n_clusters=2, random_state=random_state).fit_predict(X)

plt.subplot(221)
plt.scatter(X[:, 0], X[:, 1], c=y_pred)
plt.title("Incorrect Number of Blobs")

# Anisotropically distributed data
transformation = [[0.60834549, -0.63667341], [-0.40887718, 0.85253229]]
X_aniso = np.dot(X, transformation)
y_pred = KMeans(n_clusters=3, random_state=random_state).fit_predict(X_aniso)

plt.subplot(222)
plt.scatter(X_aniso[:, 0], X_aniso[:, 1], c=y_pred)
plt.title("Anisotropically Distributed Blobs")

# Different variance
X_varied, y_varied = make_blobs(n_samples=n_samples,
                                 cluster_std=[1.0, 2.5, 0.5],
                                 random_state=random_state)
y_pred = KMeans(n_clusters=3, random_state=random_state).fit_predict(X_varied)

plt.subplot(223)
plt.scatter(X_varied[:, 0], X_varied[:, 1], c=y_pred)
plt.title("Unequal Variance")

# Unevenly sized blobs
X_filtered = np.vstack((X[y == 0][:500], X[y == 1][:100], X[y == 2][:10]))
y_pred = KMeans(n_clusters=3,
                random_state=random_state).fit_predict(X_filtered)

plt.subplot(224)
plt.scatter(X_filtered[:, 0], X_filtered[:, 1], c=y_pred)
plt.title("Unevenly Sized Blobs")

plt.show()

```

Total running time of the script: (0 minutes 0.163 seconds)

Note: Click [here](#) to download the full example code

5.6.4 Online learning of a dictionary of parts of faces

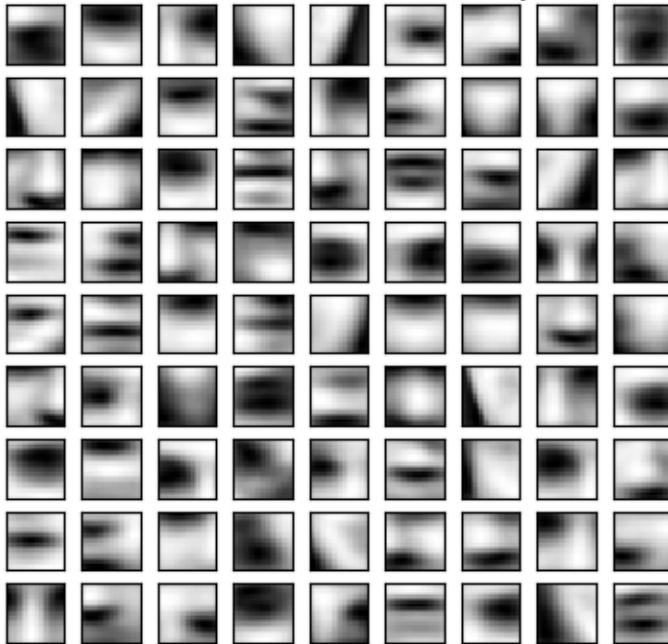
This example uses a large dataset of faces to learn a set of 20 x 20 images patches that constitute faces.

From the programming standpoint, it is interesting because it shows how to use the online API of the scikit-learn to process a very large dataset by chunks. The way we proceed is that we load an image at a time and extract randomly 50 patches from this image. Once we have accumulated 500 of these patches (using 10 images), we run the `partial_fit` method of the online KMeans object, MiniBatchKMeans.

The verbose setting on the MiniBatchKMeans enables us to see that some clusters are reassigned during the successive calls to partial-fit. This is because the number of patches that they represent has become too low, and it is better to choose a random new cluster.

Patches of faces

Train time 5.1s on 3200 patches



Out:

```
Learning the dictionary...
Partial fit of 100 out of 2400
Partial fit of 200 out of 2400
[MiniBatchKMeans] Reassigning 16 cluster centers.
Partial fit of 300 out of 2400
Partial fit of 400 out of 2400
Partial fit of 500 out of 2400
Partial fit of 600 out of 2400
Partial fit of 700 out of 2400
Partial fit of 800 out of 2400
Partial fit of 900 out of 2400
Partial fit of 1000 out of 2400
Partial fit of 1100 out of 2400
Partial fit of 1200 out of 2400
Partial fit of 1300 out of 2400
Partial fit of 1400 out of 2400
Partial fit of 1500 out of 2400
Partial fit of 1600 out of 2400
Partial fit of 1700 out of 2400
Partial fit of 1800 out of 2400
Partial fit of 1900 out of 2400
Partial fit of 2000 out of 2400
Partial fit of 2100 out of 2400
Partial fit of 2200 out of 2400
Partial fit of 2300 out of 2400
Partial fit of 2400 out of 2400
done in 5.06s.
```

```

print(__doc__)

import time

import matplotlib.pyplot as plt
import numpy as np

from sklearn import datasets
from sklearn.cluster import MiniBatchKMeans
from sklearn.feature_extraction.image import extract_patches_2d

faces = datasets.fetch_olivetti_faces()

# ######
# Learn the dictionary of images

print('Learning the dictionary... ')
rng = np.random.RandomState(0)
kmeans = MiniBatchKMeans(n_clusters=81, random_state=rng, verbose=True)
patch_size = (20, 20)

buffer = []
t0 = time.time()

# The online learning part: cycle over the whole dataset 6 times
index = 0
for _ in range(6):
    for img in faces.images:
        data = extract_patches_2d(img, patch_size, max_patches=50,
                                 random_state=rng)
        data = np.reshape(data, (len(data), -1))
        buffer.append(data)
        index += 1
        if index % 10 == 0:
            data = np.concatenate(buffer, axis=0)
            data -= np.mean(data, axis=0)
            data /= np.std(data, axis=0)
            kmeans.partial_fit(data)
            buffer = []
        if index % 100 == 0:
            print('Partial fit of %4i out of %i'
                  % (index, 6 * len(faces.images)))

dt = time.time() - t0
print('done in %.2fs.' % dt)

# #####
# Plot the results
plt.figure(figsize=(4.2, 4))
for i, patch in enumerate(kmeans.cluster_centers_):
    plt.subplot(9, 9, i + 1)
    plt.imshow(patch.reshape(patch_size), cmap=plt.cm.gray,
               interpolation='nearest')
    plt.xticks(())

```

```
plt.yticks(())

plt.suptitle('Patches of faces\nTrain time %.1fs on %d patches' %
             (dt, 8 * len(faces.images)), fontsize=16)
plt.subplots_adjust(0.08, 0.02, 0.92, 0.85, 0.08, 0.23)

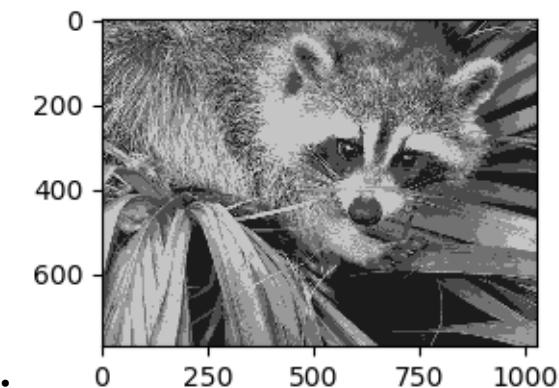
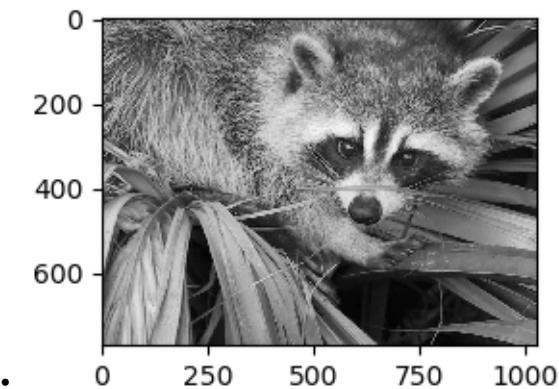
plt.show()
```

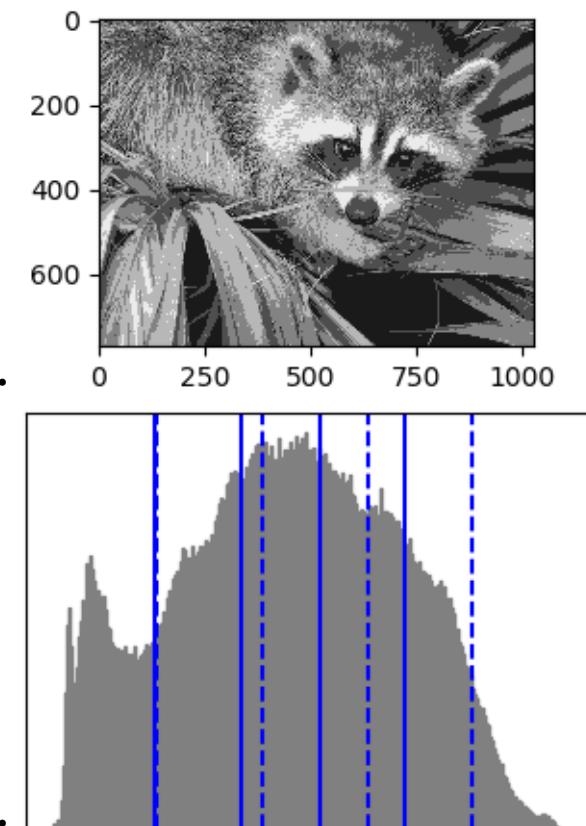
Total running time of the script: (0 minutes 6.148 seconds)

Note: Click [here](#) to download the full example code

5.6.5 Vector Quantization Example

Face, a 1024 x 768 size image of a raccoon face, is used here to illustrate how k-means is used for vector quantization.





```

print(__doc__)

# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

from sklearn import cluster

try:    # SciPy >= 0.16 have face in misc
    from scipy.misc import face
    face = face(gray=True)
except ImportError:
    face = sp.face(gray=True)

n_clusters = 5
np.random.seed(0)

X = face.reshape((-1, 1))    # We need an (n_sample, n_feature) array
k_means = cluster.KMeans(n_clusters=n_clusters, n_init=4)
k_means.fit(X)
values = k_means.cluster_centers_.squeeze()
labels = k_means.labels_

```

```
# create an array from labels and values
face_compressed = np.choose(labels, values)
face_compressed.shape = face.shape

vmin = face.min()
vmax = face.max()

# original face
plt.figure(1, figsize=(3, 2.2))
plt.imshow(face, cmap=plt.cm.gray, vmin=vmin, vmax=256)

# compressed face
plt.figure(2, figsize=(3, 2.2))
plt.imshow(face_compressed, cmap=plt.cm.gray, vmin=vmin, vmax=vmax)

# equal bins face
regular_values = np.linspace(0, 256, n_clusters + 1)
regular_labels = np.searchsorted(regular_values, face) - 1
regular_values = .5 * (regular_values[1:] + regular_values[:-1]) # mean
regular_face = np.choose(regular_labels.ravel(), regular_values, mode="clip")
regular_face.shape = face.shape
plt.figure(3, figsize=(3, 2.2))
plt.imshow(regular_face, cmap=plt.cm.gray, vmin=vmin, vmax=vmax)

# histogram
plt.figure(4, figsize=(3, 2.2))
plt.clf()
plt.axes([.01, .01, .98, .98])
plt.hist(X, bins=256, color='.5', edgecolor='.5')
plt.yticks(())
plt.xticks(regular_values)
values = np.sort(values)
for center_1, center_2 in zip(values[:-1], values[1:]):
    plt.axvline(.5 * (center_1 + center_2), color='b')

for center_1, center_2 in zip(regular_values[:-1], regular_values[1:]):
    plt.axvline(.5 * (center_1 + center_2), color='b', linestyle='--')

plt.show()
```

Total running time of the script: (0 minutes 3.752 seconds)

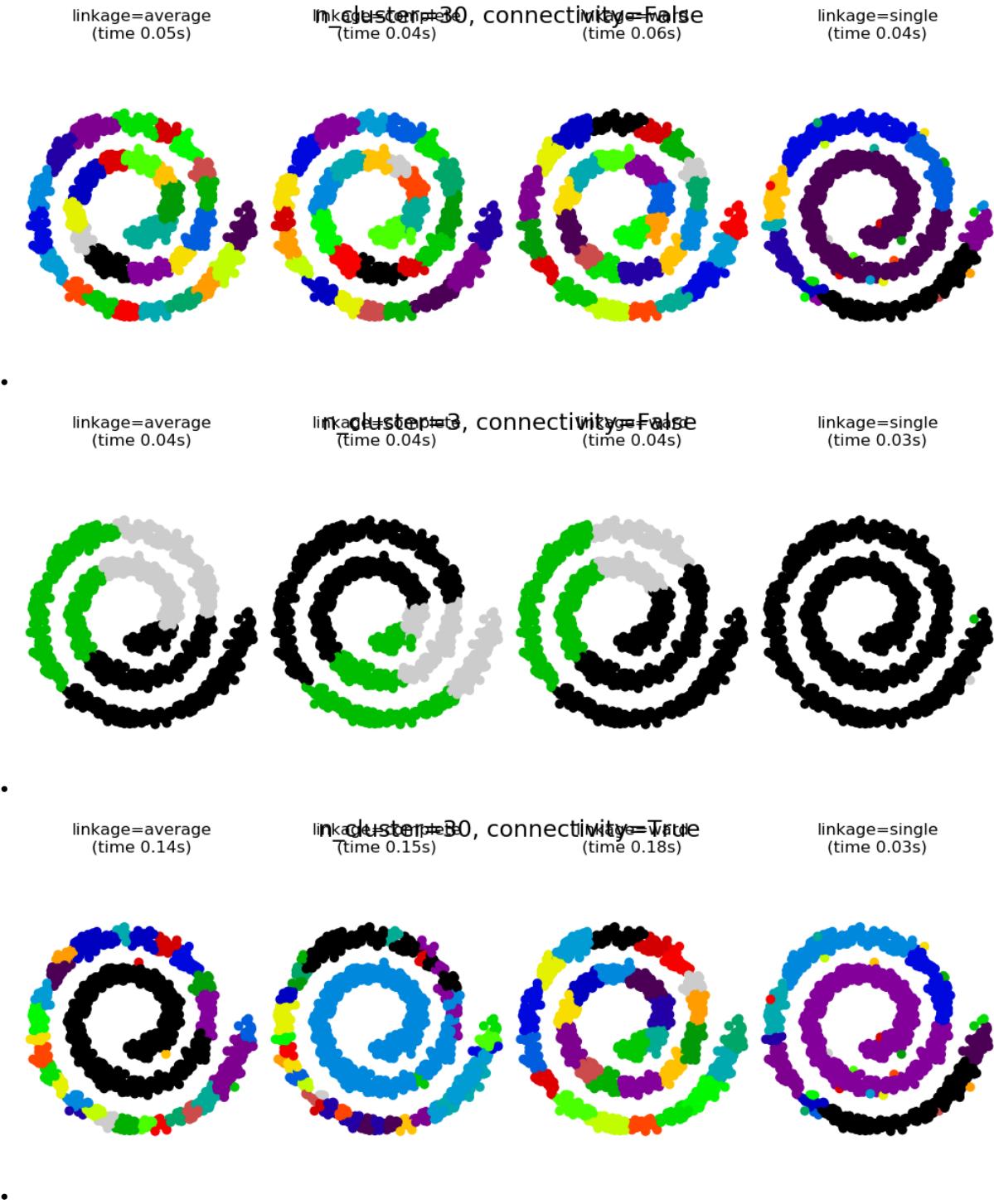
Note: Click [here](#) to download the full example code

5.6.6 Agglomerative clustering with and without structure

This example shows the effect of imposing a connectivity graph to capture local structure in the data. The graph is simply the graph of 20 nearest neighbors.

Two consequences of imposing a connectivity can be seen. First clustering with a connectivity matrix is much faster. Second, when using a connectivity matrix, single, average and complete linkage are unstable and tend to create a few clusters that grow very quickly. Indeed, average and complete linkage fight this percolation behavior by considering all the distances between two clusters when merging them (while single linkage exaggerates the behaviour by considering

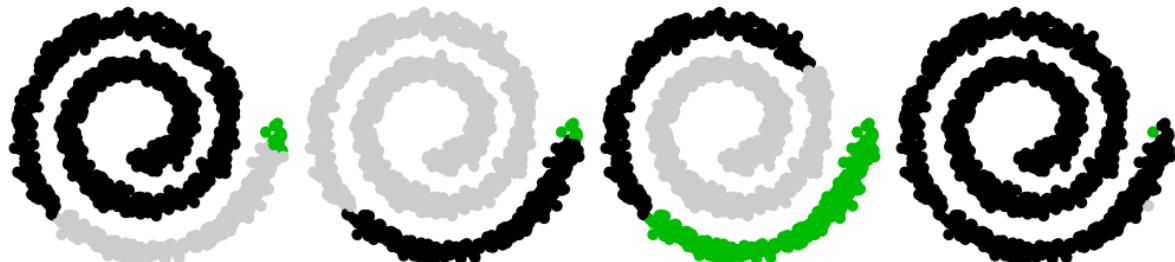
only the shortest distance between clusters). The connectivity graph breaks this mechanism for average and complete linkage, making them resemble the more brittle single linkage. This effect is more pronounced for very sparse graphs (try decreasing the number of neighbors in `kneighbors_graph`) and with complete linkage. In particular, having a very small number of neighbors in the graph, imposes a geometry that is close to that of single linkage, which is well known to have this percolation instability.



linkage=average
(time 0.12s)

linkage=average,
n_clusters=3, connectivity=True
(time 0.12s)

linkage=single
(time 0.03s)



```
# Authors: Gael Varoquaux, Nelle Varoquaux
# License: BSD 3 clause

import time
import matplotlib.pyplot as plt
import numpy as np

from sklearn.cluster import AgglomerativeClustering
from sklearn.neighbors import kneighbors_graph

# Generate sample data
n_samples = 1500
np.random.seed(0)
t = 1.5 * np.pi * (1 + 3 * np.random.rand(1, n_samples))
x = t * np.cos(t)
y = t * np.sin(t)

X = np.concatenate((x, y))
X += .7 * np.random.randn(2, n_samples)
X = X.T

# Create a graph capturing local connectivity. Larger number of neighbors
# will give more homogeneous clusters to the cost of computation
# time. A very large number of neighbors gives more evenly distributed
# cluster sizes, but may not impose the local manifold structure of
# the data
knn_graph = kneighbors_graph(X, 30, include_self=False)

for connectivity in (None, knn_graph):
    for n_clusters in (30, 3):
        plt.figure(figsize=(10, 4))
        for index, linkage in enumerate(('average',
                                         'complete',
                                         'ward',
                                         'single')):
            plt.subplot(1, 4, index + 1)
            model = AgglomerativeClustering(linkage=linkage,
                                              connectivity=connectivity,
                                              n_clusters=n_clusters)
            t0 = time.time()
```

```
model.fit(X)
elapsed_time = time.time() - t0
plt.scatter(X[:, 0], X[:, 1], c=model.labels_,
            cmap=plt.cm.nipy_spectral)
plt.title('linkage=%s\n(time %.2fs)' % (linkage, elapsed_time),
           fontdict=dict(verticalalignment='top'))
plt.axis('equal')
plt.axis('off')

plt.subplots_adjust(bottom=0, top=.89, wspace=0,
                    left=0, right=1)
plt.suptitle('n_cluster=%i, connectivity=%r' %
             (n_clusters, connectivity is not None), size=17)

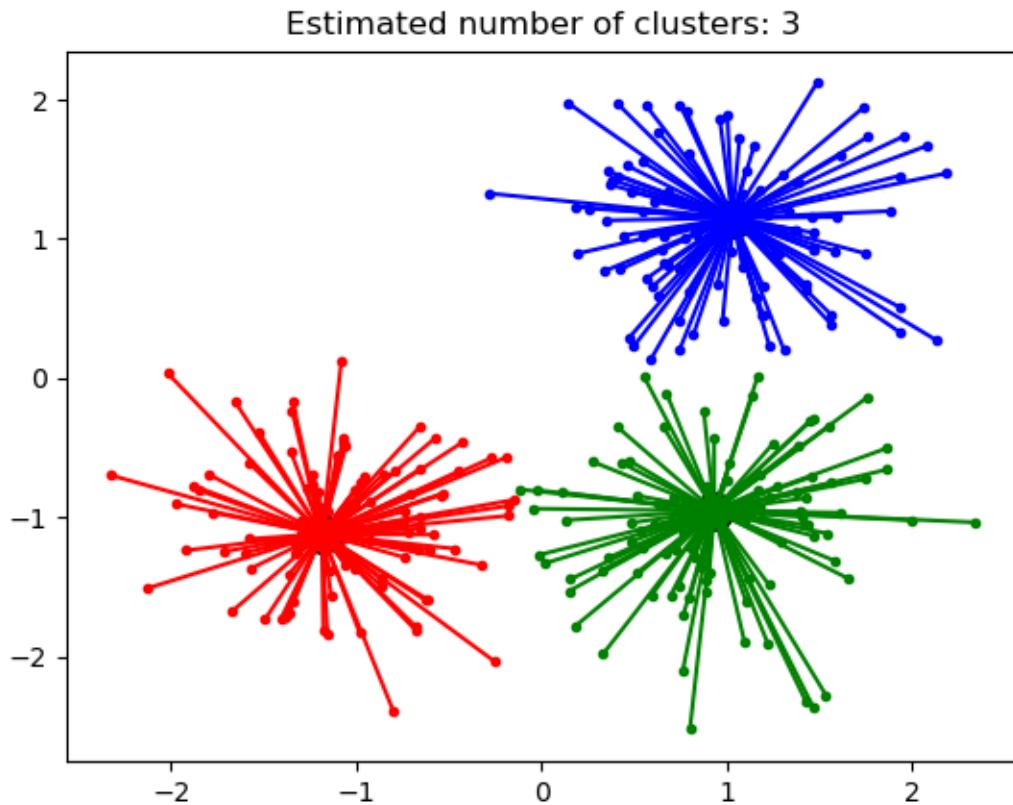
plt.show()
```

Total running time of the script: (0 minutes 1.743 seconds)

Note: Click [here](#) to download the full example code

5.6.7 Demo of affinity propagation clustering algorithm

Reference: Brendan J. Frey and Delbert Dueck, “Clustering by Passing Messages Between Data Points”, Science Feb. 2007



Out:

```
Estimated number of clusters: 3
Homogeneity: 0.872
Completeness: 0.872
V-measure: 0.872
Adjusted Rand Index: 0.912
Adjusted Mutual Information: 0.871
Silhouette Coefficient: 0.753
```

```
print(__doc__)

from sklearn.cluster import AffinityPropagation
from sklearn import metrics
from sklearn.datasets.samples_generator import make_blobs

# ##########
# Generate sample data
centers = [[1, 1], [-1, -1], [1, -1]]
X, labels_true = make_blobs(n_samples=300, centers=centers, cluster_std=0.5,
                           random_state=0)
```

```

# ##########
# Compute Affinity Propagation
af = AffinityPropagation(preference=-50).fit(X)
cluster_centers_indices = af.cluster_centers_indices_
labels = af.labels_

n_clusters_ = len(cluster_centers_indices)

print('Estimated number of clusters: %d' % n_clusters_)
print("Homogeneity: %0.3f" % metrics.homogeneity_score(labels_true, labels))
print("Completeness: %0.3f" % metrics.completeness_score(labels_true, labels))
print("V-measure: %0.3f" % metrics.v_measure_score(labels_true, labels))
print("Adjusted Rand Index: %0.3f"
      % metrics.adjusted_rand_score(labels_true, labels))
print("Adjusted Mutual Information: %0.3f"
      % metrics.adjusted_mutual_info_score(labels_true, labels,
                                             average_method='arithmetic'))
print("Silhouette Coefficient: %0.3f"
      % metrics.silhouette_score(X, labels, metric='sqeuclidean'))

# #####
# Plot result
import matplotlib.pyplot as plt
from itertools import cycle

plt.close('all')
plt.figure(1)
plt.clf()

colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
for k, col in zip(range(n_clusters_), colors):
    class_members = labels == k
    cluster_center = X[cluster_centers_indices[k]]
    plt.plot(X[class_members, 0], X[class_members, 1], col + '.')
    plt.plot(cluster_center[0], cluster_center[1], 'o',
             markerfacecolor=col,
             markeredgecolor='k', markersize=14)
    for x in X[class_members]:
        plt.plot([cluster_center[0], x[0]], [cluster_center[1], x[1]], col)

plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()

```

Total running time of the script: (0 minutes 0.569 seconds)

Note: Click [here](#) to download the full example code

5.6.8 Segmenting the picture of greek coins in regions

This example uses *Spectral clustering* on a graph created from voxel-to-voxel difference on an image to break this image into multiple partly-homogeneous regions.

This procedure (spectral clustering on an image) is an efficient approximate solution for finding normalized graph cuts.

There are two options to assign labels:

- with ‘kmeans’ spectral clustering will cluster samples in the embedding space using a kmeans algorithm
- whereas ‘discrete’ will iteratively search for the closest partition space to the embedding space.

```

print(__doc__)

# Author: Gael Varoquaux <gael.varoquaux@normalesup.org>, Brian Cheung
# License: BSD 3 clause

import time

import numpy as np
from distutils.version import LooseVersion
from scipy.ndimage.filters import gaussian_filter
import matplotlib.pyplot as plt
import skimage
from skimage.data import coins
from skimage.transform import rescale

from sklearn.feature_extraction import image
from sklearn.cluster import spectral_clustering

# these were introduced in skimage-0.14
if LooseVersion(skimage.__version__) >= '0.14':
    rescale_params = {'anti_aliasing': False, 'multichannel': False}
else:
    rescale_params = {}

# load the coins as a numpy array
orig_coins = coins()

# Resize it to 20% of the original size to speed up the processing
# Applying a Gaussian filter for smoothing prior to down-scaling
# reduces aliasing artifacts.
smoothened_coins = gaussian_filter(orig_coins, sigma=2)
rescaled_coins = rescale(smoothened_coins, 0.2, mode="reflect",
                        **rescale_params)

# Convert the image into a graph with the value of the gradient on the
# edges.
graph = image.img_to_graph(rescaled_coins)

# Take a decreasing function of the gradient: an exponential
# The smaller beta is, the more independent the segmentation is of the
# actual image. For beta=1, the segmentation is close to a voronoi
beta = 10
eps = 1e-6
graph.data = np.exp(-beta * graph.data / graph.data.std()) + eps

# Apply spectral clustering (this step goes much faster if you have pyamg
# installed)
N_REGIONS = 25

```

Visualize the resulting regions

```

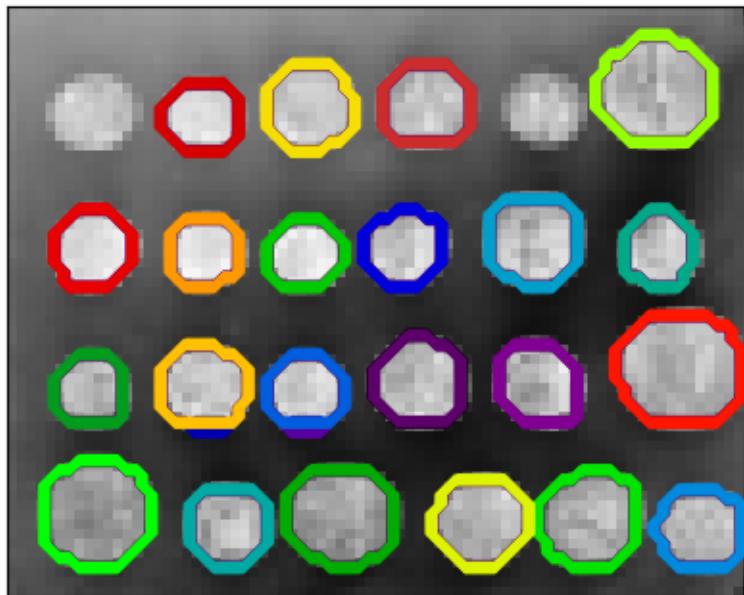
for assign_labels in ('kmeans', 'discretize'):
    t0 = time.time()
    labels = spectral_clustering(graph, n_clusters=N_REGIONS,
                                  assign_labels=assign_labels, random_state=42)

```

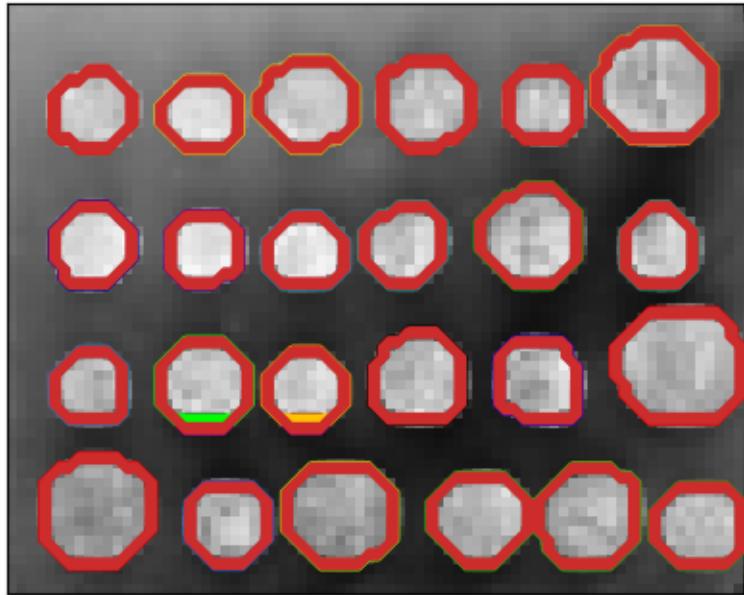
```
t1 = time.time()
labels = labels.reshape(rescaled_coins.shape)

plt.figure(figsize=(5, 5))
plt.imshow(rescaled_coins, cmap=plt.cm.gray)
for l in range(N_REGIONS):
    plt.contour(labels == l,
                colors=[plt.cm.nipy_spectral(l / float(N_REGIONS))])
plt.xticks(())
plt.yticks(())
title = 'Spectral clustering: %s, %.2fs' % (assign_labels, (t1 - t0))
print(title)
plt.title(title)
plt.show()
```

Spectral clustering: kmeans, 5.15s



Spectral clustering: discretize, 5.98s



Out:

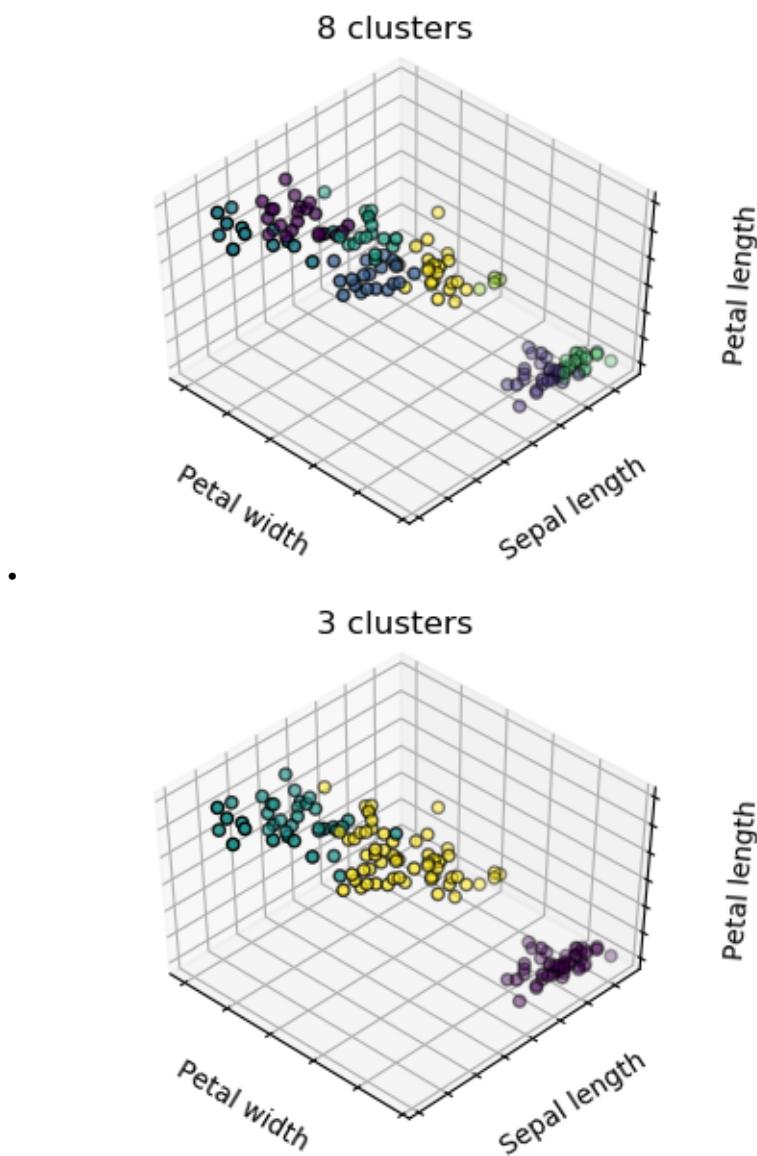
```
Spectral clustering: kmeans, 5.15s
Spectral clustering: discretize, 5.98s
```

Total running time of the script: (0 minutes 11.961 seconds)

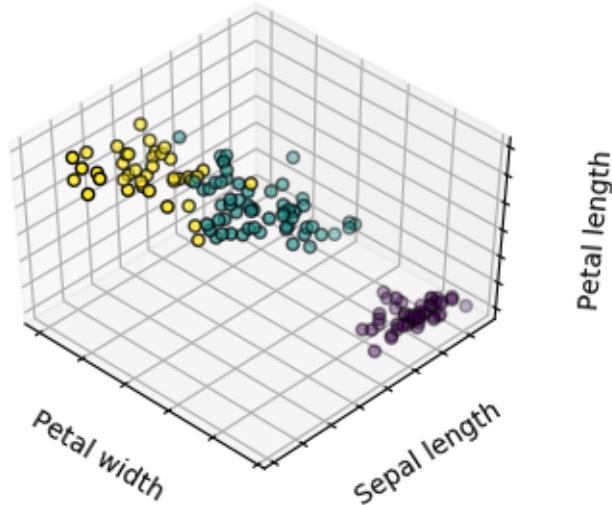
Note: Click [here](#) to download the full example code

5.6.9 K-means Clustering

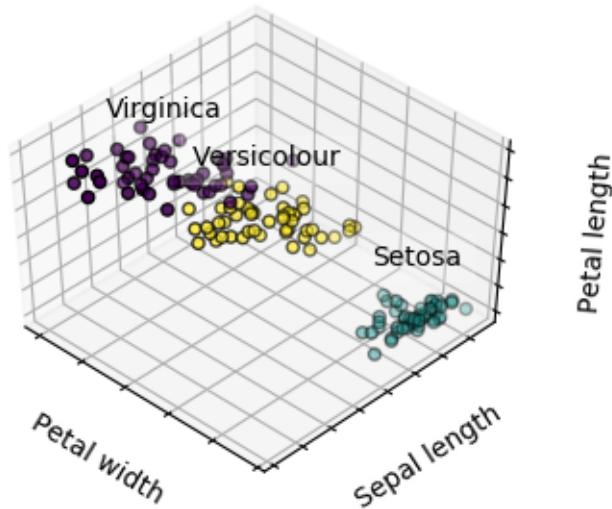
The plots display firstly what a K-means algorithm would yield using three clusters. It is then shown what the effect of a bad initialization is on the classification process: By setting n_init to only 1 (default is 10), the amount of times that the algorithm will be run with different centroid seeds is reduced. The next plot displays what using eight clusters would deliver and finally the ground truth.



3 clusters, bad initialization



Ground Truth



```
print(__doc__)

# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
# Though the following import is not directly being used, it is required
# for 3D projection to work
from mpl_toolkits.mplot3d import Axes3D

from sklearn.cluster import KMeans
from sklearn import datasets

np.random.seed(5)
```

```

iris = datasets.load_iris()
X = iris.data
y = iris.target

estimators = [('k_means_iris_8', KMeans(n_clusters=8)),
              ('k_means_iris_3', KMeans(n_clusters=3)),
              ('k_means_iris_bad_init', KMeans(n_clusters=3, n_init=1,
                                              init='random'))]

fignum = 1
titles = ['8 clusters', '3 clusters', '3 clusters, bad initialization']
for name, est in estimators:
    fig = plt.figure(fignum, figsize=(4, 3))
    ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)
    est.fit(X)
    labels = est.labels_

    ax.scatter(X[:, 3], X[:, 0], X[:, 2],
               c=labels.astype(np.float), edgecolor='k')

    ax.w_xaxis.set_ticklabels([])
    ax.w_yaxis.set_ticklabels([])
    ax.w_zaxis.set_ticklabels([])
    ax.set_xlabel('Petal width')
    ax.set_ylabel('Sepal length')
    ax.set_zlabel('Petal length')
    ax.set_title(titles[fignum - 1])
    ax.dist = 12
    fignum = fignum + 1

# Plot the ground truth
fig = plt.figure(fignum, figsize=(4, 3))
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)

for name, label in [('Setosa', 0),
                     ('Versicolour', 1),
                     ('Virginica', 2)]:
    ax.text3D(X[y == label, 3].mean(),
              X[y == label, 0].mean(),
              X[y == label, 2].mean() + 2, name,
              horizontalalignment='center',
              bbox=dict(alpha=.2, edgecolor='w', facecolor='w'))
# Reorder the labels to have colors matching the cluster results
y = np.choose(y, [1, 2, 0]).astype(np.float)
ax.scatter(X[:, 3], X[:, 0], X[:, 2], c=y, edgecolor='k')

ax.w_xaxis.set_ticklabels([])
ax.w_yaxis.set_ticklabels([])
ax.w_zaxis.set_ticklabels([])
ax.set_xlabel('Petal width')
ax.set_ylabel('Sepal length')
ax.set_zlabel('Petal length')
ax.set_title('Ground Truth')
ax.dist = 12

fig.show()

```

Total running time of the script: (0 minutes 0.241 seconds)

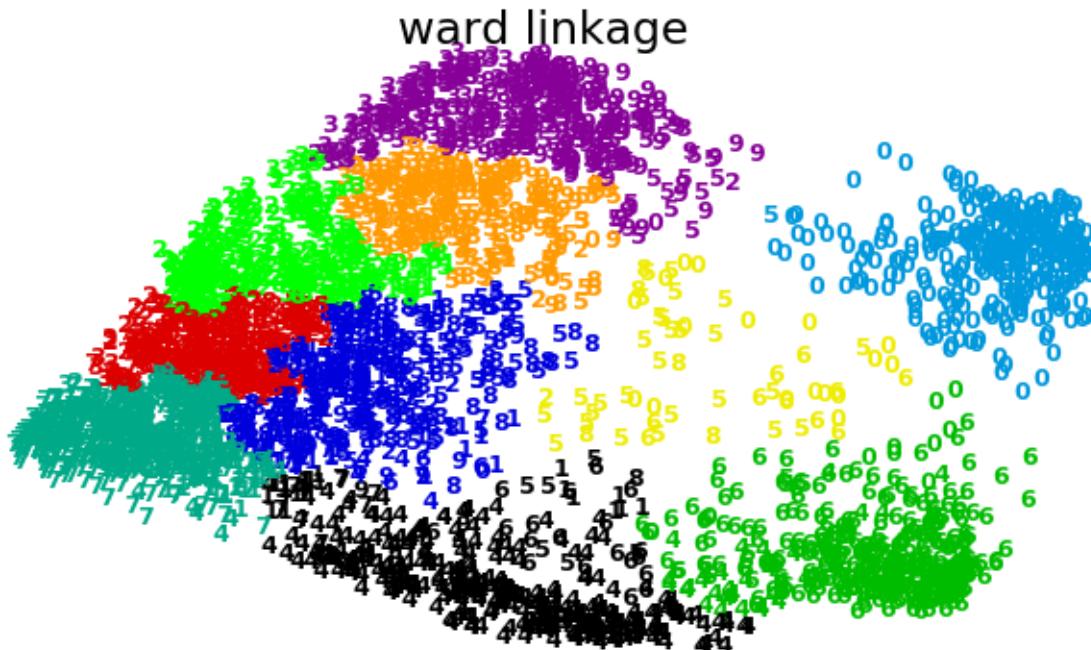
Note: Click [here](#) to download the full example code

5.6.10 Various Agglomerative Clustering on a 2D embedding of digits

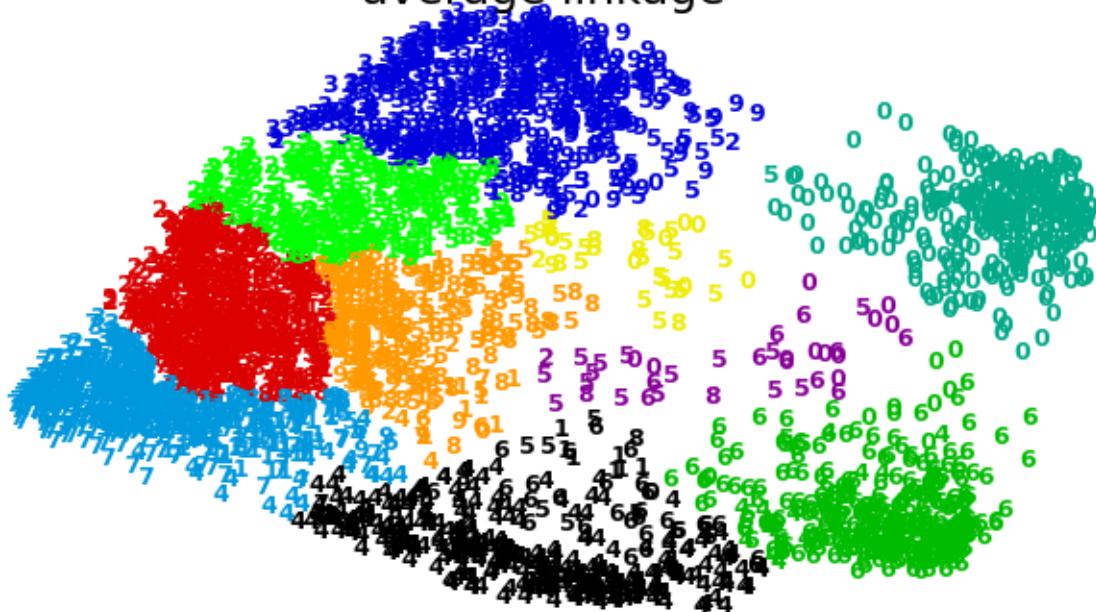
An illustration of various linkage option for agglomerative clustering on a 2D embedding of the digits dataset.

The goal of this example is to show intuitively how the metrics behave, and not to find good clusters for the digits. This is why the example works on a 2D embedding.

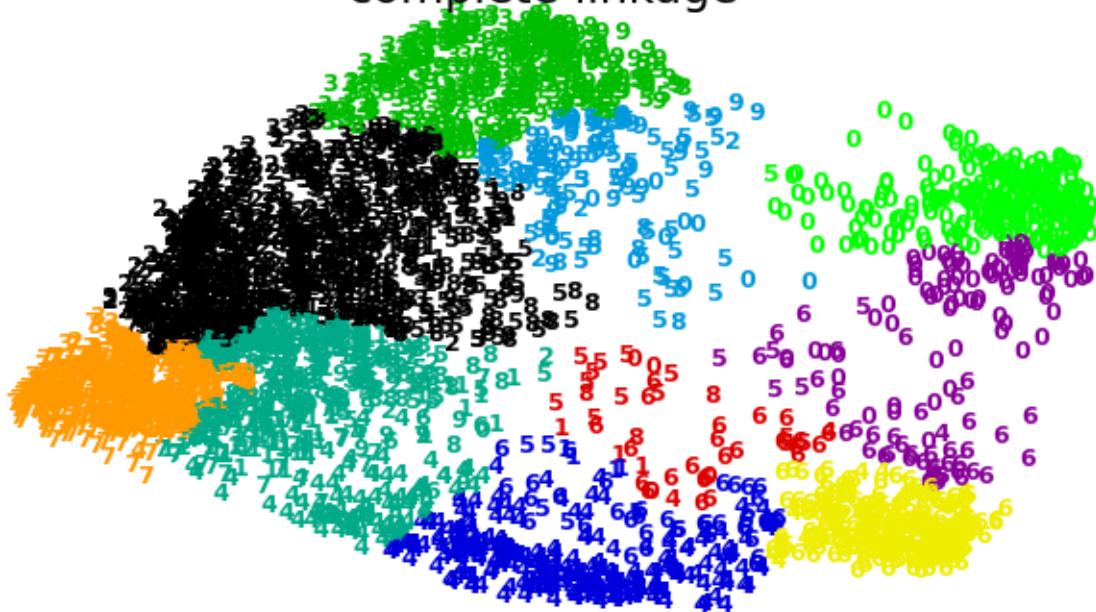
What this example shows us is the behavior “rich getting richer” of agglomerative clustering that tends to create uneven cluster sizes. This behavior is pronounced for the average linkage strategy, that ends up with a couple of singleton clusters, while in the case of single linkage we get a single central cluster with all other clusters being drawn from noise points around the fringes.

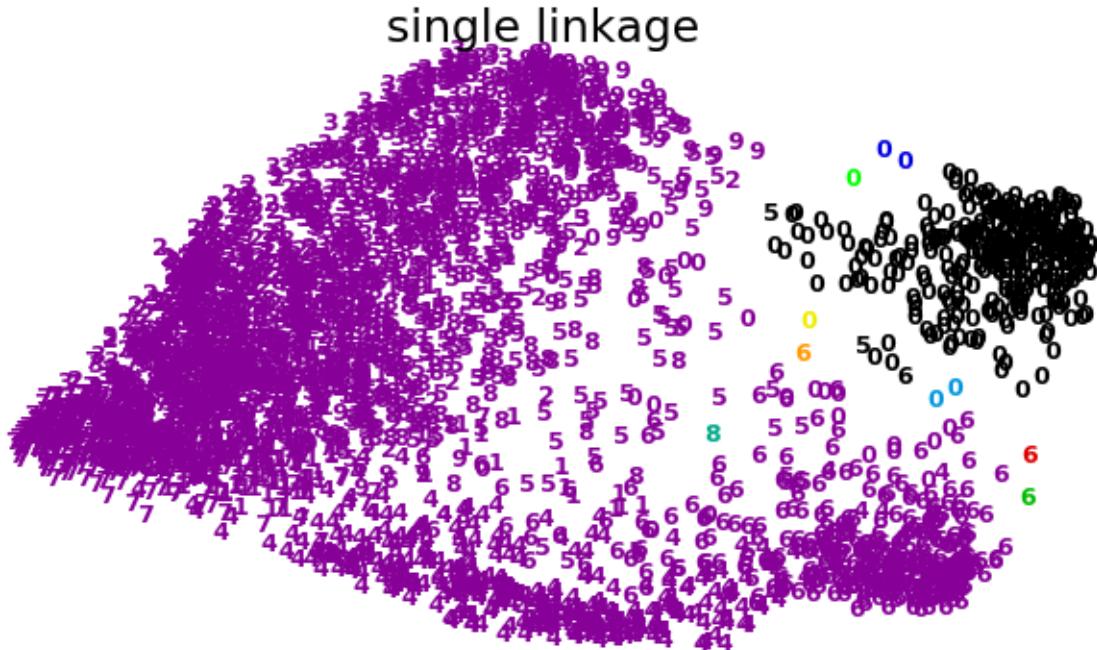


average linkage



complete linkage





Out:

```
Computing embedding
Done.
ward : 0.46s
average : 0.36s
complete : 0.38s
single : 0.17s
```

```
# Authors: Gael Varoquaux
# License: BSD 3 clause (C) INRIA 2014

print(__doc__)
from time import time

import numpy as np
from scipy import ndimage
from matplotlib import pyplot as plt

from sklearn import manifold, datasets

digits = datasets.load_digits(n_class=10)
X = digits.data
y = digits.target
n_samples, n_features = X.shape
```

```

np.random.seed(0)

def nudge_images(X, y):
    # Having a larger dataset shows more clearly the behavior of the
    # methods, but we multiply the size of the dataset only by 2, as the
    # cost of the hierarchical clustering methods are strongly
    # super-linear in n_samples
    shift = lambda x: ndimage.shift(x.reshape((8, 8)),
                                    .3 * np.random.normal(size=2),
                                    mode='constant',
                                    ).ravel()
    X = np.concatenate([X, np.apply_along_axis(shift, 1, X)])
    Y = np.concatenate([y, y], axis=0)
    return X, Y

X, y = nudge_images(X, y)

#-----
# Visualize the clustering
def plot_clustering(X_red, labels, title=None):
    x_min, x_max = np.min(X_red, axis=0), np.max(X_red, axis=0)
    X_red = (X_red - x_min) / (x_max - x_min)

    plt.figure(figsize=(6, 4))
    for i in range(X_red.shape[0]):
        plt.text(X_red[i, 0], X_red[i, 1], str(y[i]),
                 color=plt.cm.nipy_spectral(labels[i] / 10.),
                 fontdict={'weight': 'bold', 'size': 9})

    plt.xticks([])
    plt.yticks([])
    if title is not None:
        plt.title(title, size=17)
    plt.axis('off')
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])

#-----
# 2D embedding of the digits dataset
print("Computing embedding")
X_red = manifold.SpectralEmbedding(n_components=2).fit_transform(X)
print("Done.")

from sklearn.cluster import AgglomerativeClustering

for linkage in ('ward', 'average', 'complete', 'single'):
    clustering = AgglomerativeClustering(linkage=linkage, n_clusters=10)
    t0 = time()
    clustering.fit(X_red)
    print("%s :%t%.2fs" % (linkage, time() - t0))

    plot_clustering(X_red, clustering.labels_, "%s linkage" % linkage)

plt.show()

```

Total running time of the script: (0 minutes 26.873 seconds)

Note: Click [here](#) to download the full example code

5.6.11 Spectral clustering for image segmentation

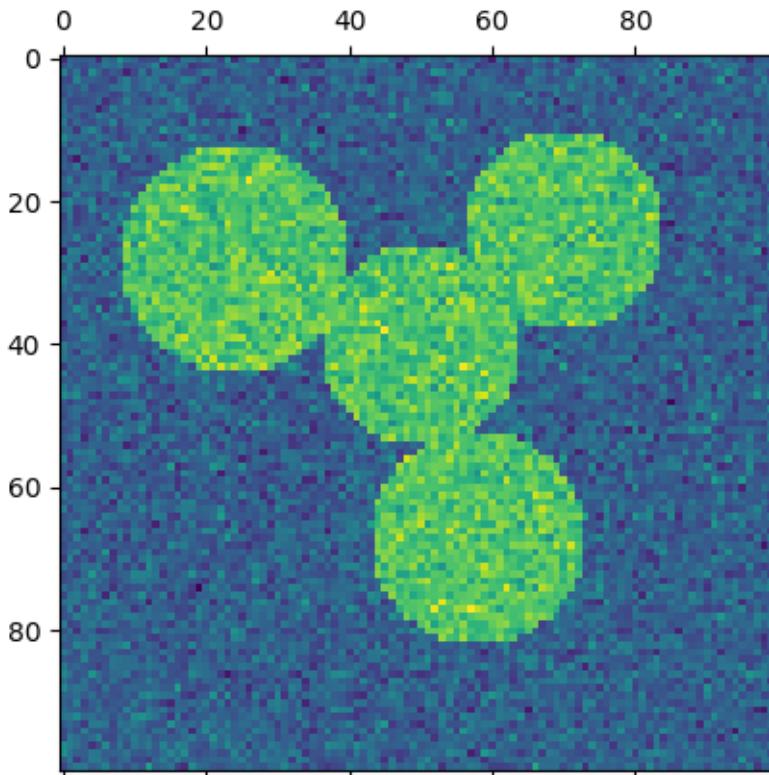
In this example, an image with connected circles is generated and spectral clustering is used to separate the circles.

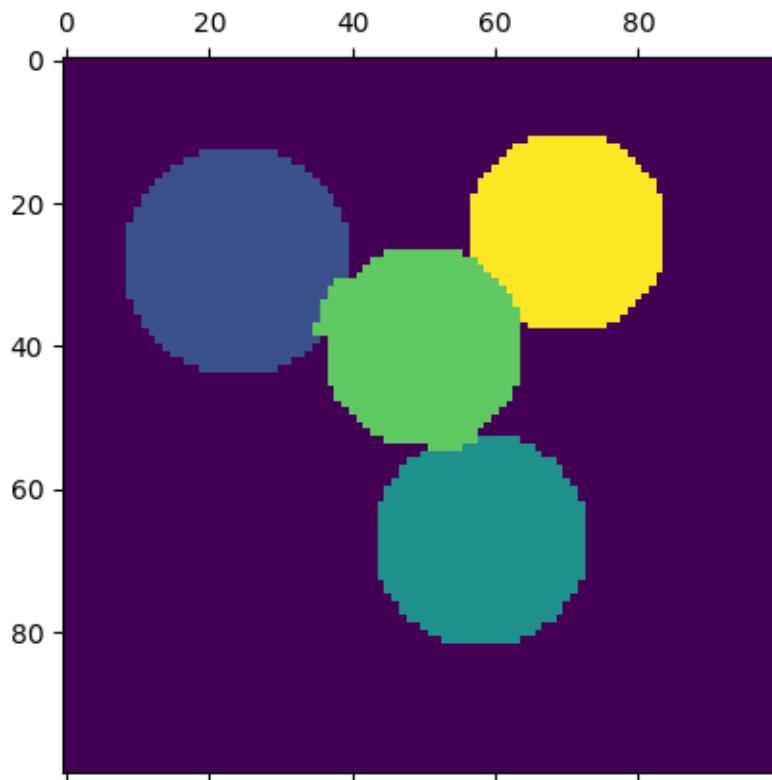
In these settings, the *Spectral clustering* approach solves the problem known as ‘normalized graph cuts’: the image is seen as a graph of connected voxels, and the spectral clustering algorithm amounts to choosing graph cuts defining regions while minimizing the ratio of the gradient along the cut, and the volume of the region.

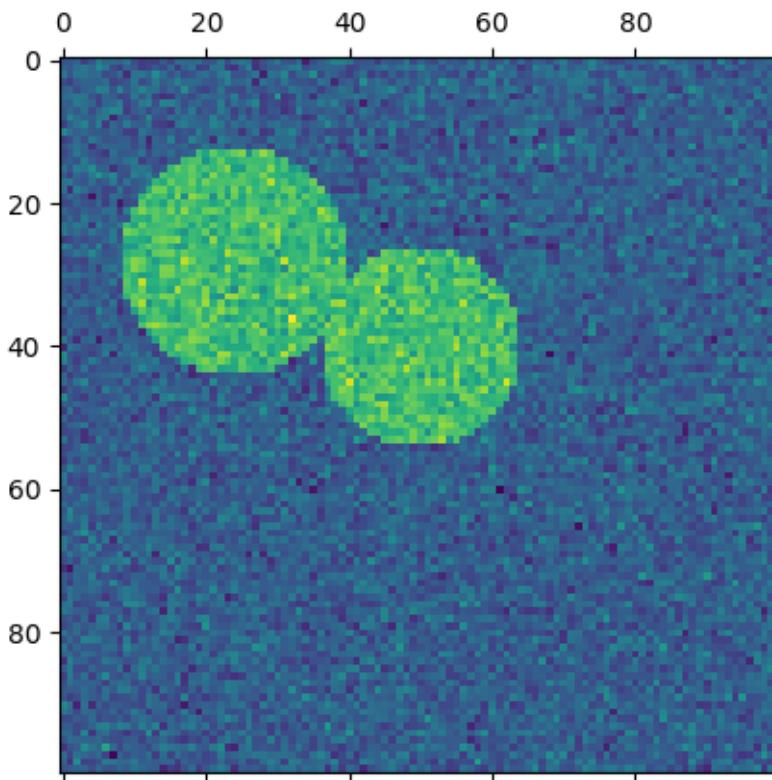
As the algorithm tries to balance the volume (ie balance the region sizes), if we take circles with different sizes, the segmentation fails.

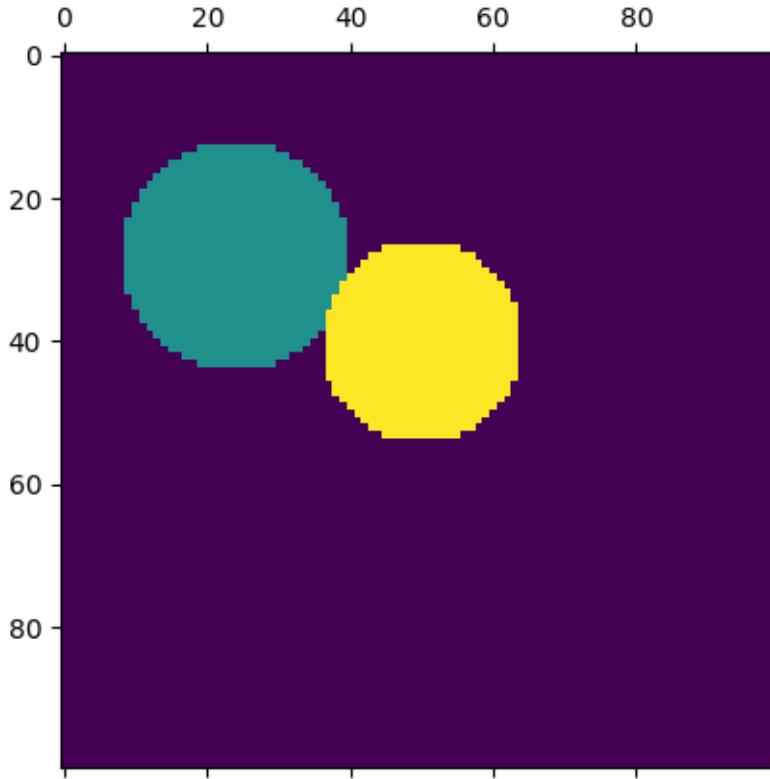
In addition, as there is no useful information in the intensity of the image, or its gradient, we choose to perform the spectral clustering on a graph that is only weakly informed by the gradient. This is close to performing a Voronoi partition of the graph.

In addition, we use the mask of the objects to restrict the graph to the outline of the objects. In this example, we are interested in separating the objects one from the other, and not from the background.









```

print(__doc__)

# Authors: Emmanuelle Gouillart <emmanuelle.gouillart@normalesup.org>
#          Gael Varoquaux <gael.varoquaux@normalesup.org>
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.feature_extraction import image
from sklearn.cluster import spectral_clustering

l = 100
x, y = np.indices((l, l))

center1 = (28, 24)
center2 = (40, 50)
center3 = (67, 58)
center4 = (24, 70)

radius1, radius2, radius3, radius4 = 16, 14, 15, 14

circle1 = (x - center1[0]) ** 2 + (y - center1[1]) ** 2 < radius1 ** 2
circle2 = (x - center2[0]) ** 2 + (y - center2[1]) ** 2 < radius2 ** 2
circle3 = (x - center3[0]) ** 2 + (y - center3[1]) ** 2 < radius3 ** 2
circle4 = (x - center4[0]) ** 2 + (y - center4[1]) ** 2 < radius4 ** 2

```

```
# ######
# 4 circles
img = circle1 + circle2 + circle3 + circle4

# We use a mask that limits to the foreground: the problem that we are
# interested in here is not separating the objects from the background,
# but separating them one from the other.
mask = img.astype(bool)

img = img.astype(float)
img += 1 + 0.2 * np.random.randn(*img.shape)

# Convert the image into a graph with the value of the gradient on the
# edges.
graph = image.img_to_graph(img, mask=mask)

# Take a decreasing function of the gradient: we take it weakly
# dependent from the gradient the segmentation is close to a voronoi
graph.data = np.exp(-graph.data / graph.data.std())

# Force the solver to be arpack, since amg is numerically
# unstable on this example
labels = spectral_clustering(graph, n_clusters=4, eigen_solver='arpack')
label_im = np.full(mask.shape, -1.)
label_im[mask] = labels

plt.matshow(img)
plt.matshow(label_im)

# #####
# 2 circles
img = circle1 + circle2
mask = img.astype(bool)
img = img.astype(float)

img += 1 + 0.2 * np.random.randn(*img.shape)

graph = image.img_to_graph(img, mask=mask)
graph.data = np.exp(-graph.data / graph.data.std())

labels = spectral_clustering(graph, n_clusters=2, eigen_solver='arpack')
label_im = np.full(mask.shape, -1.)
label_im[mask] = labels

plt.matshow(img)
plt.matshow(label_im)

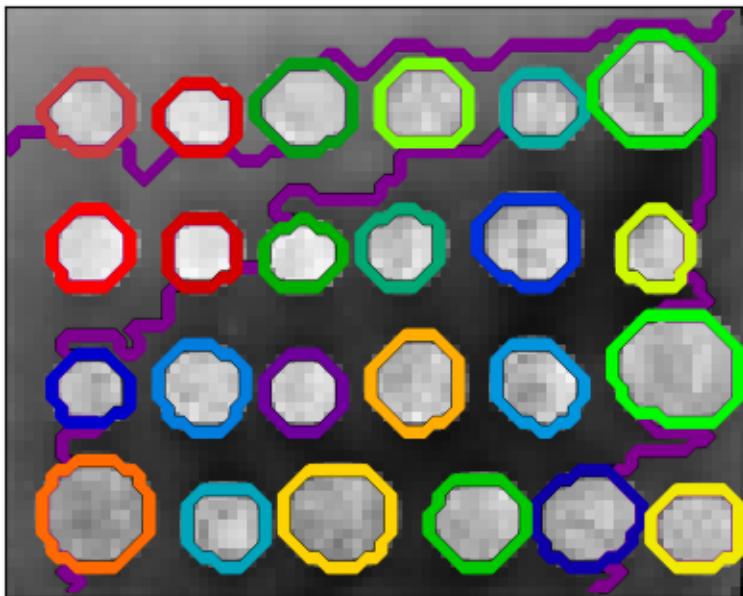
plt.show()
```

Total running time of the script: (0 minutes 0.675 seconds)

Note: Click [here](#) to download the full example code

5.6.12 A demo of structured Ward hierarchical clustering on an image of coins

Compute the segmentation of a 2D image with Ward hierarchical clustering. The clustering is spatially constrained in order for each segmented region to be in one piece.



Out:

```
Compute structured hierarchical clustering...
Elapsed time: 0.2273859977722168
Number of pixels: 4697
Number of clusters: 27
```

```
# Author : Vincent Michel, 2010
#          Alexandre Gramfort, 2011
# License: BSD 3 clause

print(__doc__)

import time as time
```

```
import numpy as np
from distutils.version import LooseVersion
from scipy.ndimage.filters import gaussian_filter

import matplotlib.pyplot as plt

import skimage
from skimage.data import coins
from skimage.transform import rescale

from sklearn.feature_extraction.image import grid_to_graph
from sklearn.cluster import AgglomerativeClustering

# these were introduced in skimage-0.14
if LooseVersion(skimage.__version__) >= '0.14':
    rescale_params = {'anti_aliasing': False, 'multichannel': False}
else:
    rescale_params = {}

#####
# Generate data
orig_coins = coins()

# Resize it to 20% of the original size to speed up the processing
# Applying a Gaussian filter for smoothing prior to down-scaling
# reduces aliasing artifacts.
smoothed_coins = gaussian_filter(orig_coins, sigma=2)
rescaled_coins = rescale(smoothed_coins, 0.2, mode="reflect",
                        **rescale_params)

X = np.reshape(rescaled_coins, (-1, 1))

#####
# Define the structure A of the data. Pixels connected to their neighbors.
connectivity = grid_to_graph(*rescaled_coins.shape)

#####
# Compute clustering
print("Compute structured hierarchical clustering...")
st = time.time()
n_clusters = 27 # number of regions
ward = AgglomerativeClustering(n_clusters=n_clusters, linkage='ward',
                               connectivity=connectivity)
ward.fit(X)
label = np.reshape(ward.labels_, rescaled_coins.shape)
print("Elapsed time: ", time.time() - st)
print("Number of pixels: ", label.size)
print("Number of clusters: ", np.unique(label).size)

#####
# Plot the results on an image
plt.figure(figsize=(5, 5))
plt.imshow(rescaled_coins, cmap=plt.cm.gray)
for l in range(n_clusters):
    plt.contour(label == l,
                colors=[plt.cm.nipy_spectral(l / float(n_clusters)), ])
plt.xticks(())
plt.yticks(())
```

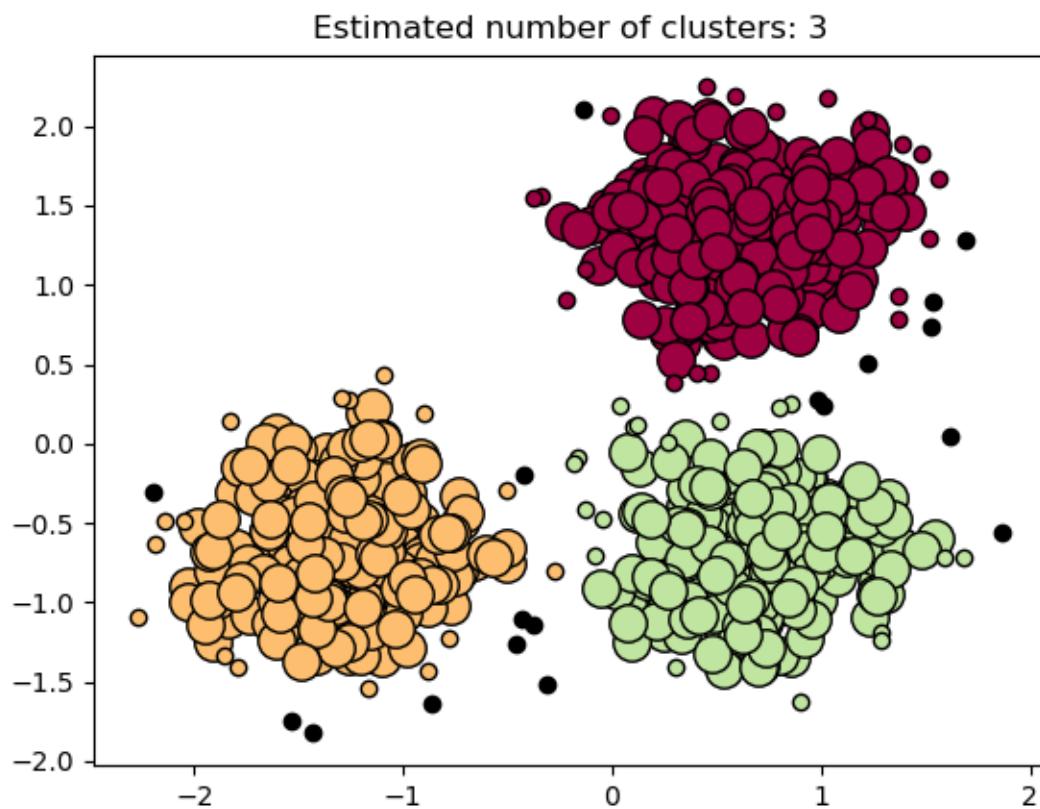
```
plt.show()
```

Total running time of the script: (0 minutes 0.897 seconds)

Note: Click [here](#) to download the full example code

5.6.13 Demo of DBSCAN clustering algorithm

Finds core samples of high density and expands clusters from them.



Out:

```
Estimated number of clusters: 3
Estimated number of noise points: 18
Homogeneity: 0.953
Completeness: 0.883
V-measure: 0.917
Adjusted Rand Index: 0.952
Adjusted Mutual Information: 0.916
Silhouette Coefficient: 0.626
```

```

print(__doc__)

import numpy as np

from sklearn.cluster import DBSCAN
from sklearn import metrics
from sklearn.datasets.samples_generator import make_blobs
from sklearn.preprocessing import StandardScaler


# ##########
# Generate sample data
centers = [[1, 1], [-1, -1], [1, -1]]
X, labels_true = make_blobs(n_samples=750, centers=centers, cluster_std=0.4,
                            random_state=0)

X = StandardScaler().fit_transform(X)

# #####
# Compute DBSCAN
db = DBSCAN(eps=0.3, min_samples=10).fit(X)
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_

# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).count(-1)

print('Estimated number of clusters: %d' % n_clusters_)
print('Estimated number of noise points: %d' % n_noise_)
print("Homogeneity: %0.3f" % metrics.homogeneity_score(labels_true, labels))
print("Completeness: %0.3f" % metrics.completeness_score(labels_true, labels))
print("V-measure: %0.3f" % metrics.v_measure_score(labels_true, labels))
print("Adjusted Rand Index: %0.3f"
      % metrics.adjusted_rand_score(labels_true, labels))
print("Adjusted Mutual Information: %0.3f"
      % metrics.adjusted_mutual_info_score(labels_true, labels,
                                             average_method='arithmetic'))
print("Silhouette Coefficient: %0.3f"
      % metrics.silhouette_score(X, labels))

# #####
# Plot result
import matplotlib.pyplot as plt

# Black removed and is used for noise instead.
unique_labels = set(labels)
colors = [plt.cm.Spectral(each)
          for each in np.linspace(0, 1, len(unique_labels))]
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

```

```

class_member_mask = (labels == k)

xy = X[class_member_mask & core_samples_mask]
plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
          markeredgecolor='k', markersize=14)

xy = X[class_member_mask & ~core_samples_mask]
plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
          markeredgecolor='k', markersize=6)

plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()

```

Total running time of the script: (0 minutes 0.043 seconds)

Note: Click [here](#) to download the full example code

5.6.14 Color Quantization using K-Means

Performs a pixel-wise Vector Quantization (VQ) of an image of the summer palace (China), reducing the number of colors required to show the image from 96,615 unique colors to 64, while preserving the overall appearance quality.

In this example, pixels are represented in a 3D-space and K-means is used to find 64 color clusters. In the image processing literature, the codebook obtained from K-means (the cluster centers) is called the color palette. Using a single byte, up to 256 colors can be addressed, whereas an RGB encoding requires 3 bytes per pixel. The GIF file format, for example, uses such a palette.

For comparison, a quantized image using a random codebook (colors picked up randomly) is also shown.

Original image (96,615 colors)



Quantized image (64 colors, K-Means)



Quantized image (64 colors, Random)



•

Out:

```
Fitting model on a small sub-sample of the data
done in 0.264s.
Predicting color indices on the full image (k-means)
done in 0.216s.
Predicting color indices on the full image (random)
done in 0.310s.
```

```
# Authors: Robert Layton <robertlayton@gmail.com>
#          Olivier Grisel <olivier.grisel@ensta.org>
#          Mathieu Blondel <mathieu@mblondel.org>
#
# License: BSD 3 clause

print(__doc__)
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics import pairwise_distances_argmin
from sklearn.datasets import load_sample_image
from sklearn.utils import shuffle
```

```

from time import time

n_colors = 64

# Load the Summer Palace photo
china = load_sample_image("china.jpg")

# Convert to floats instead of the default 8 bits integer coding. Dividing by
# 255 is important so that plt.imshow behaves works well on float data (need to
# be in the range [0-1])
china = np.array(china, dtype=np.float64) / 255

# Load Image and transform to a 2D numpy array.
w, h, d = original_shape = tuple(china.shape)
assert d == 3
image_array = np.reshape(china, (w * h, d))

print("Fitting model on a small sub-sample of the data")
t0 = time()
image_array_sample = shuffle(image_array, random_state=0)[:1000]
kmeans = KMeans(n_clusters=n_colors, random_state=0).fit(image_array_sample)
print("done in %0.3fs." % (time() - t0))

# Get labels for all points
print("Predicting color indices on the full image (k-means)")
t0 = time()
labels = kmeans.predict(image_array)
print("done in %0.3fs." % (time() - t0))

codebook_random = shuffle(image_array, random_state=0)[:n_colors]
print("Predicting color indices on the full image (random)")
t0 = time()
labels_random = pairwise_distances_argmin(codebook_random,
                                            image_array,
                                            axis=0)
print("done in %0.3fs." % (time() - t0))

def recreate_image(codebook, labels, w, h):
    """Recreate the (compressed) image from the code book & labels"""
    d = codebook.shape[1]
    image = np.zeros((w, h, d))
    label_idx = 0
    for i in range(w):
        for j in range(h):
            image[i][j] = codebook[labels[label_idx]]
            label_idx += 1
    return image

# Display all results, alongside original image
plt.figure(1)
plt.clf()
plt.axis('off')
plt.title('Original image (96,615 colors)')
plt.imshow(china)

plt.figure(2)

```

```
plt.clf()
plt.axis('off')
plt.title('Quantized image (64 colors, K-Means)')
plt.imshow(recreate_image(kmeans.cluster_centers_, labels, w, h))

plt.figure(3)
plt.clf()
plt.axis('off')
plt.title('Quantized image (64 colors, Random)')
plt.imshow(recreate_image(codebook_random, labels_random, w, h))
plt.show()
```

Total running time of the script: (0 minutes 1.400 seconds)

Note: Click [here](#) to download the full example code

5.6.15 Hierarchical clustering: structured vs unstructured ward

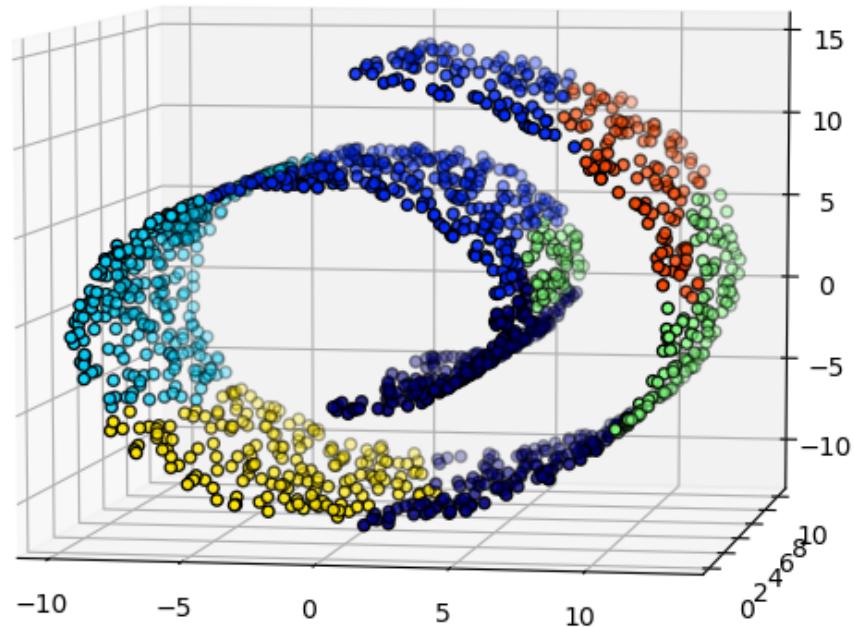
Example builds a swiss roll dataset and runs hierarchical clustering on their position.

For more information, see [Hierarchical clustering](#).

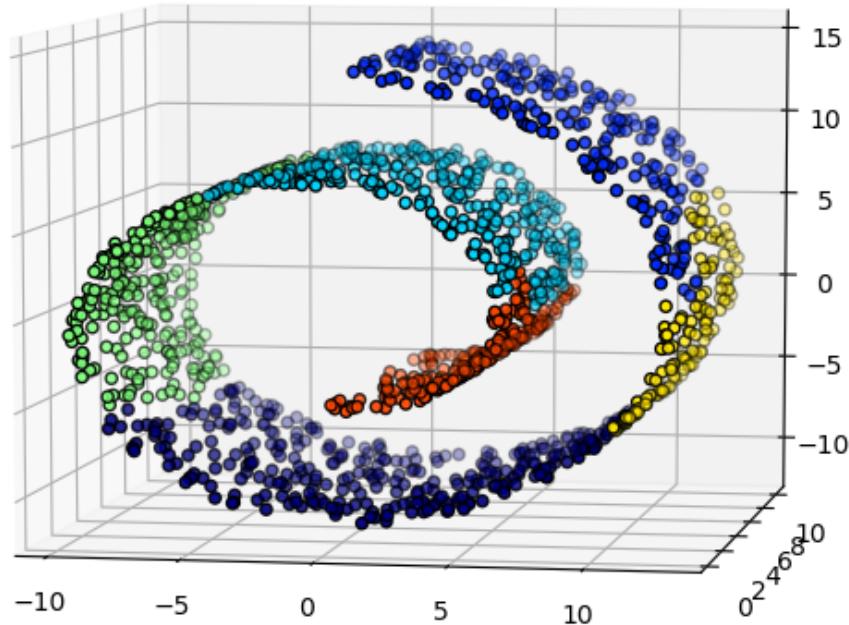
In a first step, the hierarchical clustering is performed without connectivity constraints on the structure and is solely based on distance, whereas in a second step the clustering is restricted to the k-Nearest Neighbors graph: it's a hierarchical clustering with structure prior.

Some of the clusters learned without connectivity constraints do not respect the structure of the swiss roll and extend across different folds of the manifolds. On the opposite, when opposing connectivity constraints, the clusters form a nice parcellation of the swiss roll.

Without connectivity constraints (time 0.08s)



With connectivity constraints (time 0.07s)



Out:

```
Compute unstructured hierarchical clustering...
Elapsed time: 0.08s
Number of points: 1500
Compute structured hierarchical clustering...
Elapsed time: 0.07s
Number of points: 1500
```

```
# Authors : Vincent Michel, 2010
#           Alexandre Gramfort, 2010
#           Gael Varoquaux, 2010
# License: BSD 3 clause

print(__doc__)

import time as time
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as p3
from sklearn.cluster import AgglomerativeClustering
from sklearn.datasets.samples_generator import make_swiss_roll
```

```

# ##########
# Generate data (swiss roll dataset)
n_samples = 1500
noise = 0.05
X, _ = make_swiss_roll(n_samples, noise)
# Make it thinner
X[:, 1] *= .5

# #####
# Compute clustering
print("Compute unstructured hierarchical clustering...")
st = time.time()
ward = AgglomerativeClustering(n_clusters=6, linkage='ward').fit(X)
elapsed_time = time.time() - st
label = ward.labels_
print("Elapsed time: %.2fs" % elapsed_time)
print("Number of points: %i" % label.size)

# #####
# Plot result
fig = plt.figure()
ax = p3.Axes3D(fig)
ax.view_init(7, -80)
for l in np.unique(label):
    ax.scatter(X[label == l, 0], X[label == l, 1], X[label == l, 2],
               color=plt.cm.jet(np.float(l) / np.max(label + 1)),
               s=20, edgecolor='k')
plt.title('Without connectivity constraints (time %.2fs)' % elapsed_time)

# #####
# Define the structure A of the data. Here a 10 nearest neighbors
from sklearn.neighbors import kneighbors_graph
connectivity = kneighbors_graph(X, n_neighbors=10, include_self=False)

# #####
# Compute clustering
print("Compute structured hierarchical clustering...")
st = time.time()
ward = AgglomerativeClustering(n_clusters=6, connectivity=connectivity,
                                linkage='ward').fit(X)
elapsed_time = time.time() - st
label = ward.labels_
print("Elapsed time: %.2fs" % elapsed_time)
print("Number of points: %i" % label.size)

# #####
# Plot result
fig = plt.figure()
ax = p3.Axes3D(fig)
ax.view_init(7, -80)
for l in np.unique(label):
    ax.scatter(X[label == l, 0], X[label == l, 1], X[label == l, 2],
               color=plt.cm.jet(float(l) / np.max(label + 1)),
               s=20, edgecolor='k')
plt.title('With connectivity constraints (time %.2fs)' % elapsed_time)

```

```
plt.show()
```

Total running time of the script: (0 minutes 0.202 seconds)

Note: Click [here](#) to download the full example code

5.6.16 Agglomerative clustering with different metrics

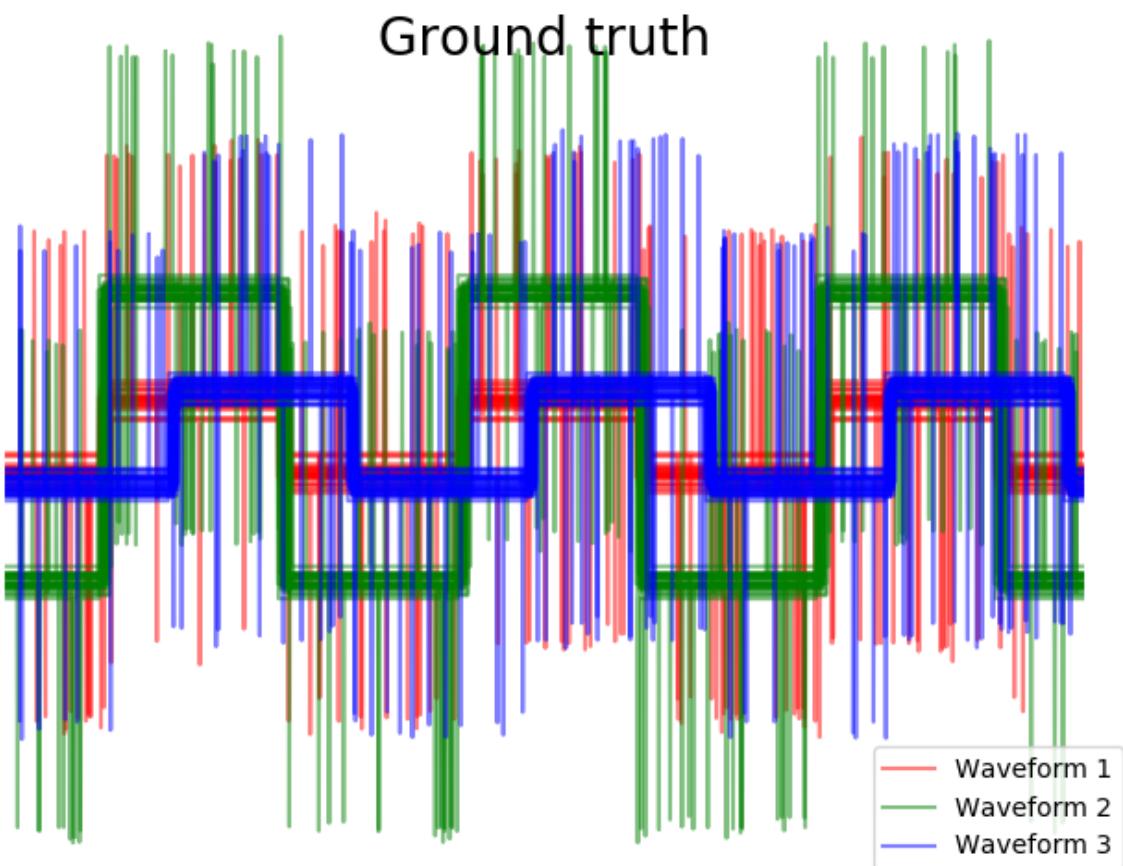
Demonstrates the effect of different metrics on the hierarchical clustering.

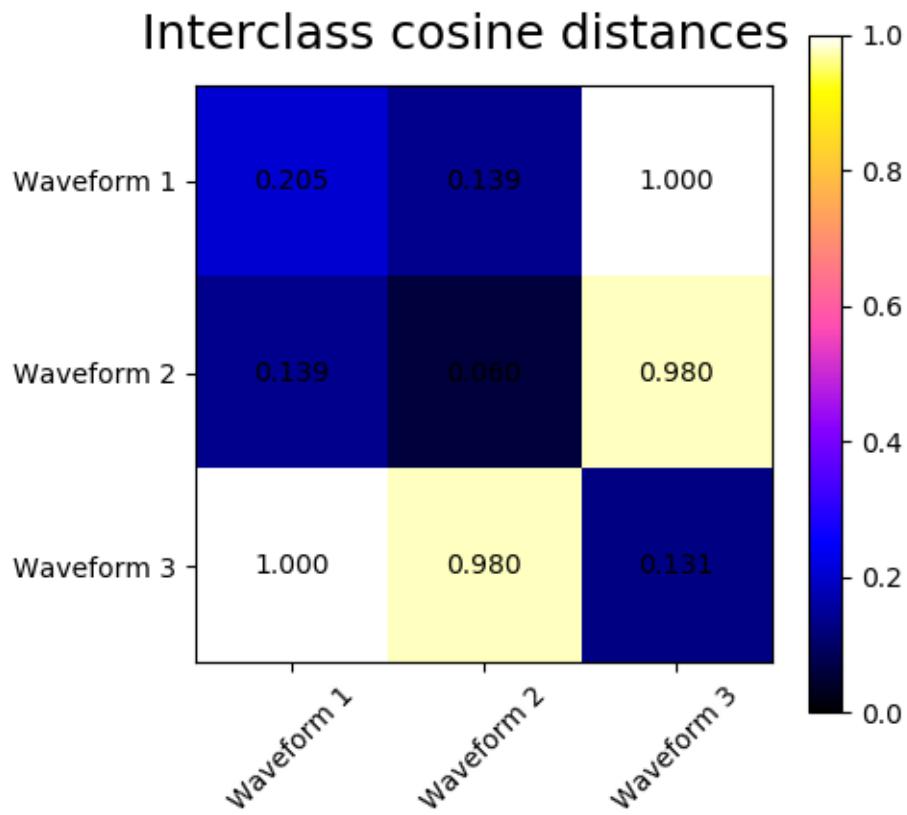
The example is engineered to show the effect of the choice of different metrics. It is applied to waveforms, which can be seen as high-dimensional vector. Indeed, the difference between metrics is usually more pronounced in high dimension (in particular for euclidean and cityblock).

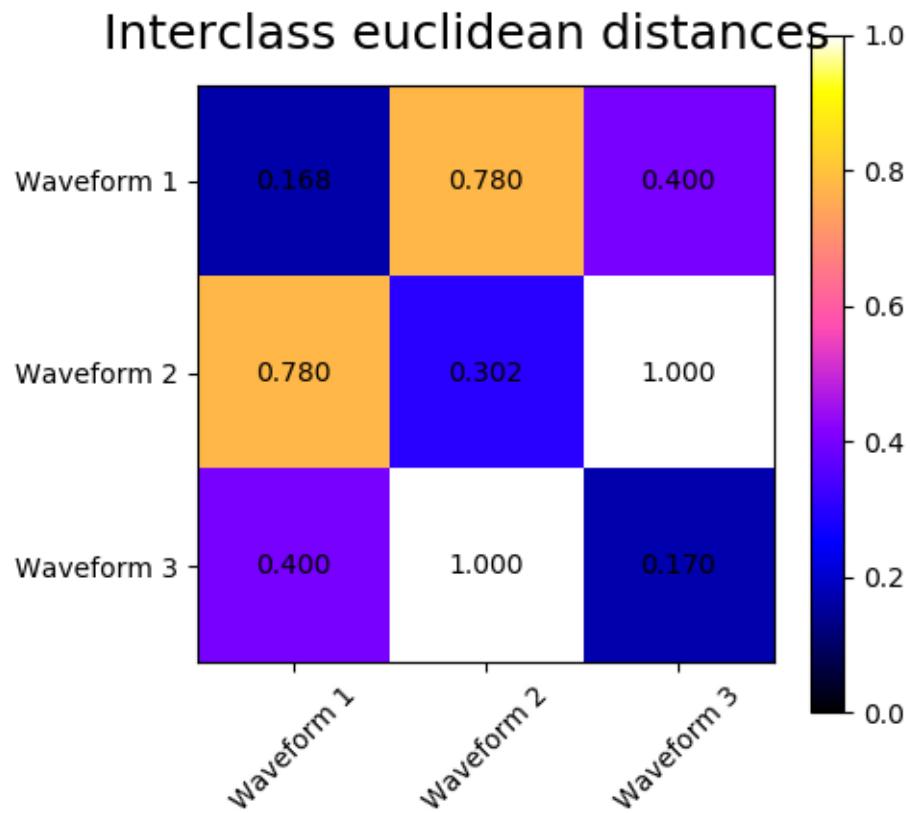
We generate data from three groups of waveforms. Two of the waveforms (waveform 1 and waveform 2) are proportional one to the other. The cosine distance is invariant to a scaling of the data, as a result, it cannot distinguish these two waveforms. Thus even with no noise, clustering using this distance will not separate out waveform 1 and 2.

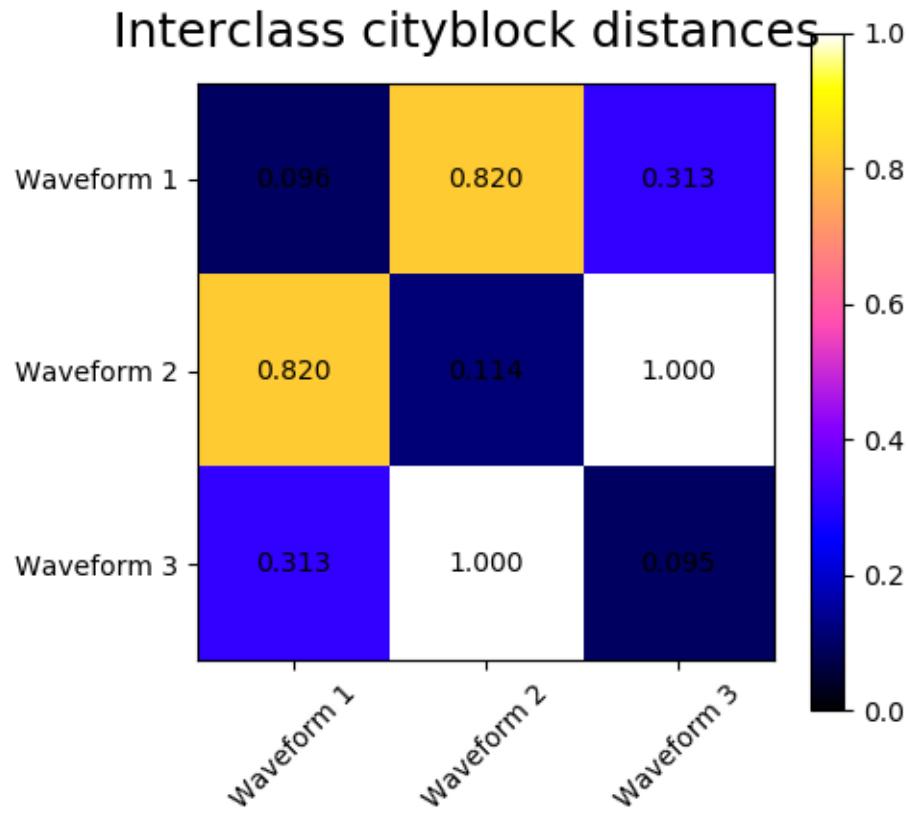
We add observation noise to these waveforms. We generate very sparse noise: only 6% of the time points contain noise. As a result, the l_1 norm of this noise (ie “cityblock” distance) is much smaller than its l_2 norm (“euclidean” distance). This can be seen on the inter-class distance matrices: the values on the diagonal, that characterize the spread of the class, are much bigger for the Euclidean distance than for the cityblock distance.

When we apply clustering to the data, we find that the clustering reflects what was in the distance matrices. Indeed, for the Euclidean distance, the classes are ill-separated because of the noise, and thus the clustering does not separate the waveforms. For the cityblock distance, the separation is good and the waveform classes are recovered. Finally, the cosine distance does not separate at all waveform 1 and 2, thus the clustering puts them in the same cluster.

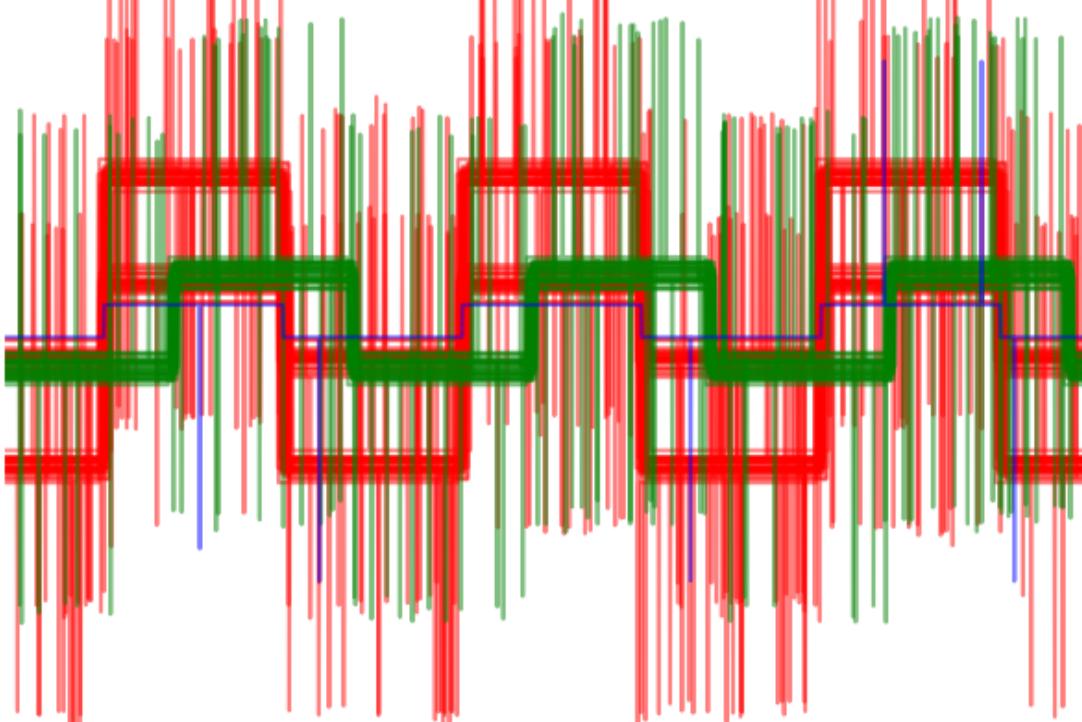




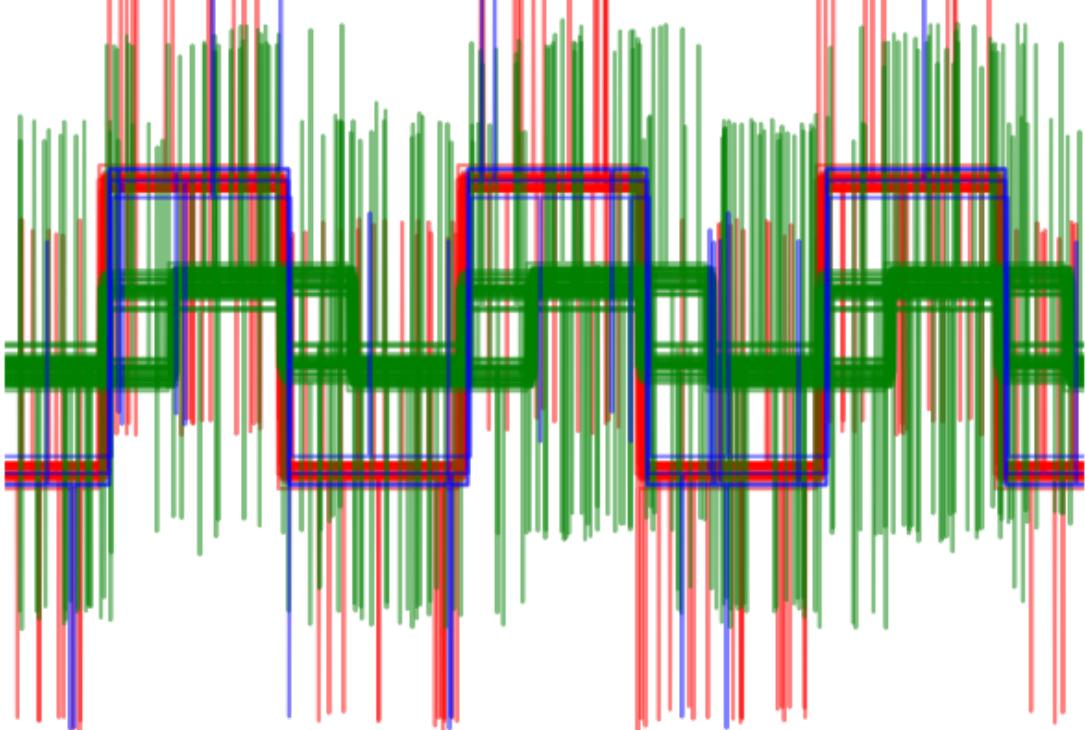




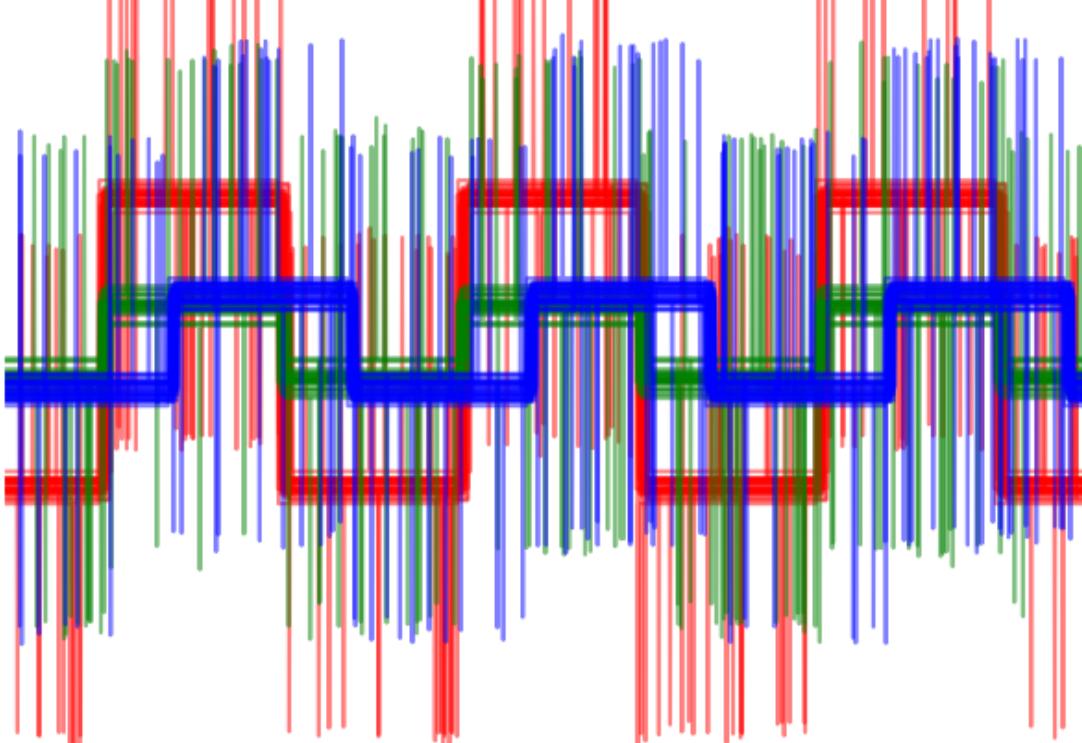
AgglomerativeClustering(affinity=cosine)



AgglomerativeClustering(affinity=euclidean)



AgglomerativeClustering(affinity=cityblock)



```
# Author: Gael Varoquaux
# License: BSD 3-Clause or CC-0

import matplotlib.pyplot as plt
import numpy as np

from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics import pairwise_distances

np.random.seed(0)

# Generate waveform data
n_features = 2000
t = np.pi * np.linspace(0, 1, n_features)

def sqr(x):
    return np.sign(np.cos(x))

X = list()
y = list()
for i, (phi, a) in enumerate([(0.5, 0.15), (0.5, 0.6), (0.3, 0.2)]):
    for _ in range(30):
        phase_noise = .01 * np.random.normal()
        amplitude_noise = .04 * np.random.normal()
        additional_noise = 1 - 2 * np.random.rand(n_features)
        # Make the noise sparse
```

```

additional_noise[np.abs(additional_noise) < .997] = 0

X.append(12 * ((a + amplitude_noise)
               * (sqr(6 * (t + phi + phase_noise))) +
               additional_noise))
y.append(i)

X = np.array(X)
y = np.array(y)

n_clusters = 3

labels = ('Waveform 1', 'Waveform 2', 'Waveform 3')

# Plot the ground-truth labelling
plt.figure()
plt.axes([0, 0, 1, 1])
for l, c, n in zip(range(n_clusters), 'rgb',
                    labels):
    lines = plt.plot(X[y == l].T, c=c, alpha=.5)
    lines[0].set_label(n)

plt.legend(loc='best')

plt.axis('tight')
plt.axis('off')
plt.suptitle("Ground truth", size=20)

# Plot the distances
for index, metric in enumerate(["cosine", "euclidean", "cityblock"]):
    avg_dist = np.zeros((n_clusters, n_clusters))
    plt.figure(figsize=(5, 4.5))
    for i in range(n_clusters):
        for j in range(n_clusters):
            avg_dist[i, j] = pairwise_distances(X[y == i], X[y == j],
                                                metric=metric).mean()

    avg_dist /= avg_dist.max()
    for i in range(n_clusters):
        for j in range(n_clusters):
            plt.text(i, j, '%.3f' % avg_dist[i, j],
                     verticalalignment='center',
                     horizontalalignment='center')

    plt.imshow(avg_dist, interpolation='nearest', cmap=plt.cm.gnuplot2,
               vmin=0)
    plt.xticks(range(n_clusters), labels, rotation=45)
    plt.yticks(range(n_clusters), labels)
    plt.colorbar()
    plt.suptitle("Interclass %s distances" % metric, size=18)
    plt.tight_layout()

# Plot clustering results
for index, metric in enumerate(["cosine", "euclidean", "cityblock"]):
    model = AgglomerativeClustering(n_clusters=n_clusters,
                                     linkage="average", affinity=metric)
    model.fit(X)

```

```

plt.figure()
plt.axes([0, 0, 1, 1])
for l, c in zip(np.arange(model.n_clusters), 'rgbk'):
    plt.plot(X[model.labels_ == l].T, c=c, alpha=.5)
plt.axis('tight')
plt.axis('off')
plt.suptitle("AgglomerativeClustering(affinity=%s)" % metric, size=20)

plt.show()

```

Total running time of the script: (0 minutes 0.584 seconds)

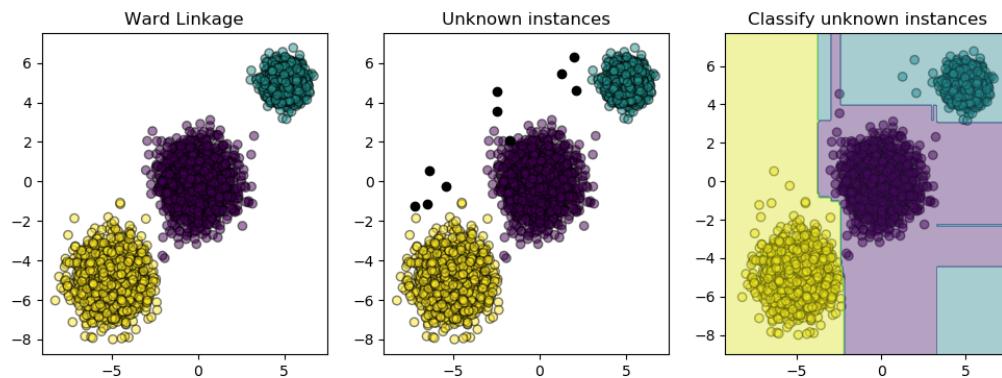
Note: Click [here](#) to download the full example code

5.6.17 Inductive Clustering

Clustering can be expensive, especially when our dataset contains millions of datapoints. Many clustering algorithms are not *inductive* and so cannot be directly applied to new data samples without recomputing the clustering, which may be intractable. Instead, we can use clustering to then learn an inductive model with a classifier, which has several benefits:

- it allows the clusters to scale and apply to new data
- unlike re-fitting the clusters to new samples, it makes sure the labelling procedure is consistent over time
- it allows us to use the inferential capabilities of the classifier to describe or explain the clusters

This example illustrates a generic implementation of a meta-estimator which extends clustering by inducing a classifier from the cluster labels.



```

# Authors: Chirag Nagpal
#          Christos Aridas
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.base import BaseEstimator, clone
from sklearn.cluster import AgglomerativeClustering
from sklearn.datasets import make_blobs
from sklearn.ensemble import RandomForestClassifier

```

```

from sklearn.utils.metaestimators import if_delegate_has_method


N_SAMPLES = 5000
RANDOM_STATE = 42


class InductiveClusterer(BaseEstimator):
    def __init__(self, clusterer, classifier):
        self.clusterer = clusterer
        self.classifier = classifier

    def fit(self, X, y=None):
        self.clusterer_ = clone(self.clusterer)
        self.classifier_ = clone(self.classifier)
        y = self.clusterer_.fit_predict(X)
        self.classifier_.fit(X, y)
        return self

    @if_delegate_has_method(delegate='classifier_')
    def predict(self, X):
        return self.classifier_.predict(X)

    @if_delegate_has_method(delegate='classifier_')
    def decision_function(self, X):
        return self.classifier_.decision_function(X)

    def plot_scatter(X, color, alpha=0.5):
        return plt.scatter(X[:, 0],
                           X[:, 1],
                           c=color,
                           alpha=alpha,
                           edgecolor='k')

# Generate some training data from clustering
X, y = make_blobs(n_samples=N_SAMPLES,
                   cluster_std=[1.0, 1.0, 0.5],
                   centers=[(-5, -5), (0, 0), (5, 5)],
                   random_state=RANDOM_STATE)

# Train a clustering algorithm on the training data and get the cluster labels
clusterer = AgglomerativeClustering(n_clusters=3)
cluster_labels = clusterer.fit_predict(X)

plt.figure(figsize=(12, 4))

plt.subplot(131)
plot_scatter(X, cluster_labels)
plt.title("Ward Linkage")

# Generate new samples and plot them along with the original dataset
X_new, y_new = make_blobs(n_samples=10,
                           centers=[(-7, -1), (-2, 4), (3, 6)],
                           random_state=RANDOM_STATE)

```

```

plt.subplot(132)
plot_scatter(X, cluster_labels)
plot_scatter(X_new, 'black', 1)
plt.title("Unknown instances")

# Declare the inductive learning model that it will be used to
# predict cluster membership for unknown instances
classifier = RandomForestClassifier(random_state=RANDOM_STATE)
inductive_learner = InductiveClusterer(clusterer, classifier).fit(X)

probable_clusters = inductive_learner.predict(X_new)

plt.subplot(133)
plot_scatter(X, cluster_labels)
plot_scatter(X_new, probable_clusters)

# Plotting decision regions
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

Z = inductive_learner.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.4)
plt.title("Classify unknown instances")

plt.show()

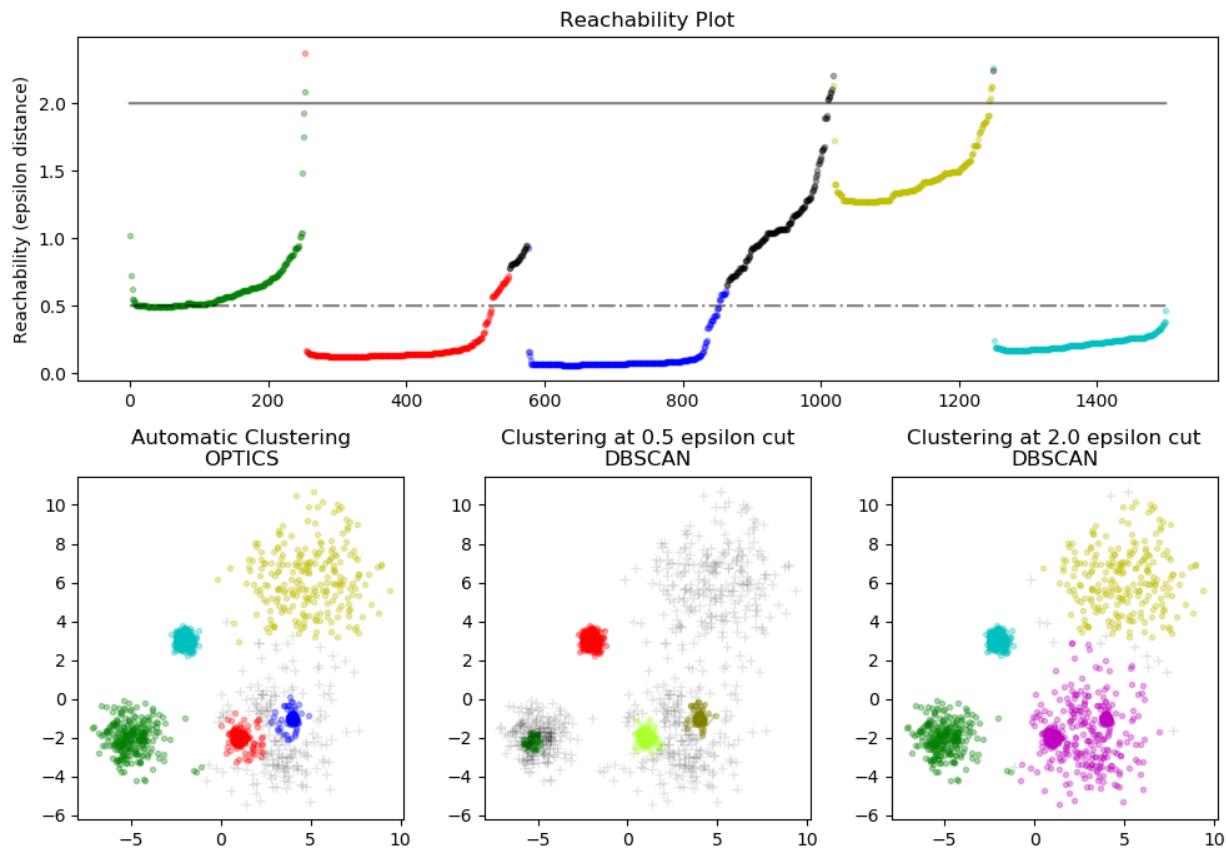
```

Total running time of the script: (0 minutes 1.436 seconds)

Note: Click [here](#) to download the full example code

5.6.18 Demo of OPTICS clustering algorithm

Finds core samples of high density and expands clusters from them. This example uses data that is generated so that the clusters have different densities. The `sklearn.cluster.OPTICS` is first used with its `Xi` cluster detection method, and then setting specific thresholds on the reachability, which corresponds to `sklearn.cluster.DBSCAN`. We can see that the different clusters of OPTICS's `Xi` method can be recovered with different choices of thresholds in DBSCAN.



```

# Authors: Shane Grigsby <refuge@rocktalus.com>
#          Adrin Jalali <adrin.jalali@gmail.com>
# License: BSD 3 clause

from sklearn.cluster import OPTICS, cluster_optics_dbscan
import matplotlib.gridspec as gridspec
import matplotlib.pyplot as plt
import numpy as np

# Generate sample data

np.random.seed(0)
n_points_per_cluster = 250

C1 = [-5, -2] + .8 * np.random.randn(n_points_per_cluster, 2)
C2 = [4, -1] + .1 * np.random.randn(n_points_per_cluster, 2)
C3 = [1, -2] + .2 * np.random.randn(n_points_per_cluster, 2)
C4 = [-2, 3] + .3 * np.random.randn(n_points_per_cluster, 2)
C5 = [3, -2] + 1.6 * np.random.randn(n_points_per_cluster, 2)
C6 = [5, 6] + 2 * np.random.randn(n_points_per_cluster, 2)
X = np.vstack((C1, C2, C3, C4, C5, C6))

clust = OPTICS(min_samples=50, xi=.05, min_cluster_size=.05)

# Run the fit
clust.fit(X)

```

```

labels_050 = cluster_optics_dbSCAN(reachability=clust.reachability_,
                                    core_distances=clust.core_distances_,
                                    ordering=clust.ordering_, eps=0.5)
labels_200 = cluster_optics_dbSCAN(reachability=clust.reachability_,
                                    core_distances=clust.core_distances_,
                                    ordering=clust.ordering_, eps=2)

space = np.arange(len(X))
reachability = clust.reachability_[clust.ordering_]
labels = clust.labels_[clust.ordering_]

plt.figure(figsize=(10, 7))
G = gridspec.GridSpec(2, 3)
ax1 = plt.subplot(G[0, :])
ax2 = plt.subplot(G[1, 0])
ax3 = plt.subplot(G[1, 1])
ax4 = plt.subplot(G[1, 2])

# Reachability plot
colors = ['g.', 'r.', 'b.', 'y.', 'c.']
for klass, color in zip(range(0, 5), colors):
    Xk = space[labels == klass]
    Rk = reachability[labels == klass]
    ax1.plot(Xk, Rk, color, alpha=0.3)
ax1.plot(space[labels == -1], reachability[labels == -1], 'k.', alpha=0.3)
ax1.plot(space, np.full_like(space, 2., dtype=float), 'k-', alpha=0.5)
ax1.plot(space, np.full_like(space, 0.5, dtype=float), 'k-.', alpha=0.5)
ax1.set_ylabel('Reachability (epsilon distance)')
ax1.set_title('Reachability Plot')

# OPTICS
colors = ['g.', 'r.', 'b.', 'y.', 'c.']
for klass, color in zip(range(0, 5), colors):
    Xk = X[clust.labels_ == klass]
    ax2.plot(Xk[:, 0], Xk[:, 1], color, alpha=0.3)
ax2.plot(X[clust.labels_ == -1, 0], X[clust.labels_ == -1, 1], 'k+', alpha=0.1)
ax2.set_title('Automatic Clustering\\nOPTICS')

# DBSCAN at 0.5
colors = ['g', 'greenyellow', 'olive', 'r', 'b', 'c']
for klass, color in zip(range(0, 6), colors):
    Xk = X[labels_050 == klass]
    ax3.plot(Xk[:, 0], Xk[:, 1], color, alpha=0.3, marker='.')
ax3.plot(X[labels_050 == -1, 0], X[labels_050 == -1, 1], 'k+', alpha=0.1)
ax3.set_title('Clustering at 0.5 epsilon cut\\nDBSCAN')

# DBSCAN at 2.
colors = ['g.', 'm.', 'y.', 'c.']
for klass, color in zip(range(0, 4), colors):
    Xk = X[labels_200 == klass]
    ax4.plot(Xk[:, 0], Xk[:, 1], color, alpha=0.3)
ax4.plot(X[labels_200 == -1, 0], X[labels_200 == -1, 1], 'k+', alpha=0.1)
ax4.set_title('Clustering at 2.0 epsilon cut\\nDBSCAN')

plt.tight_layout()
plt.show()

```

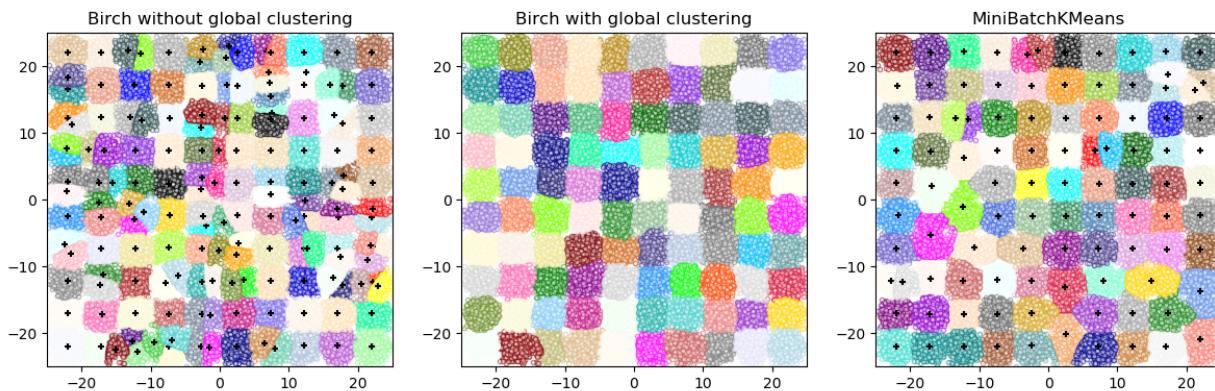
Total running time of the script: (0 minutes 0.935 seconds)

Note: Click [here](#) to download the full example code

5.6.19 Compare BIRCH and MiniBatchKMeans

This example compares the timing of Birch (with and without the global clustering step) and MiniBatchKMeans on a synthetic dataset having 100,000 samples and 2 features generated using make_blobs.

If n_clusters is set to None, the data is reduced from 100,000 samples to a set of 158 clusters. This can be viewed as a preprocessing step before the final (global) clustering step that further reduces these 158 clusters to 100 clusters.



Out:

```
Birch without global clustering as the final step took 2.47 seconds
n_clusters : 158
Birch with global clustering as the final step took 2.90 seconds
n_clusters : 100
Time taken to run MiniBatchKMeans 3.60 seconds
```

```
# Authors: Manoj Kumar <manojkumarsivaraj334@gmail.com>
#          Alexandre Gramfort <alexandre.gramfort@telecom-paristech.fr>
# License: BSD 3 clause

print(__doc__)

from itertools import cycle
from time import time
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors

from sklearn.cluster import Birch, MiniBatchKMeans
from sklearn.datasets.samples_generator import make_blobs

# Generate centers for the blobs so that it forms a 10 X 10 grid.
```

```

xx = np.linspace(-22, 22, 10)
yy = np.linspace(-22, 22, 10)
xx, yy = np.meshgrid(xx, yy)
n_centres = np.hstack((np.ravel(xx)[:, np.newaxis],
                       np.ravel(yy)[:, np.newaxis])))

# Generate blobs to do a comparison between MiniBatchKMeans and Birch.
X, y = make_blobs(n_samples=100000, centers=n_centres, random_state=0)

# Use all colors that matplotlib provides by default.
colors_ = cycle(colors.cnames.keys())

fig = plt.figure(figsize=(12, 4))
fig.subplots_adjust(left=0.04, right=0.98, bottom=0.1, top=0.9)

# Compute clustering with Birch with and without the final clustering step
# and plot.
birch_models = [Birch(threshold=1.7, n_clusters=None),
                Birch(threshold=1.7, n_clusters=100)]
final_step = ['without global clustering', 'with global clustering']

for ind, (birch_model, info) in enumerate(zip(birch_models, final_step)):
    t = time()
    birch_model.fit(X)
    time_ = time() - t
    print("Birch %s as the final step took %0.2f seconds" % (
        info, (time() - t)))

    # Plot result
    labels = birch_model.labels_
    centroids = birch_model.subcluster_centers_
    n_clusters = np.unique(labels).size
    print("n_clusters : %d" % n_clusters)

    ax = fig.add_subplot(1, 3, ind + 1)
    for this_centroid, k, col in zip(centroids, range(n_clusters), colors_):
        mask = labels == k
        ax.scatter(X[mask, 0], X[mask, 1],
                   c='w', edgecolor=col, marker='.', alpha=0.5)
        if birch_model.n_clusters is None:
            ax.scatter(this_centroid[0], this_centroid[1], marker='+',
                       c='k', s=25)
    ax.set_xlim([-25, 25])
    ax.set_ylim([-25, 25])
    ax.set_autoscaley_on(False)
    ax.set_title('Birch %s' % info)

# Compute clustering with MiniBatchKMeans.
mbk = MiniBatchKMeans(init='k-means++', n_clusters=100, batch_size=100,
                      n_init=10, max_no_improvement=10, verbose=0,
                      random_state=0)
t0 = time()
mbk.fit(X)
t_mini_batch = time() - t0
print("Time taken to run MiniBatchKMeans %0.2f seconds" % t_mini_batch)
mbk_means_labels_unique = np.unique(mbk.labels_)

ax = fig.add_subplot(1, 3, 3)

```

```
for this_centroid, k, col in zip(mbk.cluster_centers_,
                                  range(n_clusters), colors_):
    mask = mbk.labels_ == k
    ax.scatter(X[mask, 0], X[mask, 1], marker='.',
               c='w', edgecolor=col, alpha=0.5)
    ax.scatter(this_centroid[0], this_centroid[1], marker='+',
               c='k', s=25)
ax.set_xlim([-25, 25])
ax.set_ylim([-25, 25])
ax.set_title("MiniBatchKMeans")
ax.set_autoscaley_on(False)
plt.show()
```

Total running time of the script: (0 minutes 10.943 seconds)

Note: Click [here](#) to download the full example code

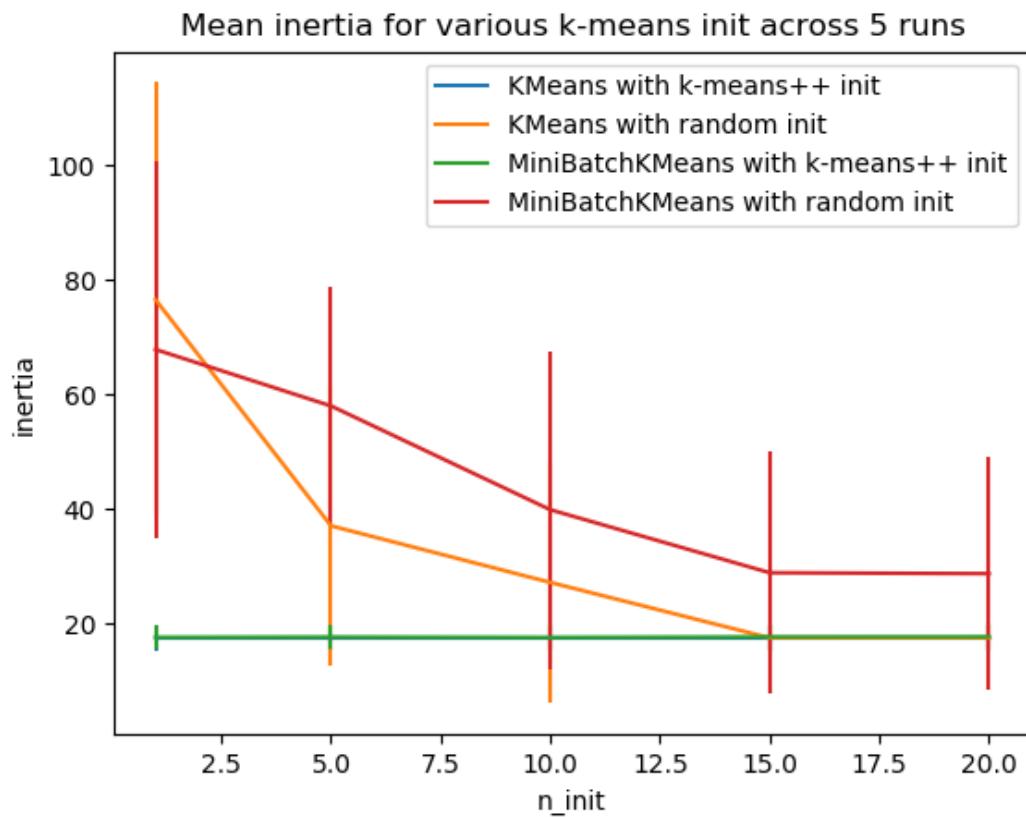
5.6.20 Empirical evaluation of the impact of k-means initialization

Evaluate the ability of k-means initializations strategies to make the algorithm convergence robust as measured by the relative standard deviation of the inertia of the clustering (i.e. the sum of squared distances to the nearest cluster center).

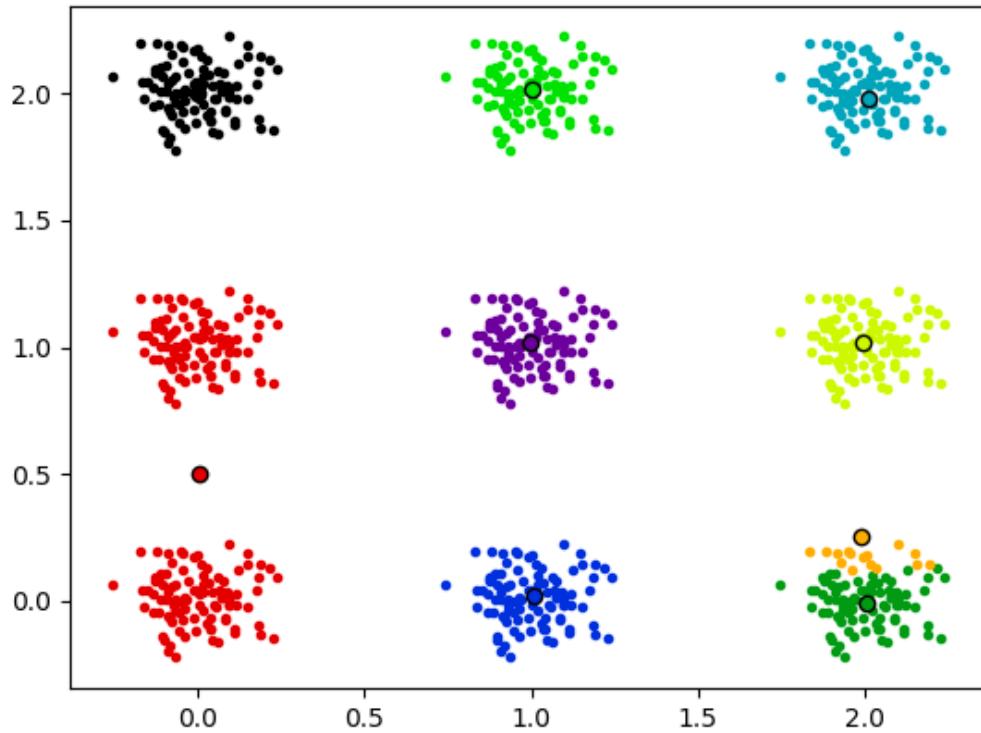
The first plot shows the best inertia reached for each combination of the model (KMeans or MiniBatchKMeans) and the init method (init="random" or init="kmeans++") for increasing values of the n_init parameter that controls the number of initializations.

The second plot demonstrate one single run of the MiniBatchKMeans estimator using a init="random" and n_init=1. This run leads to a bad convergence (local optimum) with estimated centers stuck between ground truth clusters.

The dataset used for evaluation is a 2D grid of isotropic Gaussian clusters widely spaced.



Example cluster allocation with a single random init
with MiniBatchKMeans



Out:

```
Evaluation of KMeans with k-means++ init
Evaluation of KMeans with random init
Evaluation of MiniBatchKMeans with k-means++ init
Evaluation of MiniBatchKMeans with random init
```

```
print(__doc__)

# Author: Olivier Grisel <olivier.grisel@ensta.org>
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

from sklearn.utils import shuffle
from sklearn.utils import check_random_state
from sklearn.cluster import MiniBatchKMeans
from sklearn.cluster import KMeans

random_state = np.random.RandomState(0)
```

```

# Number of run (with randomly generated dataset) for each strategy so as
# to be able to compute an estimate of the standard deviation
n_runs = 5

# k-means models can do several random init so as to be able to trade
# CPU time for convergence robustness
n_init_range = np.array([1, 5, 10, 15, 20])

# Datasets generation parameters
n_samples_per_center = 100
grid_size = 3
scale = 0.1
n_clusters = grid_size ** 2

def make_data(random_state, n_samples_per_center, grid_size, scale):
    random_state = check_random_state(random_state)
    centers = np.array([[i, j]
                        for i in range(grid_size)
                        for j in range(grid_size)])
    n_clusters_true, n_features = centers.shape

    noise = random_state.normal(
        scale=scale, size=(n_samples_per_center, centers.shape[1]))

    X = np.concatenate([c + noise for c in centers])
    y = np.concatenate([[i] * n_samples_per_center
                           for i in range(n_clusters_true)])
    return shuffle(X, y, random_state=random_state)

# Part 1: Quantitative evaluation of various init methods

plt.figure()
plots = []
legends = []

cases = [
    (KMeans, 'k-means++', {}),
    (KMeans, 'random', {}),
    (MiniBatchKMeans, 'k-means++', {'max_no_improvement': 3}),
    (MiniBatchKMeans, 'random', {'max_no_improvement': 3, 'init_size': 500}),
]
]

for factory, init, params in cases:
    print("Evaluation of %s with %s init" % (factory.__name__, init))
    inertia = np.empty((len(n_init_range), n_runs))

    for run_id in range(n_runs):
        X, y = make_data(run_id, n_samples_per_center, grid_size, scale)
        for i, n_init in enumerate(n_init_range):
            km = factory(n_clusters=n_clusters, init=init, random_state=run_id,
                         n_init=n_init, **params).fit(X)
            inertia[i, run_id] = km.inertia_
    p = plt.errorbar(n_init_range, inertia.mean(axis=1), inertia.std(axis=1))
    plots.append(p[0])
    legends.append("%s with %s init" % (factory.__name__, init))

```

```
plt.xlabel('n_init')
plt.ylabel('inertia')
plt.legend(plots, legends)
plt.title("Mean inertia for various k-means init across %d runs" % n_runs)

# Part 2: Qualitative visual inspection of the convergence

X, y = make_data(random_state, n_samples_per_center, grid_size, scale)
km = MiniBatchKMeans(n_clusters=n_clusters, init='random', n_init=1,
                     random_state=random_state).fit(X)

plt.figure()
for k in range(n_clusters):
    my_members = km.labels_ == k
    color = cm.nipy_spectral(float(k) / n_clusters, 1)
    plt.plot(X[my_members, 0], X[my_members, 1], 'o', marker='.', c=color)
    cluster_center = km.cluster_centers_[k]
    plt.plot(cluster_center[0], cluster_center[1], 'o',
              markerfacecolor=color, markeredgecolor='k', markersize=6)
plt.title("Example cluster allocation with a single random init\n"
          "with MiniBatchKMeans")

plt.show()
```

Total running time of the script: (0 minutes 2.695 seconds)

Note: Click [here](#) to download the full example code

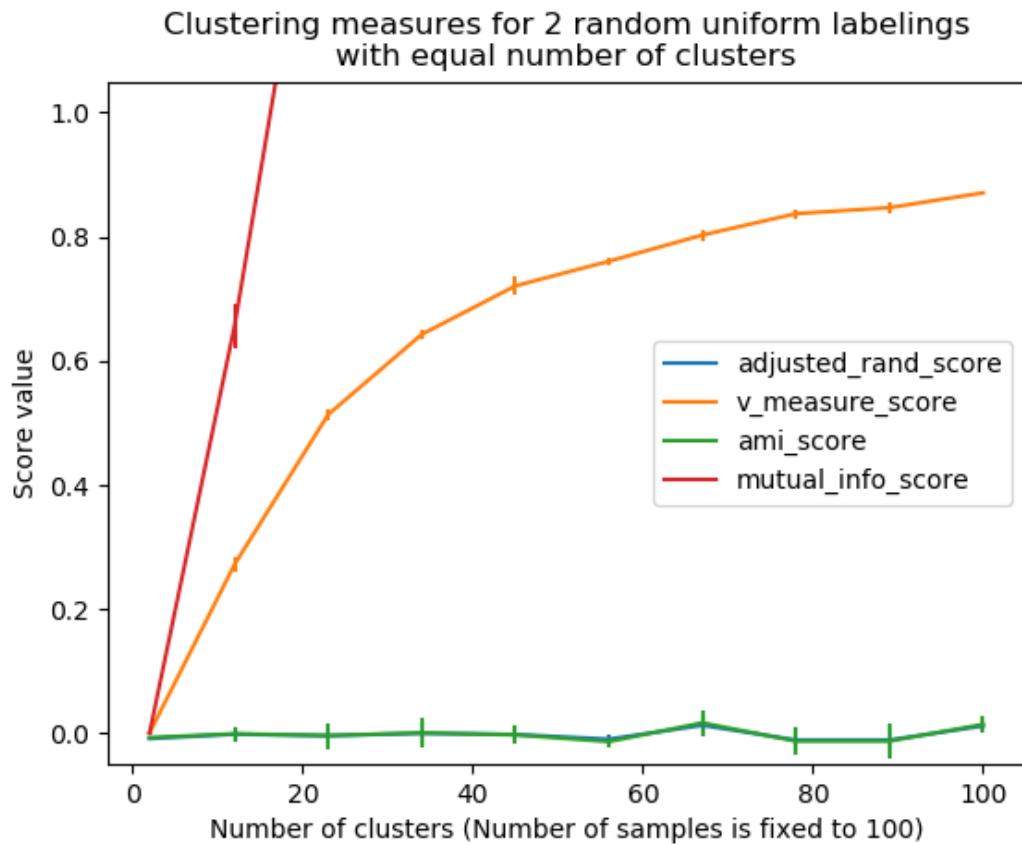
5.6.21 Adjustment for chance in clustering performance evaluation

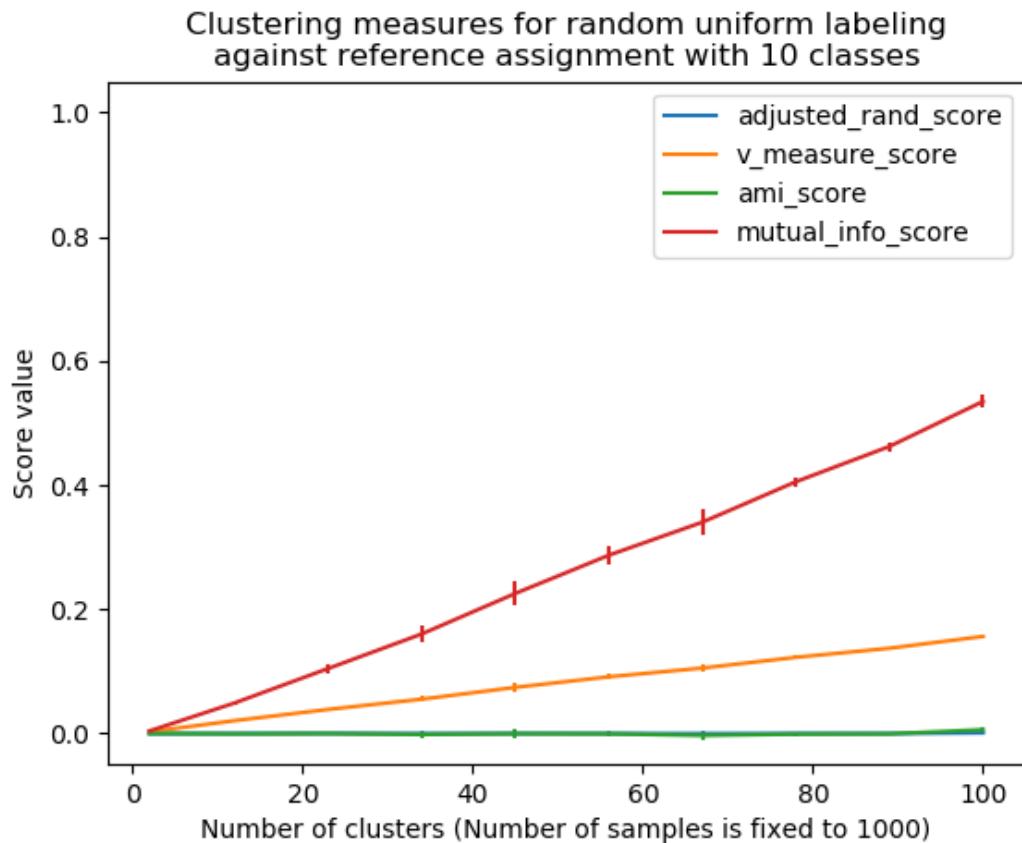
The following plots demonstrate the impact of the number of clusters and number of samples on various clustering performance evaluation metrics.

Non-adjusted measures such as the V-Measure show a dependency between the number of clusters and the number of samples: the mean V-Measure of random labeling increases significantly as the number of clusters is closer to the total number of samples used to compute the measure.

Adjusted for chance measure such as ARI display some random variations centered around a mean score of 0.0 for any number of samples and clusters.

Only adjusted measures can hence safely be used as a consensus index to evaluate the average stability of clustering algorithms for a given value of k on various overlapping sub-samples of the dataset.





Out:

```
Computing adjusted_rand_score for 10 values of n_clusters and n_samples=100
done in 0.026s
Computing v_measure_score for 10 values of n_clusters and n_samples=100
done in 0.040s
Computing ami_score for 10 values of n_clusters and n_samples=100
done in 0.349s
Computing mutual_info_score for 10 values of n_clusters and n_samples=100
done in 0.033s
Computing adjusted_rand_score for 10 values of n_clusters and n_samples=1000
done in 0.041s
Computing v_measure_score for 10 values of n_clusters and n_samples=1000
done in 0.053s
Computing ami_score for 10 values of n_clusters and n_samples=1000
done in 0.208s
Computing mutual_info_score for 10 values of n_clusters and n_samples=1000
done in 0.043s
```

```
print(__doc__)

# Author: Olivier Grisel <olivier.grisel@ensta.org>
```

```
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from time import time
from sklearn import metrics

def uniform_labelings_scores(score_func, n_samples, n_clusters_range,
                             fixed_n_classes=None, n_runs=5, seed=42):
    """Compute score for 2 random uniform cluster labelings.

    Both random labelings have the same number of clusters for each value
    possible value in ``n_clusters_range``.

    When fixed_n_classes is not None the first labeling is considered a ground
    truth class assignment with fixed number of classes.
    """
    random_labels = np.random.RandomState(seed).randint
    scores = np.zeros((len(n_clusters_range), n_runs))

    if fixed_n_classes is not None:
        labels_a = random_labels(low=0, high=fixed_n_classes, size=n_samples)

    for i, k in enumerate(n_clusters_range):
        for j in range(n_runs):
            if fixed_n_classes is None:
                labels_a = random_labels(low=0, high=k, size=n_samples)
                labels_b = random_labels(low=0, high=k, size=n_samples)
                scores[i, j] = score_func(labels_a, labels_b)
            else:
                labels_a = random_labels(low=0, high=fixed_n_classes, size=n_samples)

    return scores

def ami_score(U, V):
    return metrics.adjusted_mutual_info_score(U, V,
                                              average_method='arithmetic')

score_funcs = [
    metrics.adjusted_rand_score,
    metrics.v_measure_score,
    ami_score,
    metrics.mutual_info_score,
]
]

# 2 independent random clusterings with equal cluster number

n_samples = 100
n_clusters_range = np.linspace(2, n_samples, 10).astype(np.int)

plt.figure(1)

plots = []
names = []
for score_func in score_funcs:
    print("Computing %s for %d values of n_clusters and n_samples=%d"
          % (score_func.__name__, len(n_clusters_range), n_samples))

    t0 = time()
    scores = uniform_labelings_scores(score_func, n_samples, n_clusters_range)
```

```
print("done in %0.3fs" % (time() - t0))
plots.append(plt.errorbar(
    n_clusters_range, np.median(scores, axis=1), scores.std(axis=1))[0])
names.append(score_func.__name__)

plt.title("Clustering measures for 2 random uniform labelings\n"
          "with equal number of clusters")
plt.xlabel('Number of clusters (Number of samples is fixed to %d)' % n_samples)
plt.ylabel('Score value')
plt.legend(plots, names)
plt.ylim(bottom=-0.05, top=1.05)

# Random labeling with varying n_clusters against ground class labels
# with fixed number of clusters

n_samples = 1000
n_clusters_range = np.linspace(2, 100, 10).astype(np.int)
n_classes = 10

plt.figure(2)

plots = []
names = []
for score_func in score_funcs:
    print("Computing %s for %d values of n_clusters and n_samples=%d"
          % (score_func.__name__, len(n_clusters_range), n_samples))

    t0 = time()
    scores = uniform_labelings_scores(score_func, n_samples, n_clusters_range,
                                       fixed_n_classes=n_classes)
    print("done in %0.3fs" % (time() - t0))
    plots.append(plt.errorbar(
        n_clusters_range, scores.mean(axis=1), scores.std(axis=1))[0])
    names.append(score_func.__name__)

plt.title("Clustering measures for random uniform labeling\n"
          "against reference assignment with %d classes" % n_classes)
plt.xlabel('Number of clusters (Number of samples is fixed to %d)' % n_samples)
plt.ylabel('Score value')
plt.ylim(bottom=-0.05, top=1.05)
plt.legend(plots, names)
plt.show()
```

Total running time of the script: (0 minutes 0.835 seconds)

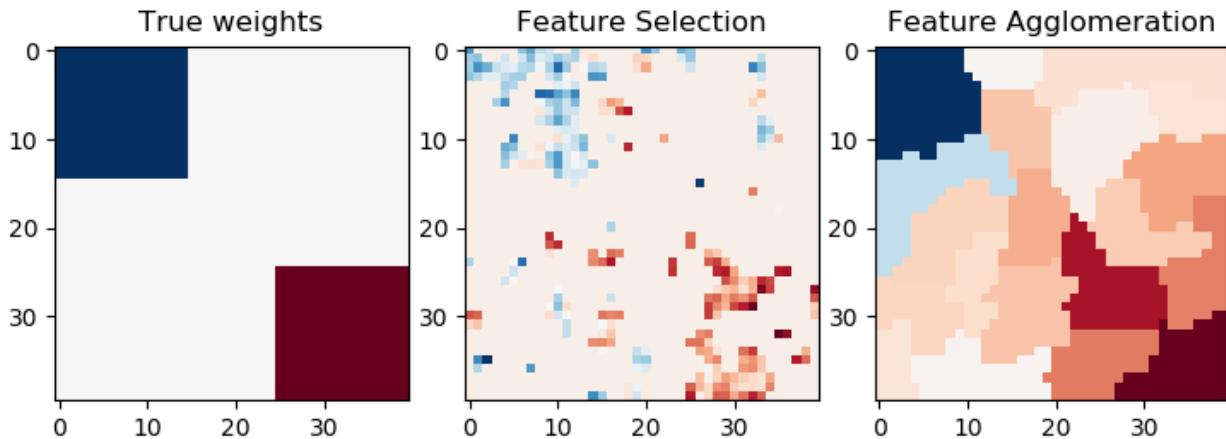
Note: Click [here](#) to download the full example code

5.6.22 Feature agglomeration vs. univariate selection

This example compares 2 dimensionality reduction strategies:

- univariate feature selection with Anova
- feature agglomeration with Ward hierarchical clustering

Both methods are compared in a regression problem using a BayesianRidge as supervised estimator.



Out:

```
[Memory] Calling sklearn.cluster.hierarchical.ward_tree...
ward_tree(array([[ -0.451933, ..., -0.675318],
   ...,
   [ 0.275706, ..., -1.085711]]),
<1600x1600 sparse matrix of type '<class \'numpy.int64\'>'  

   with 7840 stored elements in COOrdinate format>, n_clusters=None, return_
   distance=False)
ward_tree - 0.1s, 0.0min

[Memory] Calling sklearn.cluster.hierarchical.ward_tree...
ward_tree(array([[ 0.905206, ...,  0.161245],
   ...,
   [-0.849835, ..., -1.091621]]),
<1600x1600 sparse matrix of type '<class \'numpy.int64\'>'  

   with 7840 stored elements in COOrdinate format>, n_clusters=None, return_
   distance=False)
ward_tree - 0.1s, 0.0min

[Memory] Calling sklearn.cluster.hierarchical.ward_tree...
ward_tree(array([[ 0.905206, ..., -0.675318],
   ...,
   [-0.849835, ..., -1.085711]]),
<1600x1600 sparse matrix of type '<class \'numpy.int64\'>'  

   with 7840 stored elements in COOrdinate format>, n_clusters=None, return_
   distance=False)
ward_tree - 0.1s, 0.0min

[Memory] Calling sklearn.feature_selection.univariate_selection.f_regression...
f_regression(array([[ -0.451933, ...,  0.275706],
   ...,
   [-0.675318, ..., -1.085711]]),
array([ 25.267703, ..., -25.026711]))
f_regression - 0.0s, 0.0min

[Memory] Calling sklearn.feature_selection.univariate_selection.f_regression...
f_regression(array([[ 0.905206, ..., -0.849835],
   ...,
```

```
[ 0.161245, ..., -1.091621]]),
array([-27.447268, ..., -112.638768]))
f_regression - 0.0s, 0.0min
[Memory] Calling sklearn.feature_selection.univariate_selection.f_regression...
f_regression(array([[ 0.905206, ..., -0.849835],
       ...,
      [-0.675318, ..., -1.085711]]),
array([-27.447268, ..., -25.026711]))
f_regression - 0.0s, 0.0min
```

```
# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause

print(__doc__)

import shutil
import tempfile

import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg, ndimage
from joblib import Memory

from sklearn.feature_extraction.image import grid_to_graph
from sklearn import feature_selection
from sklearn.cluster import FeatureAgglomeration
from sklearn.linear_model import BayesianRidge
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import KFold

#####
# Generate data
n_samples = 200
size = 40 # image size
roi_size = 15
snr = 5.
np.random.seed(0)
mask = np.ones([size, size], dtype=np.bool)

coef = np.zeros((size, size))
coef[0:roi_size, 0:roi_size] = -1.
coef[-roi_size:, -roi_size:] = 1.

X = np.random.randn(n_samples, size ** 2)
for x in X: # smooth data
    x[:] = ndimage.gaussian_filter(x.reshape(size, size), sigma=1.0).ravel()
X -= X.mean(axis=0)
X /= X.std(axis=0)

y = np.dot(X, coef.ravel())
noise = np.random.randn(y.shape[0])
```

```

noise_coef = (linalg.norm(y, 2) / np.exp(snr / 20.)) / linalg.norm(noise, 2)
y += noise_coef * noise # add noise

# ######
# Compute the coefs of a Bayesian Ridge with GridSearch
cv = KFold(2) # cross-validation generator for model selection
ridge = BayesianRidge()
cachedir = tempfile.mkdtemp()
mem = Memory(location=cachedir, verbose=1)

# Ward agglomeration followed by BayesianRidge
connectivity = grid_to_graph(n_x=size, n_y=size)
ward = FeatureAgglomeration(n_clusters=10, connectivity=connectivity,
                             memory=mem)
clf = Pipeline([('ward', ward), ('ridge', ridge)])
# Select the optimal number of parcels with grid search
clf = GridSearchCV(clf, {'ward_n_clusters': [10, 20, 30]}, n_jobs=1, cv=cv)
clf.fit(X, y) # set the best parameters
coef_ = clf.best_estimator_.steps[-1][1].coef_
coef_ = clf.best_estimator_.steps[0][1].inverse_transform(coef_)
coef_agglomeration_ = coef_.reshape(size, size)

# Anova univariate feature selection followed by BayesianRidge
f_regression = mem.cache(feature_selection.f_regression) # caching function
anova = feature_selection.SelectPercentile(f_regression)
clf = Pipeline([('anova', anova), ('ridge', ridge)])
# Select the optimal percentage of features with grid search
clf = GridSearchCV(clf, {'anova_percentile': [5, 10, 20]}, cv=cv)
clf.fit(X, y) # set the best parameters
coef_ = clf.best_estimator_.steps[-1][1].coef_
coef_ = clf.best_estimator_.steps[0][1].inverse_transform(coef_.reshape(1, -1))
coef_selection_ = coef_.reshape(size, size)

# #####
# Inverse the transformation to plot the results on an image
plt.close('all')
plt.figure(figsize=(7.3, 2.7))
plt.subplot(1, 3, 1)
plt.imshow(coef, interpolation="nearest", cmap=plt.cm.RdBu_r)
plt.title("True weights")
plt.subplot(1, 3, 2)
plt.imshow(coef_selection_, interpolation="nearest", cmap=plt.cm.RdBu_r)
plt.title("Feature Selection")
plt.subplot(1, 3, 3)
plt.imshow(coef_agglomeration_, interpolation="nearest", cmap=plt.cm.RdBu_r)
plt.title("Feature Agglomeration")
plt.subplots_adjust(0.04, 0.0, 0.98, 0.94, 0.16, 0.26)
plt.show()

# Attempt to remove the temporary cachedir, but don't worry if it fails
shutil.rmtree(cachedir, ignore_errors=True)

```

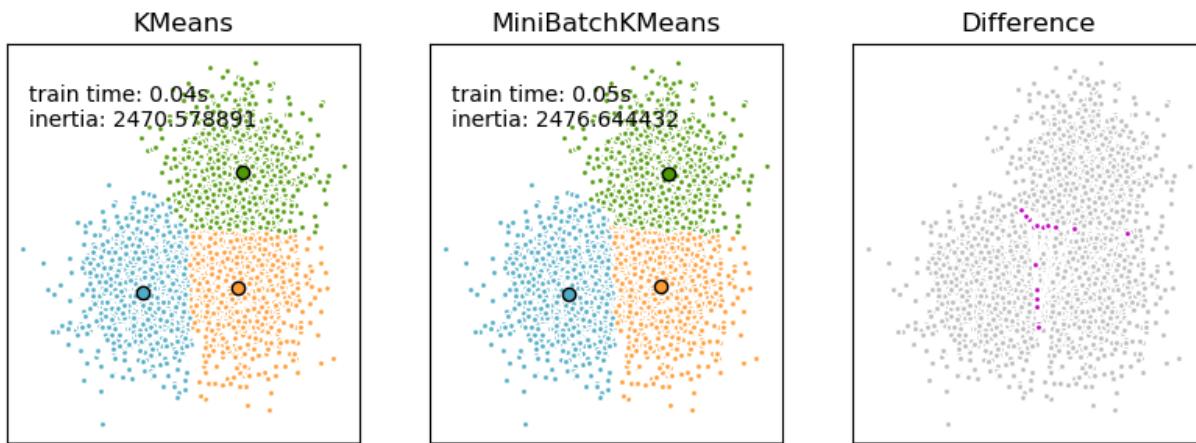
Total running time of the script: (0 minutes 0.623 seconds)

Note: Click [here](#) to download the full example code

5.6.23 Comparison of the K-Means and MiniBatchKMeans clustering algorithms

We want to compare the performance of the MiniBatchKMeans and KMeans: the MiniBatchKMeans is faster, but gives slightly different results (see [Mini Batch K-Means](#)).

We will cluster a set of data, first with KMeans and then with MiniBatchKMeans, and plot the results. We will also plot the points that are labelled differently between the two algorithms.



```
print(__doc__)

import time

import numpy as np
import matplotlib.pyplot as plt

from sklearn.cluster import MiniBatchKMeans, KMeans
from sklearn.metrics.pairwise import pairwise_distances_argmin
from sklearn.datasets.samples_generator import make_blobs

# ######
# Generate sample data
np.random.seed(0)

batch_size = 45
centers = [[1, 1], [-1, -1], [1, -1]]
n_clusters = len(centers)
X, labels_true = make_blobs(n_samples=3000, centers=centers, cluster_std=0.7)

# ######
# Compute clustering with Means

k_means = KMeans(init='k-means++', n_clusters=3, n_init=10)
t0 = time.time()
k_means.fit(X)
t_batch = time.time() - t0

# ######
# Compute clustering with MiniBatchKMeans

mbk = MiniBatchKMeans(init='k-means++', n_clusters=3, batch_size=batch_size,
                      n_init=10, max_no_improvement=10, verbose=0)
t0 = time.time()
```

```

mbk.fit(X)
t_mini_batch = time.time() - t0

# ##### Plot result #####
# We want to have the same colors for the same cluster from the
# MiniBatchKMeans and the KMeans algorithm. Let's pair the cluster centers per
# closest one.
k_means_cluster_centers = np.sort(k_means.cluster_centers_, axis=0)
mbk_means_cluster_centers = np.sort(mbk.cluster_centers_, axis=0)
k_means_labels = pairwise_distances_argmin(X, k_means_cluster_centers)
mbk_means_labels = pairwise_distances_argmin(X, mbk_means_cluster_centers)
order = pairwise_distances_argmin(k_means_cluster_centers,
                                  mbk_means_cluster_centers)

# KMeans
ax = fig.add_subplot(1, 3, 1)
for k, col in zip(range(n_clusters), colors):
    my_members = k_means_labels == k
    cluster_center = k_means_cluster_centers[k]
    ax.plot(X[my_members, 0], X[my_members, 1], 'w',
            markerfacecolor=col, marker='.')
    ax.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
            markeredgecolor='k', markersize=6)
ax.set_title('KMeans')
ax.set_xticks(())
ax.set_yticks(())
plt.text(-3.5, 1.8, 'train time: %.2fs\ninertia: %f' %
         (t_batch, k_means.inertia_))

# MiniBatchKMeans
ax = fig.add_subplot(1, 3, 2)
for k, col in zip(range(n_clusters), colors):
    my_members = mbk_means_labels == order[k]
    cluster_center = mbk_means_cluster_centers[order[k]]
    ax.plot(X[my_members, 0], X[my_members, 1], 'w',
            markerfacecolor=col, marker='.')
    ax.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
            markeredgecolor='k', markersize=6)
ax.set_title('MiniBatchKMeans')
ax.set_xticks(())
ax.set_yticks(())
plt.text(-3.5, 1.8, 'train time: %.2fs\ninertia: %f' %
         (t_mini_batch, mbk.inertia_))

# Initialise the different array to all False
different = (mbk_means_labels == 4)
ax = fig.add_subplot(1, 3, 3)

for k in range(n_clusters):
    different += ((k_means_labels == k) != (mbk_means_labels == order[k]))

identic = np.logical_not(different)

```

```
ax.plot(X[identic, 0], X[identic, 1], 'w',
        markerfacecolor='#bbbbbb', marker='.')
ax.plot(X[different, 0], X[different, 1], 'w',
        markerfacecolor='m', marker='.')
ax.set_title('Difference')
ax.set_xticks(())
ax.set_yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.127 seconds)

Note: Click [here](#) to download the full example code

5.6.24 A demo of K-Means clustering on the handwritten digits data

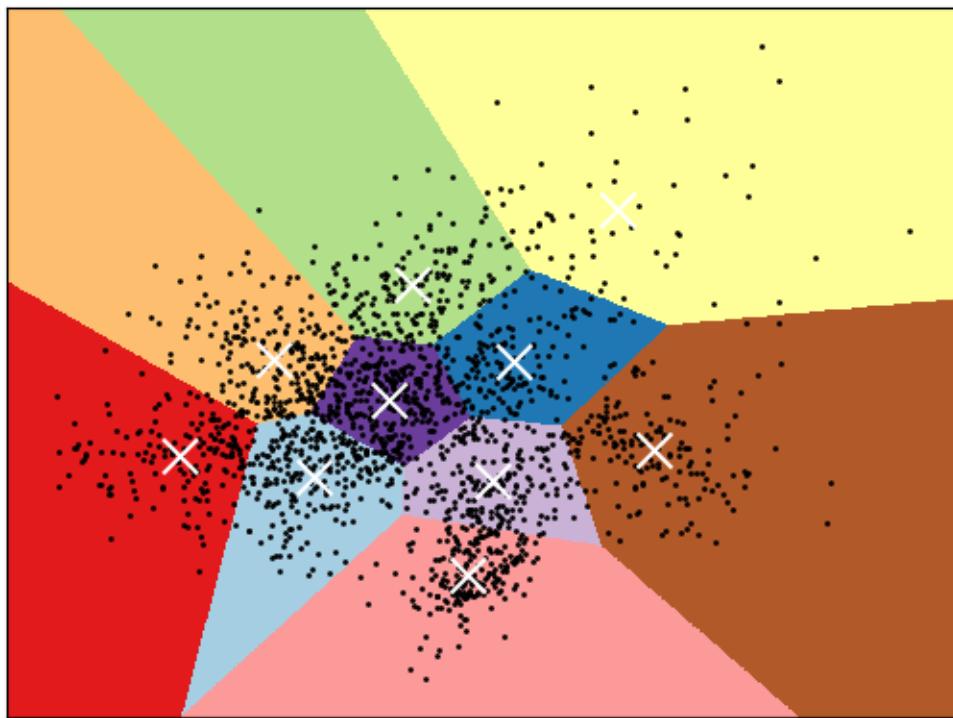
In this example we compare the various initialization strategies for K-means in terms of runtime and quality of the results.

As the ground truth is known here, we also apply different cluster quality metrics to judge the goodness of fit of the cluster labels to the ground truth.

Cluster quality metrics evaluated (see [Clustering performance evaluation](#) for definitions and discussions of the metrics):

Shorthand	full name
homo	homogeneity score
compl	completeness score
v-meas	V measure
ARI	adjusted Rand index
AMI	adjusted mutual information
silhouette	silhouette coefficient

K-means clustering on the digits dataset (PCA-reduced data)
Centroids are marked with white cross



Out:

n_digits: 10,		n_samples 1797,		n_features 64					
init	time	inertia	homo	compl	v-meas	ARI	AMI	silhouette	
k-means++	0.37s	69432	0.602	0.650	0.625	0.465	0.621	0.146	
random	0.27s	69694	0.669	0.710	0.689	0.553	0.686	0.147	
PCA-based	0.03s	70804	0.671	0.698	0.684	0.561	0.681	0.118	

```
print(__doc__)

from time import time
import numpy as np
import matplotlib.pyplot as plt

from sklearn import metrics
from sklearn.cluster import KMeans
from sklearn.datasets import load_digits
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale
```

```

np.random.seed(42)

digits = load_digits()
data = scale(digits.data)

n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target

sample_size = 300

print("n_digits: %d, \t n_samples %d, \t n_features %d"
      % (n_digits, n_samples, n_features))

print(82 * '_')
print('init\t\ttime\tinertia\thomo\tcompl\tv-meas\tARI\tAMI\tsilhouette')

def bench_k_means(estimator, name, data):
    t0 = time()
    estimator.fit(data)
    print('%-9s\t%.2fs\t%i\t%.3f\t%.3f\t%.3f\t%.3f\t%.3f\t%.3f'
          % (name, (time() - t0), estimator.inertia_,
             metrics.homogeneity_score(labels, estimator.labels_),
             metrics.completeness_score(labels, estimator.labels_),
             metrics.v_measure_score(labels, estimator.labels_),
             metrics.adjusted_rand_score(labels, estimator.labels_),
             metrics.adjusted_mutual_info_score(labels, estimator.labels_,
                                                average_method='arithmetic'),
             metrics.silhouette_score(data, estimator.labels_,
                                       metric='euclidean',
                                       sample_size=sample_size)))
    bench_k_means(KMeans(init='k-means++', n_clusters=n_digits, n_init=10),
                  name="k-means++", data=data)

    bench_k_means(KMeans(init='random', n_clusters=n_digits, n_init=10),
                  name="random", data=data)

    # in this case the seeding of the centers is deterministic, hence we run the
    # kmeans algorithm only once with n_init=1
    pca = PCA(n_components=n_digits).fit(data)
    bench_k_means(KMeans(init=pca.components_, n_clusters=n_digits, n_init=1),
                  name="PCA-based",
                  data=data)
    print(82 * '_')

    #####
    # Visualize the results on PCA-reduced data

    reduced_data = PCA(n_components=2).fit_transform(data)
    kmeans = KMeans(init='k-means++', n_clusters=n_digits, n_init=10)
    kmeans.fit(reduced_data)

    # Step size of the mesh. Decrease to increase the quality of the VQ.
    h = .02      # point in the mesh [x_min, x_max]x[y_min, y_max].

```

```

# Plot the decision boundary. For that, we will assign a color to each
x_min, x_max = reduced_data[:, 0].min() - 1, reduced_data[:, 0].max() + 1
y_min, y_max = reduced_data[:, 1].min() - 1, reduced_data[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

# Obtain labels for each point in mesh. Use last trained model.
Z = kmeans.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure(1)
plt.clf()
plt.imshow(Z, interpolation='nearest',
           extent=(xx.min(), xx.max(), yy.min(), yy.max()),
           cmap=plt.cm.Paired,
           aspect='auto', origin='lower')

plt.plot(reduced_data[:, 0], reduced_data[:, 1], 'k.', markersize=2)
# Plot the centroids as a white X
centroids = kmeans.cluster_centers_
plt.scatter(centroids[:, 0], centroids[:, 1],
            marker='x', s=169, linewidths=3,
            color='w', zorder=10)
plt.title('K-means clustering on the digits dataset (PCA-reduced data)\n'
          'Centroids are marked with white cross')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())
plt.show()

```

Total running time of the script: (0 minutes 1.230 seconds)

Note: Click [here](#) to download the full example code

5.6.25 Comparing different hierarchical linkage methods on toy datasets

This example shows characteristics of different linkage methods for hierarchical clustering on datasets that are “interesting” but still in 2D.

The main observations to make are:

- single linkage is fast, and can perform well on non-globular data, but it performs poorly in the presence of noise.
- average and complete linkage perform well on cleanly separated globular clusters, but have mixed results otherwise.
- Ward is the most effective method for noisy data.

While these examples give some intuition about the algorithms, this intuition might not apply to very high dimensional data.

```

print(__doc__)

import time

```

```

import warnings

import numpy as np
import matplotlib.pyplot as plt

from sklearn import cluster, datasets
from sklearn.preprocessing import StandardScaler
from itertools import cycle, islice

np.random.seed(0)

```

Generate datasets. We choose the size big enough to see the scalability of the algorithms, but not too big to avoid too long running times

```

n_samples = 1500
noisy_circles = datasets.make_circles(n_samples=n_samples, factor=.5,
                                      noise=.05)
noisy_moons = datasets.make_moons(n_samples=n_samples, noise=.05)
blobs = datasets.make_blobs(n_samples=n_samples, random_state=8)
no_structure = np.random.rand(n_samples, 2), None

# Anisotropically distributed data
random_state = 170
X, y = datasets.make_blobs(n_samples=n_samples, random_state=random_state)
transformation = [[0.6, -0.6], [-0.4, 0.8]]
X_aniso = np.dot(X, transformation)
aniso = (X_aniso, y)

# blobs with varied variances
varied = datasets.make_blobs(n_samples=n_samples,
                             cluster_std=[1.0, 2.5, 0.5],
                             random_state=random_state)

```

Run the clustering and plot

```

# Set up cluster parameters
plt.figure(figsize=(9 * 1.3 + 2, 14.5))
plt.subplots_adjust(left=.02, right=.98, bottom=.001, top=.96, wspace=.05,
                    hspace=.01)

plot_num = 1

default_base = {'n_neighbors': 10,
                'n_clusters': 3}

datasets = [
    (noisy_circles, {'n_clusters': 2}),
    (noisy_moons, {'n_clusters': 2}),
    (varied, {'n_neighbors': 2}),
    (aniso, {'n_neighbors': 2}),
    (blobs, {}),
    (no_structure, {})]

for i_dataset, (dataset, algo_params) in enumerate(datasets):
    # update parameters with dataset-specific values
    params = default_base.copy()
    params.update(algo_params)

```

```

X, y = dataset

# normalize dataset for easier parameter selection
X = StandardScaler().fit_transform(X)

# =====
# Create cluster objects
# =====
ward = cluster.AgglomerativeClustering(
    n_clusters=params['n_clusters'], linkage='ward')
complete = cluster.AgglomerativeClustering(
    n_clusters=params['n_clusters'], linkage='complete')
average = cluster.AgglomerativeClustering(
    n_clusters=params['n_clusters'], linkage='average')
single = cluster.AgglomerativeClustering(
    n_clusters=params['n_clusters'], linkage='single')

clustering_algorithms = (
    ('Single Linkage', single),
    ('Average Linkage', average),
    ('Complete Linkage', complete),
    ('Ward Linkage', ward),
)
for name, algorithm in clustering_algorithms:
    t0 = time.time()

    # catch warnings related to kneighbors_graph
    with warnings.catch_warnings():
        warnings.filterwarnings(
            "ignore",
            message="the number of connected components of the " +
            "connectivity matrix is [0-9]{1,2}" +
            " > 1. Completing it to avoid stopping the tree early.",
            category=UserWarning)
        algorithm.fit(X)

    t1 = time.time()
    if hasattr(algorithm, 'labels_'):
        y_pred = algorithm.labels_.astype(np.int)
    else:
        y_pred = algorithm.predict(X)

    plt.subplot(len(datasets), len(clustering_algorithms), plot_num)
    if i_dataset == 0:
        plt.title(name, size=18)

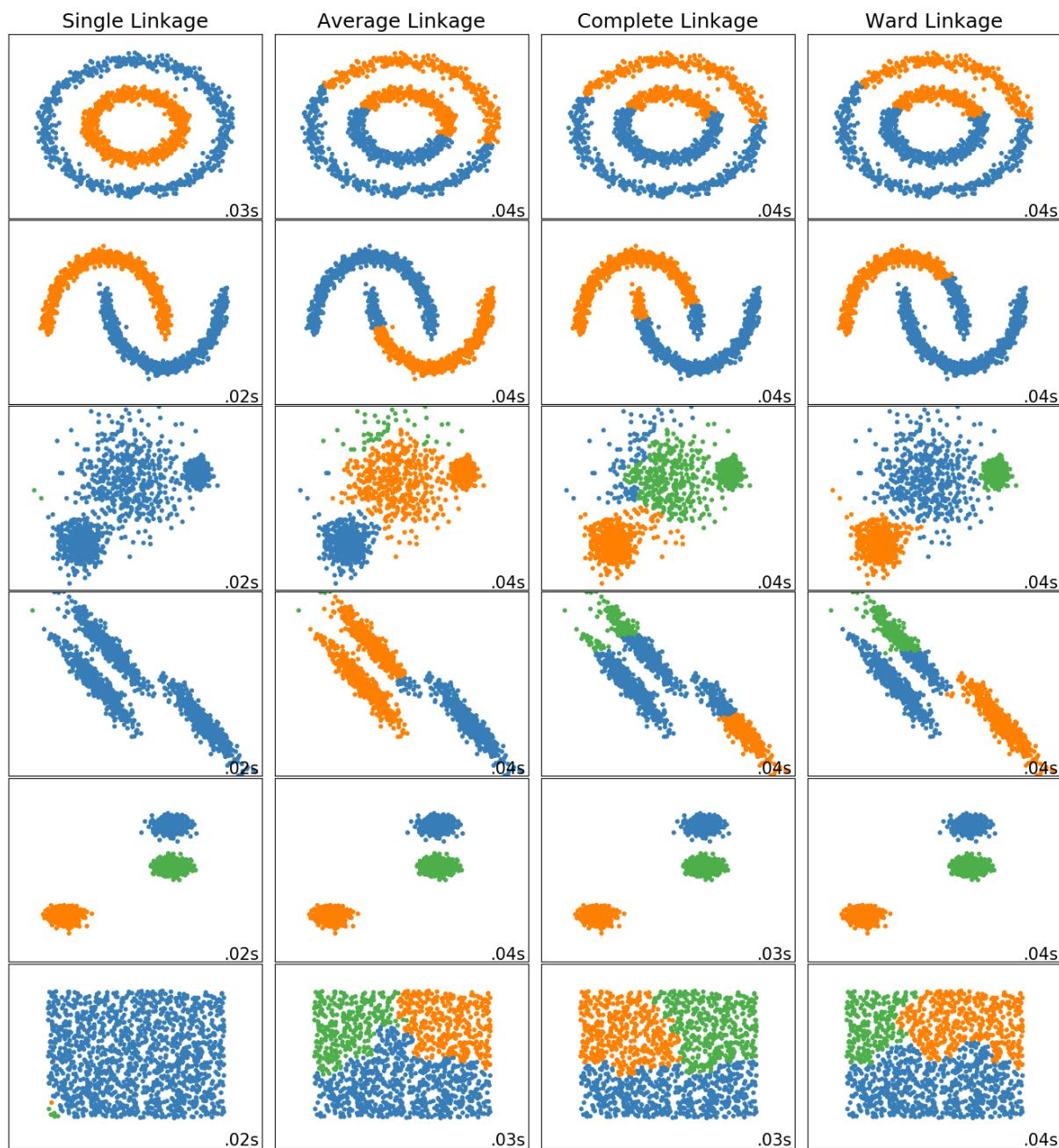
    colors = np.array(list(islice(cycle(['#377eb8', '#ff7f00', '#4daf4a',
                                         '#f781bf', '#a65628', '#984ea3',
                                         '#999999', '#e41a1c', '#dede00']), int(max(y_pred) + 1))))
    plt.scatter(X[:, 0], X[:, 1], s=10, color=colors[y_pred])

    plt.xlim(-2.5, 2.5)
    plt.ylim(-2.5, 2.5)
    plt.xticks(())
    plt.yticks(())
    plt.text(.99, .01, ('%.2fs' % (t1 - t0)).lstrip('0'),
)

```

```
    transform=plt.gca().transAxes, size=15,
    horizontalalignment='right')
plot_num += 1

plt.show()
```



Total running time of the script: (0 minutes 1.577 seconds)

Note: Click [here](#) to download the full example code

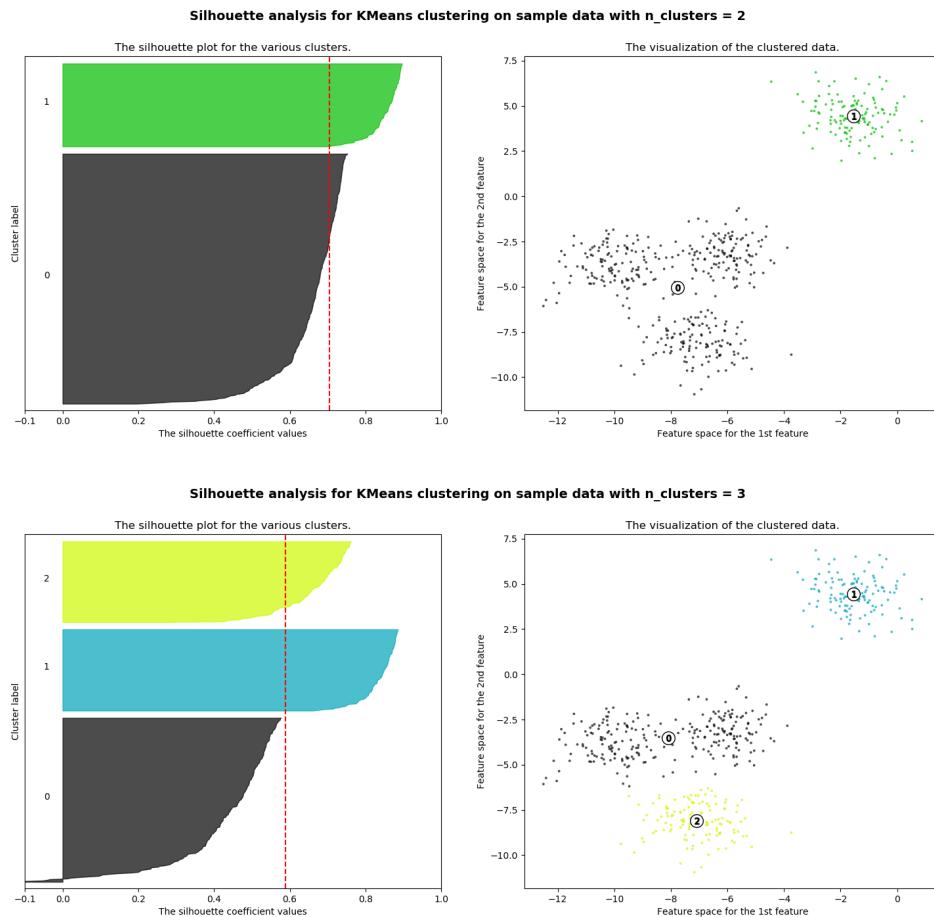
5.6.26 Selecting the number of clusters with silhouette analysis on KMeans clustering

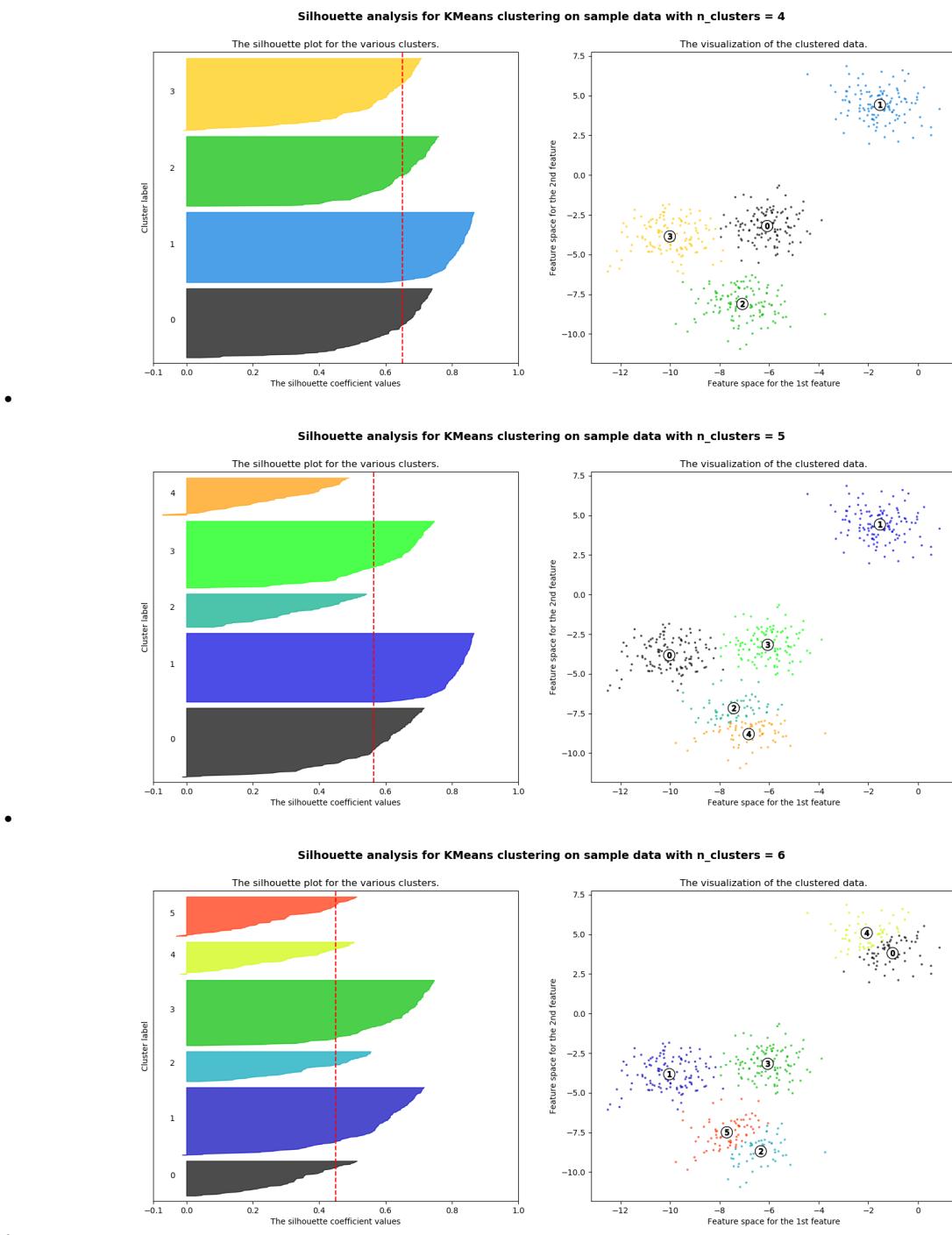
Silhouette analysis can be used to study the separation distance between the resulting clusters. The silhouette plot displays a measure of how close each point in one cluster is to points in the neighboring clusters and thus provides a way to assess parameters like number of clusters visually. This measure has a range of [-1, 1].

Silhouette coefficients (as these values are referred to as) near +1 indicate that the sample is far away from the neighboring clusters. A value of 0 indicates that the sample is on or very close to the decision boundary between two neighboring clusters and negative values indicate that those samples might have been assigned to the wrong cluster.

In this example the silhouette analysis is used to choose an optimal value for `n_clusters`. The silhouette plot shows that the `n_clusters` value of 3, 5 and 6 are a bad pick for the given data due to the presence of clusters with below average silhouette scores and also due to wide fluctuations in the size of the silhouette plots. Silhouette analysis is more ambivalent in deciding between 2 and 4.

Also from the thickness of the silhouette plot the cluster size can be visualized. The silhouette plot for cluster 0 when `n_clusters` is equal to 2, is bigger in size owing to the grouping of the 3 sub clusters into one big cluster. However when the `n_clusters` is equal to 4, all the plots are more or less of similar thickness and hence are of similar sizes as can be also verified from the labelled scatter plot on the right.





Out:

```
For n_clusters = 2 The average silhouette_score is : 0.7049787496083262
For n_clusters = 3 The average silhouette_score is : 0.5882004012129721
For n_clusters = 4 The average silhouette_score is : 0.6505186632729437
For n_clusters = 5 The average silhouette_score is : 0.56376469026194
For n_clusters = 6 The average silhouette_score is : 0.4504666294372765
```

```

from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score

import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np

print(__doc__)

# Generating the sample data from make_blobs
# This particular setting has one distinct cluster and 3 clusters placed close
# together.
X, y = make_blobs(n_samples=500,
                   n_features=2,
                   centers=4,
                   cluster_std=1,
                   center_box=(-10.0, 10.0),
                   shuffle=True,
                   random_state=1) # For reproducibility

range_n_clusters = [2, 3, 4, 5, 6]

for n_clusters in range_n_clusters:
    # Create a subplot with 1 row and 2 columns
    fig, (ax1, ax2) = plt.subplots(1, 2)
    fig.set_size_inches(18, 7)

    # The 1st subplot is the silhouette plot
    # The silhouette coefficient can range from -1, 1 but in this example all
    # lie within [-0.1, 1]
    ax1.set_xlim([-0.1, 1])
    # The (n_clusters+1)*10 is for inserting blank space between silhouette
    # plots of individual clusters, to demarcate them clearly.
    ax1.set_yscale([0, len(X) + (n_clusters + 1) * 10])

    # Initialize the clusterer with n_clusters value and a random generator
    # seed of 10 for reproducibility.
    clusterer = KMeans(n_clusters=n_clusters, random_state=10)
    cluster_labels = clusterer.fit_predict(X)

    # The silhouette_score gives the average value for all the samples.
    # This gives a perspective into the density and separation of the formed
    # clusters
    silhouette_avg = silhouette_score(X, cluster_labels)
    print("For n_clusters =", n_clusters,
          "The average silhouette_score is :", silhouette_avg)

    # Compute the silhouette scores for each sample
    sample_silhouette_values = silhouette_samples(X, cluster_labels)

    y_lower = 10
    for i in range(n_clusters):
        # Aggregate the silhouette scores for samples belonging to
        # cluster i, and sort them

```

```

ith_cluster_silhouette_values = \
    sample_silhouette_values[cluster_labels == i]

ith_cluster_silhouette_values.sort()

size_cluster_i = ith_cluster_silhouette_values.shape[0]
y_upper = y_lower + size_cluster_i

color = cm.nipy_spectral(float(i) / n_clusters)
ax1.fill_betweenx(np.arange(y_lower, y_upper),
                  0, ith_cluster_silhouette_values,
                  facecolor=color, edgecolor=color, alpha=0.7)

# Label the silhouette plots with their cluster numbers at the middle
ax1.text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))

# Compute the new y_lower for next plot
y_lower = y_upper + 10 # 10 for the 0 samples

ax1.set_title("The silhouette plot for the various clusters.")
ax1.set_xlabel("The silhouette coefficient values")
ax1.set_ylabel("Cluster label")

# The vertical line for average silhouette score of all the values
ax1.axvline(x=silhouette_avg, color="red", linestyle="--")

ax1.set_yticks([]) # Clear the yaxis labels / ticks
ax1.set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])

# 2nd Plot showing the actual clusters formed
colors = cm.nipy_spectral(cluster_labels.astype(float) / n_clusters)
ax2.scatter(X[:, 0], X[:, 1], marker='.', s=30, lw=0, alpha=0.7,
            c=colors, edgecolor='k')

# Labeling the clusters
centers = clusterer.cluster_centers_
# Draw white circles at cluster centers
ax2.scatter(centers[:, 0], centers[:, 1], marker='o',
            c="white", alpha=1, s=200, edgecolor='k')

for i, c in enumerate(centers):
    ax2.scatter(c[0], c[1], marker='$_d$' % i, alpha=1,
                s=50, edgecolor='k')

ax2.set_title("The visualization of the clustered data.")
ax2.set_xlabel("Feature space for the 1st feature")
ax2.set_ylabel("Feature space for the 2nd feature")

plt.suptitle(("Silhouette analysis for KMeans clustering on sample data "
              "with n_clusters = %d" % n_clusters),
              fontsize=14, fontweight='bold')

plt.show()

```

Total running time of the script: (0 minutes 0.629 seconds)

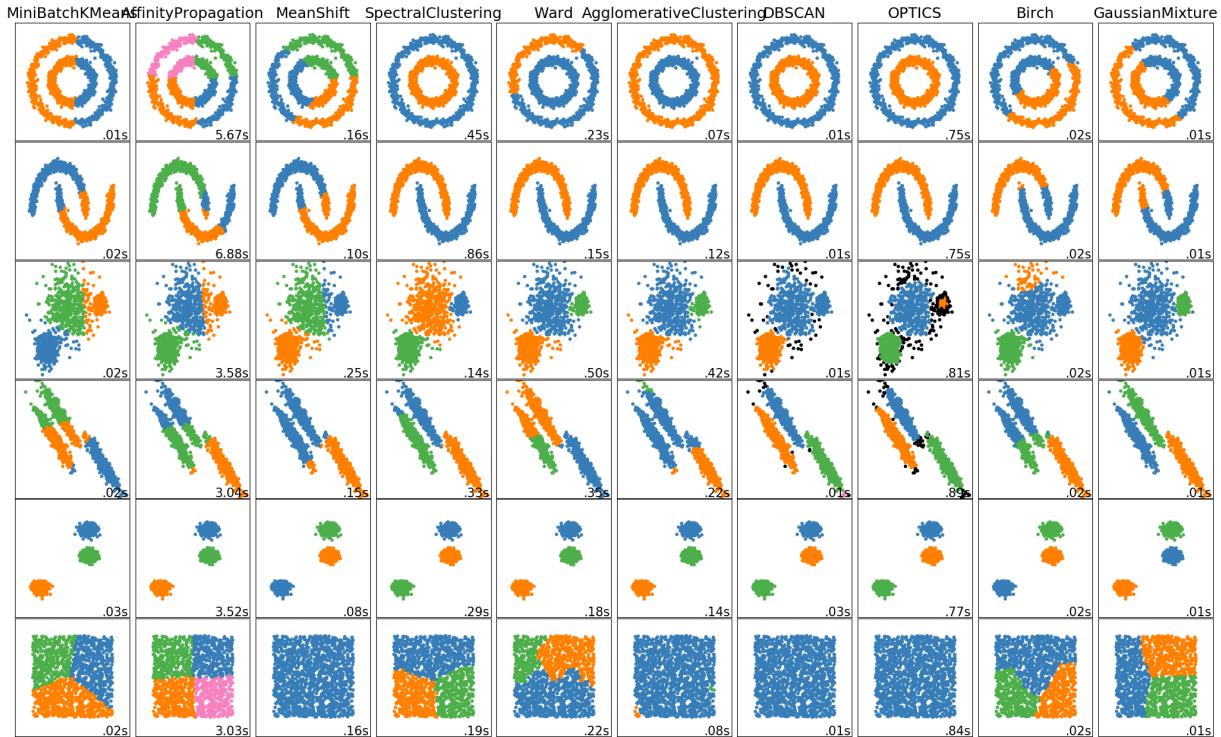
Note: Click [here](#) to download the full example code

5.6.27 Comparing different clustering algorithms on toy datasets

This example shows characteristics of different clustering algorithms on datasets that are “interesting” but still in 2D. With the exception of the last dataset, the parameters of each of these dataset-algorithm pairs has been tuned to produce good clustering results. Some algorithms are more sensitive to parameter values than others.

The last dataset is an example of a ‘null’ situation for clustering: the data is homogeneous, and there is no good clustering. For this example, the null dataset uses the same parameters as the dataset in the row above it, which represents a mismatch in the parameter values and the data structure.

While these examples give some intuition about the algorithms, this intuition might not apply to very high dimensional data.



```
print(__doc__)

import time
import warnings

import numpy as np
import matplotlib.pyplot as plt

from sklearn import cluster, datasets, mixture
from sklearn.neighbors import kneighbors_graph
from sklearn.preprocessing import StandardScaler
from itertools import cycle, islice

np.random.seed(0)
```

```

# =====
# Generate datasets. We choose the size big enough to see the scalability
# of the algorithms, but not too big to avoid too long running times
# =====
n_samples = 1500
noisy_circles = datasets.make_circles(n_samples=n_samples, factor=.5,
                                       noise=.05)
noisy_moons = datasets.make_moons(n_samples=n_samples, noise=.05)
blobs = datasets.make_blobs(n_samples=n_samples, random_state=8)
no_structure = np.random.rand(n_samples, 2), None

# Anisotropically distributed data
random_state = 170
X, y = datasets.make_blobs(n_samples=n_samples, random_state=random_state)
transformation = [[0.6, -0.6], [-0.4, 0.8]]
X_aniso = np.dot(X, transformation)
aniso = (X_aniso, y)

# blobs with varied variances
varied = datasets.make_blobs(n_samples=n_samples,
                             cluster_std=[1.0, 2.5, 0.5],
                             random_state=random_state)

# =====
# Set up cluster parameters
# =====
plt.figure(figsize=(9 * 2 + 3, 12.5))
plt.subplots_adjust(left=.02, right=.98, bottom=.001, top=.96, wspace=.05,
                    hspace=.01)

plot_num = 1

default_base = {'quantile': .3,
                'eps': .3,
                'damping': .9,
                'preference': -200,
                'n_neighbors': 10,
                'n_clusters': 3,
                'min_samples': 20,
                'xi': 0.05,
                'min_cluster_size': 0.1}

datasets = [
    (noisy_circles, {'damping': .77, 'preference': -240,
                    'quantile': .2, 'n_clusters': 2,
                    'min_samples': 20, 'xi': 0.25}),
    (noisy_moons, {'damping': .75, 'preference': -220, 'n_clusters': 2}),
    (varied, {'eps': .18, 'n_neighbors': 2,
              'min_samples': 5, 'xi': 0.035, 'min_cluster_size': .2}),
    (aniso, {'eps': .15, 'n_neighbors': 2,
              'min_samples': 20, 'xi': 0.1, 'min_cluster_size': .2}),
    (blobs, {}),
    (no_structure, {})]

for i_dataset, (dataset, algo_params) in enumerate(datasets):
    # update parameters with dataset-specific values
    params = default_base.copy()

```

```

params.update(algo_params)

X, y = dataset

# normalize dataset for easier parameter selection
X = StandardScaler().fit_transform(X)

# estimate bandwidth for mean shift
bandwidth = cluster.estimate_bandwidth(X, quantile=params['quantile'])

# connectivity matrix for structured Ward
connectivity = kneighbors_graph(
    X, n_neighbors=params['n_neighbors'], include_self=False)
# make connectivity symmetric
connectivity = 0.5 * (connectivity + connectivity.T)

# =====
# Create cluster objects
# =====

ms = cluster.MeanShift(bandwidth=bandwidth, bin_seeding=True)
two_means = cluster.MiniBatchKMeans(n_clusters=params['n_clusters'])
ward = cluster.AgglomerativeClustering(
    n_clusters=params['n_clusters'], linkage='ward',
    connectivity=connectivity)
spectral = cluster.SpectralClustering(
    n_clusters=params['n_clusters'], eigen_solver='arpack',
    affinity="nearest_neighbors")
dbSCAN = cluster.DBSCAN(eps=params['eps'])
optics = cluster.OPTICS(min_samples=params['min_samples'],
                        xi=params['xi'],
                        min_cluster_size=params['min_cluster_size'])
affinity_propagation = cluster.AffinityPropagation(
    damping=params['damping'], preference=params['preference'])
average_linkage = cluster.AgglomerativeClustering(
    linkage="average", affinity="cityblock",
    n_clusters=params['n_clusters'], connectivity=connectivity)
birch = cluster.Birch(n_clusters=params['n_clusters'])
gmm = mixture.GaussianMixture(
    n_components=params['n_clusters'], covariance_type='full')

clustering_algorithms = (
    ('MiniBatchKMeans', two_means),
    ('AffinityPropagation', affinity_propagation),
    ('MeanShift', ms),
    ('SpectralClustering', spectral),
    ('Ward', ward),
    ('AgglomerativeClustering', average_linkage),
    ('DBSCAN', dbSCAN),
    ('OPTICS', optics),
    ('Birch', birch),
    ('GaussianMixture', gmm)
)

for name, algorithm in clustering_algorithms:
    t0 = time.time()

# catch warnings related to kneighbors_graph
    with warnings.catch_warnings():

```

```
warnings.filterwarnings(
    "ignore",
    message="the number of connected components of the " +
    "connectivity matrix is [0-9]{1,2}" +
    " > 1. Completing it to avoid stopping the tree early.",
    category=UserWarning)
warnings.filterwarnings(
    "ignore",
    message="Graph is not fully connected, spectral embedding" +
    " may not work as expected.",
    category=UserWarning)
algorithm.fit(X)

t1 = time.time()
if hasattr(algorithm, 'labels_'):
    y_pred = algorithm.labels_.astype(np.int)
else:
    y_pred = algorithm.predict(X)

plt.subplot(len(datasets), len(clustering_algorithms), plot_num)
if i_dataset == 0:
    plt.title(name, size=18)

colors = np.array(list(islice(cycle(['#377eb8', '#ff7f00', '#4daf4a',
                                    '#f781bf', '#a65628', '#984ea3',
                                    '#999999', '#e41a1c', '#dede00']), int(max(y_pred) + 1))))
# add black color for outliers (if any)
colors = np.append(colors, ["#000000"])
plt.scatter(X[:, 0], X[:, 1], s=10, color=colors[y_pred])

plt.xlim(-2.5, 2.5)
plt.ylim(-2.5, 2.5)
plt.xticks(())
plt.yticks(())
plt.text(.99, .01, ('%.2fs' % (t1 - t0)).lstrip('0'),
         transform=plt.gca().transAxes, size=15,
         horizontalalignment='right')
plot_num += 1

plt.show()
```

Total running time of the script: (0 minutes 39.530 seconds)

5.7 Pipelines and composite estimators

Examples of how to compose transformers and pipelines from other estimators. See the *User Guide*.

Note: Click [here](#) to download the full example code

5.7.1 Concatenating multiple feature extraction methods

In many real-world examples, there are many ways to extract features from a dataset. Often it is beneficial to combine several methods to obtain good performance. This example shows how to use FeatureUnion to combine features obtained by PCA and univariate selection.

Combining features using this transformer has the benefit that it allows cross validation and grid searches over the whole process.

The combination used in this example is not particularly helpful on this dataset and is only used to illustrate the usage of FeatureUnion.

Out:

```
Combined space has 3 features
Fitting 5 folds for each of 18 candidates, totalling 90 fits
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=0.1
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=0.1, score=0.
↪933, total= 0.0s
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=0.1
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=0.1, score=0.
↪933, total= 0.0s
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=0.1
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=0.1, score=0.
↪867, total= 0.0s
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=0.1
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=0.1, score=0.
↪933, total= 0.0s
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=0.1
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=0.1, score=1.
↪000, total= 0.0s
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=1
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=1, score=0.
↪900, total= 0.0s
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=1
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=1, score=0.
↪000, total= 0.0s
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=1
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=1, score=0.
↪867, total= 0.0s
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=1
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=1, score=0.
↪933, total= 0.0s
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=1
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=1, score=1.
↪000, total= 0.0s
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=10
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=10, score=0.
↪933, total= 0.0s
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=10
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=10, score=1.
↪000, total= 0.0s
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=10
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=10, score=0.
↪900, total= 0.0s
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=10
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=10, score=0.
↪933, total= 0.0s
[CV] features_pca_n_components=1, features_univ_select_k=1, svm_C=10
```



```
[CV] features_pca_n_components=3, features_univ_select_k=1, svm_C=10
[CV] features_pca_n_components=3, features_univ_select_k=1, svm_C=10, score=0.
↪967, total= 0.0s
[CV] features_pca_n_components=3, features_univ_select_k=1, svm_C=10
[CV] features_pca_n_components=3, features_univ_select_k=1, svm_C=10, score=1.
↪000, total= 0.0s
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=0.1
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=0.1, score=0.
↪967, total= 0.0s
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=0.1
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=0.1, score=1.
↪000, total= 0.0s
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=0.1
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=0.1, score=0.
↪933, total= 0.0s
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=0.1
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=0.1, score=0.
↪967, total= 0.0s
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=0.1
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=0.1, score=1.
↪000, total= 0.0s
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=1
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=1, score=0.
↪967, total= 0.0s
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=1
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=1, score=1.
↪000, total= 0.0s
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=1
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=1, score=0.
↪967, total= 0.0s
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=1
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=1, score=0.
↪967, total= 0.0s
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=1
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=1, score=1.
↪000, total= 0.0s
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=10
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=10, score=1.
↪000, total= 0.0s
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=10
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=10, score=1.
↪000, total= 0.0s
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=10
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=10, score=0.
↪900, total= 0.0s
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=10
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=10, score=0.
↪967, total= 0.0s
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=10
[CV] features_pca_n_components=3, features_univ_select_k=2, svm_C=10, score=1.
↪000, total= 0.0s
Pipeline(steps=[('features',
                  FeatureUnion(transformer_list=[('pca', PCA(n_components=3)),
                                                ('univ_select',
                                                 SelectKBest(k=1)))]),
                  ('svm', SVC(C=10, kernel='linear'))])
```

```
# Author: Andreas Mueller <amueller@ais.uni-bonn.de>
#
# License: BSD 3 clause

from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest

iris = load_iris()

X, y = iris.data, iris.target

# This dataset is way too high-dimensional. Better do PCA:
pca = PCA(n_components=2)

# Maybe some original features where good, too?
selection = SelectKBest(k=1)

# Build estimator from PCA and Univariate selection:

combined_features = FeatureUnion([("pca", pca), ("univ_select", selection)])

# Use combined features to transform dataset:
X_features = combined_features.fit(X, y).transform(X)
print("Combined space has", X_features.shape[1], "features")

svm = SVC(kernel="linear")

# Do grid search over k, n_components and C:

pipeline = Pipeline([("features", combined_features), ("svm", svm)])

param_grid = dict(features_pca_n_components=[1, 2, 3],
                  features_univ_select_k=[1, 2],
                  svm_C=[0.1, 1, 10])

grid_search = GridSearchCV(pipeline, param_grid=param_grid, cv=5, verbose=10)
grid_search.fit(X, y)
print(grid_search.best_estimator_)
```

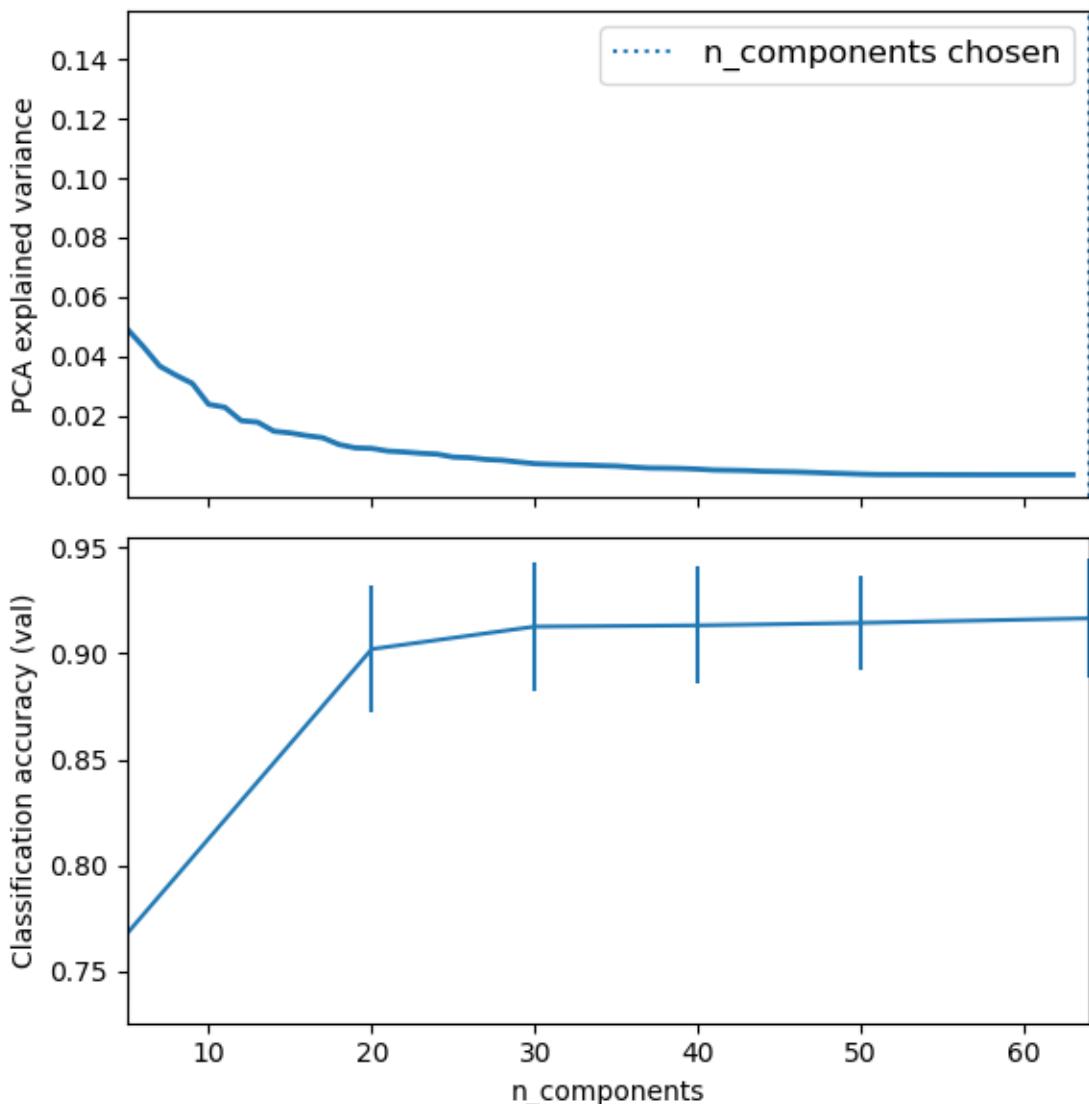
Total running time of the script: (0 minutes 0.865 seconds)

Note: Click [here](#) to download the full example code

5.7.2 Pipelining: chaining a PCA and a logistic regression

The PCA does an unsupervised dimensionality reduction, while the logistic regression does the prediction.

We use a GridSearchCV to set the dimensionality of the PCA



Out:

```
Best parameter (CV score=0.917):  
{'logistic_alpha': 0.01, 'pca_n_components': 64}
```

```
print(__doc__)  
  
# Code source: Gaël Varoquaux  
# Modified for documentation by Jaques Grobler  
# License: BSD 3 clause
```

```

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

from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.linear_model import SGDClassifier
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV

# Define a pipeline to search for the best combination of PCA truncation
# and classifier regularization.
logistic = SGDClassifier(loss='log', penalty='l2', early_stopping=True,
                          max_iter=10000, tol=1e-5, random_state=0)
pca = PCA()
pipe = Pipeline(steps=[('pca', pca), ('logistic', logistic)])

digits = datasets.load_digits()
X_digits = digits.data
y_digits = digits.target

# Parameters of pipelines can be set using '__' separated parameter names:
param_grid = {
    'pca_n_components': [5, 20, 30, 40, 50, 64],
    'logistic_alpha': np.logspace(-4, 4, 5),
}
search = GridSearchCV(pipe, param_grid, iid=False, cv=5)
search.fit(X_digits, y_digits)
print("Best parameter (CV score=%0.3f):" % search.best_score_)
print(search.best_params_)

# Plot the PCA spectrum
pca.fit(X_digits)

fig, (ax0, ax1) = plt.subplots(nrows=2, sharex=True, figsize=(6, 6))
ax0.plot(pca.explained_variance_ratio_, linewidth=2)
ax0.set_ylabel('PCA explained variance')

ax0.axvline(search.best_estimator_.named_steps['pca'].n_components,
            linestyle=':', label='n_components chosen')
ax0.legend(prop=dict(size=12))

# For each number of components, find the best classifier results
results = pd.DataFrame(search.cv_results_)
components_col = 'param_pca_n_components'
best_clfs = results.groupby(components_col).apply(
    lambda g: g.nlargest(1, 'mean_test_score'))

best_clfs.plot(x=components_col, y='mean_test_score', yerr='std_test_score',
                legend=False, ax=ax1)
ax1.set_ylabel('Classification accuracy (val)')
ax1.set_xlabel('n_components')

plt.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 20.748 seconds)

Note: Click [here](#) to download the full example code

5.7.3 Column Transformer with Mixed Types

This example illustrates how to apply different preprocessing and feature extraction pipelines to different subsets of features, using `sklearn.compose.ColumnTransformer`. This is particularly handy for the case of datasets that contain heterogeneous data types, since we may want to scale the numeric features and one-hot encode the categorical ones.

In this example, the numeric data is standard-scaled after mean-imputation, while the categorical data is one-hot encoded after imputing missing values with a new category ('missing').

Finally, the preprocessing pipeline is integrated in a full prediction pipeline using `sklearn.pipeline.Pipeline`, together with a simple classification model.

```
# Author: Pedro Morales <part.morales@gmail.com>
#
# License: BSD 3 clause

import pandas as pd
import numpy as np

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, GridSearchCV

np.random.seed(0)

# Read data from Titanic dataset.
titanic_url = ('https://raw.githubusercontent.com/amueller/'
               'scipy-2017-sklearn/091d371/notebooks/datasets/titanic3.csv')
data = pd.read_csv(titanic_url)

# We will train our classifier with the following features:
# Numeric Features:
# - age: float.
# - fare: float.
# Categorical Features:
# - embarked: categories encoded as strings {'C', 'S', 'Q'}.
# - sex: categories encoded as strings {'female', 'male'}.
# - pclass: ordinal integers {1, 2, 3}.

# We create the preprocessing pipelines for both numeric and categorical data.
numeric_features = ['age', 'fare']
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_features = ['embarked', 'sex', 'pclass']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])
```

```
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])  
  
# Append classifier to preprocessing pipeline.  
# Now we have a full prediction pipeline.  
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('classifier', LogisticRegression(solver='lbfgs'))])  
  
X = data.drop('survived', axis=1)  
y = data['survived']  
  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)  
  
clf.fit(X_train, y_train)  
print("model score: %.3f" % clf.score(X_test, y_test))
```

Out:

```
model score: 0.790
```

Using the prediction pipeline in a grid search

Grid search can also be performed on the different preprocessing steps defined in the `ColumnTransformer` object, together with the classifier's hyperparameters as part of the `Pipeline`. We will search for both the imputer strategy of the numeric preprocessing and the regularization parameter of the logistic regression using `sklearn.model_selection.GridSearchCV`.

```
param_grid = {  
    'preprocessor_num_imputer_strategy': ['mean', 'median'],  
    'classifier_C': [0.1, 1.0, 10, 100],  
}  
  
grid_search = GridSearchCV(clf, param_grid, cv=10, iid=False)  
grid_search.fit(X_train, y_train)  
  
print(("best logistic regression from grid search: %.3f"
      % grid_search.score(X_test, y_test)))
```

Out:

```
best logistic regression from grid search: 0.798
```

Total running time of the script: (0 minutes 2.103 seconds)

Note: Click [here](#) to download the full example code

5.7.4 Selecting dimensionality reduction with Pipeline and GridSearchCV

This example constructs a pipeline that does dimensionality reduction followed by prediction with a support vector classifier. It demonstrates the use of `GridSearchCV` and `Pipeline` to optimize over different classes of estimators

in a single CV run – unsupervised PCA and NMF dimensionality reductions are compared to univariate feature selection during the grid search.

Additionally, `Pipeline` can be instantiated with the `memory` argument to memoize the transformers within the pipeline, avoiding to fit again the same transformers over and over.

Note that the use of `memory` to enable caching becomes interesting when the fitting of a transformer is costly.

Illustration of Pipeline and GridSearchCV

This section illustrates the use of a `Pipeline` with `GridSearchCV`

```
# Authors: Robert McGibbon, Joel Nothman, Guillaume Lemaitre

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.svm import LinearSVC
from sklearn.decomposition import PCA, NMF
from sklearn.feature_selection import SelectKBest, chi2

print(__doc__)

pipe = Pipeline([
    # the reduce_dim stage is populated by the param_grid
    ('reduce_dim', 'passthrough'),
    ('classify', LinearSVC(dual=False, max_iter=10000))
])

N_FEATURES_OPTIONS = [2, 4, 8]
C_OPTIONS = [1, 10, 100, 1000]
param_grid = [
    {
        'reduce_dim': [PCA(iterated_power=7), NMF()],
        'reduce_dim_n_components': N_FEATURES_OPTIONS,
        'classify_C': C_OPTIONS
    },
    {
        'reduce_dim': [SelectKBest(chi2)],
        'reduce_dim_k': N_FEATURES_OPTIONS,
        'classify_C': C_OPTIONS
    },
]
reducer_labels = ['PCA', 'NMF', 'KBest(chi2)']

grid = GridSearchCV(pipe, cv=5, n_jobs=1, param_grid=param_grid, iid=False)
digits = load_digits()
grid.fit(digits.data, digits.target)

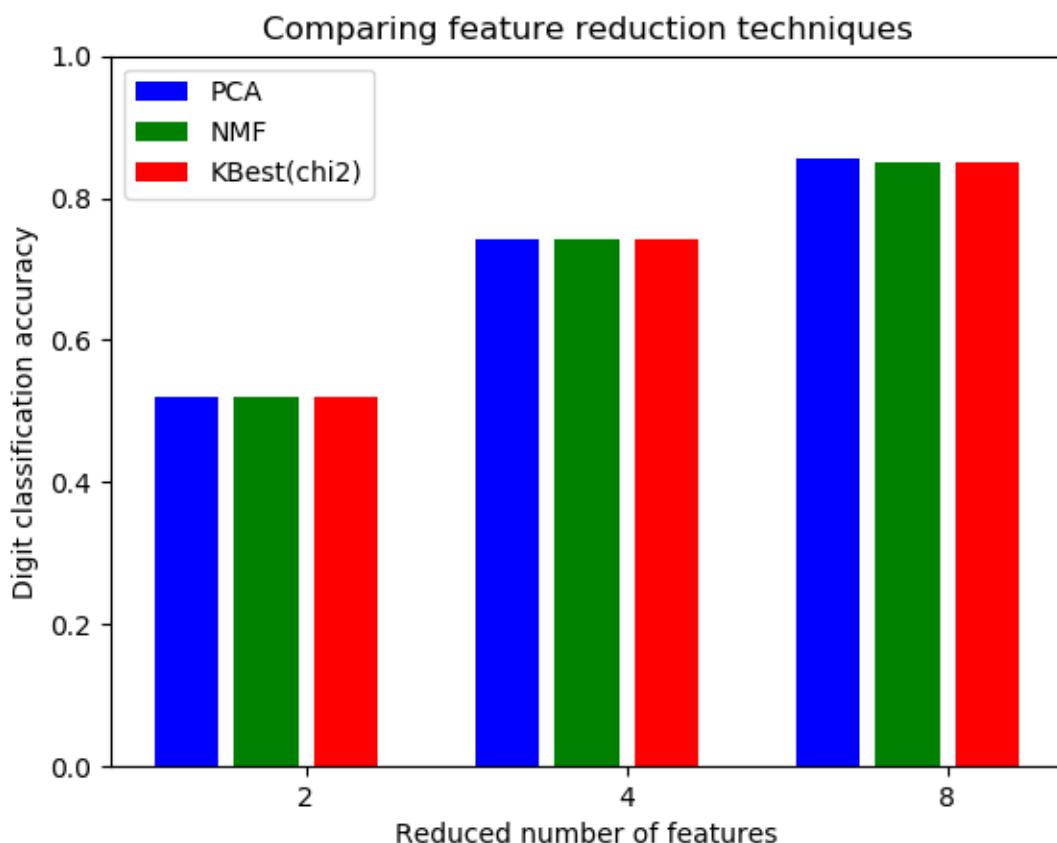
mean_scores = np.array(grid.cv_results_['mean_test_score'])
# scores are in the order of param_grid iteration, which is alphabetical
mean_scores = mean_scores.reshape(len(C_OPTIONS), -1, len(N_FEATURES_OPTIONS))
# select score for best C
mean_scores = mean_scores.max(axis=0)
bar_offsets = (np.arange(len(N_FEATURES_OPTIONS)) *
```

```
(len(reducer_labels) + 1) + .5)

plt.figure()
COLORS = 'bgrcmyk'
for i, (label, reducer_scores) in enumerate(zip(reducer_labels, mean_scores)):
    plt.bar(bar_offsets + i, reducer_scores, label=label, color=COLORS[i])

plt.title("Comparing feature reduction techniques")
plt.xlabel('Reduced number of features')
plt.xticks(bar_offsets + len(reducer_labels) / 2, N_FEATURES_OPTIONS)
plt.ylabel('Digit classification accuracy')
plt.ylim((0, 1))
plt.legend(loc='upper left')

plt.show()
```



Caching transformers within a Pipeline

It is sometimes worthwhile storing the state of a specific transformer since it could be used again. Using a pipeline in `GridSearchCV` triggers such situations. Therefore, we use the argument `memory` to enable caching.

Warning: Note that this example is, however, only an illustration since for this specific case fitting PCA is not necessarily slower than loading the cache. Hence, use the `memory` constructor parameter when the fitting of a transformer is costly.

```
from tempfile import mkdtemp
from shutil import rmtree
from joblib import Memory

# Create a temporary folder to store the transformers of the pipeline
cachedir = mkdtemp()
memory = Memory(location=cachedir, verbose=10)
cached_pipe = Pipeline([('reduce_dim', PCA()),
                       ('classify', LinearSVC(dual=False, max_iter=10000))],
                      memory=memory)

# This time, a cached pipeline will be used within the grid search
grid = GridSearchCV(cached_pipe, cv=5, n_jobs=1, param_grid=param_grid,
                     iid=False)
digits = load_digits()
grid.fit(digits.data, digits.target)

# Delete the temporary cache before exiting
rmtree(cachedir)
```

Out:

```
[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(PCA(iterated_power=7, n_components=2), array([[0., ..., 0.],
   ...
   [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline',
↪message=None)
                                         fit_transform_one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(PCA(iterated_power=7, n_components=2), array([[0., ..., 0.],
   ...
   [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline',
↪message=None)
                                         fit_transform_one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(PCA(iterated_power=7, n_components=2), array([[0., ..., 0.],
   ...
   [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline',
↪message=None)
                                         fit_transform_one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(PCA(iterated_power=7, n_components=2), array([[0., ..., 0.],
   ...
   [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline',
↪message=None)
                                         fit_transform_one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fit_transform_one...
```

```

_fi t _transform _one (PCA (iterated _power = 7, n _components = 2), array ([[0., ..., 0.],
    ...,
    [0., ..., 0.])), array ([0, ..., 9]), None, message _clsname ='Pipeline', ↵
↳ message = None)
                                            fi t _transform _one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fi t _transform _one...
_fi t _transform _one (PCA (iterated _power = 7, n _components = 4), array ([[0., ..., 0.],
    ...,
    [0., ..., 0.])), array ([0, ..., 8]), None, message _clsname ='Pipeline', ↵
↳ message = None)
                                            fi t _transform _one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fi t _transform _one...
_fi t _transform _one (PCA (iterated _power = 7, n _components = 4), array ([[0., ..., 0.],
    ...,
    [0., ..., 0.])), array ([0, ..., 8]), None, message _clsname ='Pipeline', ↵
↳ message = None)
                                            fi t _transform _one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fi t _transform _one...
_fi t _transform _one (PCA (iterated _power = 7, n _components = 4), array ([[0., ..., 0.],
    ...,
    [0., ..., 0.])), array ([0, ..., 8]), None, message _clsname ='Pipeline', ↵
↳ message = None)
                                            fi t _transform _one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fi t _transform _one...
_fi t _transform _one (PCA (iterated _power = 7, n _components = 4), array ([[0., ..., 0.],
    ...,
    [0., ..., 0.])), array ([0, ..., 8]), None, message _clsname ='Pipeline', ↵
↳ message = None)
                                            fi t _transform _one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fi t _transform _one...
_fi t _transform _one (PCA (iterated _power = 7, n _components = 4), array ([[0., ..., 0.],
    ...,
    [0., ..., 0.])), array ([0, ..., 8]), None, message _clsname ='Pipeline', ↵
↳ message = None)
                                            fi t _transform _one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fi t _transform _one...
_fi t _transform _one (PCA (iterated _power = 7, n _components = 4), array ([[0., ..., 0.],
    ...,
    [0., ..., 0.])), array ([0, ..., 9]), None, message _clsname ='Pipeline', ↵
↳ message = None)
                                            fi t _transform _one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fi t _transform _one...
_fi t _transform _one (PCA (iterated _power = 7, n _components = 8), array ([[0., ..., 0.],
    ...,
    [0., ..., 0.])), array ([0, ..., 8]), None, message _clsname ='Pipeline', ↵
↳ message = None)
                                            fi t _transform _one - 0.1s, 0.0min

[Memory] Calling sklearn.pipeline._fi t _transform _one...
_fi t _transform _one (PCA (iterated _power = 7, n _components = 8), array ([[0., ..., 0.],
    ...,
    [0., ..., 0.])), array ([0, ..., 8]), None, message _clsname ='Pipeline', ↵
↳ message = None)
                                            fi t _transform _one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fi t _transform _one...
_fi t _transform _one (PCA (iterated _power = 7, n _components = 8), array ([[0., ..., 0.],
    ...,
    [0., ..., 0.])), array ([0, ..., 8]), None, message _clsname ='Pipeline', ↵
↳ message = None)
                                            fi t _transform _one - 0.0s, 0.0min

```

```
[0., ... , 0.]], array([0, ... , 8]), None, message_cliname='Pipeline',  
↪message=None)  
fit_transform_one - 0.0s, 0.0min  


---

  
[Memory] Calling sklearn.pipeline._fit_transform_one...  
_fit_transform_one(PCA(iterated_power=7, n_components=8), array([[0., ... , 0.],  
    ...,  
    [0., ... , 0.]], array([0, ... , 8]), None, message_cliname='Pipeline',  
↪message=None)  
fit_transform_one - 0.0s, 0.0min  


---

  
[Memory] Calling sklearn.pipeline._fit_transform_one...  
_fit_transform_one(PCA(iterated_power=7, n_components=8), array([[0., ... , 0.],  
    ...,  
    [0., ... , 0.]], array([0, ... , 9]), None, message_cliname='Pipeline',  
↪message=None)  
fit_transform_one - 0.0s, 0.0min  


---

  
[Memory] Calling sklearn.pipeline._fit_transform_one...  
_fit_transform_one(NMF(n_components=2), array([[0., ... , 0.],  
    ...,  
    [0., ... , 0.]], array([0, ... , 8]), None, message_cliname='Pipeline',  
↪message=None)  
fit_transform_one - 0.1s, 0.0min  


---

  
[Memory] Calling sklearn.pipeline._fit_transform_one...  
_fit_transform_one(NMF(n_components=2), array([[0., ... , 0.],  
    ...,  
    [0., ... , 0.]], array([0, ... , 8]), None, message_cliname='Pipeline',  
↪message=None)  
fit_transform_one - 0.0s, 0.0min  


---

  
[Memory] Calling sklearn.pipeline._fit_transform_one...  
_fit_transform_one(NMF(n_components=2), array([[0., ... , 0.],  
    ...,  
    [0., ... , 0.]], array([0, ... , 8]), None, message_cliname='Pipeline',  
↪message=None)  
fit_transform_one - 0.1s, 0.0min  


---

  
[Memory] Calling sklearn.pipeline._fit_transform_one...  
_fit_transform_one(NMF(n_components=2), array([[0., ... , 0.],  
    ...,  
    [0., ... , 0.]], array([0, ... , 8]), None, message_cliname='Pipeline',  
↪message=None)  
fit_transform_one - 0.0s, 0.0min  


---

  
[Memory] Calling sklearn.pipeline._fit_transform_one...  
_fit_transform_one(NMF(n_components=2), array([[0., ... , 0.],  
    ...,  
    [0., ... , 0.]], array([0, ... , 9]), None, message_cliname='Pipeline',  
↪message=None)  
fit_transform_one - 0.0s, 0.0min  


---

  
[Memory] Calling sklearn.pipeline._fit_transform_one...  
_fit_transform_one(NMF(n_components=4), array([[0., ... , 0.],  
    ...,  
    [0., ... , 0.]], array([0, ... , 8]), None, message_cliname='Pipeline',  
↪message=None)
```

```
fit_transform_one - 0.1s, 0.0min
[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(NMF(n_components=4), array([[0., ..., 0.],
   ...,
   [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline', ↵
message=None)
fit_transform_one - 0.0s, 0.0min
[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(NMF(n_components=4), array([[0., ..., 0.],
   ...,
   [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline', ↵
message=None)
fit_transform_one - 0.0s, 0.0min
[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(NMF(n_components=4), array([[0., ..., 0.],
   ...,
   [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline', ↵
message=None)
fit_transform_one - 0.0s, 0.0min
[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(NMF(n_components=4), array([[0., ..., 0.],
   ...,
   [0., ..., 0.]]), array([0, ..., 9]), None, message_cliname='Pipeline', ↵
message=None)
fit_transform_one - 0.1s, 0.0min
[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(NMF(n_components=8), array([[0., ..., 0.],
   ...,
   [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline', ↵
message=None)
fit_transform_one - 0.1s, 0.0min
[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(NMF(n_components=8), array([[0., ..., 0.],
   ...,
   [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline', ↵
message=None)
fit_transform_one - 0.1s, 0.0min
[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(NMF(n_components=8), array([[0., ..., 0.],
   ...,
   [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline', ↵
message=None)
fit_transform_one - 0.2s, 0.0min
[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(NMF(n_components=8), array([[0., ..., 0.],
   ...,
   [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline', ↵
message=None)
fit_transform_one - 0.1s, 0.0min
```

```
[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(NMF(n_components=8), array([[0., ..., 0.,
    ...
    [0., ..., 0.]]), array([0, ..., 9]), None, message_clsname='Pipeline',_
message=None)
                                         fit_transform_one - 0.1s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/591acae8e60c9632aa990e4fa0962a67
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/323909961fcc2a4950defb581f08007f
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/3654e1d61587aee4f9980c99541d55d3
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/835820e75be3172b4e2e257028147bb2
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/90604ce8bb756e3ff1acd4c1ce065b1a
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/3c66df347f488dfe044ecf991ca2498b
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/bf4bec5fe5245bf3c0d271e31bc2342f
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/ce0f911df93a8e38932c48f983e7fde6
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/285efba3f9e5b1a35d6825e37a718019
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/74083d6e1ed743fa1e756a41ba467e6c
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/b72f745822a406f2191b6ebfd8ca9df9
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/997e4ce30ba5143e90330db664bf61c5
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/3140177f6ccbe72992772342347a166f
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/f843896ad42a20c4a50c0fd29512111a
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/d91a2d083b8c7311c87edd6592fc446
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/50eb0f4152cd0f7dce3b471add31d07d
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/3b28284b39c934cb519f2a31e03a951b
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
→sklearn/pipeline/_fit_transform_one/aded0eb932b9c692e801e6e56de2bda2
```

```

    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/cdf3d1530a8a2388ba80a16c66413409
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/ff5666a0cb943711a4e4dfd5b5cd8587
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/536a5fbb5a6c8d69b4fb426f0b51d9eb
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/26cd3a8242acb986f98cf38291ef5baa
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/8b6094f421dab9fc88fb54c2f32e16b4
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/d0e8598c62dcb5f059a5c2b541c23b09
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/34affb50a3d55c5302e1c7f97aec9679
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/b3635522e8f89bbe9e56ab7e7f4693fc
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/7389ad9bae0ad1181b7da9e4dd39449d
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/ab59173e00ab4a7bdf7740e5c6d50123
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/42894538fd0ab2b4235e55ef53b9cd59
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/580596a10617556e39d5e069c4de096a
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/591aca8e60c9632aa990e4fa0962a67
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/323909961fcc2a4950defb581f08007f
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/3654e1d61587aee4f9980c99541d55d3
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/835820e75be3172b4e2e257028147bb2
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/90604ce8bb756e3ff1acd4c1ce065b1a
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/3c66df347f488dfe044ecf991ca2498b
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/bf4bec5fe5245bf3c0d271e31bc2342f
    _____fit_transform_one cache loaded - 0.0s, 0.0min

```

```
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/ce0f911df93a8e38932c48f983e7fde6
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/285efba3f9e5b1a35d6825e37a718019
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/74083d6e1ed743fa1e756a41ba467e6c
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/b72f745822a406f2191b6ebfd8ca9df9
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/997e4ce30ba5143e90330db664bf61c5
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/3140177f6ccbe72992772342347a166f
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/f843896ad42a20c4a50c0fd29512111a
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/d91a2d083b8c7311c87edd6592fc446
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/50eb0f4152cd0f7dce3b471add31d07d
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/3b28284b39c934cb519f2a31e03a951b
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/aded0eb932b9c692c801c6c56dc2bda2
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/cdf3d1530a8a2388ba80a16c66413409
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/ff5666a0cb943711a4e4dfd5b5cd8587
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/536a5fbb5a6c8d69b4fb426f0b51d9eb
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/26cd3a8242acb986f98cf38291ef5baa
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/8b6094f421dab9fc88fb54c2f32e16b4
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/d0e8598c62dcb5f059a5c2b541c23b09
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/34affb50a3d55c5302e1c7f97aec9679
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/b3635522e8f89bbe9e56ab7e7f4693fc
    _____fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/7389ad9bae0ad1181b7da9e4dd39449d
```

```

[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/ab59173e00ab4a7bdf7740e5c6d50123
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/42894538fd0ab2b4235e55ef53b9cd59
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/580596a10617556e39d5e069c4de096a
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/591acaee8e60c9632aa990e4fa0962a67
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/323909961fcc2a4950defb581f08007f
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/3654e1d61587aee4f9980c99541d55d3
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/835820e75be3172b4e2e257028147bb2
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/90604ce8bb756e3ff1acd4c1ce065b1a
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/3c66df347f488dfe044ecf991ca2498b
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/bf4bec5fe5245bf3c0d271e31bc2342f
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/ce0f911df93a8e38932c48f983e7fde6
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/285efba3f9e5b1a35d6825e37a718019
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/74083d6e1ed743fa1e756a41ba467e6c
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/b72f745822a406f2191b6ebfd8ca9df9
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/997e4ce30ba5143e90330db664bf61c5
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/3140177f6ccbe72992772342347a166f
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/f843896ad42a20c4a50c0fd29512111a
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/d91a2d083b8c7311c87edd6592fc446
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/50eb0f4152cd0f7dce3b471add31d07d
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/50eb0f4152cd0f7dce3b471add31d07d

```

```
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/3b28284b39c934cb519f2a31e03a951b
˓→                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/aded0eb932b9c692c801c6c56dc2bda2
˓→                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/cdf3d1530a8a2388ba80a16c66413409
˓→                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/ff5666a0cb943711a4e4dfd5b5cd8587
˓→                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/536a5fbb5a6c8d69b4fb426f0b51d9eb
˓→                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/26cd3a8242acb986f98cf38291ef5baa
˓→                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/8b6094f421dab9fc88fb54c2f32e16b4
˓→                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/d0e8598c62dcb5f059a5c2b541c23b09
˓→                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/34affb50a3d55c5302e1c7f97aec9679
˓→                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/b3635522e8f89bbe9e56ab7e7f4693fc
˓→                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/7389ad9bae0ad1181b7da9e4dd39449d
˓→                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/ab59173e00ab4a7bdf7740e5c6d50123
˓→                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/42894538fd0ab2b4235e55ef53b9cd59
˓→                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/580596a10617556e39d5e069c4de096a
˓→                                         fit_transform_one cache loaded - 0.0s, 0.0min
_____
[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(SelectKBest(k=2, score_func=<function chi2 at 0x7efe30bb2268>),
˓→array([[0., ..., 0.],
˓→       ...,
˓→       [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline',
˓→message=None)
˓→                                         fit_transform_one - 0.0s, 0.0min
_____
[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(SelectKBest(k=2, score_func=<function chi2 at 0x7efe30bb2268>),
˓→array([[0., ..., 0.],
˓→       ...,
˓→       [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline',
˓→message=None)
˓→                                         fit_transform_one - 0.0s, 0.0min
_____
```

```
[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(SelectKBest(k=2, score_func=<function chi2 at 0x7efe30bb2268>),
array([[0., ..., 0.],
       ...,
       [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline',
message=None)
                                         fit_transform_one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(SelectKBest(k=2, score_func=<function chi2 at 0x7efe30bb2268>),
array([[0., ..., 0.],
       ...,
       [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline',
message=None)
                                         fit_transform_one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(SelectKBest(k=2, score_func=<function chi2 at 0x7efe30bb2268>),
array([[0., ..., 0.],
       ...,
       [0., ..., 0.]]), array([0, ..., 9]), None, message_cliname='Pipeline',
message=None)
                                         fit_transform_one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(SelectKBest(k=4, score_func=<function chi2 at 0x7efe30bb2268>),
array([[0., ..., 0.],
       ...,
       [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline',
message=None)
                                         fit_transform_one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(SelectKBest(k=4, score_func=<function chi2 at 0x7efe30bb2268>),
array([[0., ..., 0.],
       ...,
       [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline',
message=None)
                                         fit_transform_one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(SelectKBest(k=4, score_func=<function chi2 at 0x7efe30bb2268>),
array([[0., ..., 0.],
       ...,
       [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline',
message=None)
                                         fit_transform_one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(SelectKBest(k=4, score_func=<function chi2 at 0x7efe30bb2268>),
array([[0., ..., 0.],
       ...,
       [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline',
message=None)
                                         fit_transform_one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fit_transform_one...
```

```

_fit_transform_one(SelectKBest(k=4, score_func=<function chi2 at 0x7efe30bb2268>),
    array([[0., ..., 0.],
        ...,
        [0., ..., 0.]]), array([0, ..., 9]), None, message_cliname='Pipeline',
    message=None)
    fit_transform_one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(SelectKBest(k=8, score_func=<function chi2 at 0x7efe30bb2268>),
    array([[0., ..., 0.],
        ...,
        [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline',
    message=None)
    fit_transform_one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(SelectKBest(k=8, score_func=<function chi2 at 0x7efe30bb2268>),
    array([[0., ..., 0.],
        ...,
        [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline',
    message=None)
    fit_transform_one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(SelectKBest(k=8, score_func=<function chi2 at 0x7efe30bb2268>),
    array([[0., ..., 0.],
        ...,
        [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline',
    message=None)
    fit_transform_one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(SelectKBest(k=8, score_func=<function chi2 at 0x7efe30bb2268>),
    array([[0., ..., 0.],
        ...,
        [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline',
    message=None)
    fit_transform_one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(SelectKBest(k=8, score_func=<function chi2 at 0x7efe30bb2268>),
    array([[0., ..., 0.],
        ...,
        [0., ..., 0.]]), array([0, ..., 8]), None, message_cliname='Pipeline',
    message=None)
    fit_transform_one - 0.0s, 0.0min

[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(SelectKBest(k=8, score_func=<function chi2 at 0x7efe30bb2268>),
    array([[0., ..., 0.],
        ...,
        [0., ..., 0.]]), array([0, ..., 9]), None, message_cliname='Pipeline',
    message=None)
    fit_transform_one - 0.0s, 0.0min

[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
    sklearn/pipeline/_fit_transform_one/0c252f8c2af87fe4a595ca853ea983cd
    fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
    sklearn/pipeline/_fit_transform_one/5feb279acc70b729ede7514b133d0172
    fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
    sklearn/pipeline/_fit_transform_one/6742bad9e26dd16b831fc9abd1883c9e
    fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
    sklearn/pipeline/_fit_transform_one/096380a7ad90487199a4a6e8ec8a5744
    fit_transform_one cache loaded - 0.0s, 0.0min

```

```
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/e5107ef0d6468f91a5cd772a596ba876
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/2efa56788b791becec2fc015ecf751f
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/d27c6056fc9ab1f3fb4327fd17346312
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/d649c35672f6e2731789f5c6a9b03066
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/4906c4afa619398fb77288e086d5dd82
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/7dc3c29273fd5c57e2913f82c9185294
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/63c1c1af7115994b5cd17a1189691d5b
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/670746700924677f7a550fa4cae5c9bb
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/f9617292c7b763d23b3f6c9d96697c29
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/9a2bcb4639d1b192e5525c2bf0dca317
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/87c7c41885d63769a203e26b244fd823
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/0c252f8c2af87fe4a595ca853ea983cd
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/5feb279acc70b729ede7514b133d0172
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/6742bad9e26dd16b831fc9abd1883c9e
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/096380a7ad90487199a4a6e8ec8a5744
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/e5107ef0d6468f91a5cd772a596ba876
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/2efa56788b791becec2fc015ecf751f
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/d27c6056fc9ab1f3fb4327fd17346312
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/d649c35672f6e2731789f5c6a9b03066
                                fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/4906c4afa619398fb77288e086d5dd82
```

```

    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/7dc3c29273fd5c57e2913f82c9185294
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/63c1c1af7115994b5cd17a1189691d5b
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/670746700924677f7a550fa4cae5c9bb
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/f9617292c7b763d23b3f6c9d96697c29
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/9a2bcb4639d1b192e5525c2bf0dca317
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/87c7c41885d63769a203e26b244fd823
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/0c252f8c2af87fe4a595ca853ea983cd
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/5feb279acc70b729ede7514b133d0172
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/6742bad9e26dd16b831fc9abd1883c9e
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/096380a7ad90487199a4a6e8ec8a5744
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/e5107ef0d6468f91a5cd772a596ba876
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/2efa56788b791becec2fc015ecf751f
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/d27c6056fc9ab1f3fb4327fd17346312
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/d649c35672f6e2731789f5c6a9b03066
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/4906c4afa619398fb77288e086d5dd82
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/7dc3c29273fd5c57e2913f82c9185294
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/63c1c1af7115994b5cd17a1189691d5b
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/670746700924677f7a550fa4cae5c9bb
    _fit_transform_one cache loaded - 0.0s, 0.0min
[Memory]0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvv/joblib/
˓→sklearn/pipeline/_fit_transform_one/f9617292c7b763d23b3f6c9d96697c29
    _fit_transform_one cache loaded - 0.0s, 0.0min

```

```
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/9a2bcb4639d1b192e5525c2bf0dca317
                                         fit_transform_one cache loaded - 0.0s, 0.0min
[Memory] 0.0s, 0.0min : Loading _fit_transform_one from /tmp/tmp15ribdvu/joblib/
˓→sklearn/pipeline/_fit_transform_one/87c7c41885d63769a203e26b244fd823
                                         fit_transform_one cache loaded - 0.0s, 0.0min
_____
[Memory] Calling sklearn.pipeline._fit_transform_one...
_fit_transform_one(PCA(iterated_power=7, n_components=8), array([[0., ..., 0.],
   ...,
   [0., ..., 0.]]), array([0, ..., 8]), None, message_clsname='Pipeline',
˓→message=None)
                                         fit_transform_one - 0.0s, 0.0min
```

The PCA fitting is only computed at the evaluation of the first configuration of the C parameter of the LinearSVC classifier. The other configurations of C will trigger the loading of the cached PCA estimator data, leading to save processing time. Therefore, the use of caching the pipeline using `memory` is highly beneficial when fitting a transformer is costly.

Total running time of the script: (0 minutes 20.695 seconds)

Note: Click [here](#) to download the full example code

5.7.5 Column Transformer with Heterogeneous Data Sources

Datasets can often contain components of that require different feature extraction and processing pipelines. This scenario might occur when:

1. Your dataset consists of heterogeneous data types (e.g. raster images and text captions)
2. Your dataset is stored in a Pandas DataFrame and different columns require different processing pipelines.

This example demonstrates how to use `sklearn.compose.ColumnTransformer` on a dataset containing different types of features. We use the 20-newsgroups dataset and compute standard bag-of-words features for the subject line and body in separate pipelines as well as ad hoc features on the body. We combine them (with weights) using a ColumnTransformer and finally train a classifier on the combined set of features.

The choice of features is not particularly helpful, but serves to illustrate the technique.

Out:

```
[Pipeline] ..... (step 1 of 3) Processing subjectbody, total= 0.1s
[Pipeline] ..... (step 2 of 3) Processing union, total= 0.3s
[Pipeline] ..... (step 3 of 3) Processing svc, total= 0.3s
precision      recall    f1-score   support
          0       0.96      0.62      0.76      494
          1       0.25      0.84      0.39       76

accuracy           0.65      570
macro avg       0.61      0.73      0.57      570
weighted avg     0.87      0.65      0.71      570
```

```

# Author: Matt Terry <matt.terry@gmail.com>
#
# License: BSD 3 clause

import numpy as np

from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.datasets import fetch_20newsgroups
from sklearn.datasets.twenty_newsgroups import strip_newsgroup_footer
from sklearn.datasets.twenty_newsgroups import strip_newsgroup_quoting
from sklearn.decomposition import TruncatedSVD
from sklearn.feature_extraction import DictVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import classification_report
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.svm import LinearSVC


class TextStats(BaseEstimator, TransformerMixin):
    """Extract features from each document for DictVectorizer"""

    def fit(self, x, y=None):
        return self

    def transform(self, posts):
        return [{ 'length': len(text),
                  'num_sentences': text.count('.') }
                for text in posts]


class SubjectBodyExtractor(BaseEstimator, TransformerMixin):
    """Extract the subject & body from a usenet post in a single pass.

    Takes a sequence of strings and produces a dict of sequences. Keys are
    `subject` and `body`.
    """

    def fit(self, x, y=None):
        return self

    def transform(self, posts):
        # construct object dtype array with two columns
        # first column = 'subject' and second column = 'body'
        features = np.empty(shape=(len(posts), 2), dtype=object)
        for i, text in enumerate(posts):
            headers, _, bod = text.partition('\n\n')
            bod = strip_newsgroup_footer(bod)
            bod = strip_newsgroup_quoting(bod)
            features[i, 1] = bod

            prefix = 'Subject:'
            sub = ''
            for line in headers.split('\n'):
                if line.startswith(prefix):
                    sub = line[len(prefix):]
                    break
            features[i, 0] = sub

```

```
    return features

pipeline = Pipeline([
    # Extract the subject & body
    ('subjectbody', SubjectBodyExtractor()),

    # Use ColumnTransformer to combine the features from subject and body
    ('union', ColumnTransformer(
        [
            # Pulling features from the post's subject line (first column)
            ('subject', TfidfVectorizer(min_df=50), 0),

            # Pipeline for standard bag-of-words model for body (second column)
            ('body_bow', Pipeline([
                ('tfidf', TfidfVectorizer()),
                ('best', TruncatedSVD(n_components=50)),
            ]), 1),
        ],
        # Pipeline for pulling ad hoc features from post's body
        ('body_stats', Pipeline([
            ('stats', TextStats()), # returns a list of dicts
            ('vect', DictVectorizer()), # list of dicts -> feature matrix
        ]), 1),
    )),
    # weight components in ColumnTransformer
    transformer_weights={
        'subject': 0.8,
        'body_bow': 0.5,
        'body_stats': 1.0,
    }
)),
    # Use a SVC classifier on the combined features
    ('svc', LinearSVC()),
], verbose=True)

# limit the list of categories to make running this example faster.
categories = ['alt.atheism', 'talk.religion.misc']
train = fetch_20newsgroups(random_state=1,
                           subset='train',
                           categories=categories,
                           )
test = fetch_20newsgroups(random_state=1,
                           subset='test',
                           categories=categories,
                           )

pipeline.fit(train.data, train.target)
y = pipeline.predict(test.data)
print(classification_report(y, test.target))
```

Total running time of the script: (0 minutes 1.258 seconds)

Note: Click [here](#) to download the full example code

5.7.6 Effect of transforming the targets in regression model

In this example, we give an overview of the `sklearn.compose.TransformedTargetRegressor`. Two examples illustrate the benefit of transforming the targets before learning a linear regression model. The first example uses synthetic data while the second example is based on the Boston housing data set.

```
# Author: Guillaume Lemaitre <guillaume.lemaitre@inria.fr>
# License: BSD 3 clause

import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from distutils.version import LooseVersion

print(__doc__)
```

Synthetic example

```
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.linear_model import RidgeCV
from sklearn.compose import TransformedTargetRegressor
from sklearn.metrics import median_absolute_error, r2_score

# `normed` is being deprecated in favor of `density` in histograms
if LooseVersion(matplotlib.__version__) >= '2.1':
    density_param = {'density': True}
else:
    density_param = {'normed': True}
```

A synthetic random regression problem is generated. The targets `y` are modified by: (i) translating all targets such that all entries are non-negative and (ii) applying an exponential function to obtain non-linear targets which cannot be fitted using a simple linear model.

Therefore, a logarithmic (`np.log1p`) and an exponential function (`np.expm1`) will be used to transform the targets before training a linear regression model and using it for prediction.

```
X, y = make_regression(n_samples=10000, noise=100, random_state=0)
y = np.exp((y + abs(y.min())) / 200)
y_trans = np.log1p(y)
```

The following illustrate the probability density functions of the target before and after applying the logarithmic functions.

```
f, (ax0, ax1) = plt.subplots(1, 2)

ax0.hist(y, bins=100, **density_param)
ax0.set_xlim([0, 2000])
ax0.set_ylabel('Probability')
ax0.set_xlabel('Target')
ax0.set_title('Target distribution')

ax1.hist(y_trans, bins=100, **density_param)
ax1.set_ylabel('Probability')
```

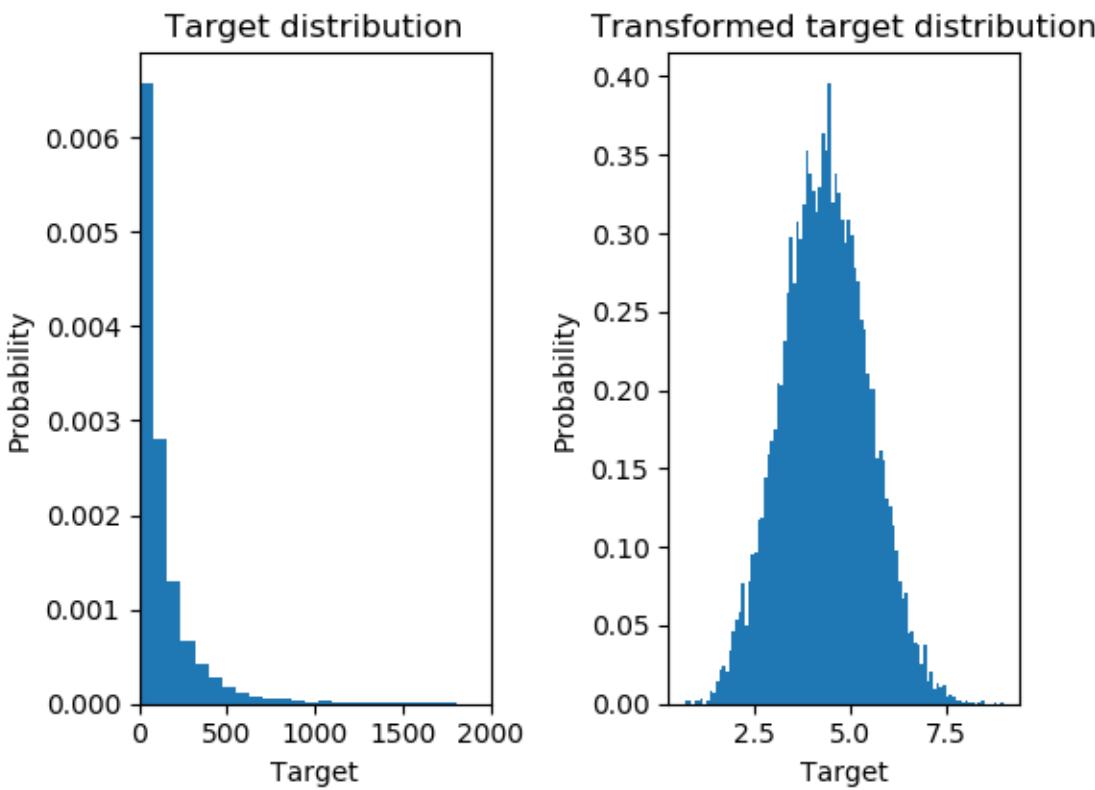
```

ax1.set_xlabel('Target')
ax1.set_title('Transformed target distribution')

f.suptitle("Synthetic data", y=0.035)
f.tight_layout(rect=[0.05, 0.05, 0.95, 0.95])

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

```



Synthetic data

At first, a linear model will be applied on the original targets. Due to the non-linearity, the model trained will not be precise during the prediction. Subsequently, a logarithmic function is used to linearize the targets, allowing better prediction even with a similar linear model as reported by the median absolute error (MAE).

```

f, (ax0, ax1) = plt.subplots(1, 2, sharey=True)

regr = RidgeCV()
regr.fit(X_train, y_train)
y_pred = regr.predict(X_test)

ax0.scatter(y_test, y_pred)
ax0.plot([0, 2000], [0, 2000], '--k')
ax0.set_ylabel('Target predicted')
ax0.set_xlabel('True Target')
ax0.set_title('Ridge regression \n without target transformation')
ax0.text(100, 1750, r'$R^2$=%.2f, MAE=%.2f' % (
    r2_score(y_test, y_pred), median_absolute_error(y_test, y_pred)))
ax0.set_xlim([0, 2000])

```

```

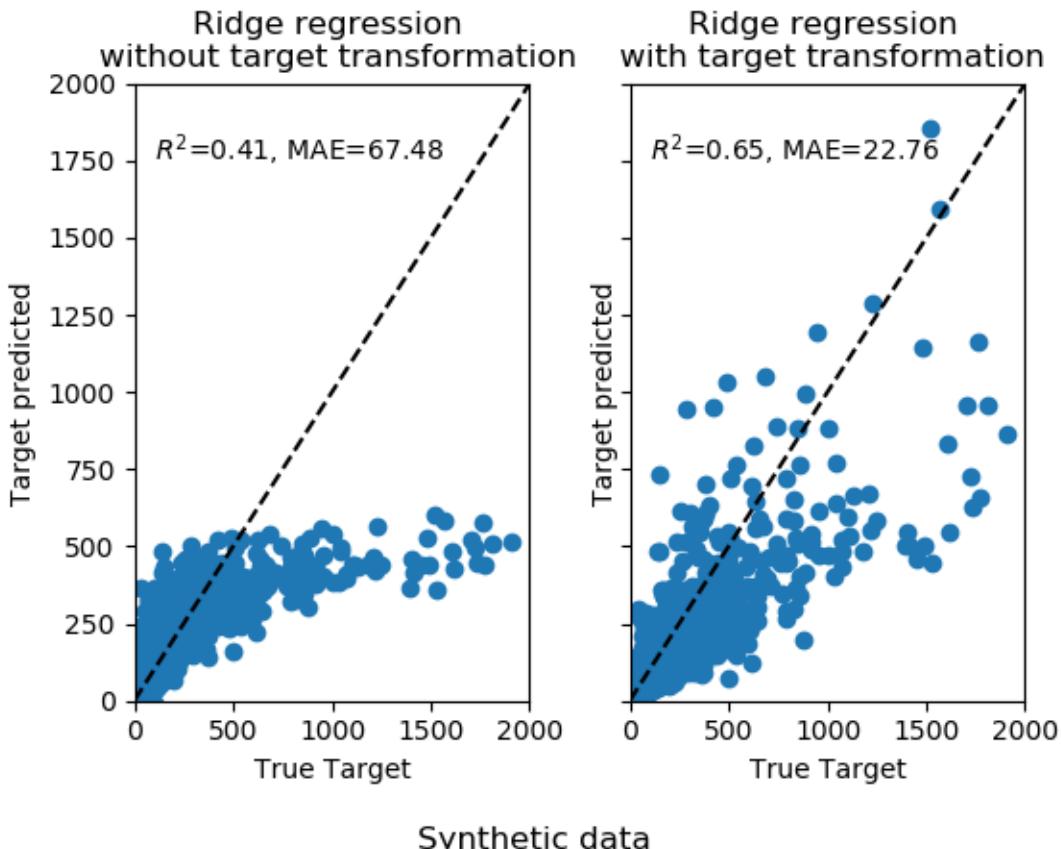
ax0.set_ylim([0, 2000])

regr_trans = TransformedTargetRegressor(regressor=RidgeCV(),
                                         func=np.log1p,
                                         inverse_func=np.expm1)
regr_trans.fit(X_train, y_train)
y_pred = regr_trans.predict(X_test)

ax1.scatter(y_test, y_pred)
ax1.plot([0, 2000], [0, 2000], '--k')
ax1.set_ylabel('Target predicted')
ax1.set_xlabel('True Target')
ax1.set_title('Ridge regression \n with target transformation')
ax1.text(100, 1750, r'$R^2$=% .2f, MAE=% .2f' % (
    r2_score(y_test, y_pred), median_absolute_error(y_test, y_pred)))
ax1.set_xlim([0, 2000])
ax1.set_ylim([0, 2000])

f.suptitle("Synthetic data", y=0.035)
f.tight_layout(rect=[0.05, 0.05, 0.95, 0.95])

```



Real-world data set

In a similar manner, the boston housing data set is used to show the impact of transforming the targets before learning a model. In this example, the targets to be predicted corresponds to the weighted distances to the five Boston employment centers.

```
from sklearn.datasets import load_boston
from sklearn.preprocessing import QuantileTransformer, quantile_transform

dataset = load_boston()
target = np.array(dataset.feature_names) == "DIS"
X = dataset.data[:, np.logical_not(target)]
y = dataset.data[:, target].squeeze()
y_trans = quantile_transform(dataset.data[:, target],
                             n_quantiles=300,
                             output_distribution='normal',
                             copy=True).squeeze()
```

A `sklearn.preprocessing.QuantileTransformer` is used such that the targets follows a normal distribution before applying a `sklearn.linear_model.RidgeCV` model.

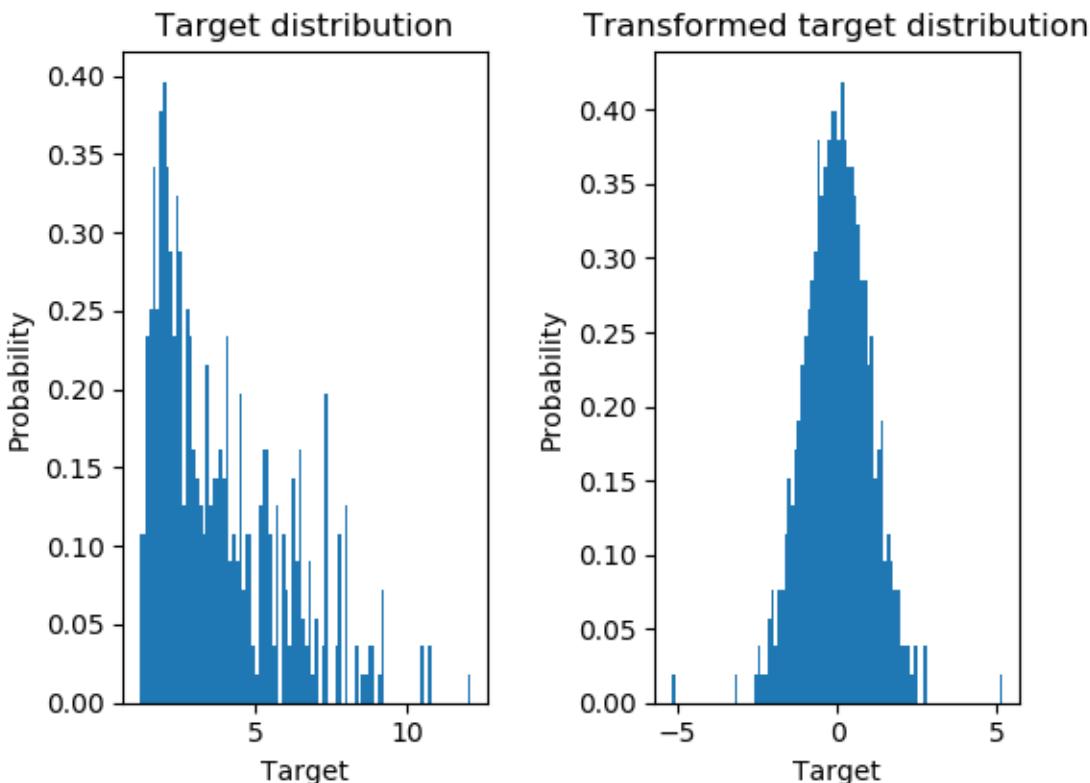
```
f, (ax0, ax1) = plt.subplots(1, 2)

ax0.hist(y, bins=100, **density_param)
ax0.set_ylabel('Probability')
ax0.set_xlabel('Target')
ax0.set_title('Target distribution')

ax1.hist(y_trans, bins=100, **density_param)
ax1.set_ylabel('Probability')
ax1.set_xlabel('Target')
ax1.set_title('Transformed target distribution')

f.suptitle("Boston housing data: distance to employment centers", y=0.035)
f.tight_layout(rect=[0.05, 0.05, 0.95, 0.95])

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)
```



Boston housing data: distance to employment centers

The effect of the transformer is weaker than on the synthetic data. However, the transform induces a decrease of the MAE.

```
f, (ax0, ax1) = plt.subplots(1, 2, sharey=True)

regr = RidgeCV()
regr.fit(X_train, y_train)
y_pred = regr.predict(X_test)

ax0.scatter(y_test, y_pred)
ax0.plot([0, 10], [0, 10], '--k')
ax0.set_ylabel('Target predicted')
ax0.set_xlabel('True Target')
ax0.set_title('Ridge regression \n without target transformation')
ax0.text(1, 9, r'$R^2$=%.{2f}, MAE=%.{2f}' %
        (r2_score(y_test, y_pred), median_absolute_error(y_test, y_pred)))
ax0.set_xlim([0, 10])
ax0.set_ylim([0, 10])

regr_trans = TransformedTargetRegressor(
    regressor=RidgeCV(),
    transformer=QuantileTransformer(n_quantiles=300,
                                    output_distribution='normal'))
regr_trans.fit(X_train, y_train)
y_pred = regr_trans.predict(X_test)
```

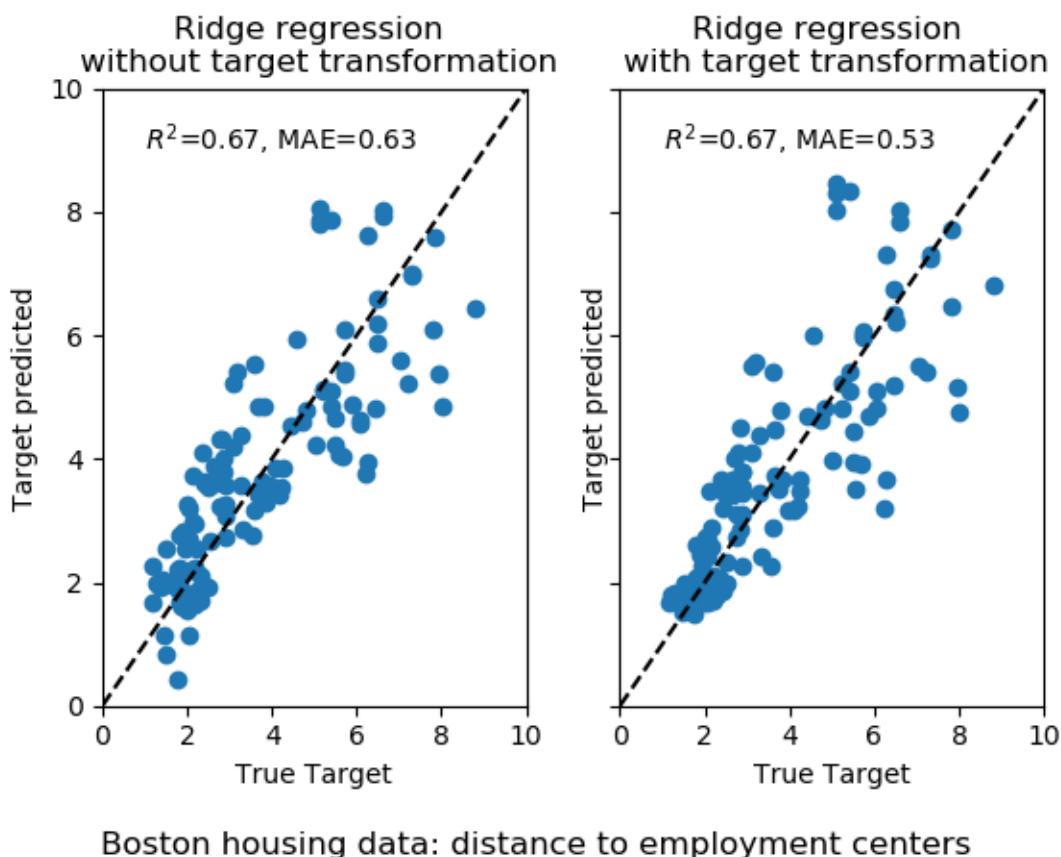
```

ax1.scatter(y_test, y_pred)
ax1.plot([0, 10], [0, 10], '--k')
ax1.set_ylabel('Target predicted')
ax1.set_xlabel('True Target')
ax1.set_title('Ridge regression \n with target transformation')
ax1.text(1, 9, r'$R^2$=%.*f, MAE=%.*f' % (
    r2_score(y_test, y_pred), median_absolute_error(y_test, y_pred)))
ax1.set_xlim([0, 10])
ax1.set_ylim([0, 10])

f.suptitle("Boston housing data: distance to employment centers", y=0.035)
f.tight_layout(rect=[0.05, 0.05, 0.95, 0.95])

plt.show()

```



Total running time of the script: (0 minutes 1.682 seconds)

5.8 Covariance estimation

Examples concerning the `sklearn.covariance` module.

Note: Click [here](#) to download the full example code

5.8.1 Ledoit-Wolf vs OAS estimation

The usual covariance maximum likelihood estimate can be regularized using shrinkage. Ledoit and Wolf proposed a close formula to compute the asymptotically optimal shrinkage parameter (minimizing a MSE criterion), yielding the Ledoit-Wolf covariance estimate.

Chen et al. proposed an improvement of the Ledoit-Wolf shrinkage parameter, the OAS coefficient, whose convergence is significantly better under the assumption that the data are Gaussian.

This example, inspired from Chen's publication [1], shows a comparison of the estimated MSE of the LW and OAS methods, using Gaussian distributed data.

[1] "Shrinkage Algorithms for MMSE Covariance Estimation" Chen et al., IEEE Trans. on Sign. Proc., Volume 58, Issue 10, October 2010.

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import toeplitz, cholesky

from sklearn.covariance import LedoitWolf, OAS

np.random.seed(0)
```

```
n_features = 100
# simulation covariance matrix (AR(1) process)
r = 0.1
real_cov = toeplitz(r ** np.arange(n_features))
coloring_matrix = cholesky(real_cov)

n_samples_range = np.arange(6, 31, 1)
repeat = 100
lw_mse = np.zeros((n_samples_range.size, repeat))
oa_mse = np.zeros((n_samples_range.size, repeat))
lw_shrinkage = np.zeros((n_samples_range.size, repeat))
oa_shrinkage = np.zeros((n_samples_range.size, repeat))
for i, n_samples in enumerate(n_samples_range):
    for j in range(repeat):
        X = np.dot(
            np.random.normal(size=(n_samples, n_features)), coloring_matrix.T)

        lw = LedoitWolf(store_precision=False, assume_centered=True)
        lw.fit(X)
        lw_mse[i, j] = lw.error_norm(real_cov, scaling=False)
        lw_shrinkage[i, j] = lw.shrinkage_

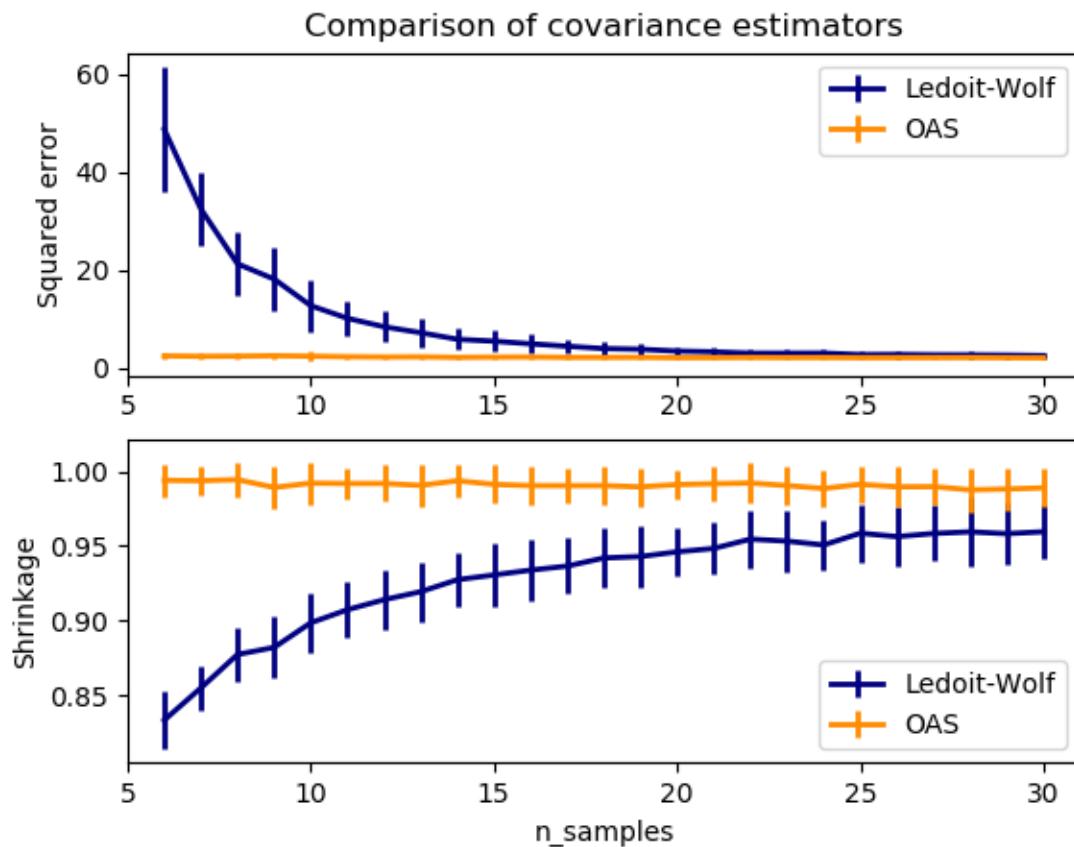
        oa = OAS(store_precision=False, assume_centered=True)
        oa.fit(X)
        oa_mse[i, j] = oa.error_norm(real_cov, scaling=False)
        oa_shrinkage[i, j] = oa.shrinkage_

# plot MSE
plt.subplot(2, 1, 1)
plt.errorbar(n_samples_range, lw_mse.mean(1), yerr=lw_mse.std(1),
             label='Ledoit-Wolf', color='navy', lw=2)
plt.errorbar(n_samples_range, oa_mse.mean(1), yerr=oa_mse.std(1),
             label='OAS', color='darkorange', lw=2)
plt.ylabel("Squared error")
```

```
plt.legend(loc="upper right")
plt.title("Comparison of covariance estimators")
plt.xlim(5, 31)

# plot shrinkage coefficient
plt.subplot(2, 1, 2)
plt.errorbar(n_samples_range, lw_shrinkage.mean(1), yerr=lw_shrinkage.std(1),
             label='Ledoit-Wolf', color='navy', lw=2)
plt.errorbar(n_samples_range, oa_shrinkage.mean(1), yerr=oa_shrinkage.std(1),
             label='OAS', color='darkorange', lw=2)
plt.xlabel("n_samples")
plt.ylabel("Shrinkage")
plt.legend(loc="lower right")
plt.ylim(plt.ylim()[0], 1. + (plt.ylim()[1] - plt.ylim()[0]) / 10.)
plt.xlim(5, 31)

plt.show()
```



Total running time of the script: (0 minutes 3.818 seconds)

Note: Click [here](#) to download the full example code

5.8.2 Sparse inverse covariance estimation

Using the GraphicalLasso estimator to learn a covariance and sparse precision from a small number of samples.

To estimate a probabilistic model (e.g. a Gaussian model), estimating the precision matrix, that is the inverse covariance matrix, is as important as estimating the covariance matrix. Indeed a Gaussian model is parametrized by the precision matrix.

To be in favorable recovery conditions, we sample the data from a model with a sparse inverse covariance matrix. In addition, we ensure that the data is not too much correlated (limiting the largest coefficient of the precision matrix) and that there are no small coefficients in the precision matrix that cannot be recovered. In addition, with a small number of observations, it is easier to recover a correlation matrix rather than a covariance, thus we scale the time series.

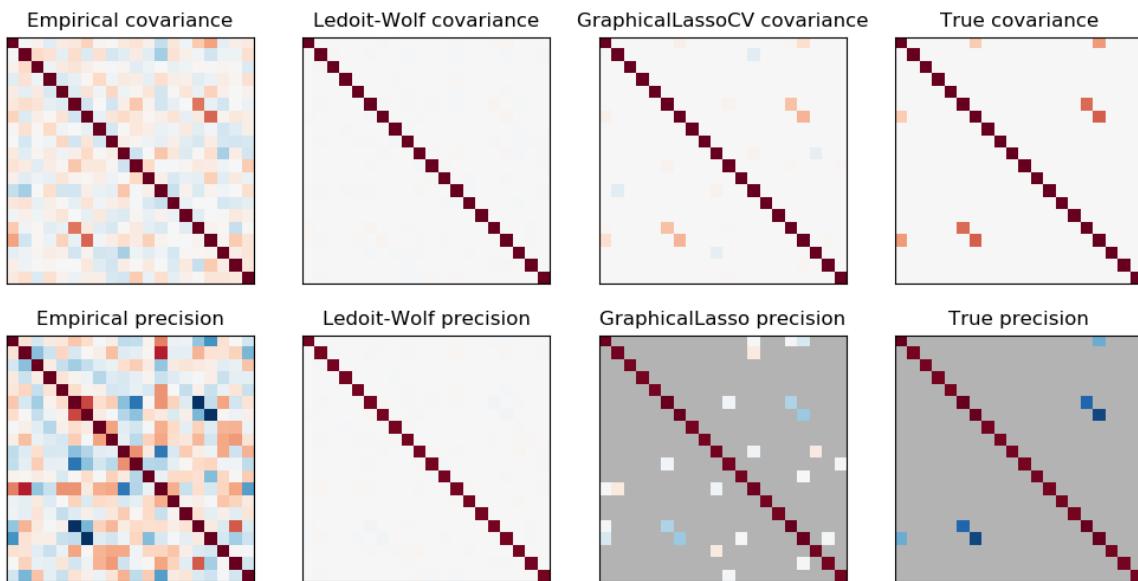
Here, the number of samples is slightly larger than the number of dimensions, thus the empirical covariance is still invertible. However, as the observations are strongly correlated, the empirical covariance matrix is ill-conditioned and as a result its inverse –the empirical precision matrix– is very far from the ground truth.

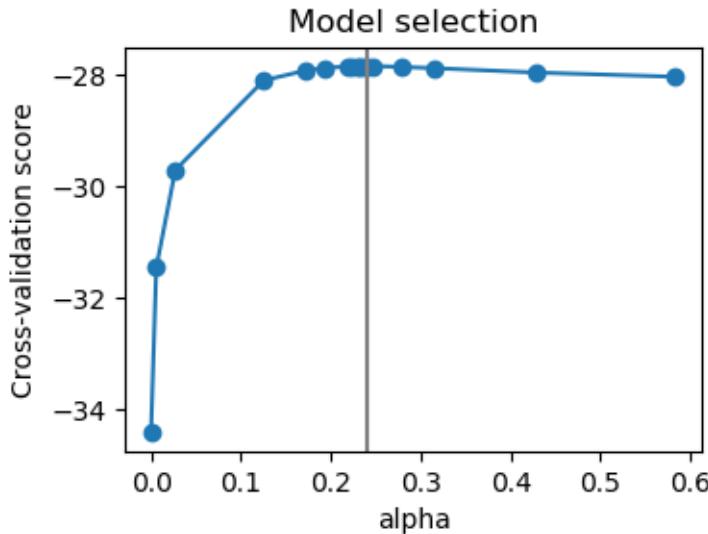
If we use l2 shrinkage, as with the Ledoit-Wolf estimator, as the number of samples is small, we need to shrink a lot. As a result, the Ledoit-Wolf precision is fairly close to the ground truth precision, that is not far from being diagonal, but the off-diagonal structure is lost.

The l1-penalized estimator can recover part of this off-diagonal structure. It learns a sparse precision. It is not able to recover the exact sparsity pattern: it detects too many non-zero coefficients. However, the highest non-zero coefficients of the l1 estimated correspond to the non-zero coefficients in the ground truth. Finally, the coefficients of the l1 precision estimate are biased toward zero: because of the penalty, they are all smaller than the corresponding ground truth value, as can be seen on the figure.

Note that, the color range of the precision matrices is tweaked to improve readability of the figure. The full range of values of the empirical precision is not displayed.

The alpha parameter of the GraphicalLasso setting the sparsity of the model is set by internal cross-validation in the GraphicalLassoCV. As can be seen on figure 2, the grid to compute the cross-validation score is iteratively refined in the neighborhood of the maximum.





```

print(__doc__)
# author: Gael Varoquaux <gael.varoquaux@inria.fr>
# License: BSD 3 clause
# Copyright: INRIA

import numpy as np
from scipy import linalg
from sklearn.datasets import make_sparse_spd_matrix
from sklearn.covariance import GraphicalLassoCV, ledoit_wolf
import matplotlib.pyplot as plt

#####
# Generate the data
n_samples = 60
n_features = 20

prng = np.random.RandomState(1)
prec = make_sparse_spd_matrix(n_features, alpha=.98,
                             smallest_coef=.4,
                             largest_coef=.7,
                             random_state=prng)

cov = linalg.inv(prec)
d = np.sqrt(np.diag(cov))
cov /= d
cov /= d[:, np.newaxis]
prec *= d
prec *= d[:, np.newaxis]
X = prng.multivariate_normal(np.zeros(n_features), cov, size=n_samples)
X -= X.mean(axis=0)
X /= X.std(axis=0)

#####
# Estimate the covariance
emp_cov = np.dot(X.T, X) / n_samples

model = GraphicalLassoCV(cv=5)
model.fit(X)
cov_ = model.covariance_

```

```

prec_ = model.precision_

lw_cov_, _ = ledoit_wolf(X)
lw_prec_ = linalg.inv(lw_cov_)

# ##### Plot the results #####
# Plot the results
plt.figure(figsize=(10, 6))
plt.subplots_adjust(left=0.02, right=0.98)

# plot the covariances
covs = [('Empirical', emp_cov), ('Ledoit-Wolf', lw_cov_),
         ('GraphicalLassoCV', cov_), ('True', cov)]
vmax = cov_.max()
for i, (name, this_cov) in enumerate(covs):
    plt.subplot(2, 4, i + 1)
    plt.imshow(this_cov, interpolation='nearest', vmin=-vmax, vmax=vmax,
               cmap=plt.cm.RdBu_r)
    plt.xticks(())
    plt.yticks(())
    plt.title('%s covariance' % name)

# plot the precisions
precs = [('Empirical', linalg.inv(emp_cov)), ('Ledoit-Wolf', lw_prec_),
          ('GraphicalLasso', prec_), ('True', prec)]
vmax = .9 * prec_.max()
for i, (name, this_prec) in enumerate(precs):
    ax = plt.subplot(2, 4, i + 5)
    plt.imshow(np.ma.masked_equal(this_prec, 0),
               interpolation='nearest', vmin=-vmax, vmax=vmax,
               cmap=plt.cm.RdBu_r)
    plt.xticks(())
    plt.yticks(())
    plt.title('%s precision' % name)
    if hasattr(ax, 'set_facecolor'):
        ax.set_facecolor('.7')
    else:
        ax.set_axis_bgcolor('.7')

# plot the model selection metric
plt.figure(figsize=(4, 3))
plt.axes([.2, .15, .75, .7])
plt.plot(model.cv_alphas_, np.mean(model.grid_scores_, axis=1), 'o-')
plt.axvline(model.alpha_, color='.5')
plt.title('Model selection')
plt.ylabel('Cross-validation score')
plt.xlabel('alpha')

plt.show()

```

Total running time of the script: (0 minutes 0.501 seconds)

Note: Click [here](#) to download the full example code

5.8.3 Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood

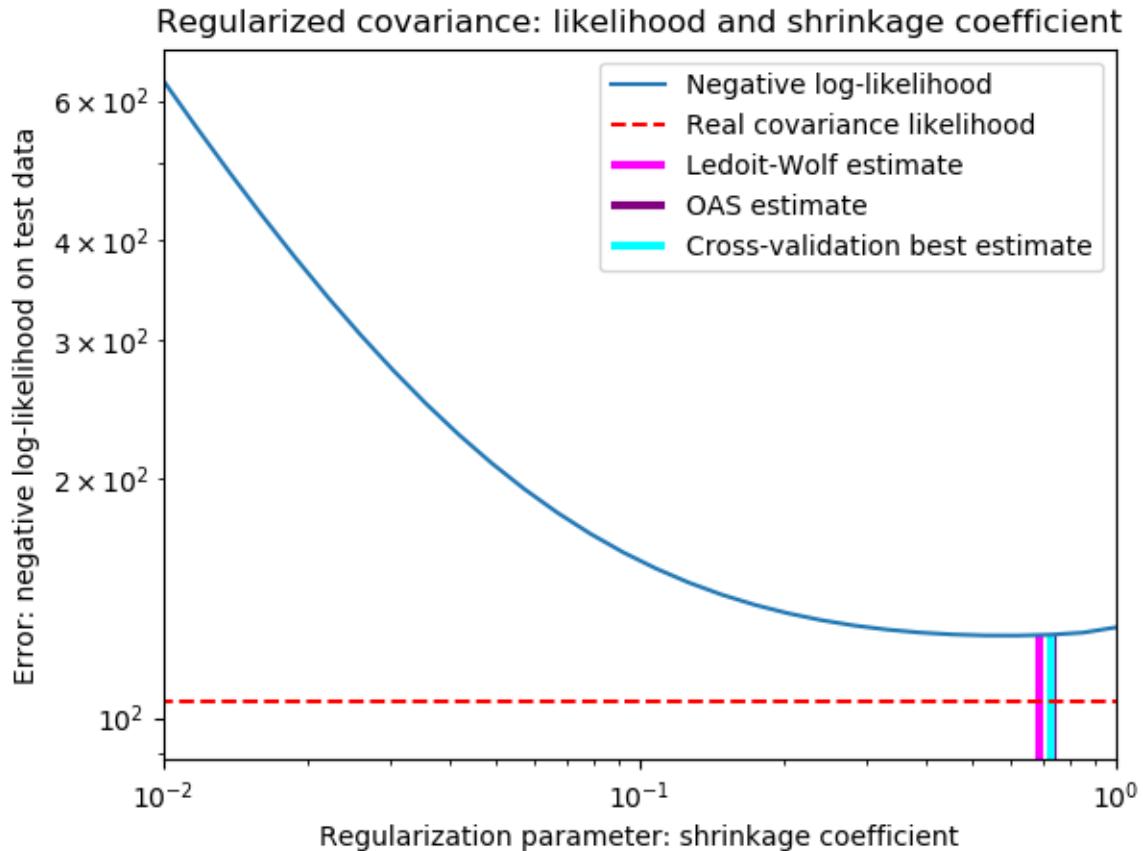
When working with covariance estimation, the usual approach is to use a maximum likelihood estimator, such as the `sklearn.covariance.EmpiricalCovariance`. It is unbiased, i.e. it converges to the true (population) covariance when given many observations. However, it can also be beneficial to regularize it, in order to reduce its variance; this, in turn, introduces some bias. This example illustrates the simple regularization used in `Shrunk Covariance` estimators. In particular, it focuses on how to set the amount of regularization, i.e. how to choose the bias-variance trade-off.

Here we compare 3 approaches:

- Setting the parameter by cross-validating the likelihood on three folds according to a grid of potential shrinkage parameters.
- A close formula proposed by Ledoit and Wolf to compute the asymptotically optimal regularization parameter (minimizing a MSE criterion), yielding the `sklearn.covariance.LedoitWolf` covariance estimate.
- An improvement of the Ledoit-Wolf shrinkage, the `sklearn.covariance.OAS`, proposed by Chen et al. Its convergence is significantly better under the assumption that the data are Gaussian, in particular for small samples.

To quantify estimation error, we plot the likelihood of unseen data for different values of the shrinkage parameter. We also show the choices by cross-validation, or with the LedoitWolf and OAS estimates.

Note that the maximum likelihood estimate corresponds to no shrinkage, and thus performs poorly. The Ledoit-Wolf estimate performs really well, as it is close to the optimal and is computational not costly. In this example, the OAS estimate is a bit further away. Interestingly, both approaches outperform cross-validation, which is significantly most computationally costly.



```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg

from sklearn.covariance import LedoitWolf, OAS, ShrunkCovariance, \
    log_likelihood, empirical_covariance
from sklearn.model_selection import GridSearchCV

# ######
# Generate sample data
n_features, n_samples = 40, 20
np.random.seed(42)
base_X_train = np.random.normal(size=(n_samples, n_features))
base_X_test = np.random.normal(size=(n_samples, n_features))

# Color samples
coloring_matrix = np.random.normal(size=(n_features, n_features))
X_train = np.dot(base_X_train, coloring_matrix)
X_test = np.dot(base_X_test, coloring_matrix)

# #####
# Compute the likelihood on test data
```

```

# spanning a range of possible shrinkage coefficient values
shrinkages = np.logspace(-2, 0, 30)
negative_logliks = [-ShrunkCovariance(shrinkage=s).fit(X_train).score(X_test)
                     for s in shrinkages]

# under the ground-truth model, which we would not have access to in real
# settings
real_cov = np.dot(coloring_matrix.T, coloring_matrix)
emp_cov = empirical_covariance(X_train)
loglik_real = -log_likelihood(emp_cov, linalg.inv(real_cov))

# ##########
# Compare different approaches to setting the parameter

# GridSearch for an optimal shrinkage coefficient
tuned_parameters = [{'shrinkage': shrinkages}]
cv = GridSearchCV(ShrunkCovariance(), tuned_parameters, cv=5)
cv.fit(X_train)

# Ledoit-Wolf optimal shrinkage coefficient estimate
lw = LedoitWolf()
loglik_lw = lw.fit(X_train).score(X_test)

# OAS coefficient estimate
oa = OAS()
loglik_oa = oa.fit(X_train).score(X_test)

# #####
# Plot results
fig = plt.figure()
plt.title("Regularized covariance: likelihood and shrinkage coefficient")
plt.xlabel('Regularization parameter: shrinkage coefficient')
plt.ylabel('Error: negative log-likelihood on test data')
# range shrinkage curve
plt.loglog(shrinkages, negative_logliks, label="Negative log-likelihood")

plt.plot(plt.xlim(), 2 * [loglik_real], '--r',
         label="Real covariance likelihood")

# adjust view
lik_max = npamax(negative_logliks)
lik_min = npamin(negative_logliks)
ymin = lik_min - 6. * np.log((plt.ylim()[1] - plt.ylim()[0]))
ymax = lik_max + 10. * np.log(lik_max - lik_min)
xmin = shrinkages[0]
xmax = shrinkages[-1]
# LW likelihood
plt.vlines(lw.shrinkage_, ymin, -loglik_lw, color='magenta',
            linewidth=3, label='Ledoit-Wolf estimate')
# OAS likelihood
plt.vlines(oa.shrinkage_, ymin, -loglik_oa, color='purple',
            linewidth=3, label='OAS estimate')
# best CV estimator likelihood
plt.vlines(cv.best_estimator_.shrinkage, ymin,
            -cv.best_estimator_.score(X_test), color='cyan',
            linewidth=3, label='Cross-validation best estimate')

plt.ylim(ymin, ymax)

```

```
plt.xlim(xmin, xmax)
plt.legend()

plt.show()
```

Total running time of the script: (0 minutes 0.183 seconds)

Note: Click [here](#) to download the full example code

5.8.4 Robust covariance estimation and Mahalanobis distances relevance

An example to show covariance estimation with the Mahalanobis distances on Gaussian distributed data.

For Gaussian distributed data, the distance of an observation x_i to the mode of the distribution can be computed using its Mahalanobis distance: $d_{(\mu, \Sigma)}(x_i)^2 = (x_i - \mu)' \Sigma^{-1} (x_i - \mu)$ where μ and Σ are the location and the covariance of the underlying Gaussian distribution.

In practice, μ and Σ are replaced by some estimates. The usual covariance maximum likelihood estimate is very sensitive to the presence of outliers in the data set and therefore, the corresponding Mahalanobis distances are. One would better have to use a robust estimator of covariance to guarantee that the estimation is resistant to “erroneous” observations in the data set and that the associated Mahalanobis distances accurately reflect the true organisation of the observations.

The Minimum Covariance Determinant estimator is a robust, high-breakdown point (i.e. it can be used to estimate the covariance matrix of highly contaminated datasets, up to $\frac{n_{\text{samples}} - n_{\text{features}} - 1}{2}$ outliers) estimator of covariance. The idea is to find $\frac{n_{\text{samples}} + n_{\text{features}} + 1}{2}$ observations whose empirical covariance has the smallest determinant, yielding a “pure” subset of observations from which to compute standards estimates of location and covariance.

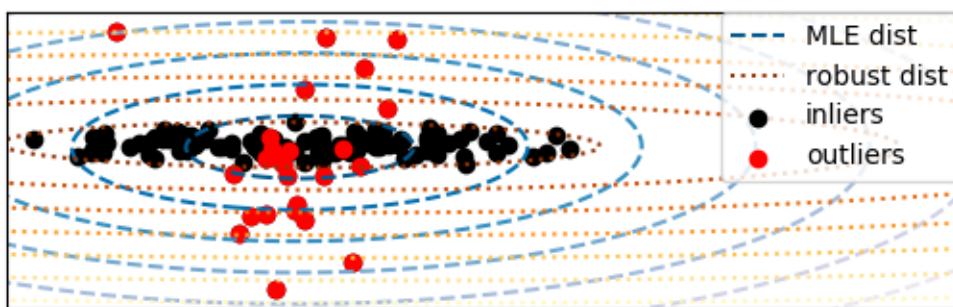
The Minimum Covariance Determinant estimator (MCD) has been introduced by P.J.Rousseeuw in [1].

This example illustrates how the Mahalanobis distances are affected by outlying data: observations drawn from a contaminating distribution are not distinguishable from the observations coming from the real, Gaussian distribution that one may want to work with. Using MCD-based Mahalanobis distances, the two populations become distinguishable. Associated applications are outliers detection, observations ranking, clustering, ... For visualization purpose, the cubic root of the Mahalanobis distances are represented in the boxplot, as Wilson and Hilmerty suggest [2]

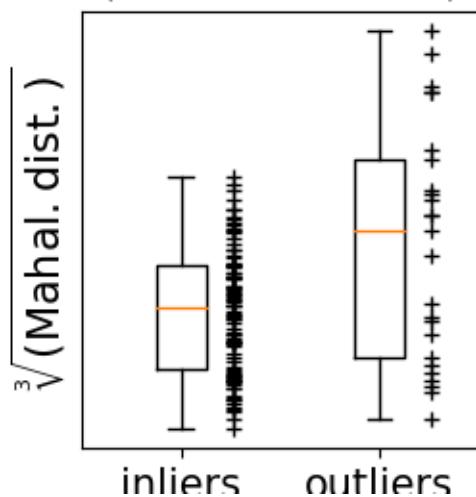
[1] P. J. Rousseeuw. Least median of squares regression. *J. Am Stat Ass*, 79:871, 1984.

[2] Wilson, E. B., & Hilferty, M. M. (1931). The distribution of chi-square. *Proceedings of the National Academy of Sciences of the United States of America*, 17, 684-688.

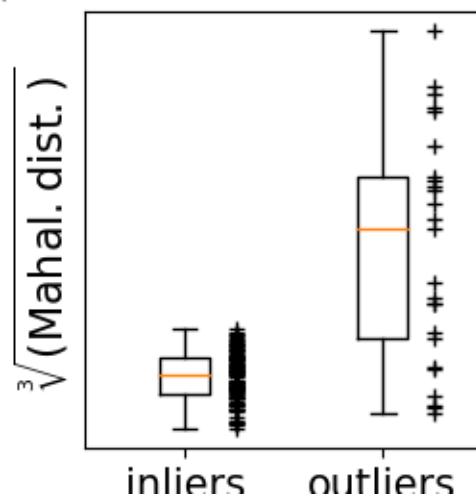
Mahalanobis distances of a contaminated data set:



1. from non-robust estimates
(Maximum Likelihood)



2. from robust estimates
(Minimum Covariance Determinant)



```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.covariance import EmpiricalCovariance, MinCovDet

n_samples = 125
n_outliers = 25
n_features = 2

# generate data
gen_cov = np.eye(n_features)
gen_cov[0, 0] = 2.
X = np.dot(np.random.randn(n_samples, n_features), gen_cov)
# add some outliers
outliers_cov = np.eye(n_features)
outliers_cov[np.arange(1, n_features), np.arange(1, n_features)] = 7.
X[-n_outliers:] = np.dot(np.random.randn(n_outliers, n_features), outliers_cov)

# fit a Minimum Covariance Determinant (MCD) robust estimator to data
robust_cov = MinCovDet().fit(X)

# compare estimators learnt from the full data set with true parameters
emp_cov = EmpiricalCovariance().fit(X)

```

```

# ##########
# Display results
fig = plt.figure()
plt.subplots_adjust(hspace=-.1, wspace=.4, top=.95, bottom=.05)

# Show data set
subfig1 = plt.subplot(3, 1, 1)
inlier_plot = subfig1.scatter(X[:, 0], X[:, 1],
                              color='black', label='inliers')
outlier_plot = subfig1.scatter(X[:, 0][-n_outliers:], X[:, 1][-n_outliers:],
                               color='red', label='outliers')
subfig1.set_xlim(subfig1.get_xlim()[0], 11.)
subfig1.set_title("Mahalanobis distances of a contaminated data set:")

# Show contours of the distance functions
xx, yy = np.meshgrid(np.linspace(plt.xlim()[0], plt.xlim()[1], 100),
                      np.linspace(plt.ylim()[0], plt.ylim()[1], 100))
zz = np.c_[xx.ravel(), yy.ravel()]

mahal_emp_cov = emp_cov.mahalanobis(zz)
mahal_emp_cov = mahal_emp_cov.reshape(xx.shape)
emp_cov_contour = subfig1.contour(xx, yy, np.sqrt(mahal_emp_cov),
                                  cmap=plt.cm.PuBu_r,
                                  linestyles='dashed')

mahal_robust_cov = robust_cov.mahalanobis(zz)
mahal_robust_cov = mahal_robust_cov.reshape(xx.shape)
robust_contour = subfig1.contour(xx, yy, np.sqrt(mahal_robust_cov),
                                 cmap=plt.cm.YlOrBr_r, linestyles='dotted')

subfig1.legend([emp_cov_contour.collections[1], robust_contour.collections[1],
                inlier_plot, outlier_plot],
               ['MLE dist', 'robust dist', 'inliers', 'outliers'],
               loc="upper right", borderaxespad=0)
plt.xticks(())
plt.yticks(())

# Plot the scores for each point
emp_mahal = emp_cov.mahalanobis(X - np.mean(X, 0)) ** (0.33)
subfig2 = plt.subplot(2, 2, 3)
subfig2.boxplot([emp_mahal[:-n_outliers], emp_mahal[-n_outliers:]], widths=.25)
subfig2.plot(np.full(n_samples - n_outliers, 1.26),
            emp_mahal[:-n_outliers], '+k', markeredgewidth=1)
subfig2.plot(np.full(n_outliers, 2.26),
            emp_mahal[-n_outliers:], '+k', markeredgewidth=1)
subfig2.axes.set_xticklabels(['inliers', 'outliers'], size=15)
subfig2.set_ylabel(r"$\sqrt{3} \cdot \text{Mahal. dist.}$", size=16)
subfig2.set_title("1. from non-robust estimates\n(Maximum Likelihood)")
plt.yticks(())

robust_mahal = robust_cov.mahalanobis(X - robust_cov.location_) ** (0.33)
subfig3 = plt.subplot(2, 2, 4)
subfig3.boxplot([robust_mahal[:-n_outliers], robust_mahal[-n_outliers:]],
               widths=.25)
subfig3.plot(np.full(n_samples - n_outliers, 1.26),
            robust_mahal[:-n_outliers], '+k', markeredgewidth=1)
subfig3.plot(np.full(n_outliers, 2.26),
            robust_mahal[-n_outliers:], '+k', markeredgewidth=1)

```

```
subfig3.axes.set_xticklabels(['inliers', 'outliers'], size=15)
subfig3.set_ylabel(r"\$\\sqrt[3]{\\rm{Mahal. dist.}}\$", size=16)
subfig3.set_title("2. from robust estimates\\n(Minimum Covariance Determinant)")
plt.yticks(())

plt.show()
```

Total running time of the script: (0 minutes 0.298 seconds)

Note: Click [here](#) to download the full example code

5.8.5 Robust vs Empirical covariance estimate

The usual covariance maximum likelihood estimate is very sensitive to the presence of outliers in the data set. In such a case, it would be better to use a robust estimator of covariance to guarantee that the estimation is resistant to “erroneous” observations in the data set.^{1,2}

Minimum Covariance Determinant Estimator

The Minimum Covariance Determinant estimator is a robust, high-breakdown point (i.e. it can be used to estimate the covariance matrix of highly contaminated datasets, up to $\frac{n_{\text{samples}} - n_{\text{features}} - 1}{2}$ outliers) estimator of covariance. The idea is to find $\frac{n_{\text{samples}} + n_{\text{features}} + 1}{2}$ observations whose empirical covariance has the smallest determinant, yielding a “pure” subset of observations from which to compute standards estimates of location and covariance. After a correction step aiming at compensating the fact that the estimates were learned from only a portion of the initial data, we end up with robust estimates of the data set location and covariance.

The Minimum Covariance Determinant estimator (MCD) has been introduced by P.J.Rousseeuw in³.

Evaluation

In this example, we compare the estimation errors that are made when using various types of location and covariance estimates on contaminated Gaussian distributed data sets:

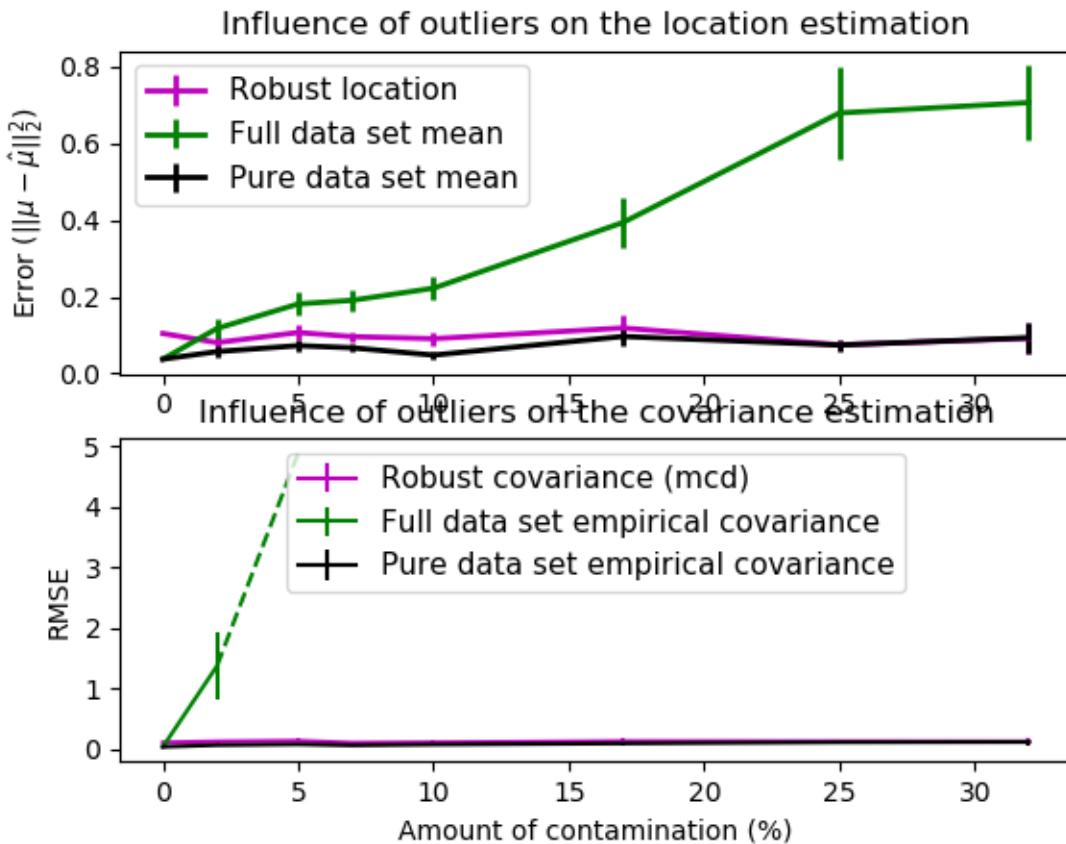
- The mean and the empirical covariance of the full dataset, which break down as soon as there are outliers in the data set
- The robust MCD, that has a low error provided $n_{\text{samples}} > 5n_{\text{features}}$
- The mean and the empirical covariance of the observations that are known to be good ones. This can be considered as a “perfect” MCD estimation, so one can trust our implementation by comparing to this case.

¹ Johanna Hardin, David M Rocke. The distribution of robust distances. Journal of Computational and Graphical Statistics. December 1, 2005, 14(4): 928-946.

² Zoubir A., Koivunen V., Chakhchoukh Y. and Muma M. (2012). Robust estimation in signal processing: A tutorial-style treatment of fundamental concepts. IEEE Signal Processing Magazine 29(4), 61-80.

³ P. J. Rousseeuw. Least median of squares regression. Journal of American Statistical Ass., 79:871, 1984.

References



```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.font_manager

from sklearn.covariance import EmpiricalCovariance, MinCovDet

# example settings
n_samples = 80
n_features = 5
repeat = 10

range_n_outliers = np.concatenate(
    (np.linspace(0, n_samples / 8, 5),
     np.linspace(n_samples / 8, n_samples / 2, 5)[1:-1])).astype(np.int)

# definition of arrays to store results
err_loc_mcd = np.zeros((range_n_outliers.size, repeat))
err_cov_mcd = np.zeros((range_n_outliers.size, repeat))
err_loc_emp_full = np.zeros((range_n_outliers.size, repeat))
err_cov_emp_full = np.zeros((range_n_outliers.size, repeat))
err_loc_emp_pure = np.zeros((range_n_outliers.size, repeat))
err_cov_emp_pure = np.zeros((range_n_outliers.size, repeat))

```

```

# computation
for i, n_outliers in enumerate(range_n_outliers):
    for j in range(repeat):

        rng = np.random.RandomState(i * j)

        # generate data
        X = rng.randn(n_samples, n_features)
        # add some outliers
        outliers_index = rng.permutation(n_samples) [:n_outliers]
        outliers_offset = 10. * \
            (np.random.randint(2, size=(n_outliers, n_features)) - 0.5)
        X[outliers_index] += outliers_offset
        inliers_mask = np.ones(n_samples).astype(bool)
        inliers_mask[outliers_index] = False

        # fit a Minimum Covariance Determinant (MCD) robust estimator to data
        mcd = MinCovDet().fit(X)
        # compare raw robust estimates with the true location and covariance
        err_loc_mcd[i, j] = np.sum(mcd.location_ ** 2)
        err_cov_mcd[i, j] = mcd.error_norm(np.eye(n_features))

        # compare estimators learned from the full data set with true
        # parameters
        err_loc_emp_full[i, j] = np.sum(X.mean(0) ** 2)
        err_cov_emp_full[i, j] = EmpiricalCovariance().fit(X).error_norm(
            np.eye(n_features))

        # compare with an empirical covariance learned from a pure data set
        # (i.e. "perfect" mcd)
        pure_X = X[inliers_mask]
        pure_location = pure_X.mean(0)
        pure_emp_cov = EmpiricalCovariance().fit(pure_X)
        err_loc_emp_pure[i, j] = np.sum(pure_location ** 2)
        err_cov_emp_pure[i, j] = pure_emp_cov.error_norm(np.eye(n_features))

# Display results
font_prop = matplotlib.font_manager.FontProperties(size=11)
plt.subplot(2, 1, 1)
lw = 2
plt.errorbar(range_n_outliers, err_loc_mcd.mean(1),
             yerr=err_loc_mcd.std(1) / np.sqrt(repeat),
             label="Robust location", lw=lw, color='m')
plt.errorbar(range_n_outliers, err_loc_emp_full.mean(1),
             yerr=err_loc_emp_full.std(1) / np.sqrt(repeat),
             label="Full data set mean", lw=lw, color='green')
plt.errorbar(range_n_outliers, err_loc_emp_pure.mean(1),
             yerr=err_loc_emp_pure.std(1) / np.sqrt(repeat),
             label="Pure data set mean", lw=lw, color='black')
plt.title("Influence of outliers on the location estimation")
plt.ylabel(r"Error ($||\mu - \hat{\mu}||_2^2$)")
plt.legend(loc="upper left", prop=font_prop)

plt.subplot(2, 1, 2)
x_size = range_n_outliers.size
plt.errorbar(range_n_outliers, err_cov_mcd.mean(1),
             yerr=err_cov_mcd.std(1),

```

```

label="Robust covariance (mcd)", color='m')
plt.errorbar(range_n_outliers[:x_size // 5 + 1],
             err_cov_emp_full.mean(1)[:x_size // 5 + 1],
             yerr=err_cov_emp_full.std(1)[:x_size // 5 + 1],
             label="Full data set empirical covariance", color='green')
plt.plot(range_n_outliers[x_size // 5:(x_size // 5) + 1],
          err_cov_emp_full.mean(1)[(x_size // 5):(x_size // 5 + 1)],
          color='green', ls='--')
plt.errorbar(range_n_outliers, err_cov_emp_pure.mean(1),
             yerr=err_cov_emp_pure.std(1),
             label="Pure data set empirical covariance", color='black')
plt.title("Influence of outliers on the covariance estimation")
plt.xlabel("Amount of contamination (%)")
plt.ylabel("RMSE")
plt.legend(loc="upper center", prop=font_prop)

plt.show()

```

Total running time of the script: (0 minutes 2.985 seconds)

5.9 Cross decomposition

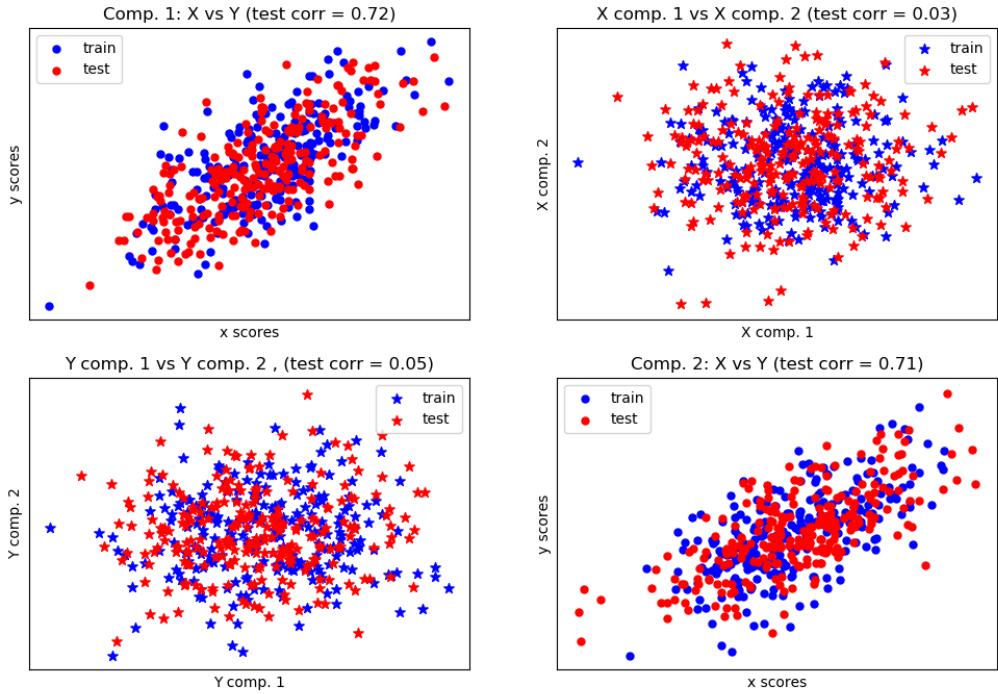
Examples concerning the `sklearn.cross_decomposition` module.

Note: Click [here](#) to download the full example code

5.9.1 Compare cross decomposition methods

Simple usage of various cross decomposition algorithms: - PLSCanonical - PLSRegression, with multivariate response, a.k.a. PLS2 - PLSRegression, with univariate response, a.k.a. PLS1 - CCA

Given 2 multivariate covarying two-dimensional datasets, X, and Y, PLS extracts the ‘directions of covariance’, i.e. the components of each datasets that explain the most shared variance between both datasets. This is apparent on the **scatterplot matrix** display: components 1 in dataset X and dataset Y are maximally correlated (points lie around the first diagonal). This is also true for components 2 in both dataset, however, the correlation across datasets for different components is weak: the point cloud is very spherical.



Out:

```

Corr(X)
[[ 1.  0.51  0.07 -0.05]
 [ 0.51  1.  0.11 -0.01]
 [ 0.07  0.11  1.  0.49]
 [-0.05 -0.01  0.49  1. ]]

Corr(Y)
[[1.  0.48  0.05  0.03]
 [0.48  1.  0.04  0.12]
 [0.05  0.04  1.  0.51]
 [0.03  0.12  0.51  1. ]]

True B (such that: Y = XB + Err)
[[1 1 1]
 [2 2 2]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]]

Estimated B
[[ 1.  1.  1. ]
 [ 2.  2.  2. ]
 [-0. -0.  0. ]
 [ 0.  0.  0. ]
 [ 0.  0.  0. ]]

```

```
[ 0.   0.  -0. ]
[-0.  -0.  -0.1]
[-0.  -0.   0. ]
[ 0.   0.   0.1]
[ 0.   0.  -0. ]]
Estimated betas
[[ 1. ]
 [ 2.1]
 [ 0. ]
 [ 0. ]
 [ 0. ]
 [-0. ]
 [-0. ]
 [ 0. ]
 [-0. ]
 [-0. ]]
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.cross_decomposition import PLSCanonical, PLSRegression, CCA

# ##### Dataset based latent variables model #####
# Dataset based latent variables model

n = 500
# 2 latents vars:
l1 = np.random.normal(size=n)
l2 = np.random.normal(size=n)

latents = np.array([l1, l1, l2, l2]).T
X = latents + np.random.normal(size=4 * n).reshape((n, 4))
Y = latents + np.random.normal(size=4 * n).reshape((n, 4))

X_train = X[:n // 2]
Y_train = Y[:n // 2]
X_test = X[n // 2:]
Y_test = Y[n // 2:]

print("Corr(X)")
print(np.round(np.corrcoef(X.T), 2))
print("Corr(Y)")
print(np.round(np.corrcoef(Y.T), 2))

# ##### Canonical (symmetric) PLS #####
# Canonical (symmetric) PLS

# Transform data
# ~~~~~
plsca = PLSCanonical(n_components=2)
plsca.fit(X_train, Y_train)
X_train_r, Y_train_r = plsca.transform(X_train, Y_train)
```

```

X_test_r, Y_test_r = plsca.transform(X_test, Y_test)

# Scatter plot of scores
# ~~~~~
# 1) On diagonal plot X vs Y scores on each components
plt.figure(figsize=(12, 8))
plt.subplot(221)
plt.scatter(X_train_r[:, 0], Y_train_r[:, 0], label="train",
            marker="o", c="b", s=25)
plt.scatter(X_test_r[:, 0], Y_test_r[:, 0], label="test",
            marker="o", c="r", s=25)
plt.xlabel("x scores")
plt.ylabel("y scores")
plt.title('Comp. 1: X vs Y (test corr = %.2f)' %
           np.corrcoef(X_test_r[:, 0], Y_test_r[:, 0])[0, 1])
plt.xticks(())
plt.yticks(())
plt.legend(loc="best")

plt.subplot(224)
plt.scatter(X_train_r[:, 1], Y_train_r[:, 1], label="train",
            marker="o", c="b", s=25)
plt.scatter(X_test_r[:, 1], Y_test_r[:, 1], label="test",
            marker="o", c="r", s=25)
plt.xlabel("x scores")
plt.ylabel("y scores")
plt.title('Comp. 2: X vs Y (test corr = %.2f)' %
           np.corrcoef(X_test_r[:, 1], Y_test_r[:, 1])[0, 1])
plt.xticks(())
plt.yticks(())
plt.legend(loc="best")

# 2) Off diagonal plot components 1 vs 2 for X and Y
plt.subplot(222)
plt.scatter(X_train_r[:, 0], X_train_r[:, 1], label="train",
            marker="*", c="b", s=50)
plt.scatter(X_test_r[:, 0], X_test_r[:, 1], label="test",
            marker="*", c="r", s=50)
plt.xlabel("X comp. 1")
plt.ylabel("X comp. 2")
plt.title('X comp. 1 vs X comp. 2 (test corr = %.2f)' %
           np.corrcoef(X_test_r[:, 0], X_test_r[:, 1])[0, 1])
plt.legend(loc="best")
plt.xticks(())
plt.yticks(())

plt.subplot(223)
plt.scatter(Y_train_r[:, 0], Y_train_r[:, 1], label="train",
            marker="*", c="b", s=50)
plt.scatter(Y_test_r[:, 0], Y_test_r[:, 1], label="test",
            marker="*", c="r", s=50)
plt.xlabel("Y comp. 1")
plt.ylabel("Y comp. 2")
plt.title('Y comp. 1 vs Y comp. 2 , (test corr = %.2f)' %
           np.corrcoef(Y_test_r[:, 0], Y_test_r[:, 1])[0, 1])
plt.legend(loc="best")
plt.xticks(())
plt.yticks())

```

```

plt.show()

# ##### PLS regression, with multivariate response, a.k.a. PLS2 #####
# each Yj = 1*X1 + 2*X2 + noize

n = 1000
q = 3
p = 10
X = np.random.normal(size=n * p).reshape((n, p))
B = np.array([[1, 2] + [0] * (p - 2)] * q).T
Y = np.dot(X, B) + np.random.normal(size=n * q).reshape((n, q)) + 5

pls2 = PLSRegression(n_components=3)
pls2.fit(X, Y)
print("True B (such that: Y = XB + Err)")
print(B)
# compare pls2.coef_ with B
print("Estimated B")
print(np.round(pls2.coef_, 1))
pls2.predict(X)

# PLS regression, with univariate response, a.k.a. PLS1

n = 1000
p = 10
X = np.random.normal(size=n * p).reshape((n, p))
y = X[:, 0] + 2 * X[:, 1] + np.random.normal(size=n * 1) + 5
pls1 = PLSRegression(n_components=3)
pls1.fit(X, y)
# note that the number of components exceeds 1 (the dimension of y)
print("Estimated betas")
print(np.round(pls1.coef_, 1))

# ##### CCA (PLS mode B with symmetric deflation)

cca = CCA(n_components=2)
cca.fit(X_train, Y_train)
X_train_r, Y_train_r = cca.transform(X_train, Y_train)
X_test_r, Y_test_r = cca.transform(X_test, Y_test)

```

Total running time of the script: (0 minutes 0.099 seconds)

5.10 Dataset examples

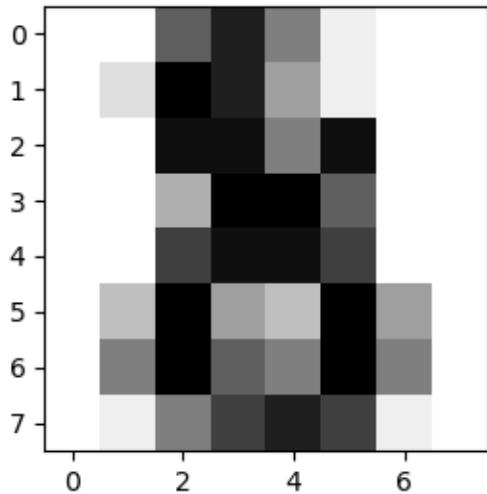
Examples concerning the `sklearn.datasets` module.

Note: Click [here](#) to download the full example code

5.10.1 The Digit Dataset

This dataset is made up of 1797 8x8 images. Each image, like the one shown below, is of a hand-written digit. In order to utilize an 8x8 figure like this, we'd have to first transform it into a feature vector with length 64.

See [here](#) for more information about this dataset.



```
print(__doc__)

# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

from sklearn import datasets

import matplotlib.pyplot as plt

#Load the digits dataset
digits = datasets.load_digits()

#Display the first digit
plt.figure(1, figsize=(3, 3))
plt.imshow(digits.images[-1], cmap=plt.cm.gray_r, interpolation='nearest')
plt.show()
```

Total running time of the script: (0 minutes 0.113 seconds)

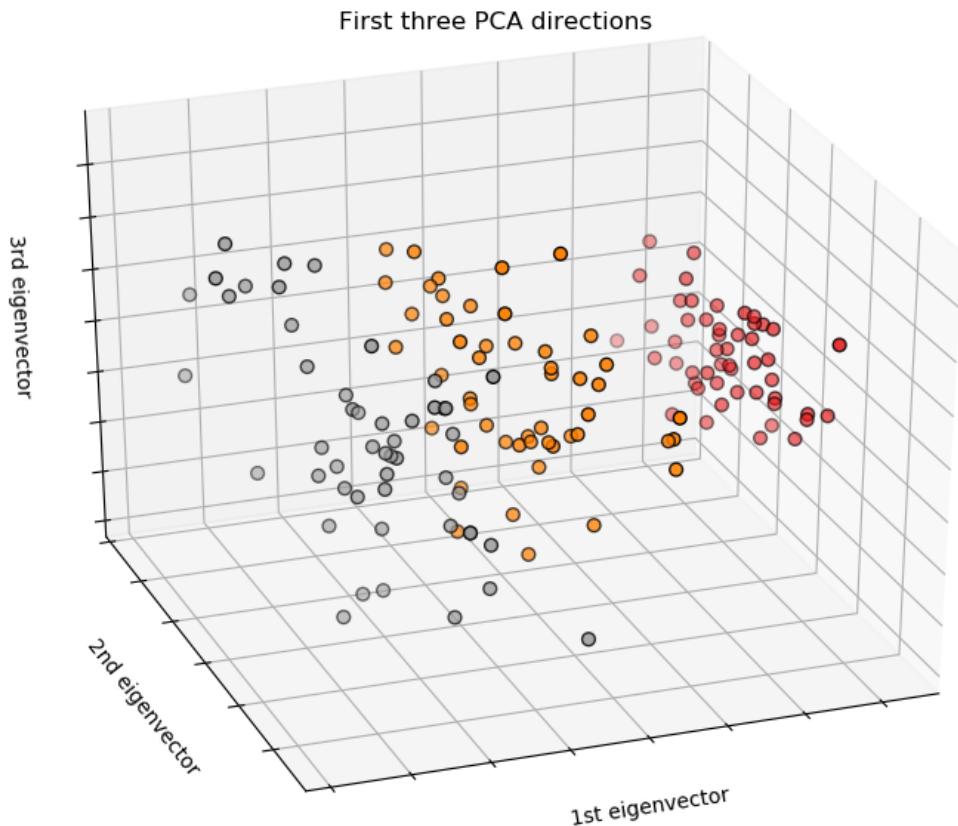
Note: Click [here](#) to download the full example code

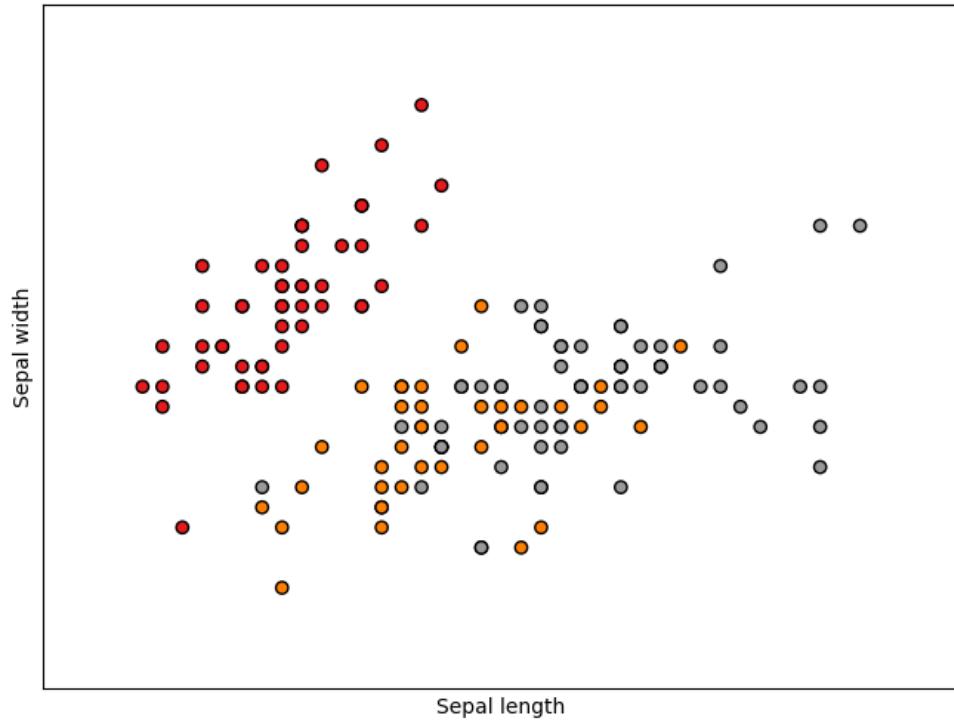
5.10.2 The Iris Dataset

This data sets consists of 3 different types of irises' (Setosa, Versicolour, and Virginica) petal and sepal length, stored in a 150x4 numpy.ndarray

The rows being the samples and the columns being: Sepal Length, Sepal Width, Petal Length and Petal Width.

The below plot uses the first two features. See [here](#) for more information on this dataset.





```
print(__doc__)

# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets
from sklearn.decomposition import PCA

# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2] # we only take the first two features.
y = iris.target

x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5

plt.figure(2, figsize=(8, 6))
plt.clf()

# Plot the training points
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Set1,
            edgecolor='k')
plt.xlabel('Sepal length')
```

```

plt.ylabel('Sepal width')

plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())

# To get a better understanding of interaction of the dimensions
# plot the first three PCA dimensions
fig = plt.figure(1, figsize=(8, 6))
ax = Axes3D(fig, elev=-150, azim=110)
X_reduced = PCA(n_components=3).fit_transform(iris.data)
ax.scatter(X_reduced[:, 0], X_reduced[:, 1], X_reduced[:, 2], c=y,
           cmap=plt.cm.Set1, edgecolor='k', s=40)
ax.set_title("First three PCA directions")
ax.set_xlabel("1st eigenvector")
ax.w_xaxis.set_ticklabels([])
ax.set_ylabel("2nd eigenvector")
ax.w_yaxis.set_ticklabels([])
ax.set_zlabel("3rd eigenvector")
ax.w_zaxis.set_ticklabels([])

plt.show()

```

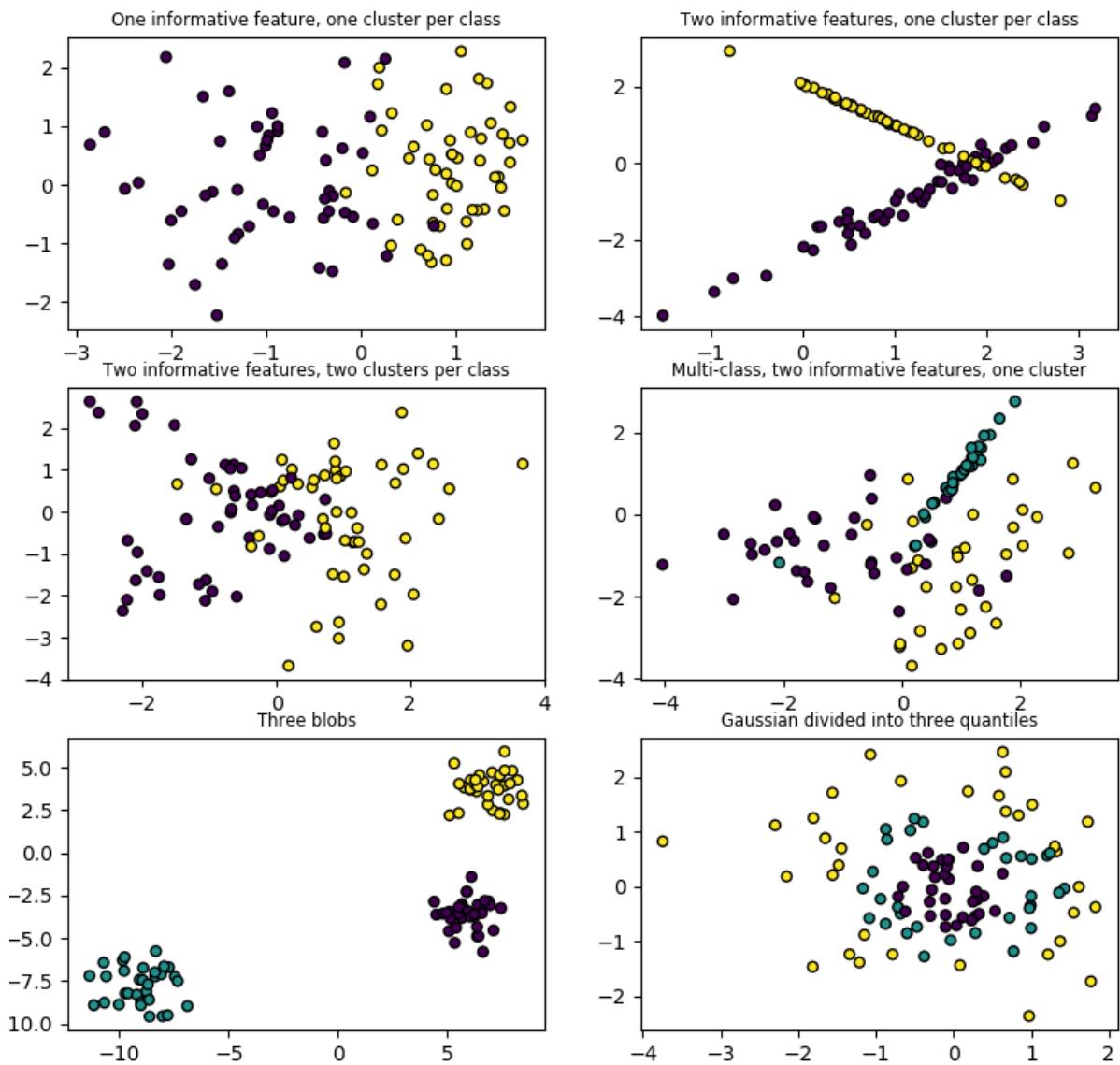
Total running time of the script: (0 minutes 0.059 seconds)

Note: Click [here](#) to download the full example code

5.10.3 Plot randomly generated classification dataset

Plot several randomly generated 2D classification datasets. This example illustrates the datasets. make_classification datasets.make_blobs and datasets.make_gaussian_quantiles functions.

For make_classification, three binary and two multi-class classification datasets are generated, with different numbers of informative features and clusters per class.



```
print(__doc__)

import matplotlib.pyplot as plt

from sklearn.datasets import make_classification
from sklearn.datasets import make_blobs
from sklearn.datasets import make_gaussian_quantiles

plt.figure(figsize=(8, 8))
plt.subplots_adjust(bottom=.05, top=.9, left=.05, right=.95)

plt.subplot(321)
plt.title("One informative feature, one cluster per class", fontsize='small')
X1, Y1 = make_classification(n_features=2, n_redundant=0, n_informative=1,
                             n_clusters_per_class=1)
```

```

plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1,
            s=25, edgecolor='k')

plt.subplot(322)
plt.title("Two informative features, one cluster per class", fontsize='small')
X1, Y1 = make_classification(n_features=2, n_redundant=0, n_informative=2,
                             n_clusters_per_class=1)
plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1,
            s=25, edgecolor='k')

plt.subplot(323)
plt.title("Two informative features, two clusters per class",
          fontsize='small')
X2, Y2 = make_classification(n_features=2, n_redundant=0, n_informative=2)
plt.scatter(X2[:, 0], X2[:, 1], marker='o', c=Y2,
            s=25, edgecolor='k')

plt.subplot(324)
plt.title("Multi-class, two informative features, one cluster",
          fontsize='small')
X1, Y1 = make_classification(n_features=2, n_redundant=0, n_informative=2,
                             n_clusters_per_class=1, n_classes=3)
plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1,
            s=25, edgecolor='k')

plt.subplot(325)
plt.title("Three blobs", fontsize='small')
X1, Y1 = make_blobs(n_features=2, centers=3)
plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1,
            s=25, edgecolor='k')

plt.subplot(326)
plt.title("Gaussian divided into three quantiles", fontsize='small')
X1, Y1 = make_gaussian_quantiles(n_features=2, n_classes=3)
plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1,
            s=25, edgecolor='k')

plt.show()

```

Total running time of the script: (0 minutes 0.097 seconds)

Note: Click [here](#) to download the full example code

5.10.4 Plot randomly generated multilabel dataset

This illustrates the `datasets.make_multilabel_classification` dataset generator. Each sample consists of counts of two features (up to 50 in total), which are differently distributed in each of two classes.

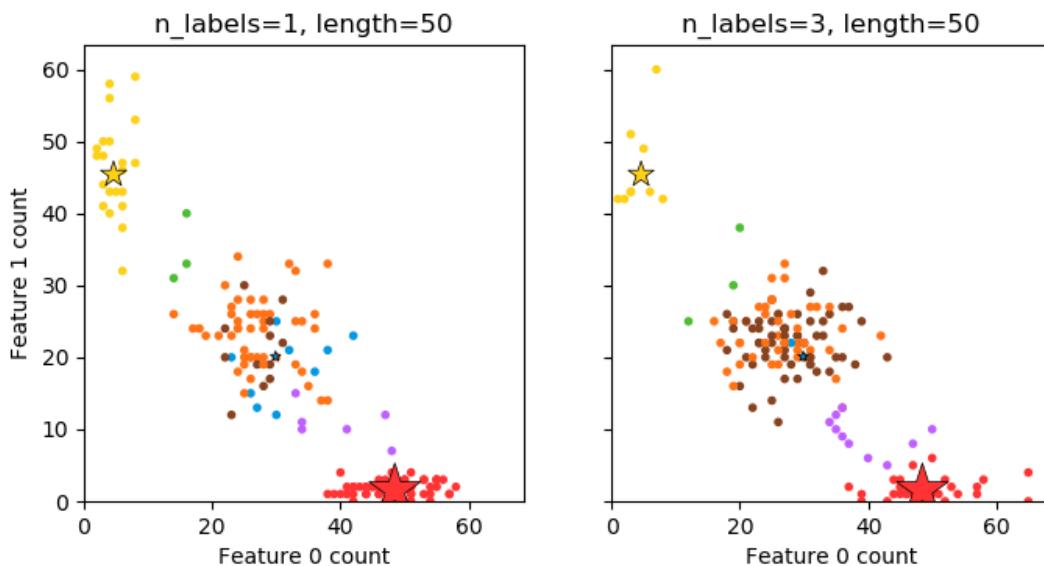
Points are labeled as follows, where Y means the class is present:

1	2	3	Color
Y	N	N	Red
N	Y	N	Blue
N	N	Y	Yellow
Y	Y	N	Purple
Y	N	Y	Orange
Y	Y	N	Green
Y	Y	Y	Brown

A star marks the expected sample for each class; its size reflects the probability of selecting that class label.

The left and right examples highlight the `n_labels` parameter: more of the samples in the right plot have 2 or 3 labels.

Note that this two-dimensional example is very degenerate: generally the number of features would be much greater than the “document length”, while here we have much larger documents than vocabulary. Similarly, with `n_classes > n_features`, it is much less likely that a feature distinguishes a particular class.



Out:

```
The data was generated from (random_state=1013):
Class    P(C)      P(w0|C)  P(w1|C)
red      0.64      0.97     0.03
blue     0.06      0.60     0.40
yellow   0.30      0.09     0.91
```

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_multilabel_classification as make_ml_clf
```

```

print(__doc__)

COLORS = np.array(['!',
                  '#FF3333', # red
                  '#0198E1', # blue
                  '#BF5FFF', # purple
                  '#FCD116', # yellow
                  '#FF7216', # orange
                  '#4DBD33', # green
                  '#87421F' # brown
                  ])

# Use same random seed for multiple calls to make_multilabel_classification to
# ensure same distributions
RANDOM_SEED = np.random.randint(2 ** 10)

def plot_2d(ax, n_labels=1, n_classes=3, length=50):
    X, Y, p_c, p_w_c = make_ml_clf(n_samples=150, n_features=2,
                                      n_classes=n_classes, n_labels=n_labels,
                                      length=length, allow_unlabeled=False,
                                      return_distributions=True,
                                      random_state=RANDOM_SEED)

    ax.scatter(X[:, 0], X[:, 1], color=COLORS.take((Y * [1, 2, 4]
                                                    .sum(axis=1))),
               marker='.')
    ax.scatter(p_w_c[0] * length, p_w_c[1] * length,
               marker='*', linewidth=.5, edgecolor='black',
               s=20 + 1500 * p_c ** 2,
               color=COLORS.take([1, 2, 4]))
    ax.set_xlabel('Feature 0 count')
    return p_c, p_w_c

_, (ax1, ax2) = plt.subplots(1, 2, sharex='row', sharey='row', figsize=(8, 4))
plt.subplots_adjust(bottom=.15)

p_c, p_w_c = plot_2d(ax1, n_labels=1)
ax1.set_title('n_labels=1, length=50')
ax1.set_ylabel('Feature 1 count')

plot_2d(ax2, n_labels=3)
ax2.set_title('n_labels=3, length=50')
ax2.set_xlim(left=0, auto=True)
ax2.set_ylim(bottom=0, auto=True)

plt.show()

print('The data was generated from (random_state=%d):' % RANDOM_SEED)
print('Class', 'P(C)', 'P(w0|C)', 'P(w1|C)', sep='\t')
for k, p, p_w in zip(['red', 'blue', 'yellow'], p_c, p_w_c.T):
    print('%s\t%0.2f\t%0.2f\t%0.2f' % (k, p, p_w[0], p_w[1]))

```

Total running time of the script: (0 minutes 0.085 seconds)

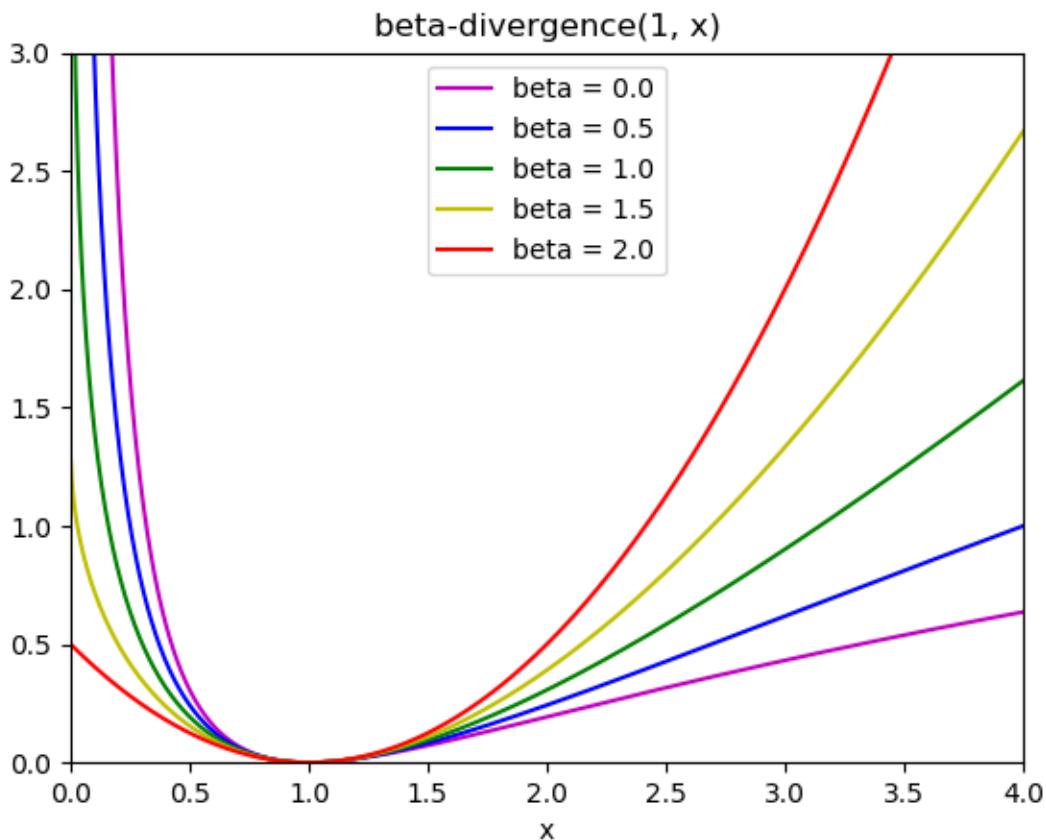
5.11 Decomposition

Examples concerning the `sklearn.decomposition` module.

Note: Click [here](#) to download the full example code

5.11.1 Beta-divergence loss functions

A plot that compares the various Beta-divergence loss functions supported by the Multiplicative-Update ('mu') solver in `sklearn.decomposition.NMF`.



```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition.nmf import _beta_divergence

print(__doc__)

x = np.linspace(0.001, 4, 1000)
y = np.zeros(x.shape)

colors = 'mbgyr'
for j, beta in enumerate((0., 0.5, 1., 1.5, 2.)):
```

```

for i, xi in enumerate(x):
    y[i] = _beta_divergence(1, xi, 1, beta)
name = "beta = %1.1f" % beta
plt.plot(x, y, label=name, color=colors[j])

plt.xlabel("x")
plt.title("beta-divergence (1, x)")
plt.legend(loc=0)
plt.axis([0, 4, 0, 3])
plt.show()

```

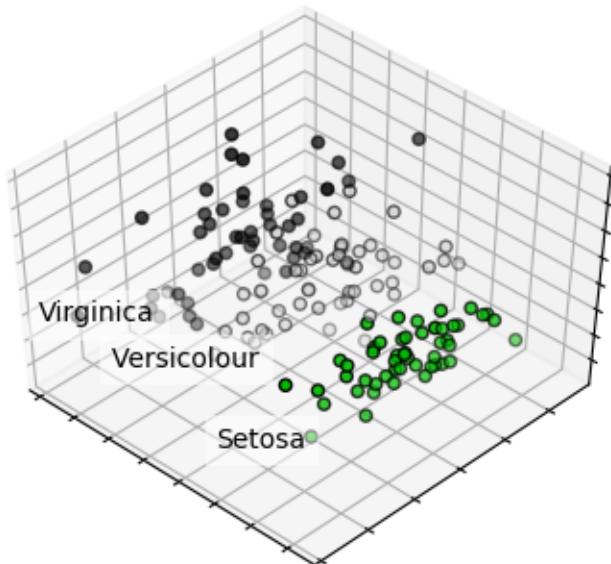
Total running time of the script: (0 minutes 0.225 seconds)

Note: Click [here](#) to download the full example code

5.11.2 PCA example with Iris Data-set

Principal Component Analysis applied to the Iris dataset.

See [here](#) for more information on this dataset.



```

print(__doc__)

# Code source: Gaël Varoquaux
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from sklearn import decomposition
from sklearn import datasets

```

```
np.random.seed(5)

centers = [[1, 1], [-1, -1], [1, -1]]
iris = datasets.load_iris()
X = iris.data
y = iris.target

fig = plt.figure(1, figsize=(4, 3))
plt.clf()
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)

plt.cla()
pca = decomposition.PCA(n_components=3)
pca.fit(X)
X = pca.transform(X)

for name, label in [('Setosa', 0), ('Versicolour', 1), ('Virginica', 2)]:
    ax.text3D(X[y == label, 0].mean(),
               X[y == label, 1].mean() + 1.5,
               X[y == label, 2].mean(), name,
               horizontalalignment='center',
               bbox=dict(alpha=.5, edgecolor='w', facecolor='w'))
# Reorder the labels to have colors matching the cluster results
y = np.choose(y, [1, 2, 0]).astype(np.float)
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=y, cmap=plt.cm.nipy_spectral,
           edgecolor='k')

ax.w_xaxis.set_ticklabels([])
ax.w_yaxis.set_ticklabels([])
ax.w_zaxis.set_ticklabels([])

plt.show()
```

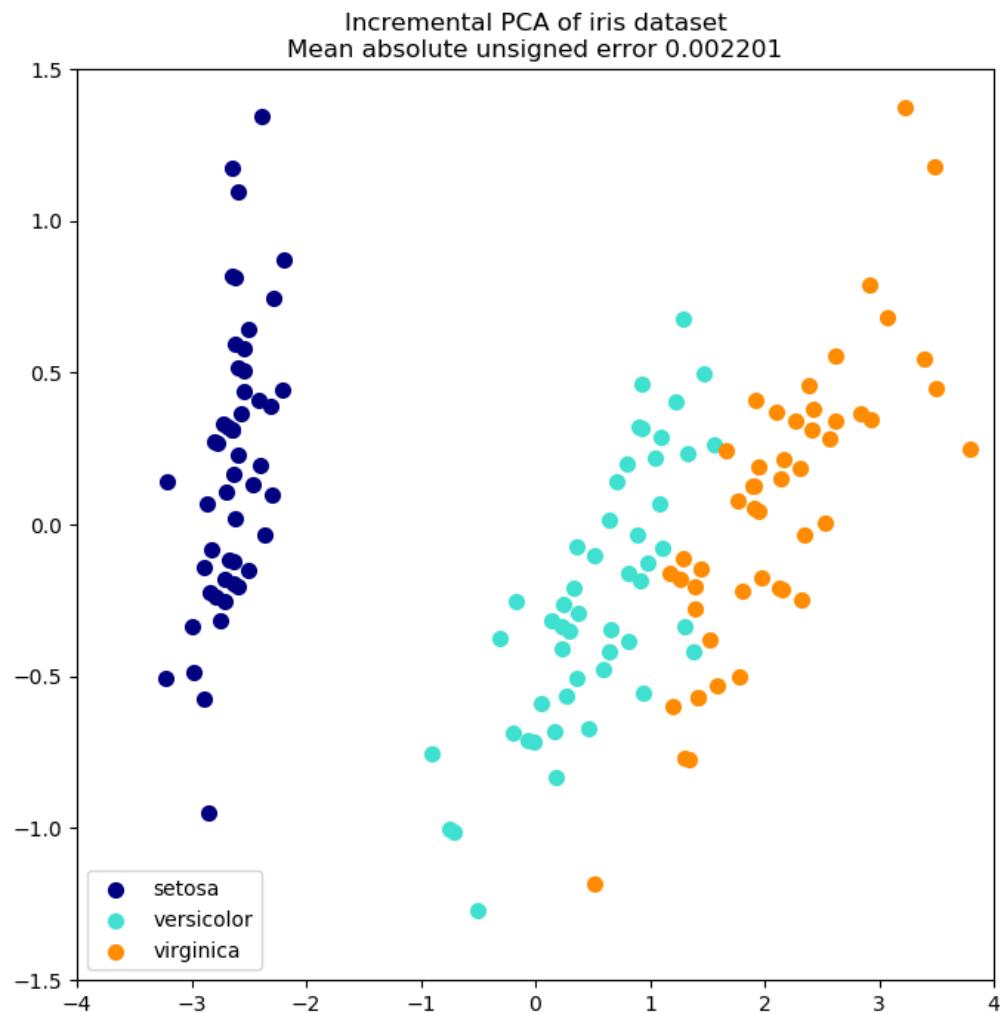
Total running time of the script: (0 minutes 0.186 seconds)

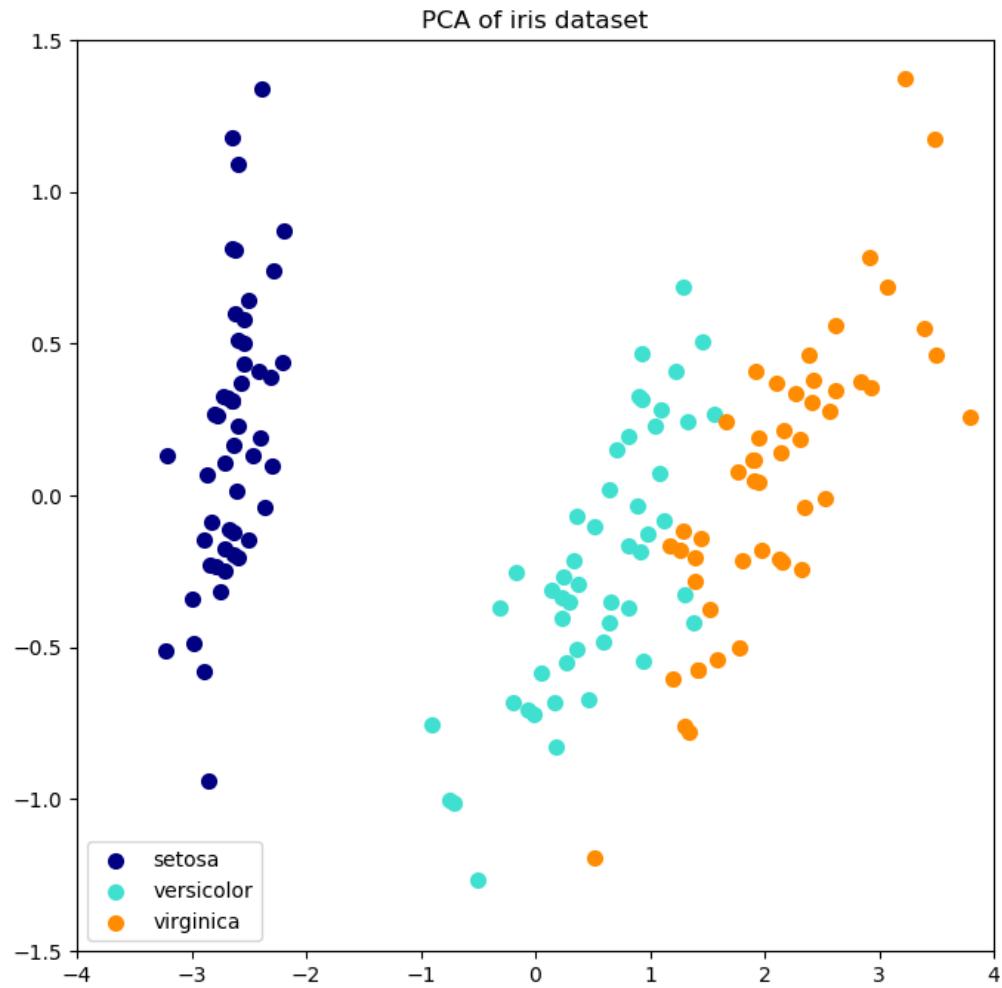
Note: Click [here](#) to download the full example code

5.11.3 Incremental PCA

Incremental principal component analysis (IPCA) is typically used as a replacement for principal component analysis (PCA) when the dataset to be decomposed is too large to fit in memory. IPCA builds a low-rank approximation for the input data using an amount of memory which is independent of the number of input data samples. It is still dependent on the input data features, but changing the batch size allows for control of memory usage.

This example serves as a visual check that IPCA is able to find a similar projection of the data to PCA (to a sign flip), while only processing a few samples at a time. This can be considered a “toy example”, as IPCA is intended for large datasets which do not fit in main memory, requiring incremental approaches.





```
print(__doc__)

# Authors: Kyle Kastner
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import load_iris
from sklearn.decomposition import PCA, IncrementalPCA

iris = load_iris()
X = iris.data
y = iris.target

n_components = 2
ipca = IncrementalPCA(n_components=n_components, batch_size=10)
```

```

X_ipca = ipca.fit_transform(X)

pca = PCA(n_components=n_components)
X_pca = pca.fit_transform(X)

colors = ['navy', 'turquoise', 'darkorange']

for X_transformed, title in [(X_ipca, "Incremental PCA"), (X_pca, "PCA")]:
    plt.figure(figsize=(8, 8))
    for color, i, target_name in zip(colors, [0, 1, 2], iris.target_names):
        plt.scatter(X_transformed[y == i, 0], X_transformed[y == i, 1],
                    color=color, lw=2, label=target_name)

    if "Incremental" in title:
        err = np.abs(np.abs(X_pca) - np.abs(X_ipca)).mean()
        plt.title(title + " of iris dataset\nMean absolute unsigned error "
                  "% .6f" % err)
    else:
        plt.title(title + " of iris dataset")
    plt.legend(loc="best", shadow=False, scatterpoints=1)
    plt.axis([-4, 4, -1.5, 1.5])

plt.show()

```

Total running time of the script: (0 minutes 0.143 seconds)

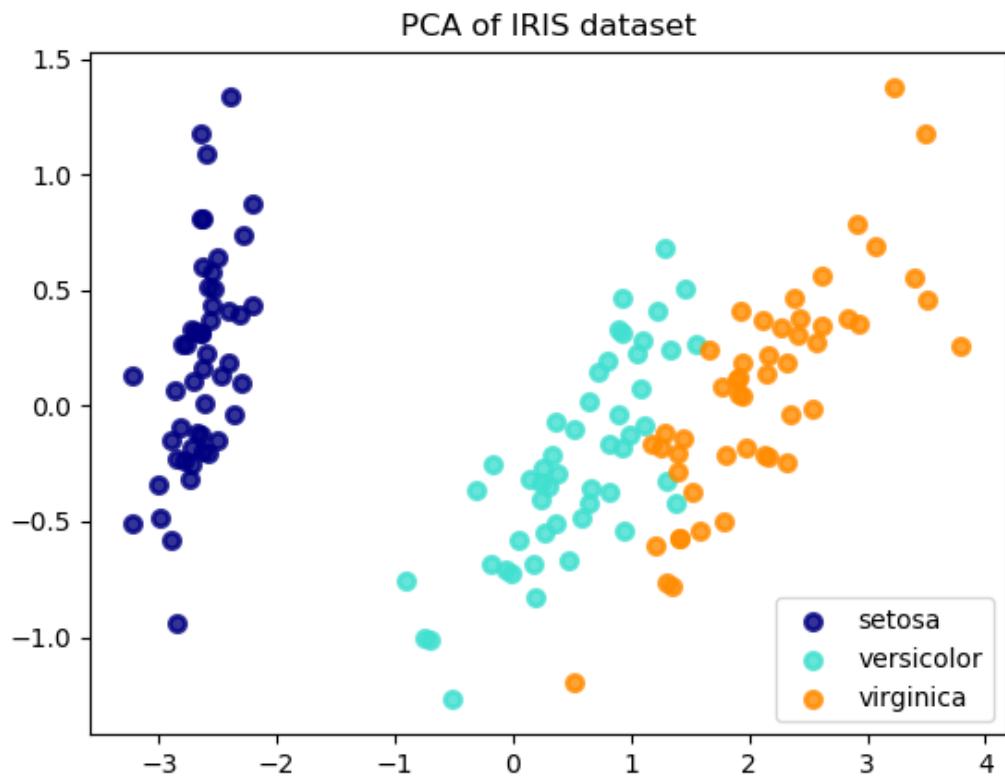
Note: Click [here](#) to download the full example code

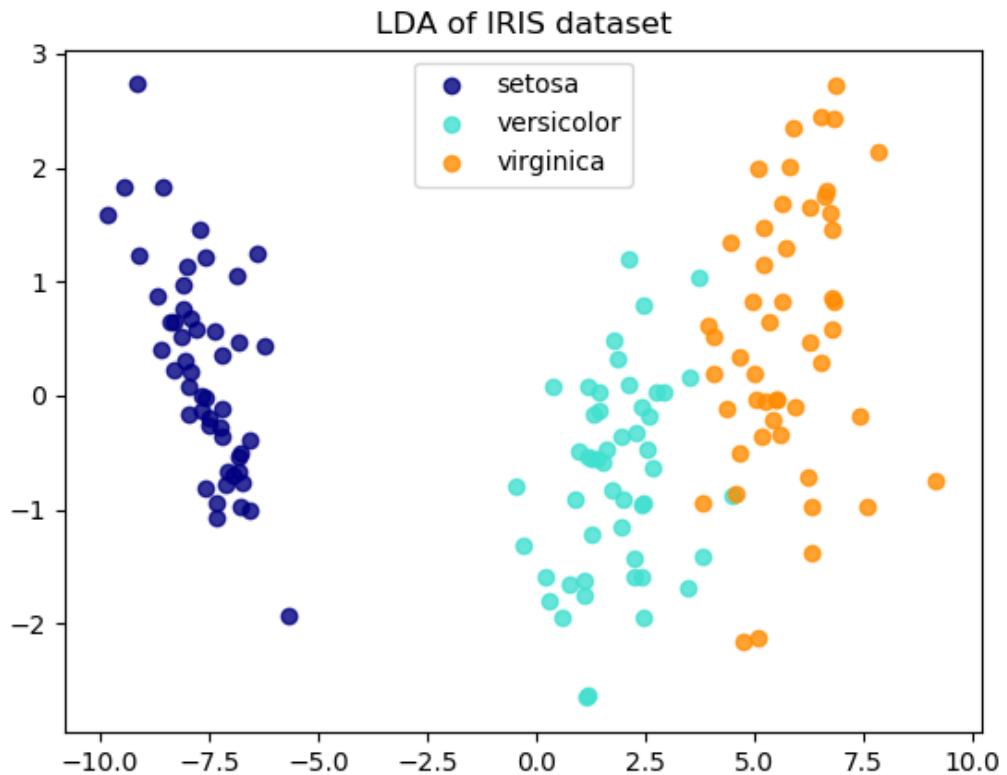
5.11.4 Comparison of LDA and PCA 2D projection of Iris dataset

The Iris dataset represents 3 kind of Iris flowers (Setosa, Versicolour and Virginica) with 4 attributes: sepal length, sepal width, petal length and petal width.

Principal Component Analysis (PCA) applied to this data identifies the combination of attributes (principal components, or directions in the feature space) that account for the most variance in the data. Here we plot the different samples on the 2 first principal components.

Linear Discriminant Analysis (LDA) tries to identify attributes that account for the most variance *between classes*. In particular, LDA, in contrast to PCA, is a supervised method, using known class labels.





Out:

```
explained variance ratio (first two components): [0.92461872 0.05306648]
```

```
print(__doc__)

import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

iris = datasets.load_iris()

X = iris.data
y = iris.target
target_names = iris.target_names

pca = PCA(n_components=2)
X_r = pca.fit(X).transform(X)

lda = LinearDiscriminantAnalysis(n_components=2)
```

```
X_r2 = lda.fit(X, y).transform(X)

# Percentage of variance explained for each components
print('explained variance ratio (first two components): %s'
      % str(pca.explained_variance_ratio_))

plt.figure()
colors = ['navy', 'turquoise', 'darkorange']
lw = 2

for color, i, target_name in zip(colors, [0, 1, 2], target_names):
    plt.scatter(X_r[y == i, 0], X_r[y == i, 1], color=color, alpha=.8, lw=lw,
                label=target_name)
plt.legend(loc='best', shadow=False, scatterpoints=1)
plt.title('PCA of IRIS dataset')

plt.figure()
for color, i, target_name in zip(colors, [0, 1, 2], target_names):
    plt.scatter(X_r2[y == i, 0], X_r2[y == i, 1], alpha=.8, color=color,
                label=target_name)
plt.legend(loc='best', shadow=False, scatterpoints=1)
plt.title('LDA of IRIS dataset')

plt.show()
```

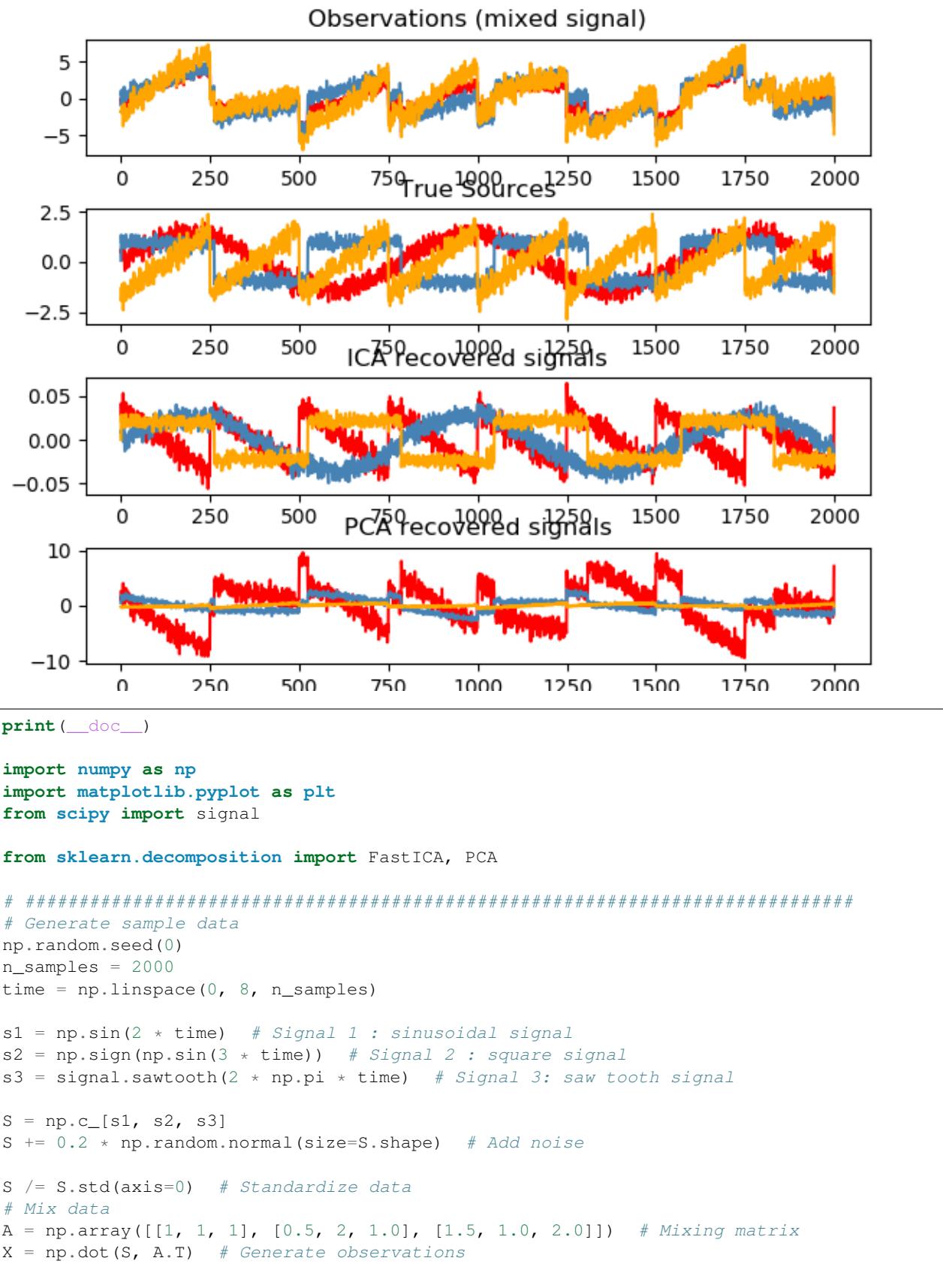
Total running time of the script: (0 minutes 0.057 seconds)

Note: Click [here](#) to download the full example code

5.11.5 Blind source separation using FastICA

An example of estimating sources from noisy data.

Independent component analysis (ICA) is used to estimate sources given noisy measurements. Imagine 3 instruments playing simultaneously and 3 microphones recording the mixed signals. ICA is used to recover the sources ie. what is played by each instrument. Importantly, PCA fails at recovering our instruments since the related signals reflect non-Gaussian processes.



```
# Compute ICA
ica = FastICA(n_components=3)
S_ = ica.fit_transform(X)    # Reconstruct signals
A_ = ica.mixing_ # Get estimated mixing matrix

# We can `prove` that the ICA model applies by reverting the unmixing.
assert np.allclose(X, np.dot(S_, A_.T) + ica.mean_)

# For comparison, compute PCA
pca = PCA(n_components=3)
H = pca.fit_transform(X)    # Reconstruct signals based on orthogonal components

# ######
# Plot results

plt.figure()

models = [X, S_, H]
names = ['Observations (mixed signal)',
         'True Sources',
         'ICA recovered signals',
         'PCA recovered signals']
colors = ['red', 'steelblue', 'orange']

for ii, (model, name) in enumerate(zip(models, names), 1):
    plt.subplot(4, 1, ii)
    plt.title(name)
    for sig, color in zip(model.T, colors):
        plt.plot(sig, color=color)

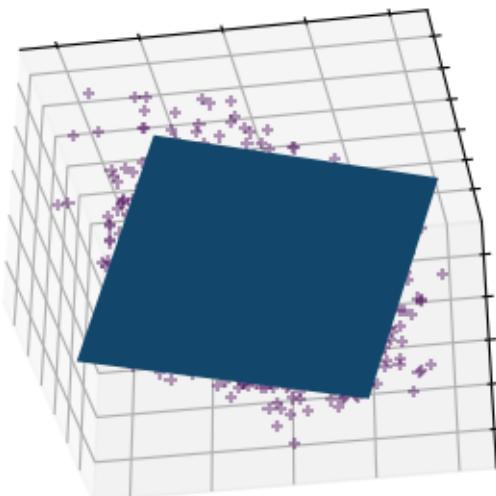
plt.subplots_adjust(0.09, 0.04, 0.94, 0.94, 0.26, 0.46)
plt.show()
```

Total running time of the script: (0 minutes 0.089 seconds)

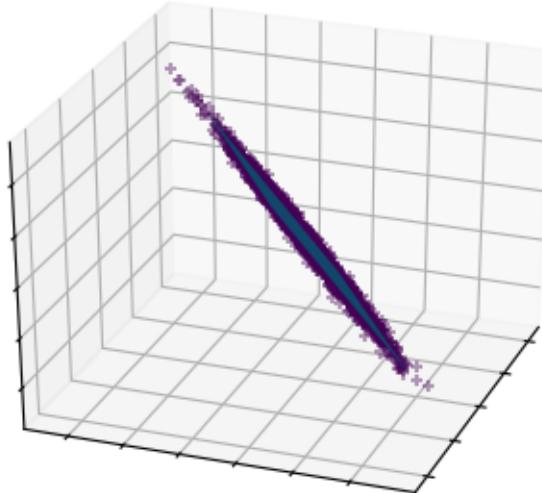
Note: Click [here](#) to download the full example code

5.11.6 Principal components analysis (PCA)

These figures aid in illustrating how a point cloud can be very flat in one direction—which is where PCA comes in to choose a direction that is not flat.



•



•

```
print(__doc__)

# Authors: Gael Varoquaux
#          Jaques Grobler
#          Kevin Hughes
# License: BSD 3 clause

from sklearn.decomposition import PCA

from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

#####
# Create the data
```

```

e = np.exp(1)
np.random.seed(4)

def pdf(x):
    return 0.5 * (stats.norm(scale=0.25 / e).pdf(x)
                  + stats.norm(scale=4 / e).pdf(x))

y = np.random.normal(scale=0.5, size=(30000))
x = np.random.normal(scale=0.5, size=(30000))
z = np.random.normal(scale=0.1, size=len(x))

density = pdf(x) * pdf(y)
pdf_z = pdf(5 * z)

density *= pdf_z

a = x + y
b = 2 * y
c = a - b + z

norm = np.sqrt(a.var() + b.var())
a /= norm
b /= norm

# ######
# Plot the figures
def plot_figs(fig_num, elev, azim):
    fig = plt.figure(fig_num, figsize=(4, 3))
    plt.clf()
    ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=elev, azim=azim)

    ax.scatter(a[::10], b[::10], c[::10], c=density[::10], marker='+', alpha=.4)
    Y = np.c_[a, b, c]

    # Using SciPy's SVD, this would be:
    # _, pca_score, V = scipy.linalg.svd(Y, full_matrices=False)

    pca = PCA(n_components=3)
    pca.fit(Y)
    pca_score = pca.explained_variance_ratio_
    V = pca.components_

    x_pca_axis, y_pca_axis, z_pca_axis = 3 * V.T
    x_pca_plane = np.r_[x_pca_axis[:2], -x_pca_axis[1:-1]]
    y_pca_plane = np.r_[y_pca_axis[:2], -y_pca_axis[1:-1]]
    z_pca_plane = np.r_[z_pca_axis[:2], -z_pca_axis[1:-1]]
    x_pca_plane.shape = (2, 2)
    y_pca_plane.shape = (2, 2)
    z_pca_plane.shape = (2, 2)
    ax.plot_surface(x_pca_plane, y_pca_plane, z_pca_plane)
    ax.w_xaxis.set_ticklabels([])
    ax.w_yaxis.set_ticklabels([])
    ax.w_zaxis.set_ticklabels([])

elev = -40

```

```

azim = -80
plot_figs(1, elev, azim)

elev = 30
azim = 20
plot_figs(2, elev, azim)

plt.show()

```

Total running time of the script: (0 minutes 0.114 seconds)

Note: Click [here](#) to download the full example code

5.11.7 FastICA on 2D point clouds

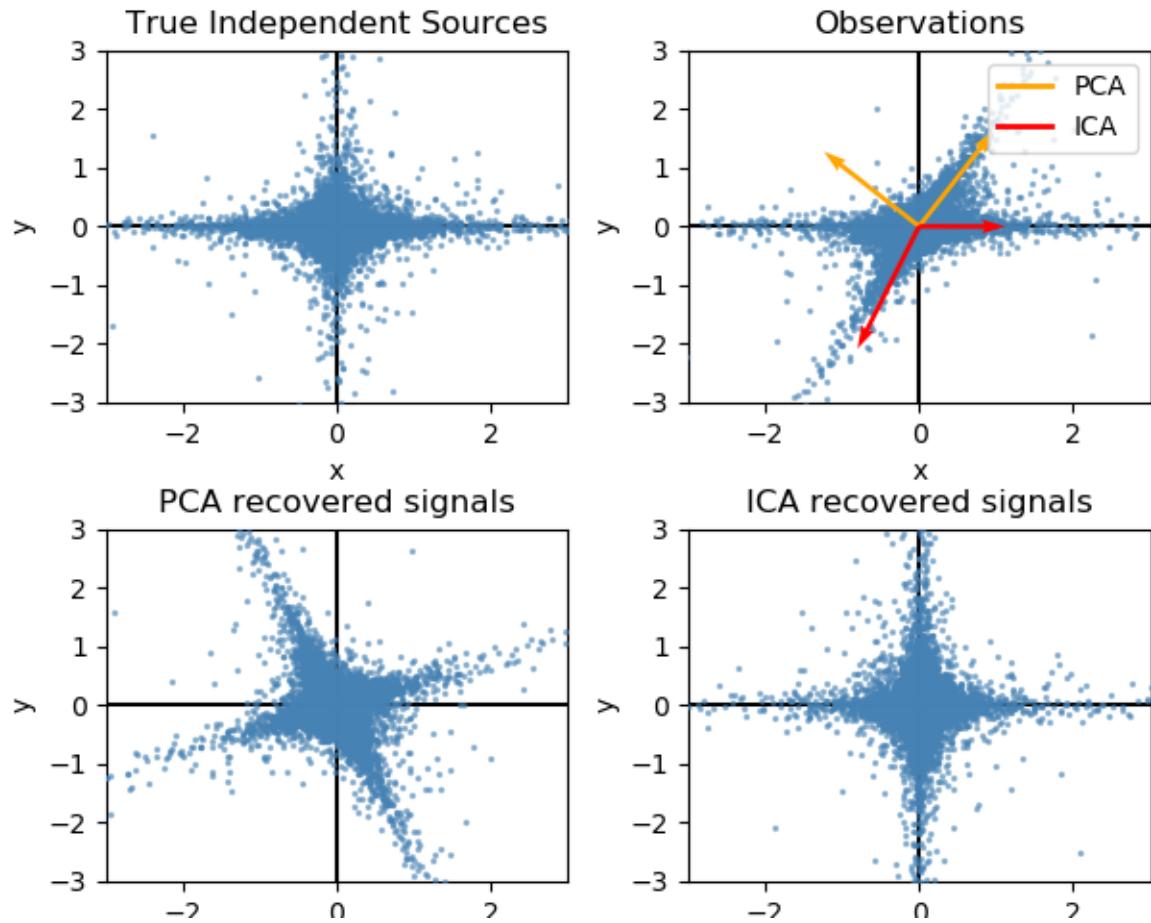
This example illustrates visually in the feature space a comparison by results using two different component analysis techniques.

Independent component analysis (ICA) vs Principal component analysis (PCA).

Representing ICA in the feature space gives the view of ‘geometric ICA’: ICA is an algorithm that finds directions in the feature space corresponding to projections with high non-Gaussianity. These directions need not be orthogonal in the original feature space, but they are orthogonal in the whitened feature space, in which all directions correspond to the same variance.

PCA, on the other hand, finds orthogonal directions in the raw feature space that correspond to directions accounting for maximum variance.

Here we simulate independent sources using a highly non-Gaussian process, 2 student T with a low number of degrees of freedom (top left figure). We mix them to create observations (top right figure). In this raw observation space, directions identified by PCA are represented by orange vectors. We represent the signal in the PCA space, after whitening by the variance corresponding to the PCA vectors (lower left). Running ICA corresponds to finding a rotation in this space to identify the directions of largest non-Gaussianity (lower right).



```
print(__doc__)

# Authors: Alexandre Gramfort, Gael Varoquaux
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.decomposition import PCA, FastICA

#####
# Generate sample data
rng = np.random.RandomState(42)
S = rng.standard_t(1.5, size=(20000, 2))
S[:, 0] *= 2.

# Mix data
A = np.array([[1, 1], [0, 2]]) # Mixing matrix

X = np.dot(S, A.T) # Generate observations

pca = PCA()
S_pca_ = pca.fit(X).transform(X)

ica = FastICA(random_state=rng)
S_ica_ = ica.fit(X).transform(X) # Estimate the sources
```

```

S_ica_ /= S_ica_.std(axis=0)

# ##### Plot results

def plot_samples(S, axis_list=None):
    plt.scatter(S[:, 0], S[:, 1], s=2, marker='o', zorder=10,
                color='steelblue', alpha=0.5)
    if axis_list is not None:
        colors = ['orange', 'red']
        for color, axis in zip(colors, axis_list):
            axis /= axis.std()
            x_axis, y_axis = axis
            # Trick to get legend to work
            plt.plot(0.1 * x_axis, 0.1 * y_axis, linewidth=2, color=color)
            plt.quiver(0, 0, x_axis, y_axis, zorder=11, width=0.01, scale=6,
                       color=color)

    plt.hlines(0, -3, 3)
    plt.vlines(0, -3, 3)
    plt.xlim(-3, 3)
    plt.ylim(-3, 3)
    plt.xlabel('x')
    plt.ylabel('y')

plt.figure()
plt.subplot(2, 2, 1)
plot_samples(S / S.std())
plt.title('True Independent Sources')

axis_list = [pca.components_.T, ica.mixing_]
plt.subplot(2, 2, 2)
plot_samples(X / np.std(X), axis_list=axis_list)
legend = plt.legend(['PCA', 'ICA'], loc='upper right')
legend.set_zorder(100)

plt.title('Observations')

plt.subplot(2, 2, 3)
plot_samples(S_pca_ / np.std(S_pca_, axis=0))
plt.title('PCA recovered signals')

plt.subplot(2, 2, 4)
plot_samples(S_ica_ / np.std(S_ica_))
plt.title('ICA recovered signals')

plt.subplots_adjust(0.09, 0.04, 0.94, 0.94, 0.26, 0.36)
plt.show()

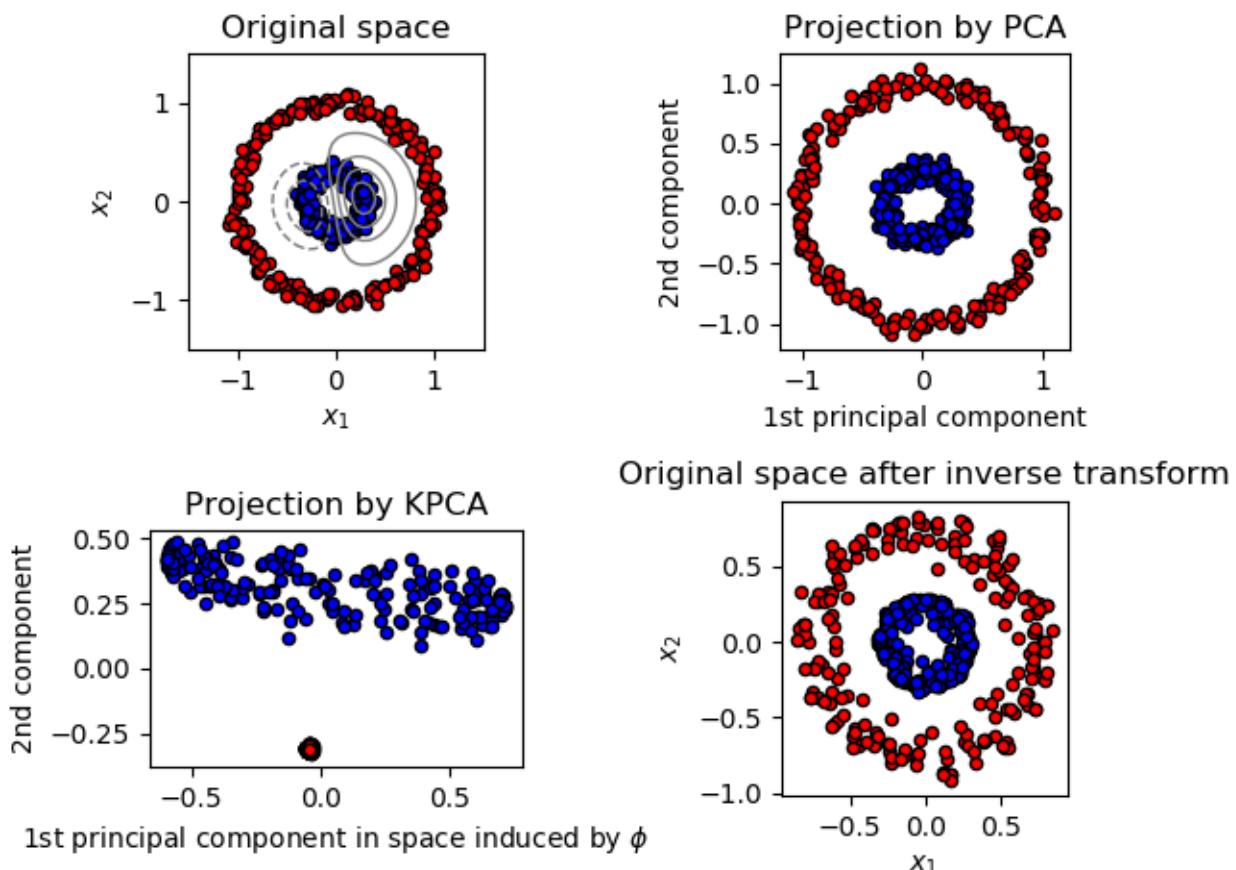
```

Total running time of the script: (0 minutes 0.357 seconds)

Note: Click [here](#) to download the full example code

5.11.8 Kernel PCA

This example shows that Kernel PCA is able to find a projection of the data that makes data linearly separable.



```
print(__doc__)

# Authors: Mathieu Blondel
#          Andreas Mueller
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.decomposition import PCA, KernelPCA
from sklearn.datasets import make_circles

np.random.seed(0)

X, y = make_circles(n_samples=400, factor=.3, noise=.05)

k pca = KernelPCA(kernel="rbf", fit_inverse_transform=True, gamma=10)
X_kpca = k pca.fit_transform(X)
X_back = k pca.inverse_transform(X_kpca)
pca = PCA()
X_pca = pca.fit_transform(X)
```

```
# Plot results

plt.figure()
plt.subplot(2, 2, 1, aspect='equal')
plt.title("Original space")
reds = y == 0
blues = y == 1

plt.scatter(X[reds, 0], X[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X[blues, 0], X[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")

X1, X2 = np.meshgrid(np.linspace(-1.5, 1.5, 50), np.linspace(-1.5, 1.5, 50))
X_grid = np.array([np.ravel(X1), np.ravel(X2)]).T
# projection on the first principal component (in the phi space)
Z_grid = kpca.transform(X_grid)[:, 0].reshape(X1.shape)
plt.contour(X1, X2, Z_grid, colors='grey', linewidths=1, origin='lower')

plt.subplot(2, 2, 2, aspect='equal')
plt.scatter(X_pca[reds, 0], X_pca[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X_pca[blues, 0], X_pca[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.title("Projection by PCA")
plt.xlabel("1st principal component")
plt.ylabel("2nd component")

plt.subplot(2, 2, 3, aspect='equal')
plt.scatter(X_kpca[reds, 0], X_kpca[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X_kpca[blues, 0], X_kpca[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.title("Projection by KPCA")
plt.xlabel(r"1st principal component in space induced by $\phi$")
plt.ylabel("2nd component")

plt.subplot(2, 2, 4, aspect='equal')
plt.scatter(X_back[reds, 0], X_back[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X_back[blues, 0], X_back[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.title("Original space after inverse transform")
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.460 seconds)

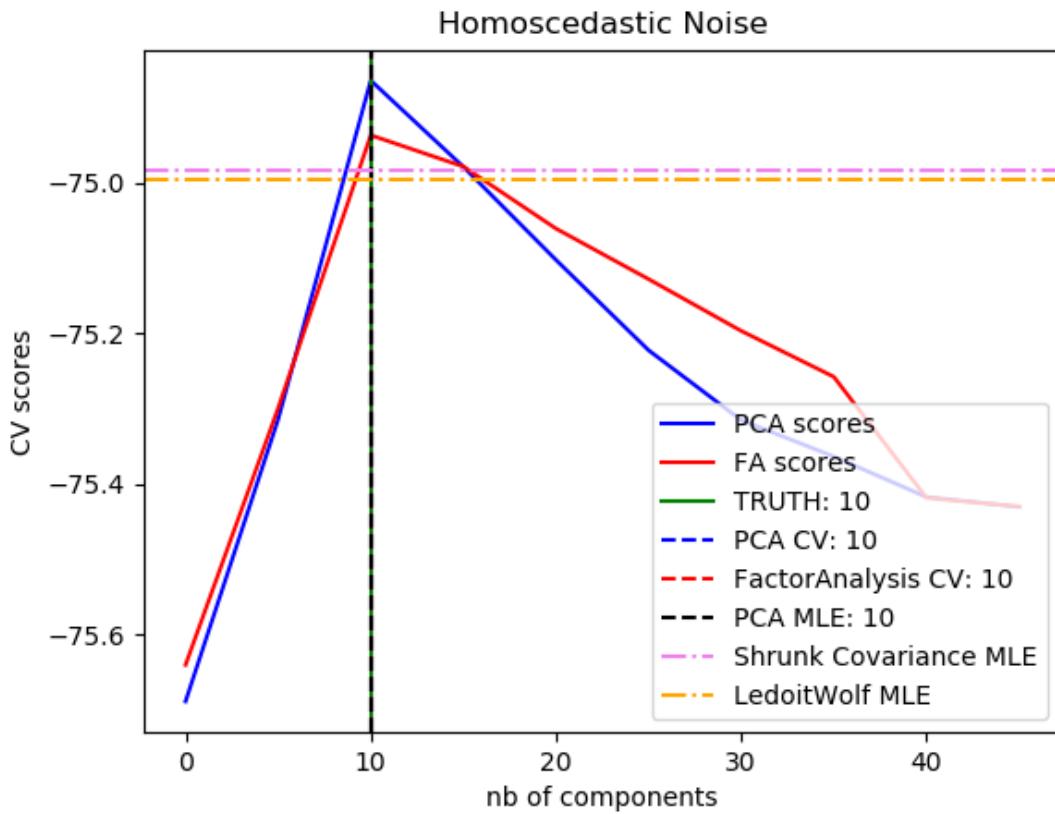
Note: Click [here](#) to download the full example code

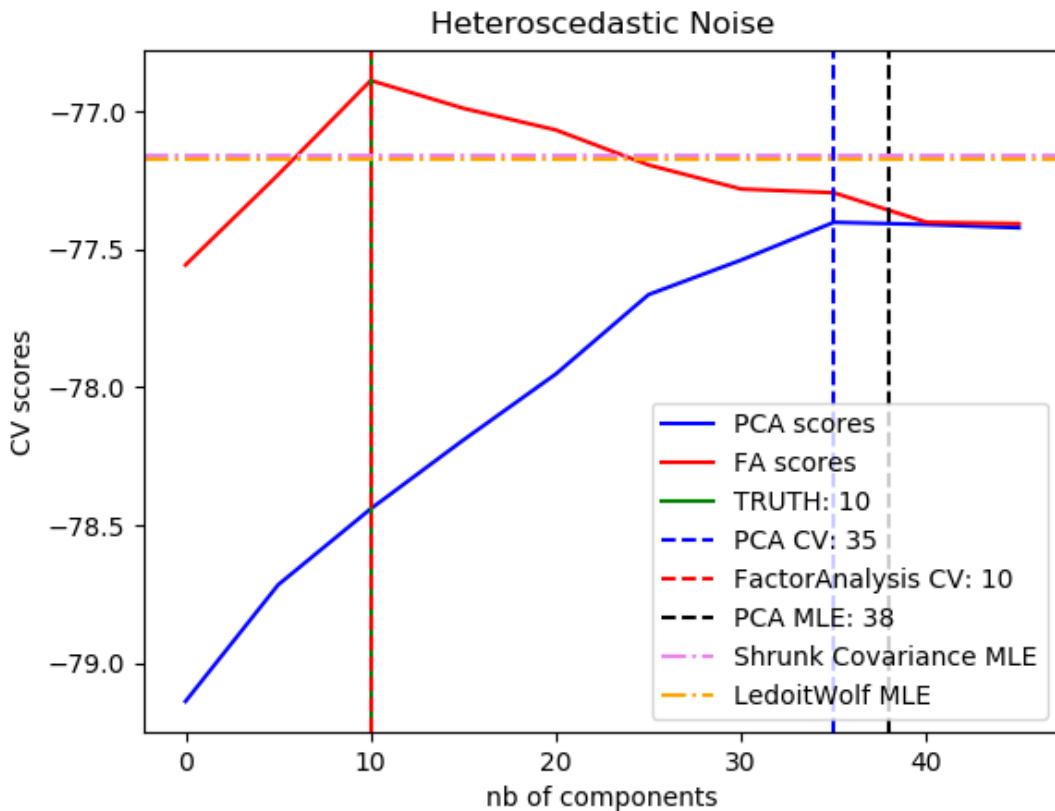
5.11.9 Model selection with Probabilistic PCA and Factor Analysis (FA)

Probabilistic PCA and Factor Analysis are probabilistic models. The consequence is that the likelihood of new data can be used for model selection and covariance estimation. Here we compare PCA and FA with cross-validation on low rank data corrupted with homoscedastic noise (noise variance is the same for each feature) or heteroscedastic noise (noise variance is the different for each feature). In a second step we compare the model likelihood to the likelihoods obtained from shrinkage covariance estimators.

One can observe that with homoscedastic noise both FA and PCA succeed in recovering the size of the low rank subspace. The likelihood with PCA is higher than FA in this case. However PCA fails and overestimates the rank when heteroscedastic noise is present. Under appropriate circumstances the low rank models are more likely than shrinkage models.

The automatic estimation from Automatic Choice of Dimensionality for PCA. NIPS 2000: 598-604 by Thomas P. Minka is also compared.





Out:

```
best n_components by PCA CV = 10
best n_components by FactorAnalysis CV = 10
best n_components by PCA MLE = 10
best n_components by PCA CV = 35
best n_components by FactorAnalysis CV = 10
best n_components by PCA MLE = 38
```

```
# Authors: Alexandre Gramfort
#          Denis A. Engemann
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg

from sklearn.decomposition import PCA, FactorAnalysis
from sklearn.covariance import ShrunkCovariance, LedoitWolf
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
```

```

print(__doc__)

# ##### Create the data #####
# Create the data

n_samples, n_features, rank = 1000, 50, 10
sigma = 1.
rng = np.random.RandomState(42)
U, _, _ = linalg.svd(rng.randn(n_features, n_features))
X = np.dot(rng.randn(n_samples, rank), U[:, :rank].T)

# Adding homoscedastic noise
X_homo = X + sigma * rng.randn(n_samples, n_features)

# Adding heteroscedastic noise
sigmas = sigma * rng.rand(n_features) + sigma / 2.
X_hetero = X + rng.randn(n_samples, n_features) * sigmas

# ##### Fit the models #####
# Fit the models

n_components = np.arange(0, n_features, 5) # options for n_components

def compute_scores(X):
    pca = PCA(svd_solver='full')
    fa = FactorAnalysis()

    pca_scores, fa_scores = [], []
    for n in n_components:
        pca.n_components = n
        fa.n_components = n
        pca_scores.append(np.mean(cross_val_score(pca, X, cv=5)))
        fa_scores.append(np.mean(cross_val_score(fa, X, cv=5)))

    return pca_scores, fa_scores

def shrunk_cov_score(X):
    shrinkages = np.logspace(-2, 0, 30)
    cv = GridSearchCV(ShrunkCovariance(), {'shrinkage': shrinkages}, cv=5)
    return np.mean(cross_val_score(cv.fit(X).best_estimator_, X, cv=5))

def lw_score(X):
    return np.mean(cross_val_score(LedoitWolf(), X, cv=5))

for X, title in [(X_homo, 'Homoscedastic Noise'),
                  (X_hetero, 'Heteroscedastic Noise')]:
    pca_scores, fa_scores = compute_scores(X)
    n_components_pca = n_components[np.argmax(pca_scores)]
    n_components_fa = n_components[np.argmax(fa_scores)]

    pca = PCA(svd_solver='full', n_components='mle')
    pca.fit(X)
    n_components_pca_mle = pca.n_components_

```

```

print("best n_components by PCA CV = %d" % n_components_pca)
print("best n_components by FactorAnalysis CV = %d" % n_components_fa)
print("best n_components by PCA MLE = %d" % n_components_pca_mle)

plt.figure()
plt.plot(n_components, pca_scores, 'b', label='PCA scores')
plt.plot(n_components, fa_scores, 'r', label='FA scores')
plt.axvline(rank, color='g', label='TRUTH: %d' % rank, linestyle='--')
plt.axvline(n_components_pca, color='b',
            label='PCA CV: %d' % n_components_pca, linestyle='--')
plt.axvline(n_components_fa, color='r',
            label='FactorAnalysis CV: %d' % n_components_fa,
            linestyle='--')
plt.axvline(n_components_pca_mle, color='k',
            label='PCA MLE: %d' % n_components_pca_mle, linestyle='--')

# compare with other covariance estimators
plt.axhline(shrunken_cov_score(X), color='violet',
            label='Shrunken Covariance MLE', linestyle='-.')
plt.axhline(lw_score(X), color='orange',
            label='LedoitWolf MLE' % n_components_pca_mle, linestyle='-.')

plt.xlabel('nb of components')
plt.ylabel('CV scores')
plt.legend(loc='lower right')
plt.title(title)

plt.show()

```

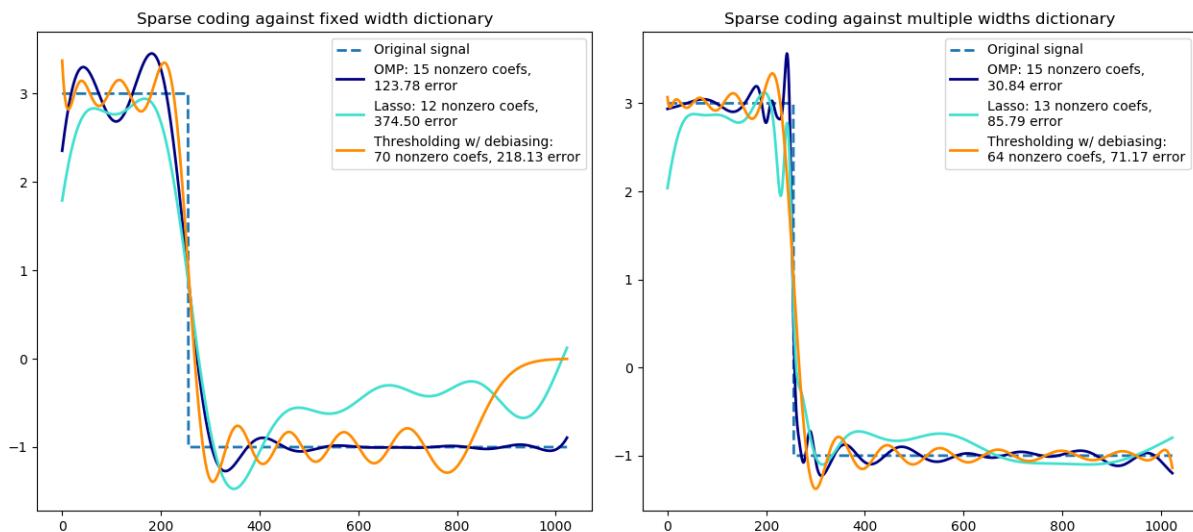
Total running time of the script: (0 minutes 17.528 seconds)

Note: Click [here](#) to download the full example code

5.11.10 Sparse coding with a precomputed dictionary

Transform a signal as a sparse combination of Ricker wavelets. This example visually compares different sparse coding methods using the `sklearn.decomposition.SparseCoder` estimator. The Ricker (also known as Mexican hat or the second derivative of a Gaussian) is not a particularly good kernel to represent piecewise constant signals like this one. It can therefore be seen how much adding different widths of atoms matters and it therefore motivates learning the dictionary to best fit your type of signals.

The richer dictionary on the right is not larger in size, heavier subsampling is performed in order to stay on the same order of magnitude.



```

print(__doc__)

from distutils.version import LooseVersion

import numpy as np
import matplotlib.pyplot as plt

from sklearn.decomposition import SparseCoder


def ricker_function(resolution, center, width):
    """Discrete sub-sampled Ricker (Mexican hat) wavelet"""
    x = np.linspace(0, resolution - 1, resolution)
    x = ((2 / ((np.sqrt(3 * width) * np.pi ** 1 / 4)))
          * (1 - ((x - center) ** 2 / width ** 2))
          * np.exp((- (x - center) ** 2) / (2 * width ** 2)))
    return x


def ricker_matrix(width, resolution, n_components):
    """Dictionary of Ricker (Mexican hat) wavelets"""
    centers = np.linspace(0, resolution - 1, n_components)
    D = np.empty((n_components, resolution))
    for i, center in enumerate(centers):
        D[i] = ricker_function(resolution, center, width)
    D /= np.sqrt(np.sum(D ** 2, axis=1))[:, np.newaxis]
    return D


resolution = 1024
subsampling = 3 # subsampling factor
width = 100
n_components = resolution // subsampling

# Compute a wavelet dictionary
D_fixed = ricker_matrix(width=width, resolution=resolution,
                        n_components=n_components)
D_multi = np.r_[tuple(ricker_matrix(width=w, resolution=resolution,

```

```

        n_components=n_components // 5)
    for w in (10, 50, 100, 500, 1000))]

# Generate a signal
y = np.linspace(0, resolution - 1, resolution)
first_quarter = y < resolution / 4
y[first_quarter] = 3.
y[np.logical_not(first_quarter)] = -1.

# List the different sparse coding methods in the following format:
# (title, transform_algorithm, transform_alpha,
# transform_n_nonzero_coefs, color)
estimators = [('OMP', 'omp', None, 15, 'navy'),
               ('Lasso', 'lasso_lars', 2, None, 'turquoise'), ]
lw = 2
# Avoid FutureWarning about default value change when numpy >= 1.14
lstsq_rcond = None if LooseVersion(np.__version__) >= '1.14' else -1

plt.figure(figsize=(13, 6))
for subplot, (D, title) in enumerate(zip((D_fixed, D_multi),
                                         ('fixed width', 'multiple widths'))):
    plt.subplot(1, 2, subplot + 1)
    plt.title('Sparse coding against %s dictionary' % title)
    plt.plot(y, lw=lw, linestyle='--', label='Original signal')
    # Do a wavelet approximation
    for title, algo, alpha, n_nonzero, color in estimators:
        coder = SparseCoder(dictionary=D, transform_n_nonzero_coefs=n_nonzero,
                             transform_alpha=alpha, transform_algorithm=algo)
        x = coder.transform(y.reshape(1, -1))
        density = len(np.flatnonzero(x))
        x = np.ravel(np.dot(x, D))
        squared_error = np.sum((y - x) ** 2)
        plt.plot(x, color=color, lw=lw,
                  label='%s: %s nonzero coefs, %.2f error'
                  % (title, density, squared_error))

    # Soft thresholding debiasing
    coder = SparseCoder(dictionary=D, transform_algorithm='threshold',
                         transform_alpha=20)
    x = coder.transform(y.reshape(1, -1))
    _, idx = np.where(x != 0)
    x[0, idx], _, _, _ = np.linalg.lstsq(D[idx, :].T, y, rcond=lstsq_rcond)
    x = np.ravel(np.dot(x, D))
    squared_error = np.sum((y - x) ** 2)
    plt.plot(x, color='darkorange', lw=lw,
              label='Thresholding w/ debiasing: %d nonzero coefs, %.2f error'
              % (len(idx), squared_error))
    plt.axis('tight')
    plt.legend(shadow=False, loc='best')
plt.subplots_adjust(.04, .07, .97, .90, .09, .2)
plt.show()

```

Total running time of the script: (0 minutes 0.238 seconds)

Note: Click [here](#) to download the full example code

5.11.11 Image denoising using dictionary learning

An example comparing the effect of reconstructing noisy fragments of a raccoon face image using firstly online *Dictionary Learning* and various transform methods.

The dictionary is fitted on the distorted left half of the image, and subsequently used to reconstruct the right half. Note that even better performance could be achieved by fitting to an undistorted (i.e. noiseless) image, but here we start from the assumption that it is not available.

A common practice for evaluating the results of image denoising is by looking at the difference between the reconstruction and the original image. If the reconstruction is perfect this will look like Gaussian noise.

It can be seen from the plots that the results of *Orthogonal Matching Pursuit (OMP)* with two non-zero coefficients is a bit less biased than when keeping only one (the edges look less prominent). It is in addition closer from the ground truth in Frobenius norm.

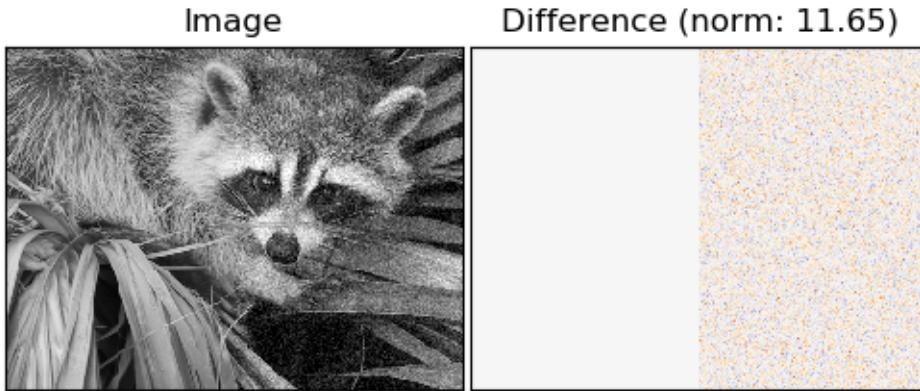
The result of *Least Angle Regression* is much more strongly biased: the difference is reminiscent of the local intensity value of the original image.

Thresholding is clearly not useful for denoising, but it is here to show that it can produce a suggestive output with very high speed, and thus be useful for other tasks such as object classification, where performance is not necessarily related to visualisation.

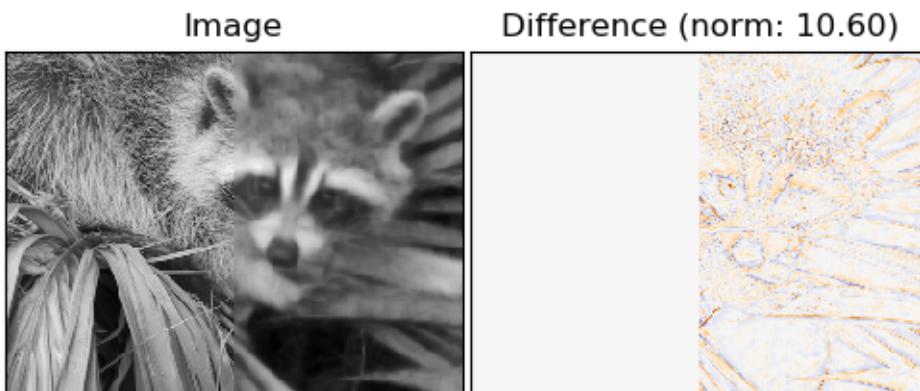
Dictionary learned from face patches
Train time 7.2s on 22692 patches



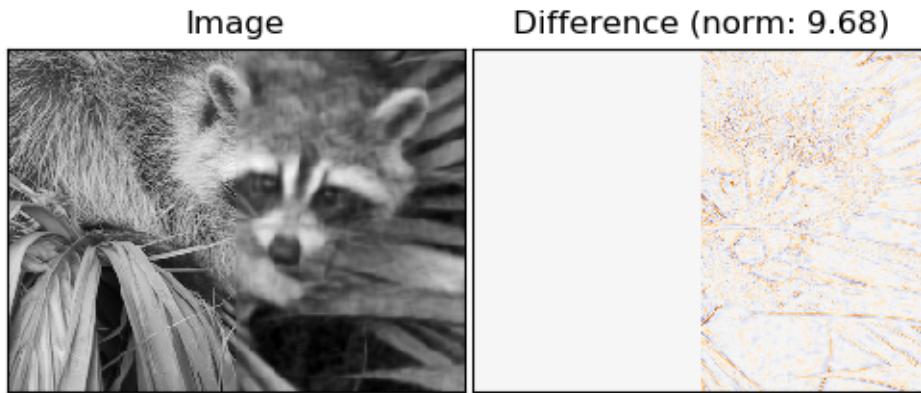
Distorted image



• Orthogonal Matching Pursuit
1 atom (time: 1.2s)

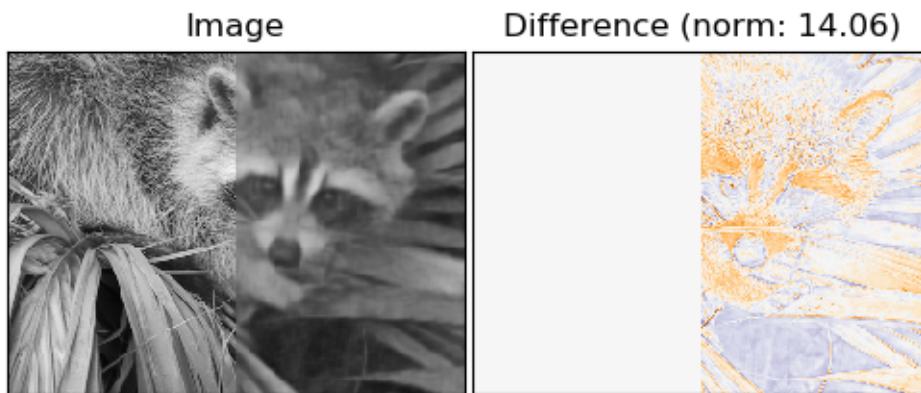


Orthogonal Matching Pursuit 2 atoms (time: 2.4s)



.

Least-angle regression 5 atoms (time: 17.5s)



.

Thresholding alpha=0.1 (time: 0.3s)



-

Out:

```
Distorting image...
Extracting reference patches...
done in 0.01s.
Learning the dictionary...
done in 7.22s.
Extracting noisy patches...
done in 0.00s.
Orthogonal Matching Pursuit
1 atom...
done in 1.19s.
Orthogonal Matching Pursuit
2 atoms...
done in 2.35s.
Least-angle regression
5 atoms...
done in 17.50s.
Thresholding
alpha=0.1...
done in 0.25s.
```

```
print(__doc__)

from time import time

import matplotlib.pyplot as plt
import numpy as np
import scipy as sp

from sklearn.decomposition import MiniBatchDictionaryLearning
```

```

from sklearn.feature_extraction.image import extract_patches_2d
from sklearn.feature_extraction.image import reconstruct_from_patches_2d

try: # SciPy >= 0.16 have face in misc
    from scipy.misc import face
    face = face(gray=True)
except ImportError:
    face = sp.face(gray=True)

# Convert from uint8 representation with values between 0 and 255 to
# a floating point representation with values between 0 and 1.
face = face / 255.

# downsample for higher speed
face = face[::4, ::4] + face[1::4, ::4] + face[::4, 1::4] + face[1::4, 1::4]
face /= 4.0
height, width = face.shape

# Distort the right half of the image
print('Distorting image....')
distorted = face.copy()
distorted[:, width // 2:] += 0.075 * np.random.randn(height, width // 2)

# Extract all reference patches from the left half of the image
print('Extracting reference patches...')
t0 = time()
patch_size = (7, 7)
data = extract_patches_2d(distorted[:, :width // 2], patch_size)
data = data.reshape(data.shape[0], -1)
data -= np.mean(data, axis=0)
data /= np.std(data, axis=0)
print('done in %.2fs.' % (time() - t0))

#####
# Learn the dictionary from reference patches

print('Learning the dictionary...')
t0 = time()
dico = MiniBatchDictionaryLearning(n_components=100, alpha=1, n_iter=500)
V = dico.fit(data).components_
dt = time() - t0
print('done in %.2fs.' % dt)

plt.figure(figsize=(4.2, 4))
for i, comp in enumerate(V[:100]):
    plt.subplot(10, 10, i + 1)
    plt.imshow(comp.reshape(patch_size), cmap=plt.cm.gray_r,
               interpolation='nearest')
    plt.xticks(())
    plt.yticks(())
plt.suptitle('Dictionary learned from face patches\n' +
             'Train time %.1fs on %d patches' % (dt, len(data)),
             fontsize=16)
plt.subplots_adjust(0.08, 0.02, 0.92, 0.85, 0.08, 0.23)

#####

```

```

# Display the distorted image

def show_with_diff(image, reference, title):
    """Helper function to display denoising"""
    plt.figure(figsize=(5, 3.3))
    plt.subplot(1, 2, 1)
    plt.title('Image')
    plt.imshow(image, vmin=0, vmax=1, cmap=plt.cm.gray,
               interpolation='nearest')
    plt.xticks(())
    plt.yticks(())
    plt.subplot(1, 2, 2)
    difference = image - reference

    plt.title('Difference (norm: %.2f)' % np.sqrt(np.sum(difference ** 2)))
    plt.imshow(difference, vmin=-0.5, vmax=0.5, cmap=plt.cm.PuOr,
               interpolation='nearest')
    plt.xticks(())
    plt.yticks(())
    plt.suptitle(title, size=16)
    plt.subplots_adjust(0.02, 0.02, 0.98, 0.79, 0.02, 0.2)

show_with_diff(distorted, face, 'Distorted image')

#####
# Extract noisy patches and reconstruct them using the dictionary

print('Extracting noisy patches... ')
t0 = time()
data = extract_patches_2d(distorted[:, width // 2:], patch_size)
data = data.reshape(data.shape[0], -1)
intercept = np.mean(data, axis=0)
data -= intercept
print('done in %.2fs.' % (time() - t0))

transform_algorithms = [
    ('Orthogonal Matching Pursuit\n1 atom', 'omp',
     {'transform_n_nonzero_coefs': 1}),
    ('Orthogonal Matching Pursuit\n2 atoms', 'omp',
     {'transform_n_nonzero_coefs': 2}),
    ('Least-angle regression\n5 atoms', 'lars',
     {'transform_n_nonzero_coefs': 5}),
    ('Thresholding\n alpha=0.1', 'threshold', {'transform_alpha': .1})]

reconstructions = {}
for title, transform_algorithm, kwargs in transform_algorithms:
    print(title + '...')
    reconstructions[title] = face.copy()
    t0 = time()
    dico.set_params(transform_algorithm=transform_algorithm, **kwargs)
    code = dico.transform(data)
    patches = np.dot(code, V)

    patches += intercept
    patches = patches.reshape(len(data), *patch_size)
    if transform_algorithm == 'threshold':
        patches -= patches.min()
        patches /= patches.max()

```

```
reconstructions[title][:, width // 2:] = reconstruct_from_patches_2d(
    patches, (height, width // 2))
dt = time() - t0
print('done in %.2fs.' % dt)
show_with_diff(reconstructions[title], face,
               title + ' (time: %.1fs)' % dt)

plt.show()
```

Total running time of the script: (0 minutes 30.244 seconds)

Note: Click [here](#) to download the full example code

5.11.12 Faces dataset decompositions

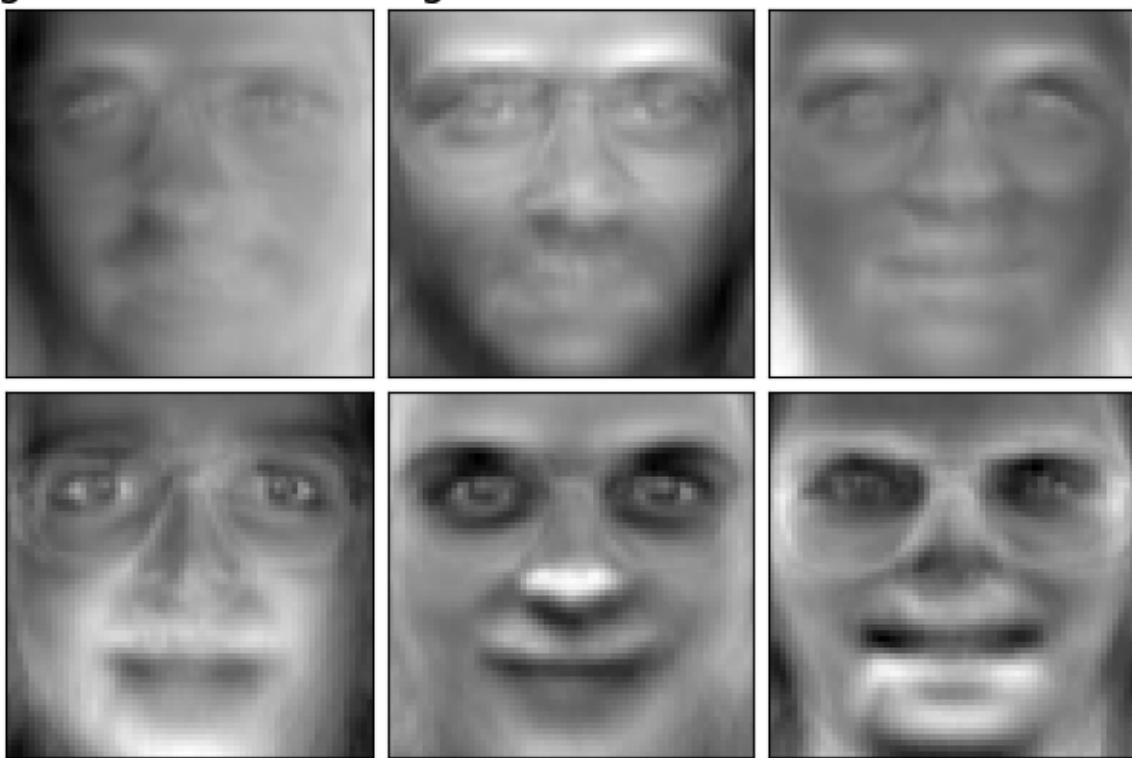
This example applies to olivetti_faces different unsupervised matrix decomposition (dimension reduction) methods from the module `sklearn.decomposition` (see the documentation chapter *Decomposing signals in components (matrix factorization problems)*).

First centered Olivetti faces

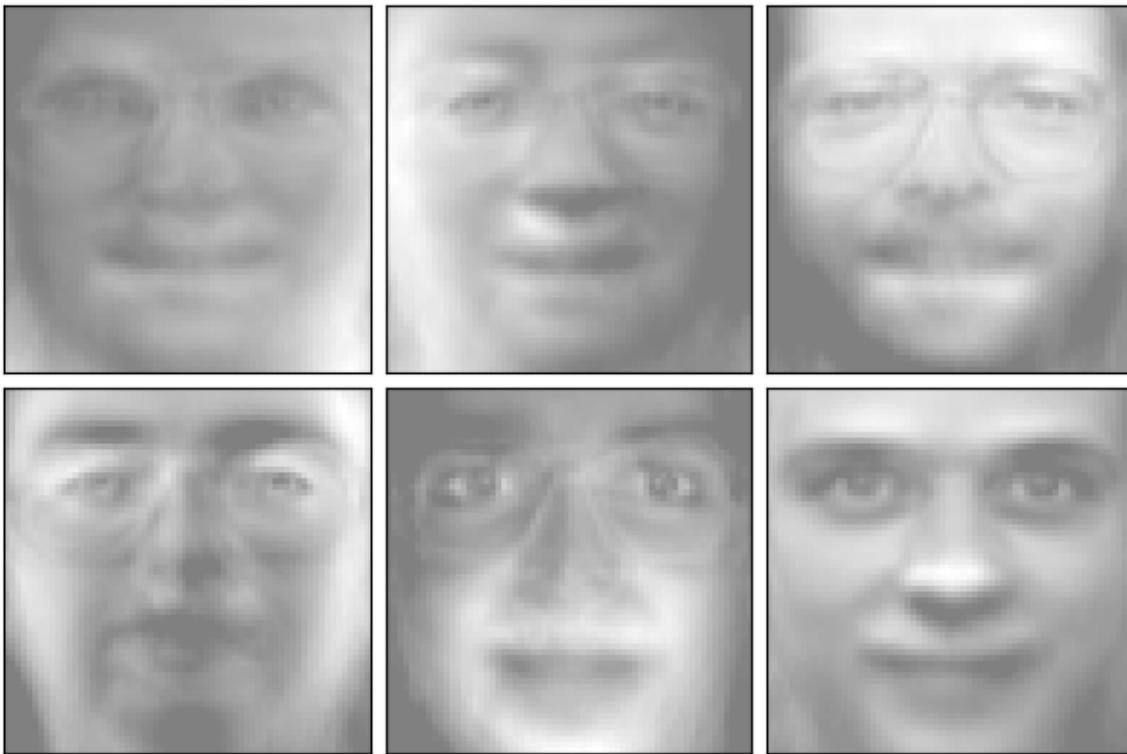


•

genfaces - PCA using randomized SVD - Train time 0.0



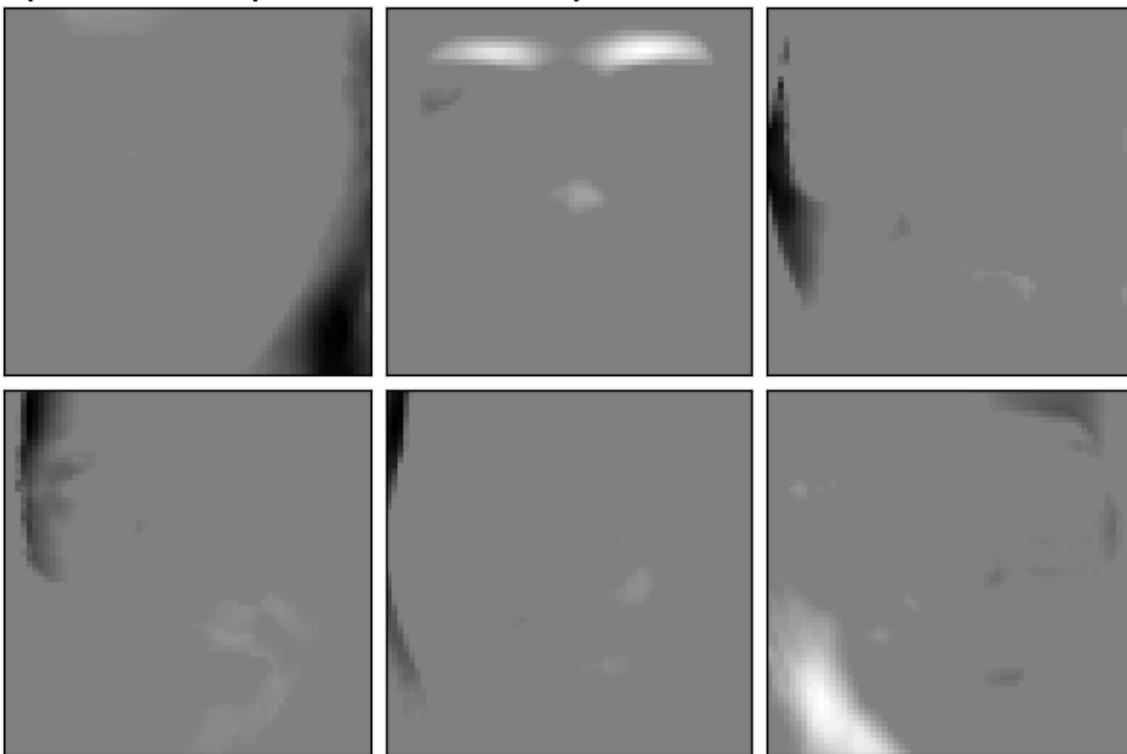
Non-negative components - NMF - Train time 0.1s



Independent components - FastICA - Train time 0.3s



Sparse comp. - MiniBatchSparsePCA - Train time 1.1s



MiniBatchDictionaryLearning - Train time 1.0s



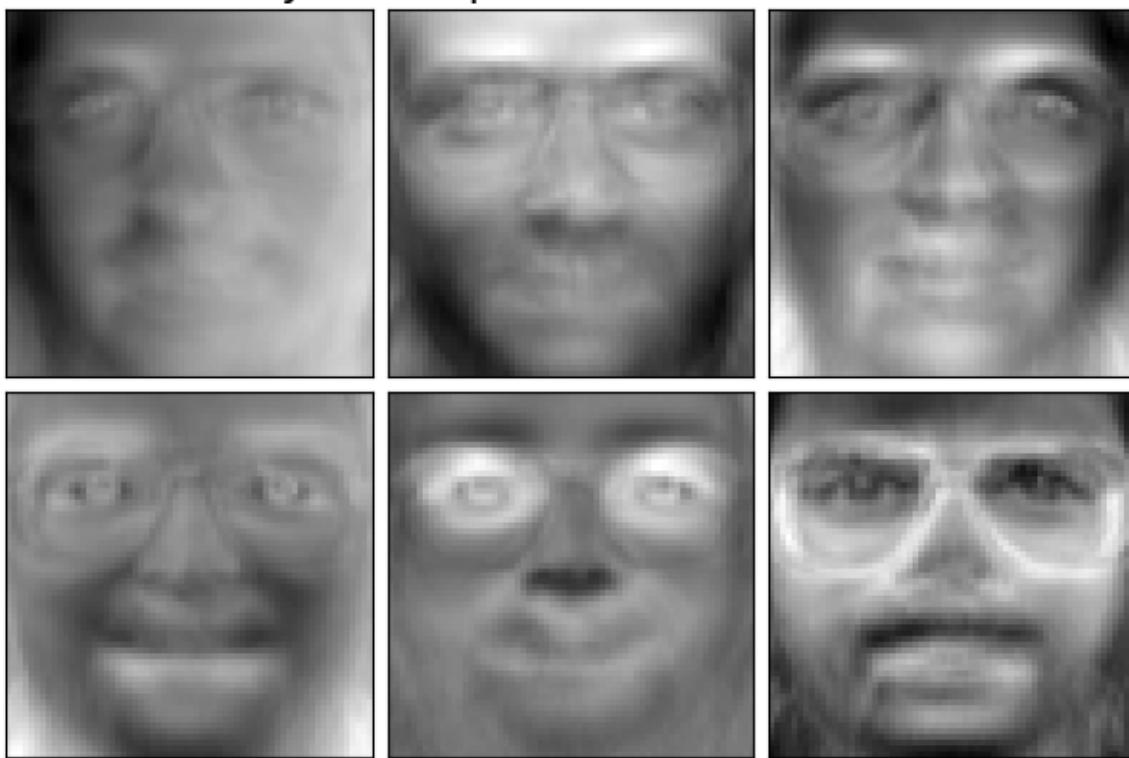
Cluster centers - MiniBatchKMeans - Train time 0.2s



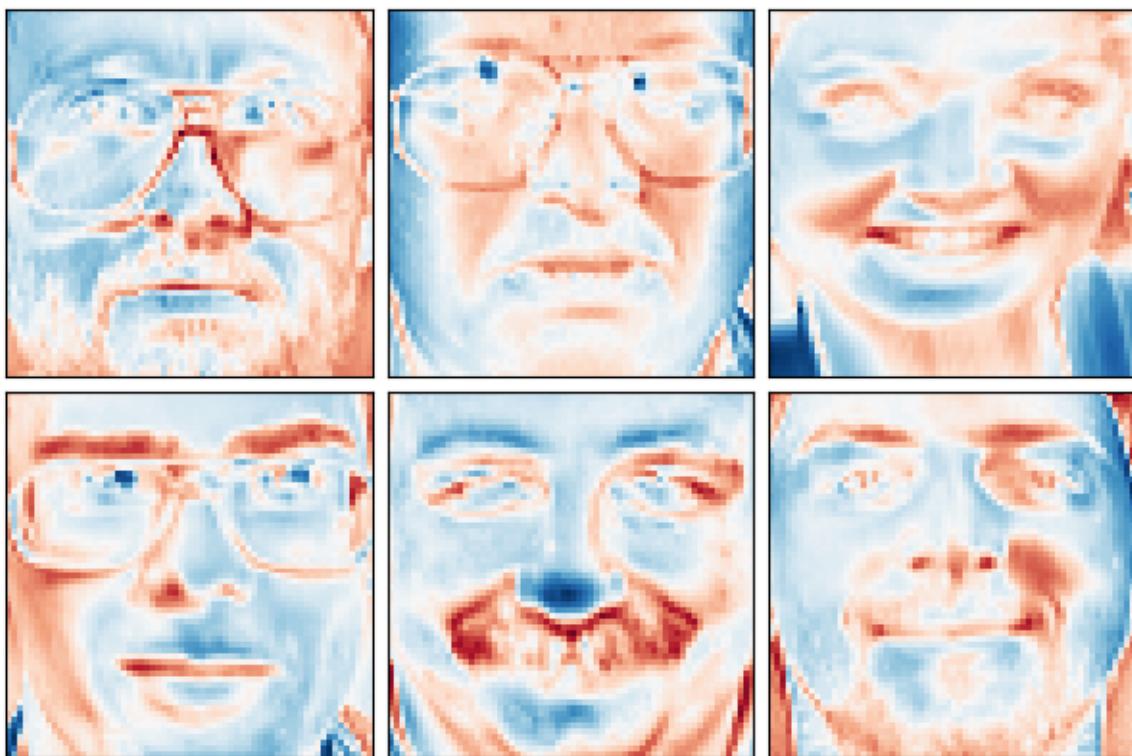
Pixelwise variance



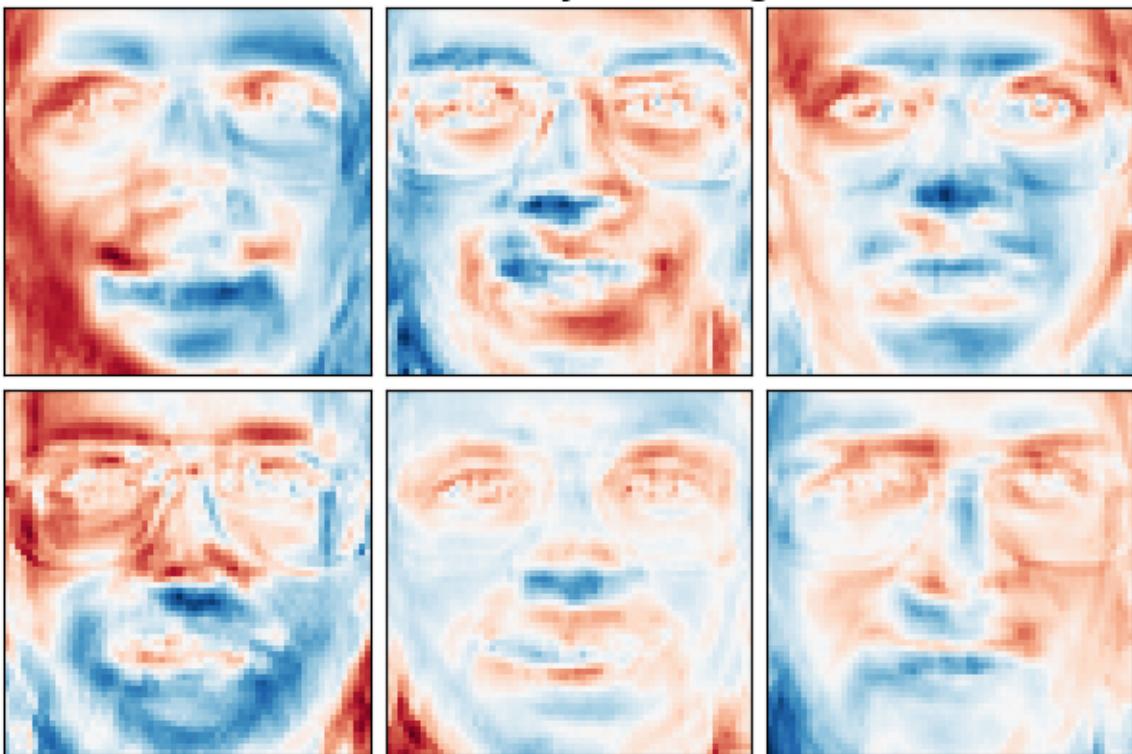
Factor Analysis components - FA - Train time 0.2s



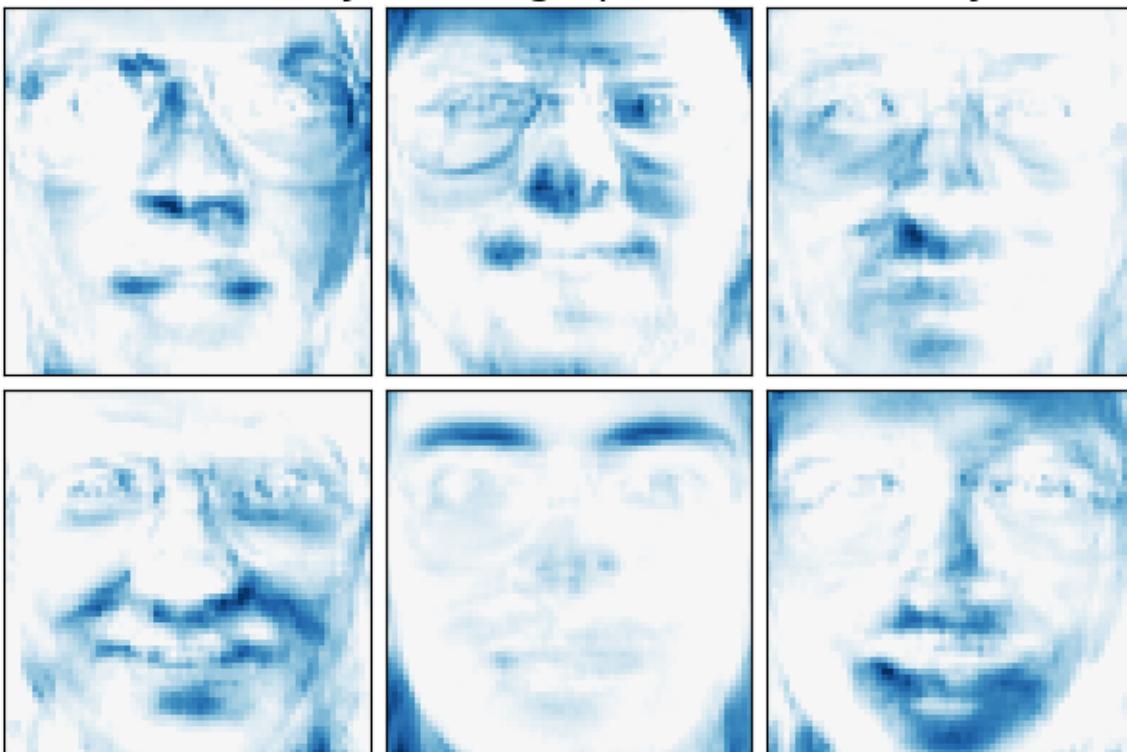
First centered Olivetti faces



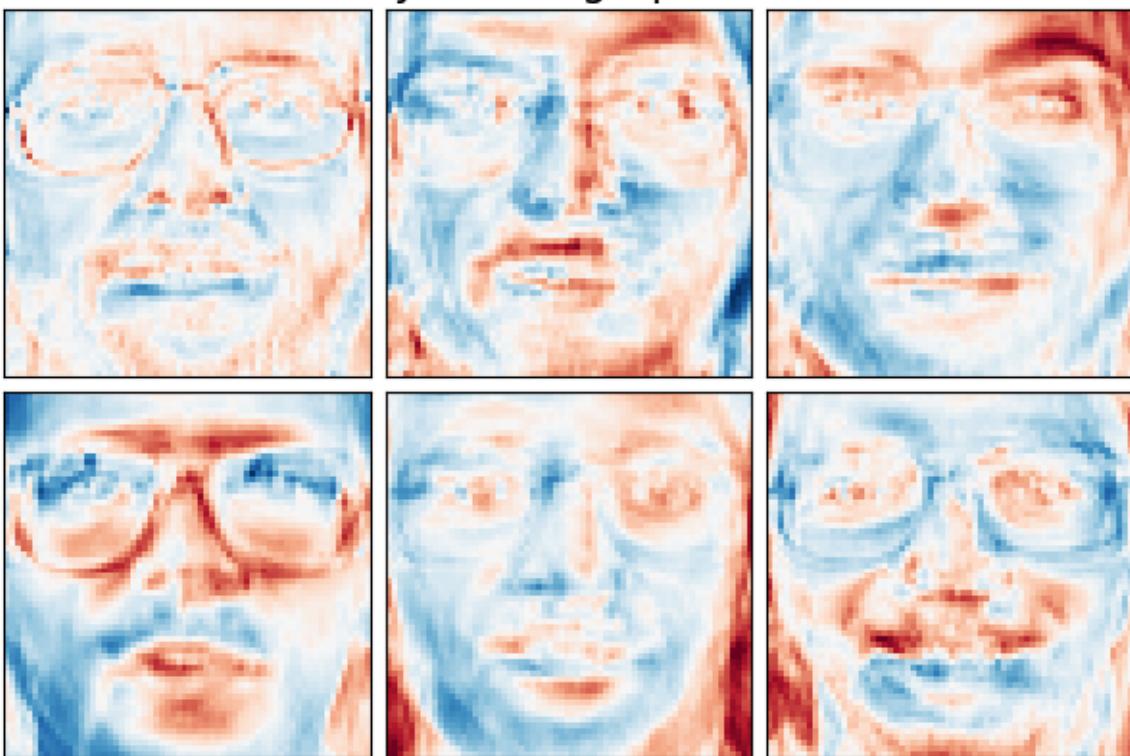
Dictionary learning



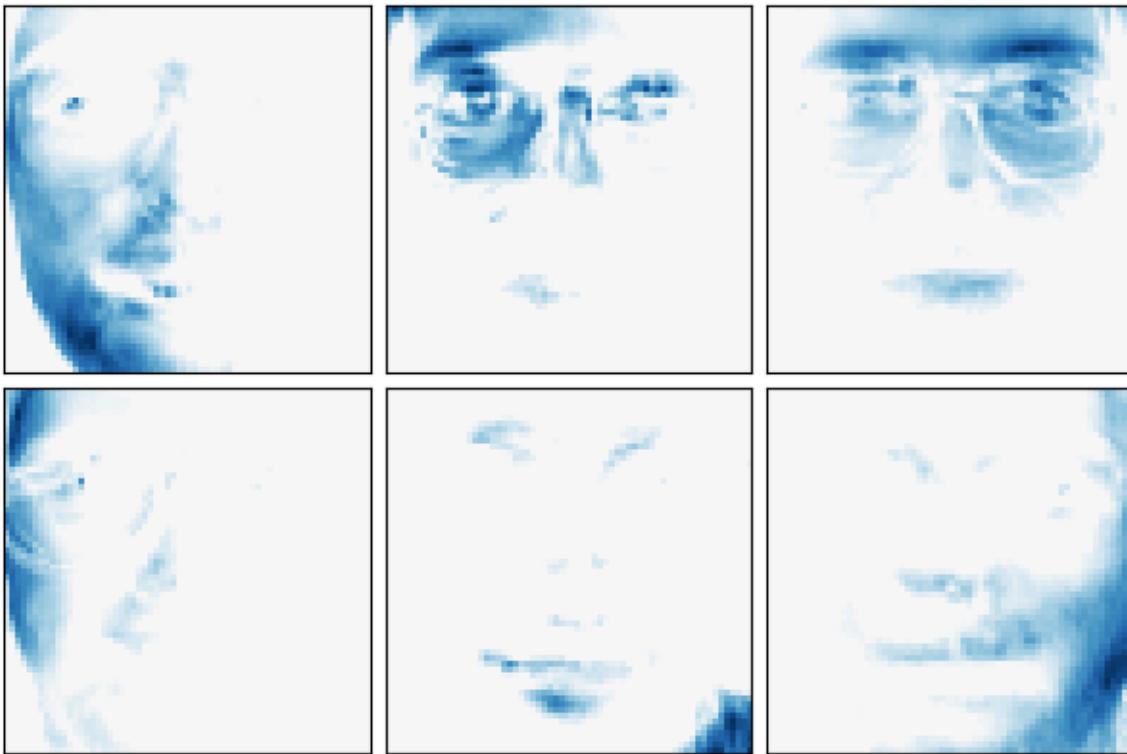
Dictionary learning - positive dictionary



Dictionary learning - positive code



Dictionary learning - positive dictionary & code



Out:

```
Dataset consists of 400 faces
Extracting the top 6 Eigenfaces - PCA using randomized SVD...
done in 0.018s
Extracting the top 6 Non-negative components - NMF...
done in 0.109s
Extracting the top 6 Independent components - FastICA...
done in 0.295s
Extracting the top 6 Sparse comp. - MiniBatchSparsePCA...
done in 1.129s
Extracting the top 6 MiniBatchDictionaryLearning...
done in 0.979s
Extracting the top 6 Cluster centers - MiniBatchKMeans...
done in 0.221s
Extracting the top 6 Factor Analysis components - FA...
done in 0.206s
Extracting the top 6 Dictionary learning...
done in 1.099s
Extracting the top 6 Dictionary learning - positive dictionary...
done in 1.213s
Extracting the top 6 Dictionary learning - positive code...
done in 0.688s
Extracting the top 6 Dictionary learning - positive dictionary & code...
done in 0.470s
```

```

print(__doc__)

# Authors: Vlad Niculae, Alexandre Gramfort
# License: BSD 3 clause

import logging
from time import time

from numpy.random import RandomState
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_olivetti_faces
from sklearn.cluster import MiniBatchKMeans
from sklearn import decomposition

# Display progress logs on stdout
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)s %(message)s')
n_row, n_col = 2, 3
n_components = n_row * n_col
image_shape = (64, 64)
rng = RandomState(0)

# ######
# Load faces data
dataset = fetch_olivetti_faces(shuffle=True, random_state=rng)
faces = dataset.data

n_samples, n_features = faces.shape

# global centering
faces_centered = faces - faces.mean(axis=0)

# local centering
faces_centered -= faces_centered.mean(axis=1).reshape(n_samples, -1)

print("Dataset consists of %d faces" % n_samples)

def plot_gallery(title, images, n_col=n_col, n_row=n_row, cmap=plt.cm.gray):
    plt.figure(figsize=(2. * n_col, 2.26 * n_row))
    plt.suptitle(title, size=16)
    for i, comp in enumerate(images):
        plt.subplot(n_row, n_col, i + 1)
        vmax = max(comp.max(), -comp.min())
        plt.imshow(comp.reshape(image_shape), cmap=cmap,
                   interpolation='nearest',
                   vmin=-vmax, vmax=vmax)
        plt.xticks(())
        plt.yticks(())
    plt.subplots_adjust(0.01, 0.05, 0.99, 0.93, 0.04, 0.)

# #####
# List of the different estimators, whether to center and transpose the
# problem, and whether the transformer uses the clustering API.
estimators = [
    ('Eigenfaces - PCA using randomized SVD',
     decomposition.PCA(n_components=n_components, svd_solver='randomized',

```

```

        whiten=True),
    True),

('Non-negative components - NMF',
 decomposition.NMF(n_components=n_components, init='nndsvda', tol=5e-3),
 False),

('Independent components - FastICA',
 decomposition.FastICA(n_components=n_components, whiten=True),
 True),

('Sparse comp. - MiniBatchSparsePCA',
 decomposition.MiniBatchSparsePCA(n_components=n_components, alpha=0.8,
                                   n_iter=100, batch_size=3,
                                   random_state=rng,
                                   normalize_components=True),
 True),

('MiniBatchDictionaryLearning',
 decomposition.MiniBatchDictionaryLearning(n_components=15, alpha=0.1,
                                            n_iter=50, batch_size=3,
                                            random_state=rng),
 True),

('Cluster centers - MiniBatchKMeans',
 MiniBatchKMeans(n_clusters=n_components, tol=1e-3, batch_size=20,
                  max_iter=50, random_state=rng),
 True),

('Factor Analysis components - FA',
 decomposition.FactorAnalysis(n_components=n_components, max_iter=20),
 True),
]

# ######
# Plot a sample of the input data

plot_gallery("First centered Olivetti faces", faces_centered[:n_components])

# #####
# Do the estimation and plot it

for name, estimator, center in estimators:
    print("Extracting the top %d %s..." % (n_components, name))
    t0 = time()
    data = faces
    if center:
        data = faces_centered
    estimator.fit(data)
    train_time = (time() - t0)
    print("done in %0.3fs" % train_time)
    if hasattr(estimator, 'cluster_centers_'):
        components_ = estimator.cluster_centers_
    else:
        components_ = estimator.components_

    # Plot an image representing the pixelwise variance provided by the

```

```

# estimator e.g. its noise_variance_ attribute. The Eigenfaces estimator,
# via the PCA decomposition, also provides a scalar noise_variance_
# (the mean of pixelwise variance) that cannot be displayed as an image
# so we skip it.
if (hasattr(estimator, 'noise_variance_') and
    estimator.noise_variance_.ndim > 0): # Skip the Eigenfaces case
    plot_gallery("Pixelwise variance",
                estimator.noise_variance_.reshape(1, -1), n_col=1,
                n_row=1)
plot_gallery('%s - Train time %.1fs' % (name, train_time),
            components_[:n_components])

plt.show()

# ######
# Various positivity constraints applied to dictionary learning.
estimators = [
    ('Dictionary learning',
     decomposition.MiniBatchDictionaryLearning(n_components=15, alpha=0.1,
                                                n_iter=50, batch_size=3,
                                                random_state=rng),
     True),
    ('Dictionary learning - positive dictionary',
     decomposition.MiniBatchDictionaryLearning(n_components=15, alpha=0.1,
                                                n_iter=50, batch_size=3,
                                                random_state=rng,
                                                positive_dict=True),
     True),
    ('Dictionary learning - positive code',
     decomposition.MiniBatchDictionaryLearning(n_components=15, alpha=0.1,
                                                n_iter=50, batch_size=3,
                                                random_state=rng,
                                                positive_code=True),
     True),
    ('Dictionary learning - positive dictionary & code',
     decomposition.MiniBatchDictionaryLearning(n_components=15, alpha=0.1,
                                                n_iter=50, batch_size=3,
                                                random_state=rng,
                                                positive_dict=True,
                                                positive_code=True),
     True),
]
]

# #####
# Plot a sample of the input data

plot_gallery("First centered Olivetti faces", faces_centered[:n_components],
             cmap=plt.cm.RdBu)

# #####
# Do the estimation and plot it

for name, estimator, center in estimators:
    print("Extracting the top %d %s..." % (n_components, name))
    t0 = time()
    data = faces
    if center:

```

```
    data = faces_centered
    estimator.fit(data)
    train_time = (time() - t0)
    print("done in %0.3fs" % train_time)
    components_ = estimator.components_
    plot_gallery(name, components_[:n_components], cmap=plt.cm.RdBu)

plt.show()
```

Total running time of the script: (0 minutes 8.614 seconds)

5.12 Ensemble methods

Examples concerning the `sklearn.ensemble` module.

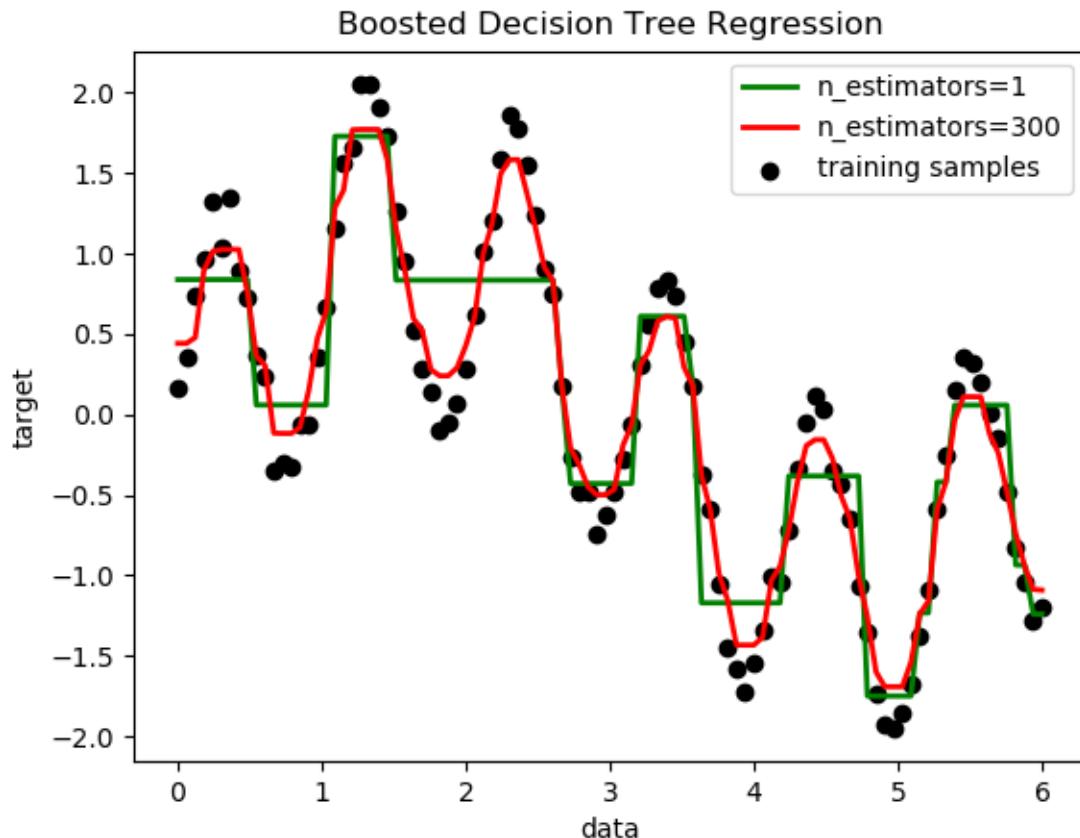
Note: Click [here](#) to download the full example code

5.12.1 Decision Tree Regression with AdaBoost

A decision tree is boosted using the AdaBoost.R2¹ algorithm on a 1D sinusoidal dataset with a small amount of Gaussian noise. 299 boosts (300 decision trees) is compared with a single decision tree regressor. As the number of boosts is increased the regressor can fit more detail.

¹

8. Drucker, “Improving Regressors using Boosting Techniques”, 1997.



```

print(__doc__)

# Author: Noel Dawe <noel.dawe@gmail.com>
#
# License: BSD 3 clause

# importing necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import AdaBoostRegressor

# Create the dataset
rng = np.random.RandomState(1)
X = np.linspace(0, 6, 100)[:, np.newaxis]
y = np.sin(X).ravel() + np.sin(6 * X).ravel() + rng.normal(0, 0.1, X.shape[0])

# Fit regression model
regr_1 = DecisionTreeRegressor(max_depth=4)

regr_2 = AdaBoostRegressor(DecisionTreeRegressor(max_depth=4),
                          n_estimators=300, random_state=rng)

regr_1.fit(X, y)
regr_2.fit(X, y)

```

```
# Predict
y_1 = regr_1.predict(X)
y_2 = regr_2.predict(X)

# Plot the results
plt.figure()
plt.scatter(X, y, c="k", label="training samples")
plt.plot(X, y_1, c="g", label="n_estimators=1", linewidth=2)
plt.plot(X, y_2, c="r", label="n_estimators=300", linewidth=2)
plt.xlabel("data")
plt.ylabel("target")
plt.title("Boosted Decision Tree Regression")
plt.legend()
plt.show()
```

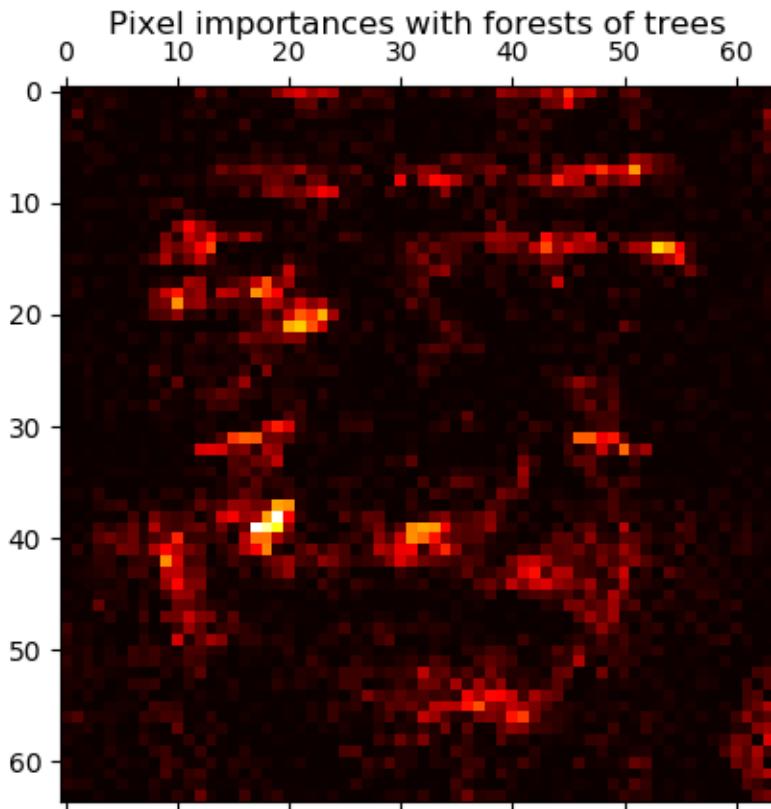
Total running time of the script: (0 minutes 0.226 seconds)

Note: Click [here](#) to download the full example code

5.12.2 Pixel importances with a parallel forest of trees

This example shows the use of forests of trees to evaluate the importance of the pixels in an image classification task (faces). The hotter the pixel, the more important.

The code below also illustrates how the construction and the computation of the predictions can be parallelized within multiple jobs.



Out:

```
Fitting ExtraTreesClassifier on faces data with 1 cores...
done in 1.028s
```

```
print(__doc__)

from time import time
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_olivetti_faces
from sklearn.ensemble import ExtraTreesClassifier

# Number of cores to use to perform parallel fitting of the forest model
n_jobs = 1

# Load the faces dataset
data = fetch_olivetti_faces()
X = data.images.reshape((len(data.images), -1))
y = data.target
```

```
mask = y < 5 # Limit to 5 classes
X = X[mask]
y = y[mask]

# Build a forest and compute the pixel importances
print("Fitting ExtraTreesClassifier on faces data with %d cores..." % n_jobs)
t0 = time()
forest = ExtraTreesClassifier(n_estimators=1000,
                              max_features=128,
                              n_jobs=n_jobs,
                              random_state=0)

forest.fit(X, y)
print("done in %0.3fs" % (time() - t0))
importances = forest.feature_importances_
importances = importances.reshape(data.images[0].shape)

# Plot pixel importances
plt.matshow(importances, cmap=plt.cm.hot)
plt.title("Pixel importances with forests of trees")
plt.show()
```

Total running time of the script: (0 minutes 1.147 seconds)

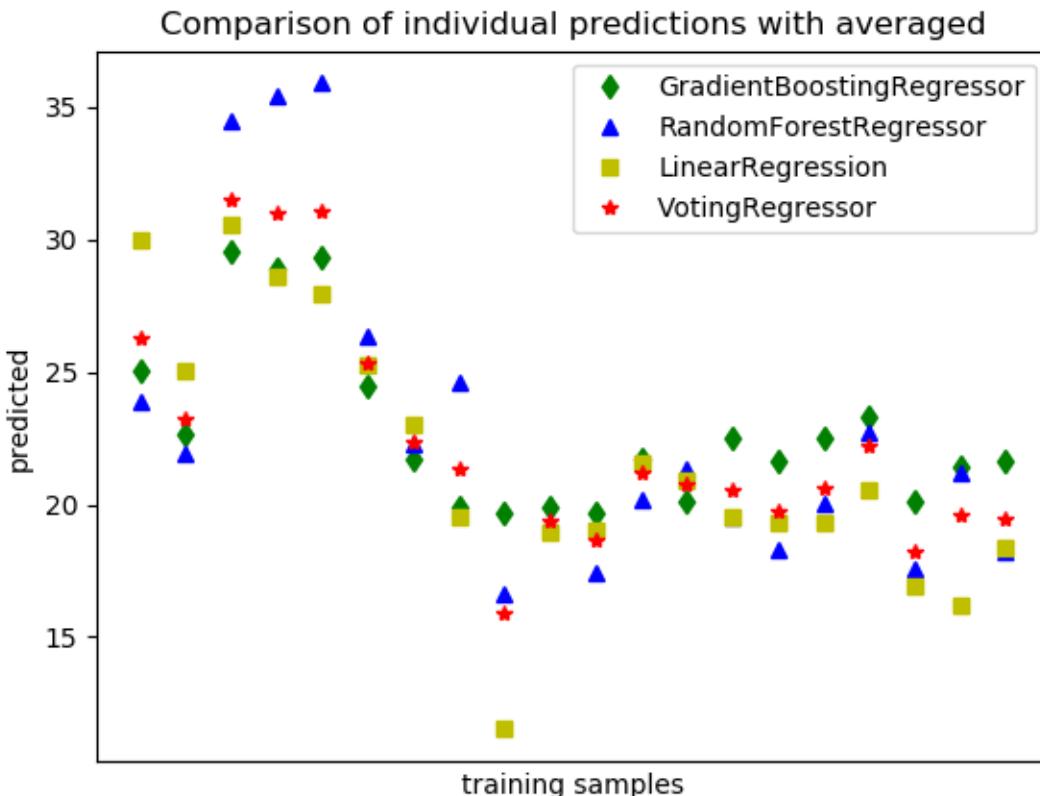
Note: Click [here](#) to download the full example code

5.12.3 Plot individual and voting regression predictions

Plot individual and averaged regression predictions for Boston dataset.

First, three exemplary regressors are initialized (`GradientBoostingRegressor`, `RandomForestRegressor`, and `LinearRegression`) and used to initialize a `VotingRegressor`.

The red starred dots are the averaged predictions.



```

print(__doc__)

import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import VotingRegressor

# Loading some example data
boston = datasets.load_boston()
X = boston.data
y = boston.target

# Training classifiers
reg1 = GradientBoostingRegressor(random_state=1, n_estimators=10)
reg2 = RandomForestRegressor(random_state=1, n_estimators=10)
reg3 = LinearRegression()
ereg = VotingRegressor([('gb', reg1), ('rf', reg2), ('lr', reg3)])
reg1.fit(X, y)
reg2.fit(X, y)
reg3.fit(X, y)
ereg.fit(X, y)

xt = X[:20]

```

```
plt.figure()
plt.plot(reg1.predict(xt), 'gd', label='GradientBoostingRegressor')
plt.plot(reg2.predict(xt), 'b^', label='RandomForestRegressor')
plt.plot(reg3.predict(xt), 'ys', label='LinearRegression')
plt.plot(ereg.predict(xt), 'r*', label='VotingRegressor')
plt.tick_params(axis='x', which='both', bottom=False, top=False,
                labelbottom=False)
plt.ylabel('predicted')
plt.xlabel('training samples')
plt.legend(loc="best")
plt.title('Comparison of individual predictions with averaged')
plt.show()
```

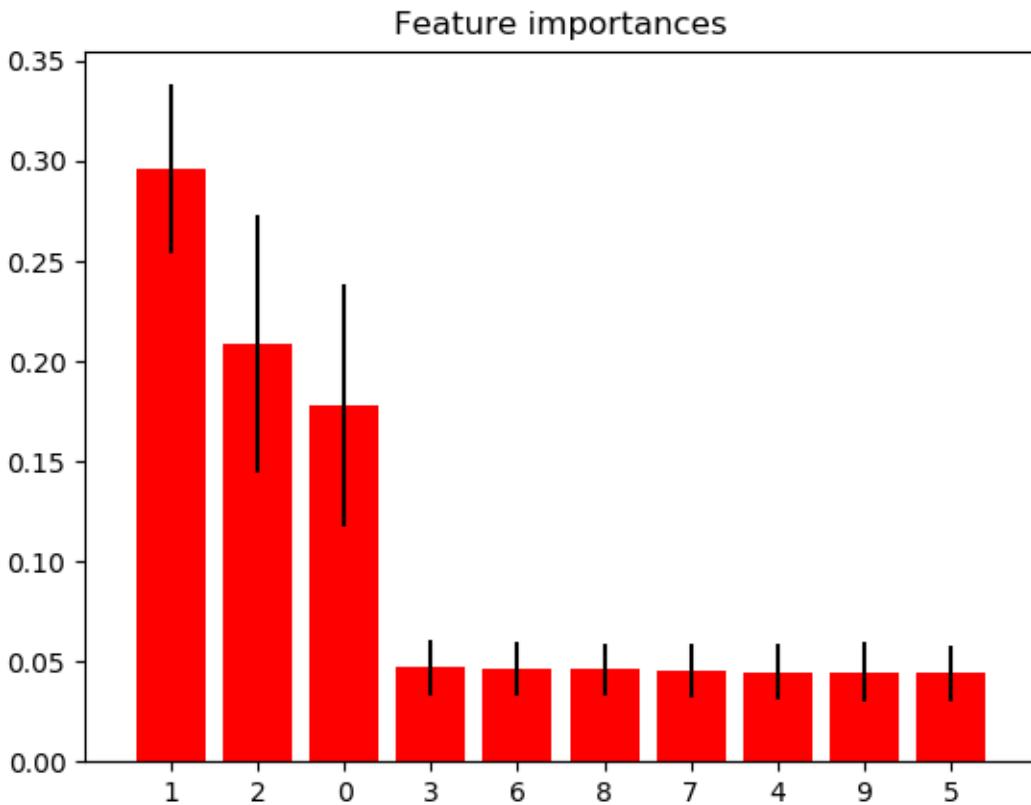
Total running time of the script: (0 minutes 0.117 seconds)

Note: Click [here](#) to download the full example code

5.12.4 Feature importances with forests of trees

This examples shows the use of forests of trees to evaluate the importance of features on an artificial classification task. The red bars are the feature importances of the forest, along with their inter-trees variability.

As expected, the plot suggests that 3 features are informative, while the remaining are not.



Out:

```
Feature ranking:  
1. feature 1 (0.295902)  
2. feature 2 (0.208351)  
3. feature 0 (0.177632)  
4. feature 3 (0.047121)  
5. feature 6 (0.046303)  
6. feature 8 (0.046013)  
7. feature 7 (0.045575)  
8. feature 4 (0.044614)  
9. feature 9 (0.044577)  
10. feature 5 (0.043912)
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_classification
from sklearn.ensemble import ExtraTreesClassifier
```

```
# Build a classification task using 3 informative features
X, y = make_classification(n_samples=1000,
                           n_features=10,
                           n_informative=3,
                           n_redundant=0,
                           n_repeated=0,
                           n_classes=2,
                           random_state=0,
                           shuffle=False)

# Build a forest and compute the feature importances
forest = ExtraTreesClassifier(n_estimators=250,
                               random_state=0)

forest.fit(X, y)
importances = forest.feature_importances_
std = np.std([tree.feature_importances_ for tree in forest.estimators_],
             axis=0)
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")

for f in range(X.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))

# Plot the feature importances of the forest
plt.figure()
plt.title("Feature importances")
plt.bar(range(X.shape[1]), importances[indices],
        color="r", yerr=std[indices], align="center")
plt.xticks(range(X.shape[1]), indices)
plt.xlim([-1, X.shape[1]])
plt.show()
```

Total running time of the script: (0 minutes 0.343 seconds)

Note: Click [here](#) to download the full example code

5.12.5 IsolationForest example

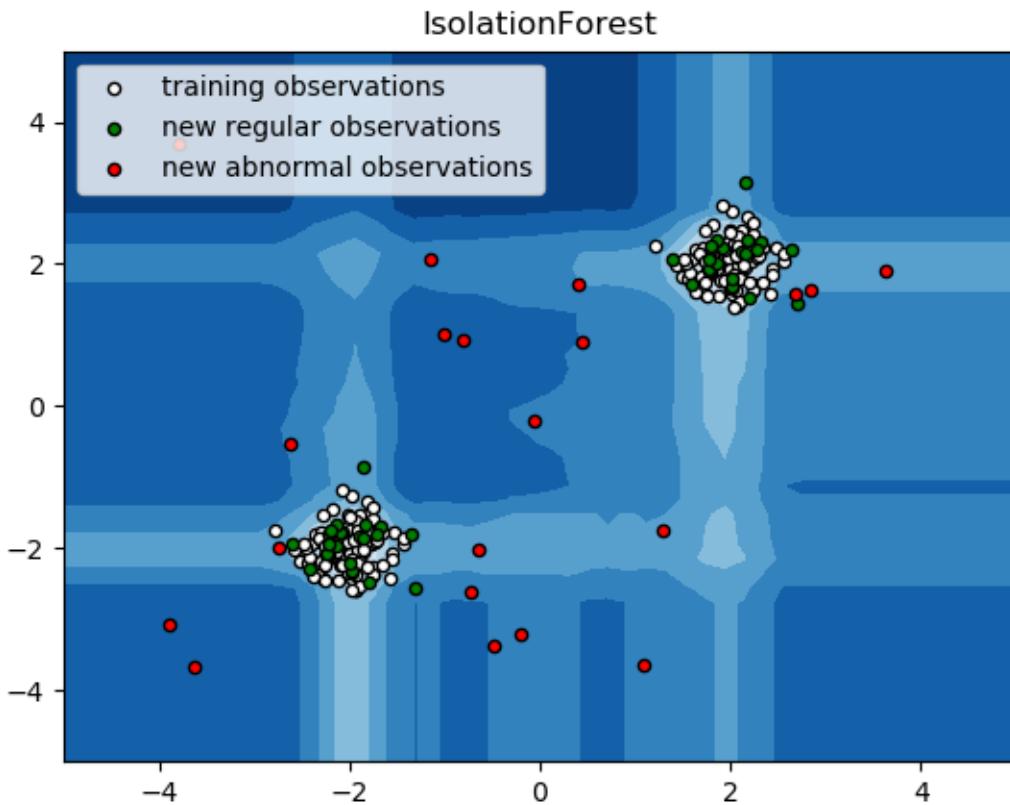
An example using `sklearn.ensemble.IsolationForest` for anomaly detection.

The IsolationForest ‘isolates’ observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature.

Since recursive partitioning can be represented by a tree structure, the number of splittings required to isolate a sample is equivalent to the path length from the root node to the terminating node.

This path length, averaged over a forest of such random trees, is a measure of normality and our decision function.

Random partitioning produces noticeable shorter paths for anomalies. Hence, when a forest of random trees collectively produce shorter path lengths for particular samples, they are highly likely to be anomalies.



```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import IsolationForest

rng = np.random.RandomState(42)

# Generate train data
X = 0.3 * rng.randn(100, 2)
X_train = np.r_[X + 2, X - 2]
# Generate some regular novel observations
X = 0.3 * rng.randn(20, 2)
X_test = np.r_[X + 2, X - 2]
# Generate some abnormal novel observations
X_outliers = rng.uniform(low=-4, high=4, size=(20, 2))

# fit the model
clf = IsolationForest(behaviour='new', max_samples=100,
                      random_state=rng, contamination='auto')
clf.fit(X_train)
y_pred_train = clf.predict(X_train)
y_pred_test = clf.predict(X_test)
y_pred_outliers = clf.predict(X_outliers)

# plot the line, the samples, and the nearest vectors to the plane

```

```
xx, yy = np.meshgrid(np.linspace(-5, 5, 50), np.linspace(-5, 5, 50))
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.title("IsolationForest")
plt.contourf(xx, yy, Z, cmap=plt.cm.Blues_r)

b1 = plt.scatter(X_train[:, 0], X_train[:, 1], c='white',
                  s=20, edgecolor='k')
b2 = plt.scatter(X_test[:, 0], X_test[:, 1], c='green',
                  s=20, edgecolor='k')
c = plt.scatter(X_outliers[:, 0], X_outliers[:, 1], c='red',
                  s=20, edgecolor='k')
plt.axis('tight')
plt.xlim((-5, 5))
plt.ylim((-5, 5))
plt.legend([b1, b2, c],
           ["training observations",
            "new regular observations", "new abnormal observations"],
           loc="upper left")
plt.show()
```

Total running time of the script: (0 minutes 0.237 seconds)

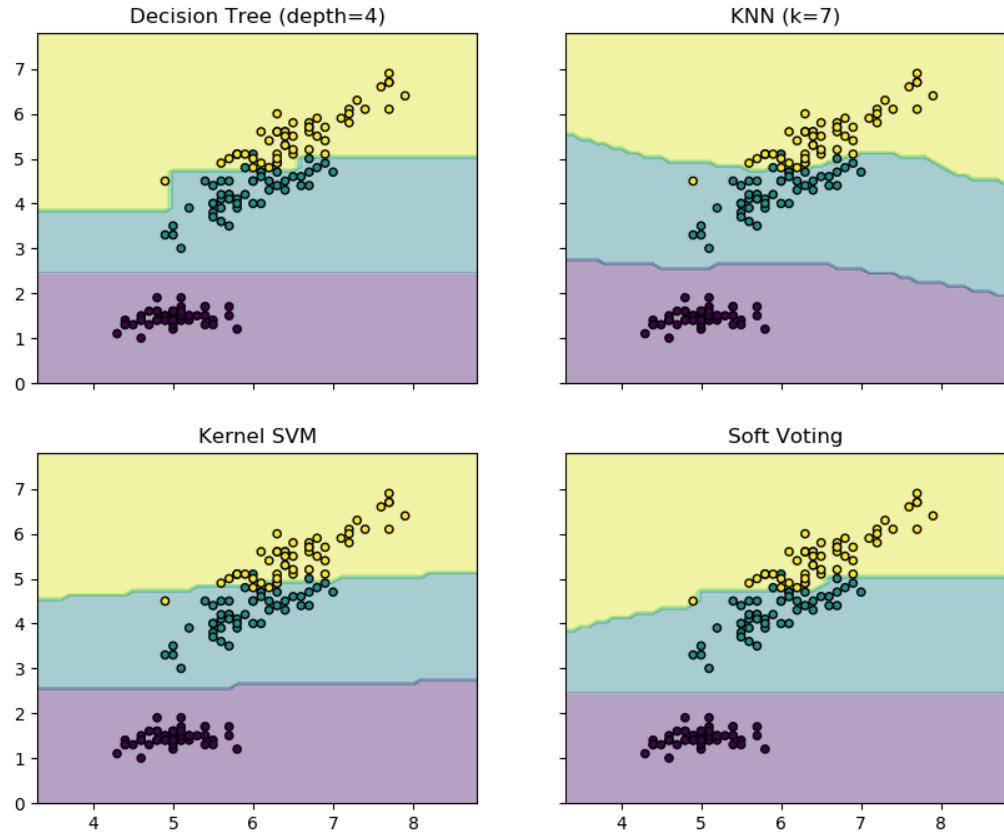
Note: Click [here](#) to download the full example code

5.12.6 Plot the decision boundaries of a VotingClassifier

Plot the decision boundaries of a `VotingClassifier` for two features of the Iris dataset.

Plot the class probabilities of the first sample in a toy dataset predicted by three different classifiers and averaged by the `VotingClassifier`.

First, three exemplary classifiers are initialized (`DecisionTreeClassifier`, `KNeighborsClassifier`, and `SVC`) and used to initialize a soft-voting `VotingClassifier` with weights [2, 1, 2], which means that the predicted probabilities of the `DecisionTreeClassifier` and `SVC` count 5 times as much as the weights of the `KNeighborsClassifier` classifier when the averaged probability is calculated.



```
print(__doc__)

from itertools import product

import numpy as np
import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.ensemble import VotingClassifier

# Loading some example data
iris = datasets.load_iris()
X = iris.data[:, [0, 2]]
y = iris.target

# Training classifiers
clf1 = DecisionTreeClassifier(max_depth=4)
clf2 = KNeighborsClassifier(n_neighbors=7)
clf3 = SVC(gamma=.1, kernel='rbf', probability=True)
clflf = VotingClassifier(estimators=[('dt', clf1), ('knn', clf2),
```

```
('svc', clf3)],
voting='soft', weights=[2, 1, 2])

clf1.fit(X, y)
clf2.fit(X, y)
clf3.fit(X, y)
eclf.fit(X, y)

# Plotting decision regions
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                      np.arange(y_min, y_max, 0.1))

f, axarr = plt.subplots(2, 2, sharex='col', sharey='row', figsize=(10, 8))

for idx, clf, tt in zip(product([0, 1], [0, 1]),
                        [clf1, clf2, clf3, eclf],
                        ['Decision Tree (depth=4)', 'KNN (k=7)',
                         'Kernel SVM', 'Soft Voting']):
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.4)
    axarr[idx[0], idx[1]].scatter(X[:, 0], X[:, 1], c=y,
                                  s=20, edgecolor='k')
    axarr[idx[0], idx[1]].set_title(tt)

plt.show()
```

Total running time of the script: (0 minutes 0.202 seconds)

Note: Click [here](#) to download the full example code

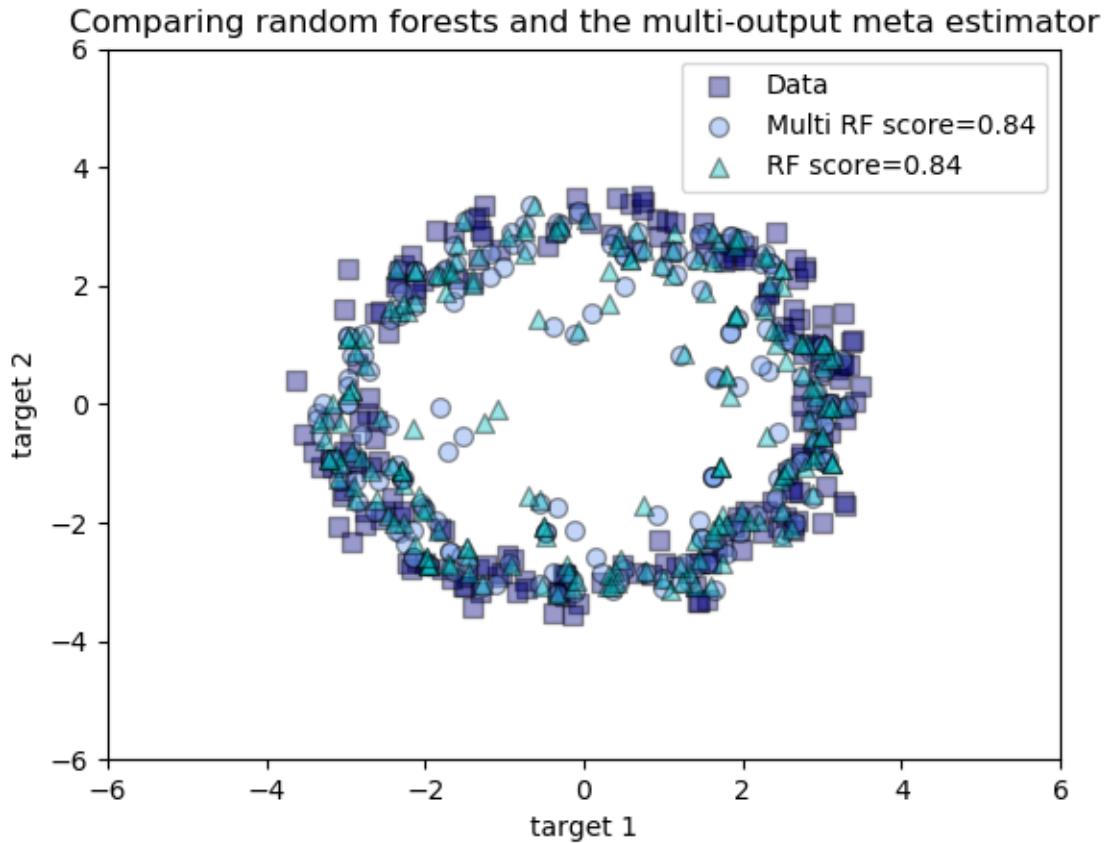
5.12.7 Comparing random forests and the multi-output meta estimator

An example to compare multi-output regression with random forest and the `multioutput.MultiOutputRegressor` meta-estimator.

This example illustrates the use of the `multioutput.MultiOutputRegressor` meta-estimator to perform multi-output regression. A random forest regressor is used, which supports multi-output regression natively, so the results can be compared.

The random forest regressor will only ever predict values within the range of observations or closer to zero for each of the targets. As a result the predictions are biased towards the centre of the circle.

Using a single underlying feature the model learns both the x and y coordinate as output.



```

print(__doc__)

# Author: Tim Head <betatim@gmail.com>
#
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.multioutput import MultiOutputRegressor

# Create a random dataset
rng = np.random.RandomState(1)
X = np.sort(200 * rng.rand(600, 1) - 100, axis=0)
y = np.array([np.pi * np.sin(X).ravel(), np.pi * np.cos(X).ravel()]).T
y += (0.5 - rng.rand(*y.shape))

X_train, X_test, y_train, y_test = train_test_split(
    X, y, train_size=400, test_size=200, random_state=4)

max_depth = 30
regr_multirf = MultiOutputRegressor(RandomForestRegressor(n_estimators=100,
                                                          max_depth=max_depth,
                                                          random_state=0))

```

```
regr_multirf.fit(X_train, y_train)

regr_rf = RandomForestRegressor(n_estimators=100, max_depth=max_depth,
                                random_state=2)
regr_rf.fit(X_train, y_train)

# Predict on new data
y_multirf = regr_multirf.predict(X_test)
y_rf = regr_rf.predict(X_test)

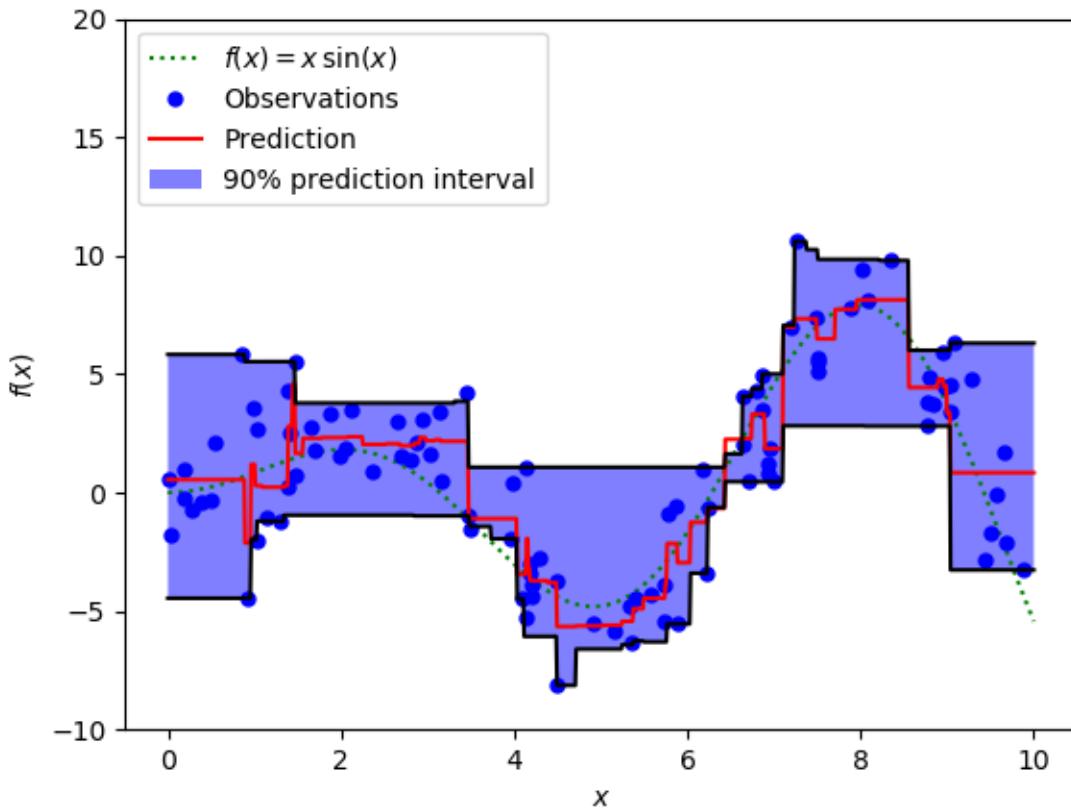
# Plot the results
plt.figure()
s = 50
a = 0.4
plt.scatter(y_test[:, 0], y_test[:, 1], edgecolor='k',
            c="navy", s=s, marker="s", alpha=a, label="Data")
plt.scatter(y_multirf[:, 0], y_multirf[:, 1], edgecolor='k',
            c="cornflowerblue", s=s, alpha=a,
            label="Multi RF score=% .2f" % regr_multirf.score(X_test, y_test))
plt.scatter(y_rf[:, 0], y_rf[:, 1], edgecolor='k',
            c="c", s=s, marker="^", alpha=a,
            label="RF score=% .2f" % regr_rf.score(X_test, y_test))
plt.xlim([-6, 6])
plt.ylim([-6, 6])
plt.xlabel("target 1")
plt.ylabel("target 2")
plt.title("Comparing random forests and the multi-output meta estimator")
plt.legend()
plt.show()
```

Total running time of the script: (0 minutes 0.299 seconds)

Note: Click [here](#) to download the full example code

5.12.8 Prediction Intervals for Gradient Boosting Regression

This example shows how quantile regression can be used to create prediction intervals.



```

import numpy as np
import matplotlib.pyplot as plt

from sklearn.ensemble import GradientBoostingRegressor

np.random.seed(1)

def f(x):
    """The function to predict."""
    return x * np.sin(x)

#-----
# First the noiseless case
X = np.atleast_2d(np.random.uniform(0, 10.0, size=100)).T
X = X.astype(np.float32)

# Observations
y = f(X).ravel()

dy = 1.5 + 1.0 * np.random.random(y.shape)
noise = np.random.normal(0, dy)
y += noise
y = y.astype(np.float32)

# Mesh the input space for evaluations of the real function, the prediction and

```

```
# its MSE
xx = np.atleast_2d(np.linspace(0, 10, 1000)).T
xx = xx.astype(np.float32)

alpha = 0.95

clf = GradientBoostingRegressor(loss='quantile', alpha=alpha,
                                 n_estimators=250, max_depth=3,
                                 learning_rate=.1, min_samples_leaf=9,
                                 min_samples_split=9)

clf.fit(X, y)

# Make the prediction on the meshed x-axis
y_upper = clf.predict(xx)

clf.set_params(alpha=1.0 - alpha)
clf.fit(X, y)

# Make the prediction on the meshed x-axis
y_lower = clf.predict(xx)

clf.set_params(loss='ls')
clf.fit(X, y)

# Make the prediction on the meshed x-axis
y_pred = clf.predict(xx)

# Plot the function, the prediction and the 90% confidence interval based on
# the MSE
fig = plt.figure()
plt.plot(xx, f(xx), 'g:', label=r'$f(x) = x\backslash\sin(x)$')
plt.plot(X, y, 'b.', markersize=10, label=u'Observations')
plt.plot(xx, y_pred, 'r-', label=u'Prediction')
plt.plot(xx, y_upper, 'k-')
plt.plot(xx, y_lower, 'k-')
plt.fill(np.concatenate([xx, xx[::-1]]),
          np.concatenate([y_upper, y_lower[::-1]]),
          alpha=.5, fc='b', ec='None', label='90% prediction interval')
plt.xlabel('$x$')
plt.ylabel('$f(x)$')
plt.ylim(-10, 20)
plt.legend(loc='upper left')
plt.show()
```

Total running time of the script: (0 minutes 0.275 seconds)

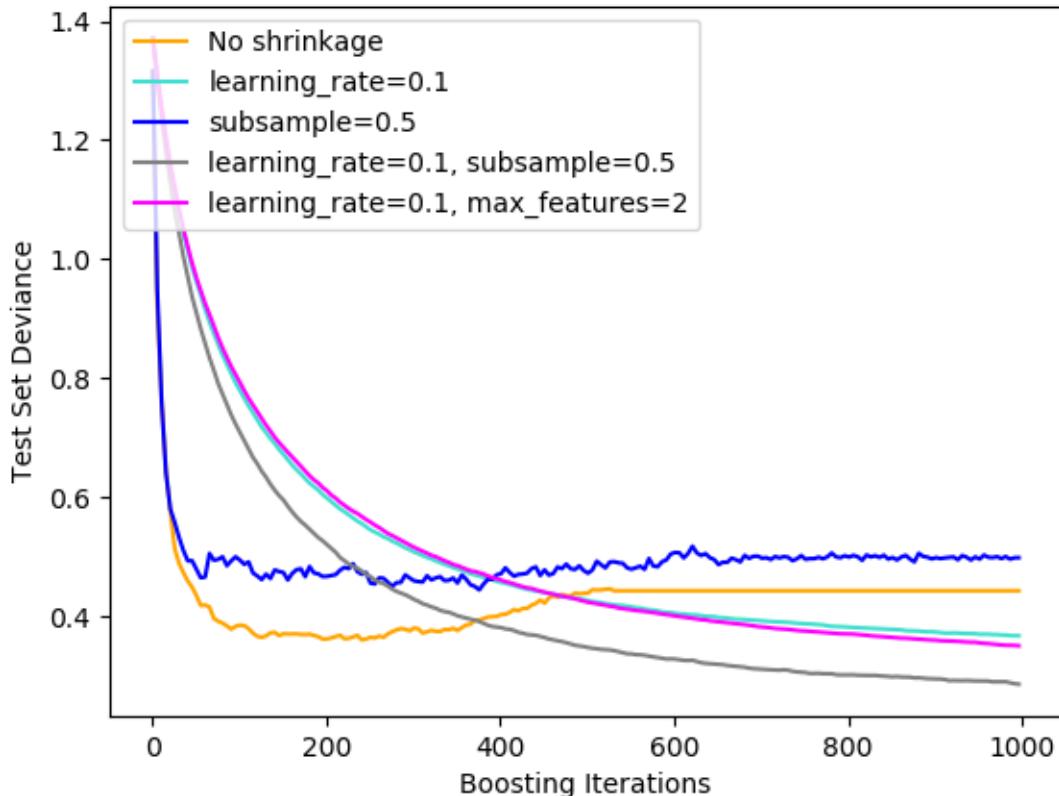
Note: Click [here](#) to download the full example code

5.12.9 Gradient Boosting regularization

Illustration of the effect of different regularization strategies for Gradient Boosting. The example is taken from Hastie et al 2009¹.

¹ T. Hastie, R. Tibshirani and J. Friedman, “Elements of Statistical Learning Ed. 2”, Springer, 2009.

The loss function used is binomial deviance. Regularization via shrinkage (`learning_rate < 1.0`) improves performance considerably. In combination with shrinkage, stochastic gradient boosting (`subsample < 1.0`) can produce more accurate models by reducing the variance via bagging. Subsampling without shrinkage usually does poorly. Another strategy to reduce the variance is by subsampling the features analogous to the random splits in Random Forests (via the `max_features` parameter).



```
print(__doc__)

# Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>
#
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn import ensemble
from sklearn import datasets

X, y = datasets.make_hastie_10_2(n_samples=12000, random_state=1)
X = X.astype(np.float32)

# map labels from {-1, 1} to {0, 1}
labels, y = np.unique(y, return_inverse=True)

X_train, X_test = X[:2000], X[2000:]
```

```
y_train, y_test = y[:2000], y[2000:]

original_params = {'n_estimators': 1000, 'max_leaf_nodes': 4, 'max_depth': None,
                   'random_state': 2,
                   'min_samples_split': 5}

plt.figure()

for label, color, setting in [('No shrinkage', 'orange',
                                {'learning_rate': 1.0, 'subsample': 1.0}),
                               ('learning_rate=0.1', 'turquoise',
                                {'learning_rate': 0.1, 'subsample': 1.0}),
                               ('subsample=0.5', 'blue',
                                {'learning_rate': 1.0, 'subsample': 0.5}),
                               ('learning_rate=0.1, subsample=0.5', 'gray',
                                {'learning_rate': 0.1, 'subsample': 0.5}),
                               ('learning_rate=0.1, max_features=2', 'magenta',
                                {'learning_rate': 0.1, 'max_features': 2})]:
    params = dict(original_params)
    params.update(setting)

    clf = ensemble.GradientBoostingClassifier(**params)
    clf.fit(X_train, y_train)

    # compute test set deviance
    test_deviance = np.zeros((params['n_estimators']),), dtype=np.float64)

    for i, y_pred in enumerate(clf.staged_decision_function(X_test)):
        # clf.loss_ assumes that y_test[i] in {0, 1}
        test_deviance[i] = clf.loss_(y_test, y_pred)

    plt.plot((np.arange(test_deviance.shape[0]) + 1)[::5], test_deviance[::5],
              '-', color=color, label=label)

plt.legend(loc='upper left')
plt.xlabel('Boosting Iterations')
plt.ylabel('Test Set Deviance')

plt.show()
```

Total running time of the script: (0 minutes 10.180 seconds)

Note: Click [here](#) to download the full example code

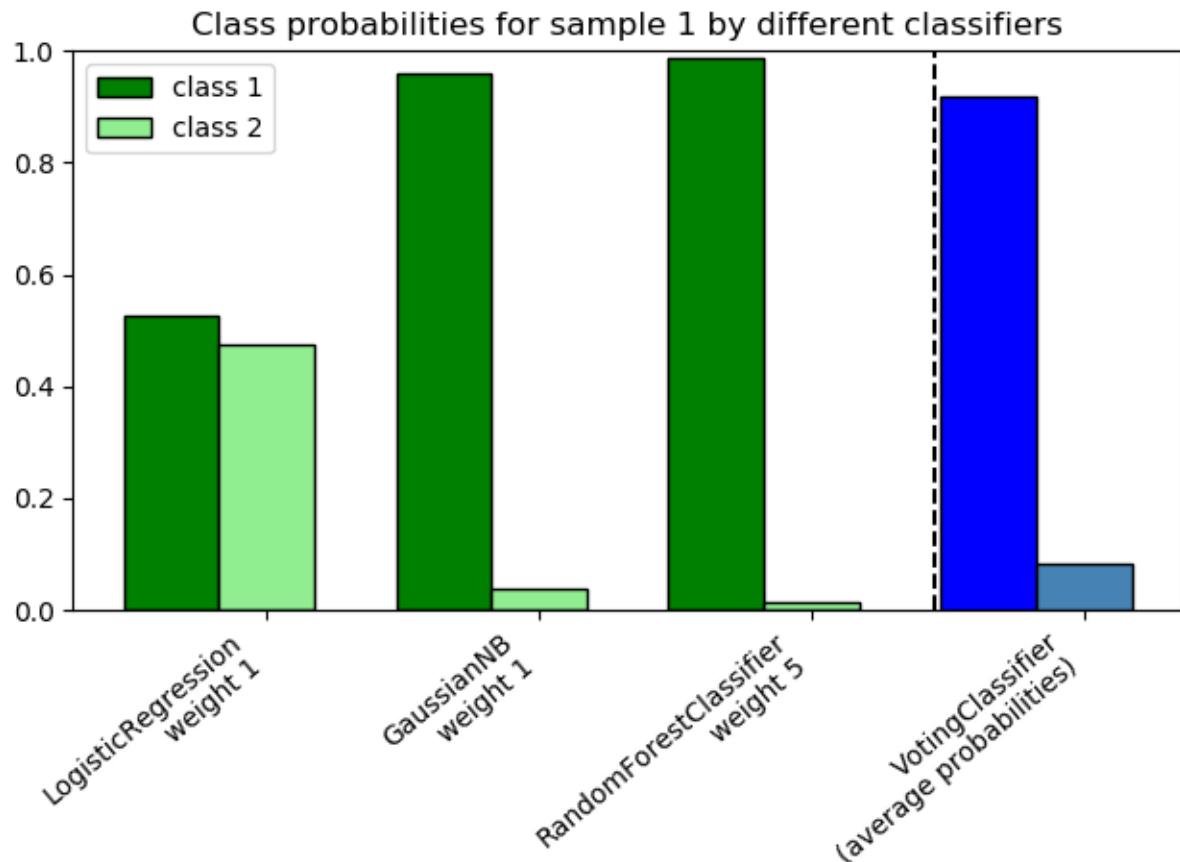
5.12.10 Plot class probabilities calculated by the VotingClassifier

Plot the class probabilities of the first sample in a toy dataset predicted by three different classifiers and averaged by the `VotingClassifier`.

First, three exemplary classifiers are initialized (`LogisticRegression`, `GaussianNB`, and `RandomForestClassifier`) and used to initialize a soft-voting `VotingClassifier` with weights [1, 1, 5], which means that the predicted probabilities of the `RandomForestClassifier` count 5 times as much as the weights of the other classifiers when the averaged probability is calculated.

To visualize the probability weighting, we fit each classifier on the training set and plot the predicted class probabilities

for the first sample in this example dataset.



```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier

clf1 = LogisticRegression(solver='lbfgs', max_iter=1000, random_state=123)
clf2 = RandomForestClassifier(n_estimators=100, random_state=123)
clf3 = GaussianNB()
X = np.array([[-1.0, -1.0], [-1.2, -1.4], [-3.4, -2.2], [1.1, 1.2]])
y = np.array([1, 1, 2, 2])

eclf = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)],
                       voting='soft',
                       weights=[1, 1, 5])

# predict class probabilities for all classifiers
probas = [c.fit(X, y).predict_proba(X) for c in (clf1, clf2, clf3, eclf)]

# get class probabilities for the first sample in the dataset
```

```
class1_1 = [pr[0, 0] for pr in probas]
class2_1 = [pr[0, 1] for pr in probas]

# plotting

N = 4 # number of groups
ind = np.arange(N) # group positions
width = 0.35 # bar width

fig, ax = plt.subplots()

# bars for classifier 1-3
p1 = ax.bar(ind, np.hstack(([class1_1[:-1], [0]])), width,
             color='green', edgecolor='k')
p2 = ax.bar(ind + width, np.hstack(([class2_1[:-1], [0]])), width,
             color='lightgreen', edgecolor='k')

# bars for VotingClassifier
p3 = ax.bar(ind, [0, 0, 0, class1_1[-1]], width,
             color='blue', edgecolor='k')
p4 = ax.bar(ind + width, [0, 0, 0, class2_1[-1]], width,
             color='steelblue', edgecolor='k')

# plot annotations
plt.axvline(2.8, color='k', linestyle='dashed')
ax.set_xticks(ind + width)
ax.set_xticklabels(['LogisticRegression\\nweight 1',
                   'GaussianNB\\nweight 1',
                   'RandomForestClassifier\\nweight 5',
                   'VotingClassifier\\n(average probabilities)'],
                   rotation=40,
                   ha='right')
plt.ylim([0, 1])
plt.title('Class probabilities for sample 1 by different classifiers')
plt.legend([p1[0], p2[0]], ['class 1', 'class 2'], loc='upper left')
plt.tight_layout()
plt.show()
```

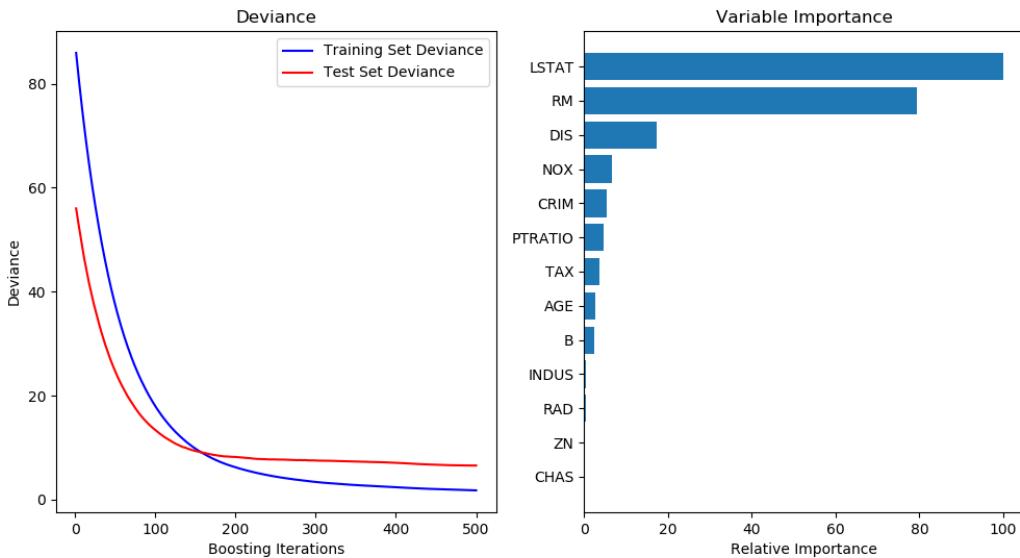
Total running time of the script: (0 minutes 0.245 seconds)

Note: Click [here](#) to download the full example code

5.12.11 Gradient Boosting regression

Demonstrate Gradient Boosting on the Boston housing dataset.

This example fits a Gradient Boosting model with least squares loss and 500 regression trees of depth 4.



Out:

```
MSE: 6.5493
```

```
print(__doc__)

# Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>
#
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn import ensemble
from sklearn import datasets
from sklearn.utils import shuffle
from sklearn.metrics import mean_squared_error

# ##########
# Load data
boston = datasets.load_boston()
X, y = shuffle(boston.data, boston.target, random_state=13)
X = X.astype(np.float32)
offset = int(X.shape[0] * 0.9)
X_train, y_train = X[:offset], y[:offset]
X_test, y_test = X[offset:], y[offset:]

# ##########
# Fit regression model
params = {'n_estimators': 500, 'max_depth': 4, 'min_samples_split': 2,
          'learning_rate': 0.01, 'loss': 'ls'}
```

```
clf = ensemble.GradientBoostingRegressor(**params)

clf.fit(X_train, y_train)
mse = mean_squared_error(y_test, clf.predict(X_test))
print("MSE: %.4f" % mse)

# ##### Plot training deviance
# Compute test set deviance
test_score = np.zeros((params['n_estimators'],), dtype=np.float64)

for i, y_pred in enumerate(clf.staged_predict(X_test)):
    test_score[i] = clf.loss_(y_test, y_pred)

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title('Deviance')
plt.plot(np.arange(params['n_estimators']) + 1, clf.train_score_, 'b-',
         label='Training Set Deviance')
plt.plot(np.arange(params['n_estimators']) + 1, test_score, 'r-',
         label='Test Set Deviance')
plt.legend(loc='upper right')
plt.xlabel('Boosting Iterations')
plt.ylabel('Deviance')

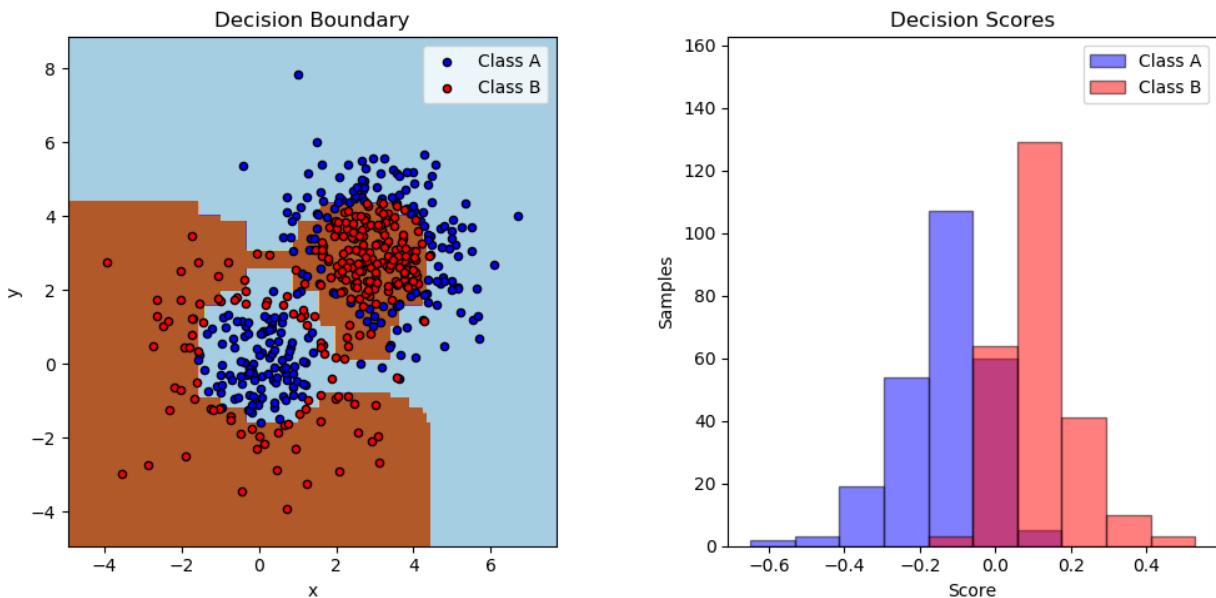
# ##### Plot feature importance
feature_importance = clf.feature_importances_
# make importances relative to max importance
feature_importance = 100.0 * (feature_importance / feature_importance.max())
sorted_idx = np.argsort(feature_importance)
pos = np.arange(sorted_idx.shape[0]) + .5
plt.subplot(1, 2, 2)
plt.barh(pos, feature_importance[sorted_idx], align='center')
plt.yticks(pos, boston.feature_names[sorted_idx])
plt.xlabel('Relative Importance')
plt.title('Variable Importance')
plt.show()
```

Total running time of the script: (0 minutes 0.416 seconds)

Note: Click [here](#) to download the full example code

5.12.12 Two-class AdaBoost

This example fits an AdaBoosted decision stump on a non-linearly separable classification dataset composed of two “Gaussian quantiles” clusters (see `sklearn.datasets.make_gaussian_quantiles`) and plots the decision boundary and decision scores. The distributions of decision scores are shown separately for samples of class A and B. The predicted class label for each sample is determined by the sign of the decision score. Samples with decision scores greater than zero are classified as B, and are otherwise classified as A. The magnitude of a decision score determines the degree of likeness with the predicted class label. Additionally, a new dataset could be constructed containing a desired purity of class B, for example, by only selecting samples with a decision score above some value.



```

print(__doc__)

# Author: Noel Dawe <noel.dawe@gmail.com>
#
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_gaussian_quantiles

# Construct dataset
X1, y1 = make_gaussian_quantiles(cov=2.,
                                  n_samples=200, n_features=2,
                                  n_classes=2, random_state=1)
X2, y2 = make_gaussian_quantiles(mean=(3, 3), cov=1.5,
                                  n_samples=300, n_features=2,
                                  n_classes=2, random_state=1)
X = np.concatenate((X1, X2))
y = np.concatenate((y1, -y2 + 1))

# Create and fit an AdaBoosted decision tree
bdt = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1),
                        algorithm="SAMME",
                        n_estimators=200)

bdt.fit(X, y)

plot_colors = "br"
plot_step = 0.02
class_names = "AB"

plt.figure(figsize=(10, 5))

```

```
# Plot the decision boundaries
plt.subplot(121)
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                      np.arange(y_min, y_max, plot_step))

Z = bdt.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
cs = plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
plt.axis("tight")

# Plot the training points
for i, n, c in zip(range(2), class_names, plot_colors):
    idx = np.where(y == i)
    plt.scatter(X[idx, 0], X[idx, 1],
                c=c, cmap=plt.cm.Paired,
                s=20, edgecolor='k',
                label="Class %s" % n)
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.legend(loc='upper right')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Decision Boundary')

# Plot the two-class decision scores
twoclass_output = bdt.decision_function(X)
plot_range = (twoclass_output.min(), twoclass_output.max())
plt.subplot(122)
for i, n, c in zip(range(2), class_names, plot_colors):
    plt.hist(twoclass_output[y == i],
             bins=10,
             range=plot_range,
             facecolor=c,
             label='Class %s' % n,
             alpha=.5,
             edgecolor='k')
x1, x2, y1, y2 = plt.axis()
plt.axis((x1, x2, y1, y2 * 1.2))
plt.legend(loc='upper right')
plt.ylabel('Samples')
plt.xlabel('Score')
plt.title('Decision Scores')

plt.tight_layout()
plt.subplots_adjust(wspace=0.35)
plt.show()
```

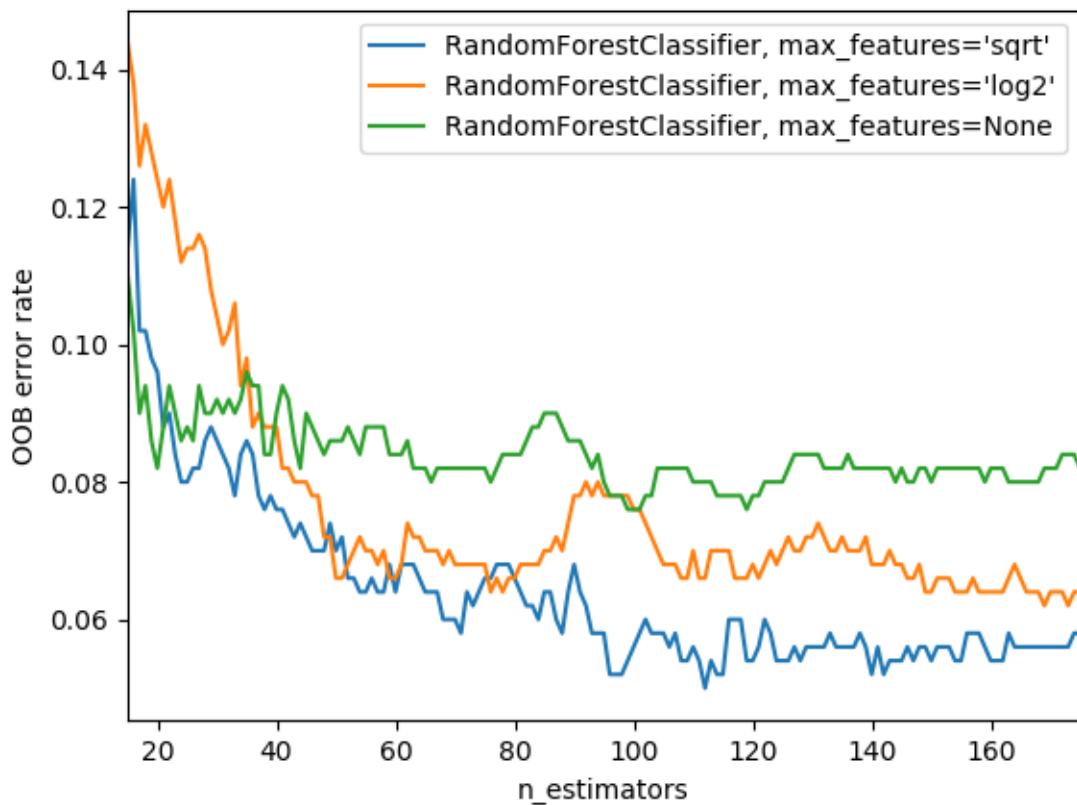
Total running time of the script: (0 minutes 2.255 seconds)

Note: Click [here](#) to download the full example code

5.12.13 OOB Errors for Random Forests

The `RandomForestClassifier` is trained using *bootstrap aggregation*, where each new tree is fit from a bootstrap sample of the training observations $z_i = (x_i, y_i)$. The *out-of-bag* (OOB) error is the average error for each z_i calculated using predictions from the trees that do not contain z_i in their respective bootstrap sample. This allows the `RandomForestClassifier` to be fit and validated whilst being trained¹.

The example below demonstrates how the OOB error can be measured at the addition of each new tree during training. The resulting plot allows a practitioner to approximate a suitable value of `n_estimators` at which the error stabilizes.



```
import matplotlib.pyplot as plt

from collections import OrderedDict
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier

# Author: Kian Ho <hui.kian.ho@gmail.com>
#         Gilles Louppe <g.louppe@gmail.com>
#         Andreas Mueller <amueller@ais.uni-bonn.de>
#
# License: BSD 3 Clause

print(__doc__)
```

¹ T. Hastie, R. Tibshirani and J. Friedman, “Elements of Statistical Learning Ed. 2”, p592-593, Springer, 2009.

```
RANDOM_STATE = 123

# Generate a binary classification dataset.
X, y = make_classification(n_samples=500, n_features=25,
                           n_clusters_per_class=1, n_informative=15,
                           random_state=RANDOM_STATE)

# NOTE: Setting the `warm_start` construction parameter to `True` disables
# support for parallelized ensembles but is necessary for tracking the OOB
# error trajectory during training.
ensemble_clfs = [
    ("RandomForestClassifier, max_features='sqrt'",
     RandomForestClassifier(n_estimators=100,
                           warm_start=True, oob_score=True,
                           max_features="sqrt",
                           random_state=RANDOM_STATE)),
    ("RandomForestClassifier, max_features='log2'",
     RandomForestClassifier(n_estimators=100,
                           warm_start=True, max_features='log2',
                           oob_score=True,
                           random_state=RANDOM_STATE)),
    ("RandomForestClassifier, max_features=None",
     RandomForestClassifier(n_estimators=100,
                           warm_start=True, max_features=None,
                           oob_score=True,
                           random_state=RANDOM_STATE))
]

# Map a classifier name to a list of (<n_estimators>, <error rate>) pairs.
error_rate = OrderedDict((label, []) for label, _ in ensemble_clfs)

# Range of `n_estimators` values to explore.
min_estimators = 15
max_estimators = 175

for label, clf in ensemble_clfs:
    for i in range(min_estimators, max_estimators + 1):
        clf.set_params(n_estimators=i)
        clf.fit(X, y)

        # Record the OOB error for each `n_estimators=i` setting.
        oob_error = 1 - clf.oob_score_
        error_rate[label].append((i, oob_error))

# Generate the "OOB error rate" vs. "n_estimators" plot.
for label, clf_err in error_rate.items():
    xs, ys = zip(*clf_err)
    plt.plot(xs, ys, label=label)

plt.xlim(min_estimators, max_estimators)
plt.xlabel("n_estimators")
plt.ylabel("OOB error rate")
plt.legend(loc="upper right")
plt.show()
```

Total running time of the script: (0 minutes 5.517 seconds)

Note: Click [here](#) to download the full example code

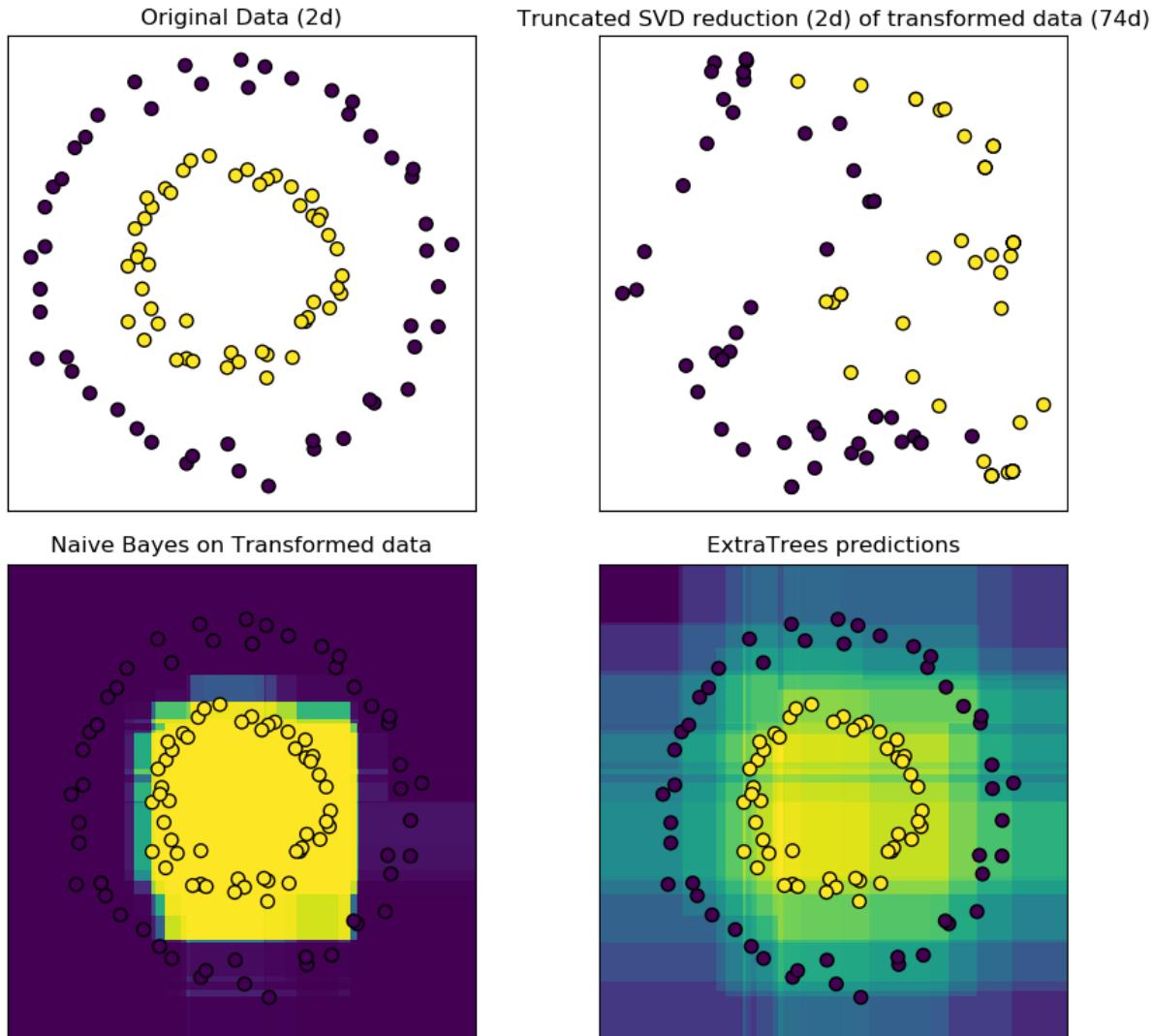
5.12.14 Hashing feature transformation using Totally Random Trees

RandomTreesEmbedding provides a way to map data to a very high-dimensional, sparse representation, which might be beneficial for classification. The mapping is completely unsupervised and very efficient.

This example visualizes the partitions given by several trees and shows how the transformation can also be used for non-linear dimensionality reduction or non-linear classification.

Points that are neighboring often share the same leaf of a tree and therefore share large parts of their hashed representation. This allows to separate two concentric circles simply based on the principal components of the transformed data with truncated SVD.

In high-dimensional spaces, linear classifiers often achieve excellent accuracy. For sparse binary data, BernoulliNB is particularly well-suited. The bottom row compares the decision boundary obtained by BernoulliNB in the transformed space with an ExtraTreesClassifier learned on the original data.



```

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_circles
from sklearn.ensemble import RandomTreesEmbedding, ExtraTreesClassifier
from sklearn.decomposition import TruncatedSVD
from sklearn.naive_bayes import BernoulliNB

# make a synthetic dataset
X, y = make_circles(factor=0.5, random_state=0, noise=0.05)

# use RandomTreesEmbedding to transform data
hasher = RandomTreesEmbedding(n_estimators=10, random_state=0, max_depth=3)
X_transformed = hasher.fit_transform(X)

# Visualize result after dimensionality reduction using truncated SVD
svd = TruncatedSVD(n_components=2)
X_reduced = svd.fit_transform(X_transformed)

# Learn a Naive Bayes classifier on the transformed data
nb = BernoulliNB()
nb.fit(X_transformed, y)

# Learn an ExtraTreesClassifier for comparison
trees = ExtraTreesClassifier(max_depth=3, n_estimators=10, random_state=0)
trees.fit(X, y)

# scatter plot of original and reduced data
fig = plt.figure(figsize=(9, 8))

ax = plt.subplot(221)
ax.scatter(X[:, 0], X[:, 1], c=y, s=50, edgecolor='k')
ax.set_title("Original Data (2d)")
ax.set_xticks(())
ax.set_yticks(())

ax = plt.subplot(222)
ax.scatter(X_reduced[:, 0], X_reduced[:, 1], c=y, s=50, edgecolor='k')
ax.set_title("Truncated SVD reduction (2d) of transformed data (%d)" % X_transformed.shape[1])
ax.set_xticks(())
ax.set_yticks(())

# Plot the decision in original space. For that, we will assign a color
# to each point in the mesh [x_min, x_max]x[y_min, y_max].
h = .01
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))

# transform grid using RandomTreesEmbedding
transformed_grid = hasher.transform(np.c_[xx.ravel(), yy.ravel()])
y_grid_pred = nb.predict_proba(transformed_grid)[:, 1]

ax = plt.subplot(223)
ax.set_title("Naive Bayes on Transformed data")

```

```

ax.pcolormesh(xx, yy, y_grid_pred.reshape(xx.shape))
ax.scatter(X[:, 0], X[:, 1], c=y, s=50, edgecolor='k')
ax.set_xlim(-1.4, 1.4)
ax.set_xticks(())
ax.set_yticks(())

# transform grid using ExtraTreesClassifier
y_grid_pred = trees.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

ax = plt.subplot(224)
ax.set_title("ExtraTrees predictions")
ax.pcolormesh(xx, yy, y_grid_pred.reshape(xx.shape))
ax.scatter(X[:, 0], X[:, 1], c=y, s=50, edgecolor='k')
ax.set_xlim(-1.4, 1.4)
ax.set_xticks(())
ax.set_yticks(())

plt.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 0.250 seconds)

Note: Click [here](#) to download the full example code

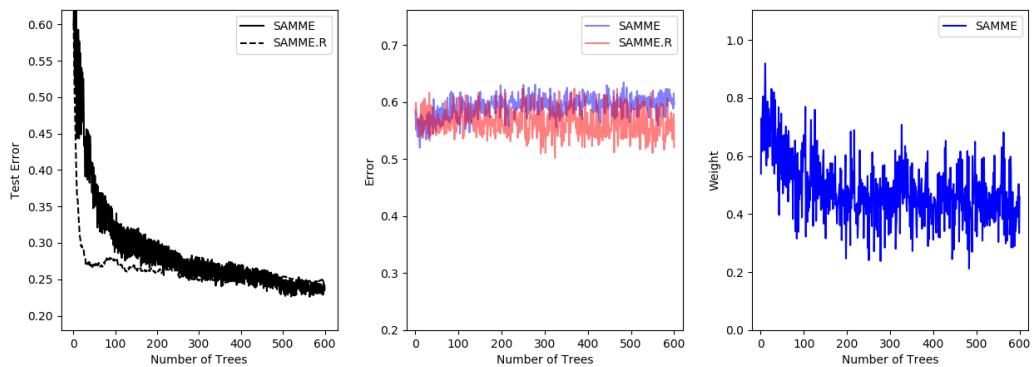
5.12.15 Multi-class AdaBoosted Decision Trees

This example reproduces Figure 1 of Zhu et al¹ and shows how boosting can improve prediction accuracy on a multi-class problem. The classification dataset is constructed by taking a ten-dimensional standard normal distribution and defining three classes separated by nested concentric ten-dimensional spheres such that roughly equal numbers of samples are in each class (quantiles of the χ^2 distribution).

The performance of the SAMME and SAMME.R¹ algorithms are compared. SAMME.R uses the probability estimates to update the additive model, while SAMME uses the classifications only. As the example illustrates, the SAMME.R algorithm typically converges faster than SAMME, achieving a lower test error with fewer boosting iterations. The error of each algorithm on the test set after each boosting iteration is shown on the left, the classification error on the test set of each tree is shown in the middle, and the boost weight of each tree is shown on the right. All trees have a weight of one in the SAMME.R algorithm and therefore are not shown.

¹

10. Zhu, H. Zou, S. Rosset, T. Hastie, “Multi-class AdaBoost”, 2009.



```

print(__doc__)

# Author: Noel Dawe <noel.dawe@gmail.com>
#
# License: BSD 3 clause

import matplotlib.pyplot as plt

from sklearn.datasets import make_gaussian_quantiles
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier


X, y = make_gaussian_quantiles(n_samples=13000, n_features=10,
                               n_classes=3, random_state=1)

n_split = 3000

X_train, X_test = X[:n_split], X[n_split:]
y_train, y_test = y[:n_split], y[n_split:]

bdt_real = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=2),
    n_estimators=600,
    learning_rate=1)

bdt_discrete = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=2),
    n_estimators=600,
    learning_rate=1.5,
    algorithm="SAMME")

bdt_real.fit(X_train, y_train)
bdt_discrete.fit(X_train, y_train)

real_test_errors = []
discrete_test_errors = []

for real_test_predict, discrete_train_predict in zip(
    bdt_real.staged_predict(X_test), bdt_discrete.staged_predict(X_test)):
    real_test_errors.append(
        1. - accuracy_score(real_test_predict, y_test))

```

```

discrete_test_errors.append(
    1. - accuracy_score(discrete_train_predict, y_test))

n_trees_discrete = len(bdt_discrete)
n_trees_real = len(bdt_real)

# Boosting might terminate early, but the following arrays are always
# n_estimators long. We crop them to the actual number of trees here:
discrete_estimator_errors = bdt_discrete.estimator_errors_[:n_trees_discrete]
real_estimator_errors = bdt_real.estimator_errors_[:n_trees_real]
discrete_estimator_weights = bdt_discrete.estimator_weights_[:n_trees_discrete]

plt.figure(figsize=(15, 5))

plt.subplot(131)
plt.plot(range(1, n_trees_discrete + 1),
          discrete_test_errors, c='black', label='SAMME')
plt.plot(range(1, n_trees_real + 1),
          real_test_errors, c='black',
          linestyle='dashed', label='SAMME.R')
plt.legend()
plt.ylim(0.18, 0.62)
plt.ylabel('Test Error')
plt.xlabel('Number of Trees')

plt.subplot(132)
plt.plot(range(1, n_trees_discrete + 1), discrete_estimator_errors,
         "b", label='SAMME', alpha=.5)
plt.plot(range(1, n_trees_real + 1), real_estimator_errors,
         "r", label='SAMME.R', alpha=.5)
plt.legend()
plt.ylabel('Error')
plt.xlabel('Number of Trees')
plt.ylim(.2,
        max(real_estimator_errors.max(),
            discrete_estimator_errors.max()) * 1.2))
plt.xlim((-20, len(bdt_discrete) + 20))

plt.subplot(133)
plt.plot(range(1, n_trees_discrete + 1), discrete_estimator_weights,
         "b", label='SAMME')
plt.legend()
plt.ylabel('Weight')
plt.xlabel('Number of Trees')
plt.ylim(0, discrete_estimator_weights.max() * 1.2))
plt.xlim((-20, n_trees_discrete + 20))

# prevent overlapping y-axis labels
plt.subplots_adjust(wspace=0.25)
plt.show()

```

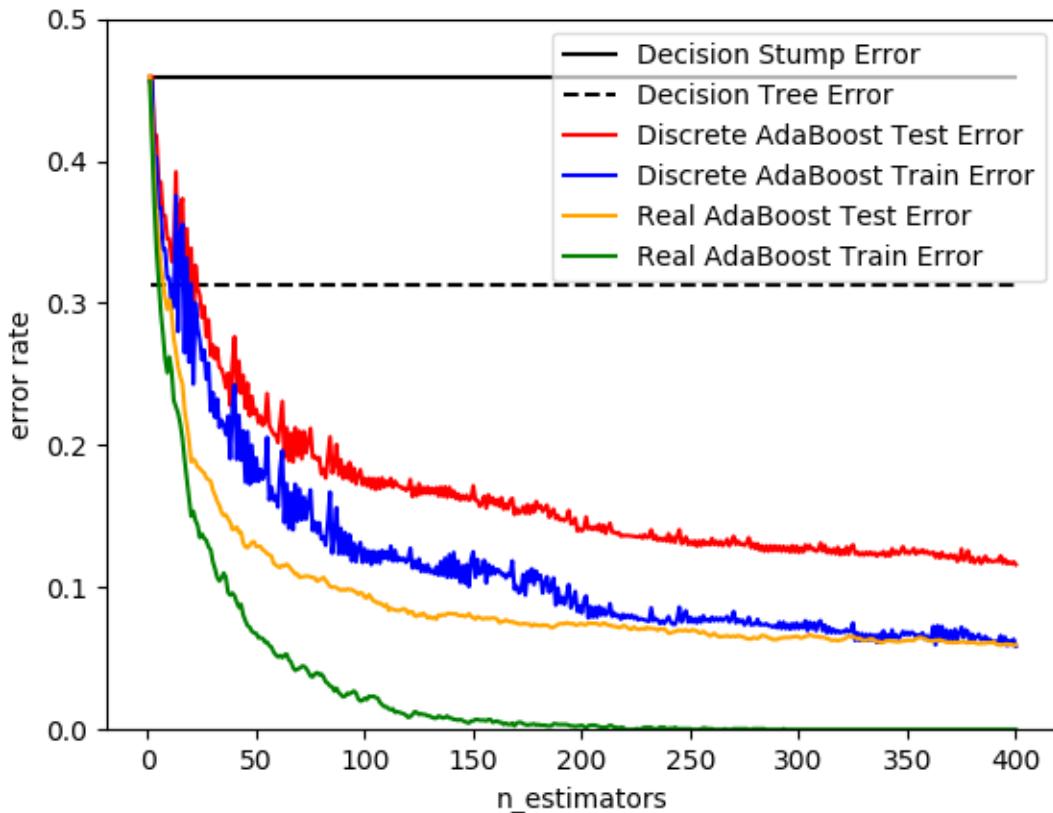
Total running time of the script: (0 minutes 11.401 seconds)

Note: Click [here](#) to download the full example code

5.12.16 Discrete versus Real AdaBoost

This example is based on Figure 10.2 from Hastie et al 2009¹ and illustrates the difference in performance between the discrete SAMME² boosting algorithm and real SAMME.R boosting algorithm. Both algorithms are evaluated on a binary classification task where the target Y is a non-linear function of 10 input features.

Discrete SAMME AdaBoost adapts based on errors in predicted class labels whereas real SAMME.R uses the predicted class probabilities.



```
print(__doc__)

# Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>,
#         Noel Dawe <noel.dawe@gmail.com>
#
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import zero_one_loss
```

¹ T. Hastie, R. Tibshirani and J. Friedman, “Elements of Statistical Learning Ed. 2”, Springer, 2009.
²

10. Zhu, H. Zou, S. Rosset, T. Hastie, “Multi-class AdaBoost”, 2009.

```

from sklearn.ensemble import AdaBoostClassifier

n_estimators = 400
# A learning rate of 1. may not be optimal for both SAMME and SAMME.R
learning_rate = 1.

X, y = datasets.make_hastie_10_2(n_samples=12000, random_state=1)

X_test, y_test = X[2000:], y[2000:]
X_train, y_train = X[:2000], y[:2000]

dt_stump = DecisionTreeClassifier(max_depth=1, min_samples_leaf=1)
dt_stump.fit(X_train, y_train)
dt_stump_err = 1.0 - dt_stump.score(X_test, y_test)

dt = DecisionTreeClassifier(max_depth=9, min_samples_leaf=1)
dt.fit(X_train, y_train)
dt_err = 1.0 - dt.score(X_test, y_test)

ada_discrete = AdaBoostClassifier(
    base_estimator=dt_stump,
    learning_rate=learning_rate,
    n_estimators=n_estimators,
    algorithm="SAMME")
ada_discrete.fit(X_train, y_train)

ada_real = AdaBoostClassifier(
    base_estimator=dt_stump,
    learning_rate=learning_rate,
    n_estimators=n_estimators,
    algorithm="SAMME.R")
ada_real.fit(X_train, y_train)

fig = plt.figure()
ax = fig.add_subplot(111)

ax.plot([1, n_estimators], [dt_stump_err] * 2, 'k-',
        label='Decision Stump Error')
ax.plot([1, n_estimators], [dt_err] * 2, 'k--',
        label='Decision Tree Error')

ada_discrete_err = np.zeros((n_estimators,))
for i, y_pred in enumerate(ada_discrete.staged_predict(X_test)):
    ada_discrete_err[i] = zero_one_loss(y_pred, y_test)

ada_discrete_err_train = np.zeros((n_estimators,))
for i, y_pred in enumerate(ada_discrete.staged_predict(X_train)):
    ada_discrete_err_train[i] = zero_one_loss(y_pred, y_train)

ada_real_err = np.zeros((n_estimators,))
for i, y_pred in enumerate(ada_real.staged_predict(X_test)):
    ada_real_err[i] = zero_one_loss(y_pred, y_test)

ada_real_err_train = np.zeros((n_estimators,))
for i, y_pred in enumerate(ada_real.staged_predict(X_train)):
    ada_real_err_train[i] = zero_one_loss(y_pred, y_train)

```

```
ax.plot(np.arange(n_estimators) + 1, ada_discrete_err,
        label='Discrete AdaBoost Test Error',
        color='red')
ax.plot(np.arange(n_estimators) + 1, ada_discrete_err_train,
        label='Discrete AdaBoost Train Error',
        color='blue')
ax.plot(np.arange(n_estimators) + 1, ada_real_err,
        label='Real AdaBoost Test Error',
        color='orange')
ax.plot(np.arange(n_estimators) + 1, ada_real_err_train,
        label='Real AdaBoost Train Error',
        color='green')

ax.set_xlim((0.0, 50))
ax.set_xlabel('n_estimators')
ax.set_ylabel('error rate')

leg = ax.legend(loc='upper right', fancybox=True)
leg.get_frame().set_alpha(0.7)

plt.show()
```

Total running time of the script: (0 minutes 4.579 seconds)

Note: Click [here](#) to download the full example code

5.12.17 Early stopping of Gradient Boosting

Gradient boosting is an ensembling technique where several weak learners (regression trees) are combined to yield a powerful single model, in an iterative fashion.

Early stopping support in Gradient Boosting enables us to find the least number of iterations which is sufficient to build a model that generalizes well to unseen data.

The concept of early stopping is simple. We specify a `validation_fraction` which denotes the fraction of the whole dataset that will be kept aside from training to assess the validation loss of the model. The gradient boosting model is trained using the training set and evaluated using the validation set. When each additional stage of regression tree is added, the validation set is used to score the model. This is continued until the scores of the model in the last `n_iter_no_change` stages do not improve by atleast `tol`. After that the model is considered to have converged and further addition of stages is “stopped early”.

The number of stages of the final model is available at the attribute `n_estimators_`.

This example illustrates how the early stopping can be used in the `sklearn.ensemble.GradientBoostingClassifier` model to achieve almost the same accuracy as compared to a model built without early stopping using many fewer estimators. This can significantly reduce training time, memory usage and prediction latency.

```
# Authors: Vighnesh Birodkar <vighneshbirodkar@nyu.edu>
#          Raghav RV <rvraghav93@gmail.com>
# License: BSD 3 clause

import time

import numpy as np
```

```

import matplotlib.pyplot as plt

from sklearn import ensemble
from sklearn import datasets
from sklearn.model_selection import train_test_split

print(__doc__)

data_list = [datasets.load_iris(), datasets.load_digits()]
data_list = [(d.data, d.target) for d in data_list]
data_list += [datasets.make_hastie_10_2()]
names = ['Iris Data', 'Digits Data', 'Hastie Data']

n_gb = []
score_gb = []
time_gb = []
n_gbes = []
score_gbes = []
time_gbes = []

n_estimators = 500

for X, y in data_list:
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                       random_state=0)

    # We specify that if the scores don't improve by atleast 0.01 for the last
    # 10 stages, stop fitting additional stages
    gbes = ensemble.GradientBoostingClassifier(n_estimators=n_estimators,
                                                validation_fraction=0.2,
                                                n_iter_no_change=5, tol=0.01,
                                                random_state=0)
    gb = ensemble.GradientBoostingClassifier(n_estimators=n_estimators,
                                              random_state=0)
    start = time.time()
    gb.fit(X_train, y_train)
    time_gb.append(time.time() - start)

    start = time.time()
    gbes.fit(X_train, y_train)
    time_gbes.append(time.time() - start)

    score_gb.append(gb.score(X_test, y_test))
    score_gbes.append(gbes.score(X_test, y_test))

    n_gb.append(gb.n_estimators_)
    n_gbes.append(gbes.n_estimators_)

bar_width = 0.2
n = len(data_list)
index = np.arange(0, n * bar_width, bar_width) * 2.5
index = index[0:n]

```

Compare scores with and without early stopping

```
plt.figure(figsize=(9, 5))

bar1 = plt.bar(index, score_gb, bar_width, label='Without early stopping',
               color='crimson')
bar2 = plt.bar(index + bar_width, score_gbes, bar_width,
               label='With early stopping', color='coral')

plt.xticks(index + bar_width, names)
plt.yticks(np.arange(0, 1.3, 0.1))

def autolabel(rects, n_estimators):
    """
    Attach a text label above each bar displaying n_estimators of each model
    """
    for i, rect in enumerate(rects):
        plt.text(rect.get_x() + rect.get_width() / 2.,
                 1.05 * rect.get_height(), 'n_est=%d' % n_estimators[i],
                 ha='center', va='bottom')

autolabel(bar1, n_gb)
autolabel(bar2, n_gbes)

plt.ylim([0, 1.3])
plt.legend(loc='best')
plt.grid(True)

plt.xlabel('Datasets')
plt.ylabel('Test score')

plt.show()
```



Compare fit times with and without early stopping

```

plt.figure(figsize=(9, 5))

bar1 = plt.bar(index, time_gb, bar_width, label='Without early stopping',
               color='crimson')
bar2 = plt.bar(index + bar_width, time_gbes, bar_width,
               label='With early stopping', color='coral')

max_y = np.amax(np.maximum(time_gb, time_gbes))

plt.xticks(index + bar_width, names)
plt.yticks(np.linspace(0, 1.3 * max_y, 13))

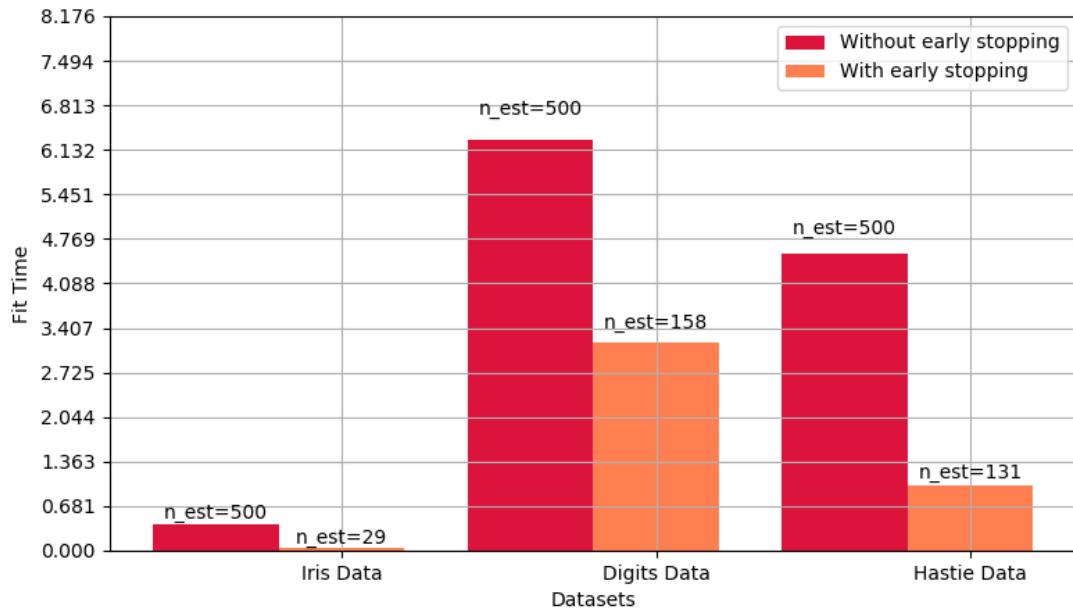
autolabel(bar1, n_gb)
autolabel(bar2, n_gbes)

plt.ylim([0, 1.3 * max_y])
plt.legend(loc='best')
plt.grid(True)

plt.xlabel('Datasets')
plt.ylabel('Fit Time')

plt.show()

```



Total running time of the script: (0 minutes 15.622 seconds)

Note: Click [here](#) to download the full example code

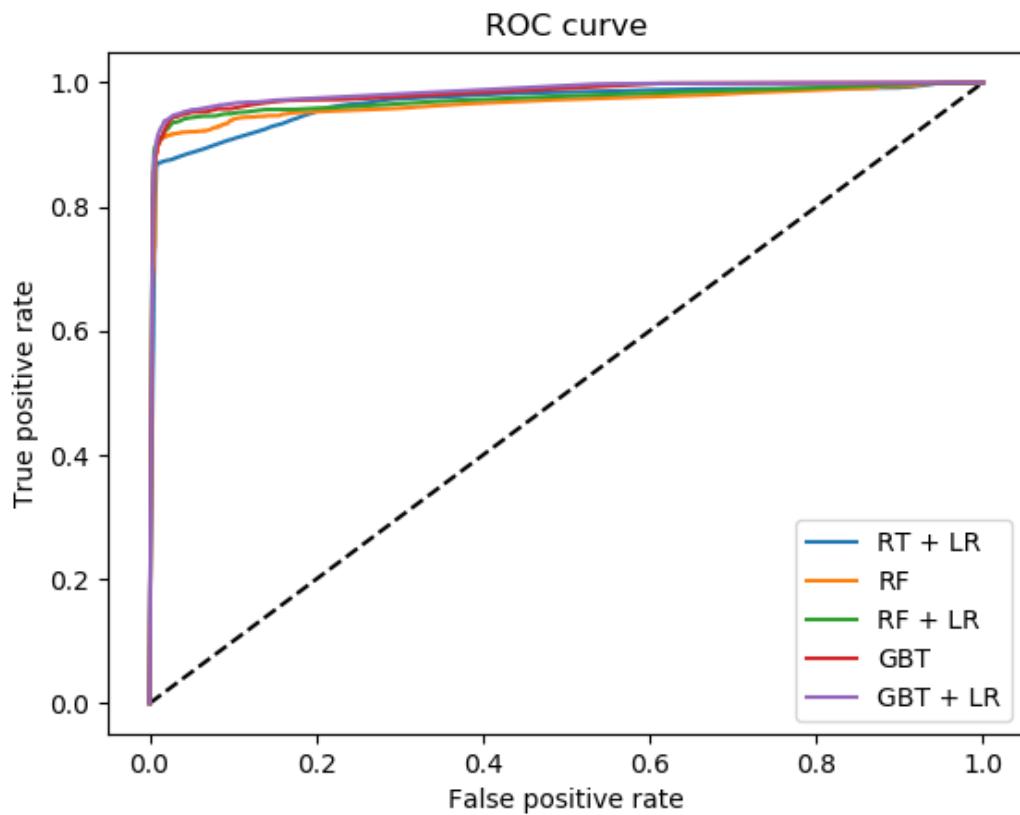
5.12.18 Feature transformations with ensembles of trees

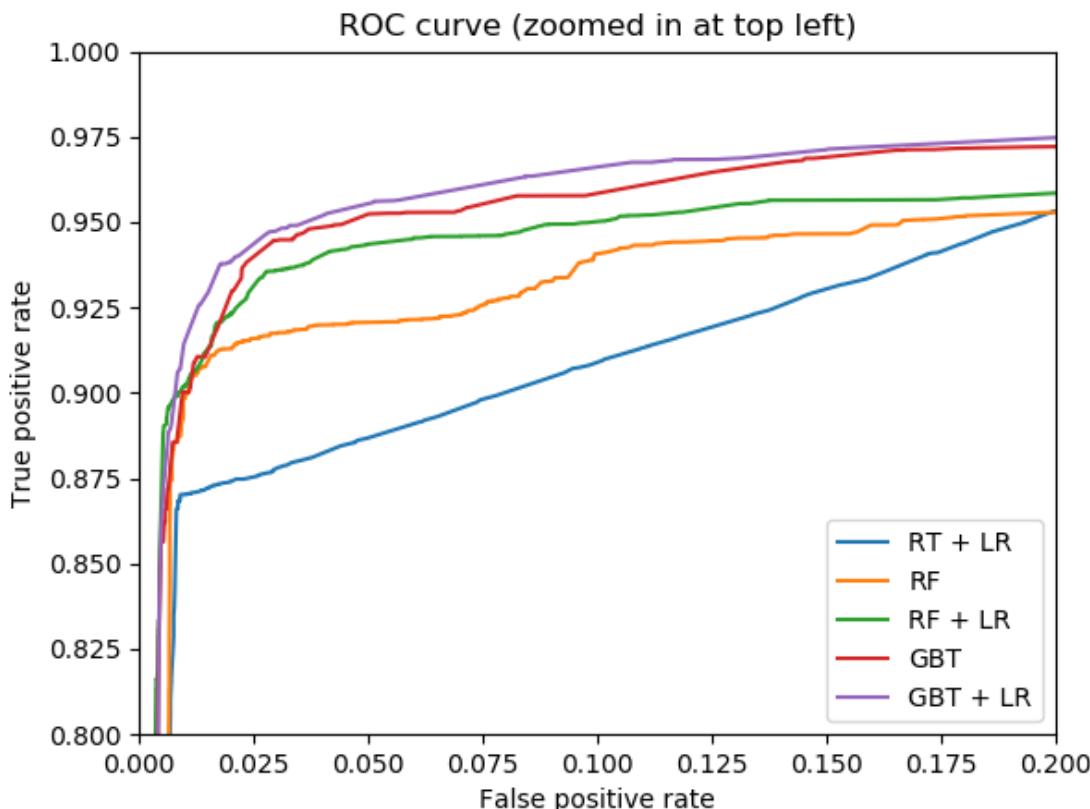
Transform your features into a higher dimensional, sparse space. Then train a linear model on these features.

First fit an ensemble of trees (totally random trees, a random forest, or gradient boosted trees) on the training set. Then each leaf of each tree in the ensemble is assigned a fixed arbitrary feature index in a new feature space. These leaf indices are then encoded in a one-hot fashion.

Each sample goes through the decisions of each tree of the ensemble and ends up in one leaf per tree. The sample is encoded by setting feature values for these leaves to 1 and the other feature values to 0.

The resulting transformer has then learned a supervised, sparse, high-dimensional categorical embedding of the data.





```
# Author: Tim Head <betatim@gmail.com>
#
# License: BSD 3 clause

import numpy as np
np.random.seed(10)

import matplotlib.pyplot as plt

from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import (RandomTreesEmbedding, RandomForestClassifier,
                             GradientBoostingClassifier)
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve
from sklearn.pipeline import make_pipeline

n_estimator = 10
X, y = make_classification(n_samples=80000)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)

# It is important to train the ensemble of trees on a different subset
# of the training data than the linear regression model to avoid
# overfitting, in particular if the total number of leaves is
# similar to the number of training samples
X_train, X_train_lr, y_train, y_train_lr = train_test_split(
```

```

X_train, y_train, test_size=0.5)

# Unsupervised transformation based on totally random trees
rt = RandomTreesEmbedding(max_depth=3, n_estimators=n_estimator,
                          random_state=0)

rt_lm = LogisticRegression(solver='lbfgs', max_iter=1000)
pipeline = make_pipeline(rt, rt_lm)
pipeline.fit(X_train, y_train)
y_pred_rt = pipeline.predict_proba(X_test)[:, 1]
fpr_rt_lm, tpr_rt_lm, _ = roc_curve(y_test, y_pred_rt)

# Supervised transformation based on random forests
rf = RandomForestClassifier(max_depth=3, n_estimators=n_estimator)
rf_enc = OneHotEncoder(categories='auto')
rf_lm = LogisticRegression(solver='lbfgs', max_iter=1000)
rf.fit(X_train, y_train)
rf_enc.fit(rf.apply(X_train))
rf_lm.fit(rf_enc.transform(rf.apply(X_train_lr)), y_train_lr)

y_pred_rf_lm = rf_lm.predict_proba(rf_enc.transform(rf.apply(X_test)))[:, 1]
fpr_rf_lm, tpr_rf_lm, _ = roc_curve(y_test, y_pred_rf_lm)

# Supervised transformation based on gradient boosted trees
grd = GradientBoostingClassifier(n_estimators=n_estimator)
grd_enc = OneHotEncoder(categories='auto')
grd_lm = LogisticRegression(solver='lbfgs', max_iter=1000)
grd.fit(X_train, y_train)
grd_enc.fit(grd.apply(X_train)[:, :, 0])
grd_lm.fit(grd_enc.transform(grd.apply(X_train_lr)[:, :, 0]), y_train_lr)

y_pred_grd_lm = grd_lm.predict_proba(
    grd_enc.transform(grd.apply(X_test)[:, :, 0]))[:, 1]
fpr_grd_lm, tpr_grd_lm, _ = roc_curve(y_test, y_pred_grd_lm)

# The gradient boosted model by itself
y_pred_grd = grd.predict_proba(X_test)[:, 1]
fpr_grd, tpr_grd, _ = roc_curve(y_test, y_pred_grd)

# The random forest model by itself
y_pred_rf = rf.predict_proba(X_test)[:, 1]
fpr_rf, tpr_rf, _ = roc_curve(y_test, y_pred_rf)

plt.figure(1)
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr_rt_lm, tpr_rt_lm, label='RT + LR')
plt.plot(fpr_rf, tpr_rf, label='RF')
plt.plot(fpr_rf_lm, tpr_rf_lm, label='RF + LR')
plt.plot(fpr_grd, tpr_grd, label='GBT')
plt.plot(fpr_grd_lm, tpr_grd_lm, label='GBT + LR')
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve')
plt.legend(loc='best')
plt.show()

plt.figure(2)
plt.xlim(0, 0.2)

```

```
plt.ylim(0.8, 1)
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr_rt_lm, tpr_rt_lm, label='RT + LR')
plt.plot(fpr_rf, tpr_rf, label='RF')
plt.plot(fpr_rf_lm, tpr_rf_lm, label='RF + LR')
plt.plot(fpr_grd, tpr_grd, label='GBT')
plt.plot(fpr_grd_lm, tpr_grd_lm, label='GBT + LR')
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve (zoomed in at top left)')
plt.legend(loc='best')
plt.show()
```

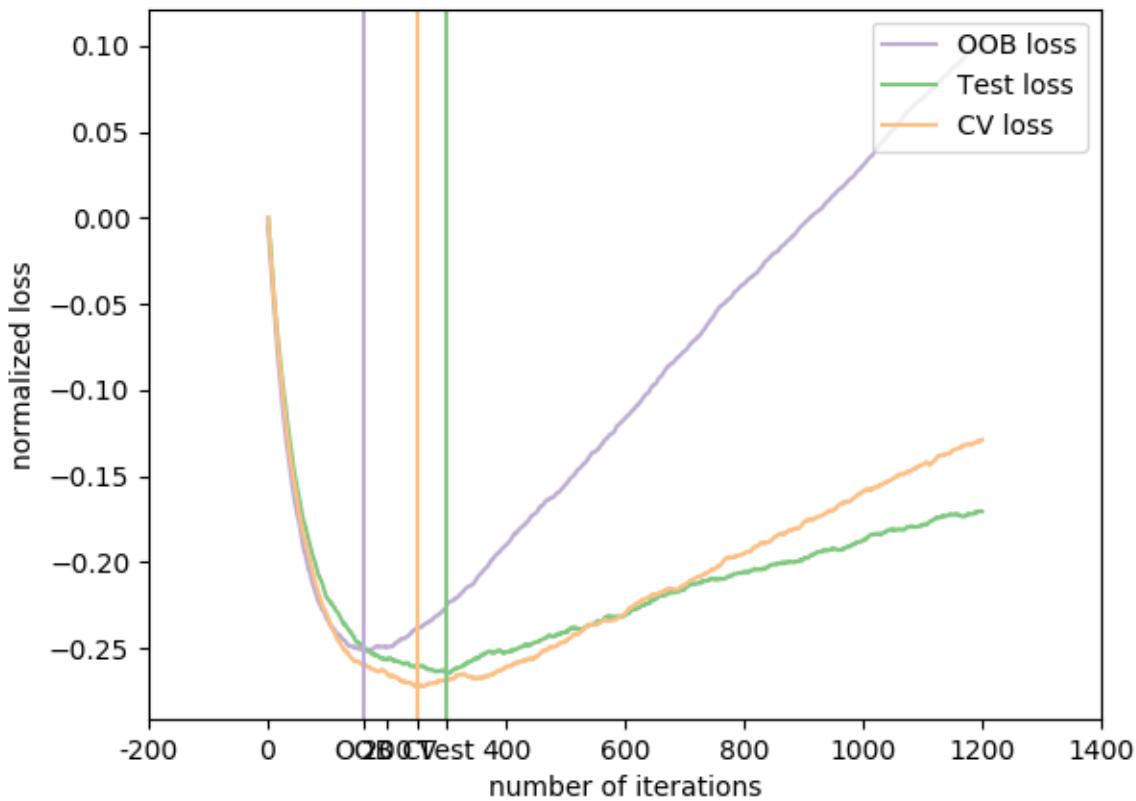
Total running time of the script: (0 minutes 2.261 seconds)

Note: Click [here](#) to download the full example code

5.12.19 Gradient Boosting Out-of-Bag estimates

Out-of-bag (OOB) estimates can be a useful heuristic to estimate the “optimal” number of boosting iterations. OOB estimates are almost identical to cross-validation estimates but they can be computed on-the-fly without the need for repeated model fitting. OOB estimates are only available for Stochastic Gradient Boosting (i.e. `subsample < 1.0`), the estimates are derived from the improvement in loss based on the examples not included in the bootstrap sample (the so-called out-of-bag examples). The OOB estimator is a pessimistic estimator of the true test loss, but remains a fairly good approximation for a small number of trees.

The figure shows the cumulative sum of the negative OOB improvements as a function of the boosting iteration. As you can see, it tracks the test loss for the first hundred iterations but then diverges in a pessimistic way. The figure also shows the performance of 3-fold cross validation which usually gives a better estimate of the test loss but is computationally more demanding.



Out:

```
Accuracy: 0.6840
```

```
print(__doc__)

# Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>
#
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn import ensemble
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split

from scipy.special import expit

# Generate data (adapted from G. Ridgeway's gbm example)
n_samples = 1000
```

```

random_state = np.random.RandomState(13)
x1 = random_state.uniform(size=n_samples)
x2 = random_state.uniform(size=n_samples)
x3 = random_state.randint(0, 4, size=n_samples)

p = expit(np.sin(3 * x1) - 4 * x2 + x3)
y = random_state.binomial(1, p, size=n_samples)

X = np.c_[x1, x2, x3]

X = X.astype(np.float32)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5,
                                                    random_state=9)

# Fit classifier with out-of-bag estimates
params = {'n_estimators': 1200, 'max_depth': 3, 'subsample': 0.5,
          'learning_rate': 0.01, 'min_samples_leaf': 1, 'random_state': 3}
clf = ensemble.GradientBoostingClassifier(**params)

clf.fit(X_train, y_train)
acc = clf.score(X_test, y_test)
print("Accuracy: {:.4f}".format(acc))

n_estimators = params['n_estimators']
x = np.arange(n_estimators) + 1

def heldout_score(clf, X_test, y_test):
    """compute deviance scores on ``X_test`` and ``y_test``. """
    score = np.zeros((n_estimators,), dtype=np.float64)
    for i, y_pred in enumerate(clf.staged_decision_function(X_test)):
        score[i] = clf.loss_(y_test, y_pred)
    return score

def cv_estimate(n_splits=None):
    cv = KFold(n_splits=n_splits)
    cv_clf = ensemble.GradientBoostingClassifier(**params)
    val_scores = np.zeros((n_estimators,), dtype=np.float64)
    for train, test in cv.split(X_train, y_train):
        cv_clf.fit(X_train[train], y_train[train])
        val_scores += heldout_score(cv_clf, X_train[test], y_train[test])
    val_scores /= n_splits
    return val_scores

# Estimate best n_estimator using cross-validation
cv_score = cv_estimate(3)

# Compute best n_estimator for test data
test_score = heldout_score(clf, X_test, y_test)

# negative cumulative sum of oob improvements
cumsum = -np.cumsum(clf.oob_improvement_)

# min loss according to OOB
oob_best_iter = x[np.argmin(cumsum)]

```

```

# min loss according to test (normalize such that first loss is 0)
test_score -= test_score[0]
test_best_iter = x[np.argmin(test_score)]

# min loss according to cv (normalize such that first loss is 0)
cv_score -= cv_score[0]
cv_best_iter = x[np.argmin(cv_score)]

# color brew for the three curves
oob_color = list(map(lambda x: x / 256.0, (190, 174, 212)))
test_color = list(map(lambda x: x / 256.0, (127, 201, 127)))
cv_color = list(map(lambda x: x / 256.0, (253, 192, 134)))

# plot curves and vertical lines for best iterations
plt.plot(x, cumsum, label='OOB loss', color=oob_color)
plt.plot(x, test_score, label='Test loss', color=test_color)
plt.plot(x, cv_score, label='CV loss', color=cv_color)
plt.axvline(x=oob_best_iter, color=oob_color)
plt.axvline(x=test_best_iter, color=test_color)
plt.axvline(x=cv_best_iter, color=cv_color)

# add three vertical lines to xticks
xticks = plt.xticks()
xticks_pos = np.array(xticks[0].tolist() +
                      [oob_best_iter, cv_best_iter, test_best_iter])
xticks_label = np.array(list(map(lambda t: int(t), xticks[0]))) +
                  ['OOB', 'CV', 'Test'])
ind = np.argsort(xticks_pos)
xticks_pos = xticks_pos[ind]
xticks_label = xticks_label[ind]
plt.xticks(xticks_pos, xticks_label)

plt.legend(loc='upper right')
plt.ylabel('normalized loss')
plt.xlabel('number of iterations')

plt.show()

```

Total running time of the script: (0 minutes 2.759 seconds)

Note: Click [here](#) to download the full example code

5.12.20 Single estimator versus bagging: bias-variance decomposition

This example illustrates and compares the bias-variance decomposition of the expected mean squared error of a single estimator against a bagging ensemble.

In regression, the expected mean squared error of an estimator can be decomposed in terms of bias, variance and noise. On average over datasets of the regression problem, the bias term measures the average amount by which the predictions of the estimator differ from the predictions of the best possible estimator for the problem (i.e., the Bayes model). The variance term measures the variability of the predictions of the estimator when fit over different instances LS of the problem. Finally, the noise measures the irreducible part of the error which is due the variability in the data.

The upper left figure illustrates the predictions (in dark red) of a single decision tree trained over a random dataset LS (the blue dots) of a toy 1d regression problem. It also illustrates the predictions (in light red) of other single decision

trees trained over other (and different) randomly drawn instances LS of the problem. Intuitively, the variance term here corresponds to the width of the beam of predictions (in light red) of the individual estimators. The larger the variance, the more sensitive are the predictions for x to small changes in the training set. The bias term corresponds to the difference between the average prediction of the estimator (in cyan) and the best possible model (in dark blue). On this problem, we can thus observe that the bias is quite low (both the cyan and the blue curves are close to each other) while the variance is large (the red beam is rather wide).

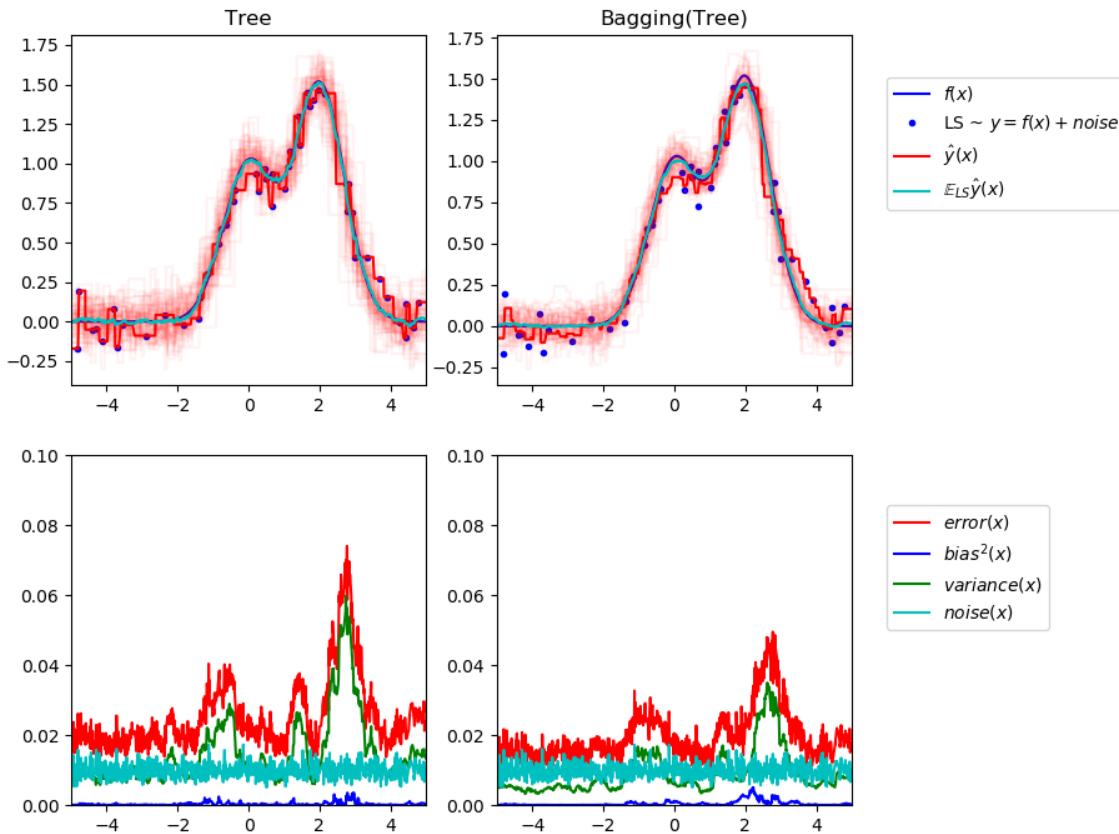
The lower left figure plots the pointwise decomposition of the expected mean squared error of a single decision tree. It confirms that the bias term (in blue) is low while the variance is large (in green). It also illustrates the noise part of the error which, as expected, appears to be constant and around 0.01.

The right figures correspond to the same plots but using instead a bagging ensemble of decision trees. In both figures, we can observe that the bias term is larger than in the previous case. In the upper right figure, the difference between the average prediction (in cyan) and the best possible model is larger (e.g., notice the offset around $x=2$). In the lower right figure, the bias curve is also slightly higher than in the lower left figure. In terms of variance however, the beam of predictions is narrower, which suggests that the variance is lower. Indeed, as the lower right figure confirms, the variance term (in green) is lower than for single decision trees. Overall, the bias-variance decomposition is therefore no longer the same. The tradeoff is better for bagging: averaging several decision trees fit on bootstrap copies of the dataset slightly increases the bias term but allows for a larger reduction of the variance, which results in a lower overall mean squared error (compare the red curves in the lower figures). The script output also confirms this intuition. The total error of the bagging ensemble is lower than the total error of a single decision tree, and this difference indeed mainly stems from a reduced variance.

For further details on bias-variance decomposition, see section 7.3 of¹.

¹ T. Hastie, R. Tibshirani and J. Friedman, “Elements of Statistical Learning”, Springer, 2009.

References



Out:

```
Tree: 0.0255 (error) = 0.0003 (bias^2) + 0.0152 (var) + 0.0098 (noise)
Bagging(Tree): 0.0196 (error) = 0.0004 (bias^2) + 0.0092 (var) + 0.0098 (noise)
```

```
print(__doc__)

# Author: Gilles Louppe <g.louppe@gmail.com>
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.ensemble import BaggingRegressor
from sklearn.tree import DecisionTreeRegressor
```

```

# Settings
n_repeat = 50          # Number of iterations for computing expectations
n_train = 50            # Size of the training set
n_test = 1000           # Size of the test set
noise = 0.1             # Standard deviation of the noise
np.random.seed(0)

# Change this for exploring the bias-variance decomposition of other
# estimators. This should work well for estimators with high variance (e.g.,
# decision trees or KNN), but poorly for estimators with low variance (e.g.,
# linear models).
estimators = [("Tree", DecisionTreeRegressor()),
              ("Bagging(Tree)", BaggingRegressor(DecisionTreeRegressor()))]

n_estimators = len(estimators)

# Generate data
def f(x):
    x = x.ravel()

    return np.exp(-x ** 2) + 1.5 * np.exp(-(x - 2) ** 2)

def generate(n_samples, noise, n_repeat=1):
    X = np.random.rand(n_samples) * 10 - 5
    X = np.sort(X)

    if n_repeat == 1:
        y = f(X) + np.random.normal(0.0, noise, n_samples)
    else:
        y = np.zeros((n_samples, n_repeat))

        for i in range(n_repeat):
            y[:, i] = f(X) + np.random.normal(0.0, noise, n_samples)

    X = X.reshape((n_samples, 1))

    return X, y

X_train = []
y_train = []

for i in range(n_repeat):
    X, y = generate(n_samples=n_train, noise=noise)
    X_train.append(X)
    y_train.append(y)

X_test, y_test = generate(n_samples=n_test, noise=noise, n_repeat=n_repeat)

plt.figure(figsize=(10, 8))

# Loop over estimators to compare
for n, (name, estimator) in enumerate(estimators):
    # Compute predictions
    y_predict = np.zeros((n_test, n_repeat))

```

```

for i in range(n_repeat):
    estimator.fit(X_train[i], y_train[i])
    y_predict[:, i] = estimator.predict(X_test)

# Bias^2 + Variance + Noise decomposition of the mean squared error
y_error = np.zeros(n_test)

for i in range(n_repeat):
    for j in range(n_repeat):
        y_error += (y_test[:, j] - y_predict[:, i]) ** 2

y_error /= (n_repeat * n_repeat)

y_noise = np.var(y_test, axis=1)
y_bias = (f(X_test) - np.mean(y_predict, axis=1)) ** 2
y_var = np.var(y_predict, axis=1)

print("{}: {:.4f} (error) = {:.4f} (bias^2) "
      " + {:.4f} (var) + {:.4f} (noise)".format(name,
                                                np.mean(y_error),
                                                np.mean(y_bias),
                                                np.mean(y_var),
                                                np.mean(y_noise)))

# Plot figures
plt.subplot(2, n_estimators, n + 1)
plt.plot(X_test, f(X_test), "b", label="$f(x)$")
plt.plot(X_train[0], y_train[0], ".b", label="LS ~ $y = f(x) + noise$")

for i in range(n_repeat):
    if i == 0:
        plt.plot(X_test, y_predict[:, i], "r", label=r"$\hat{y}(x)$")
    else:
        plt.plot(X_test, y_predict[:, i], "r", alpha=0.05)

plt.plot(X_test, np.mean(y_predict, axis=1), "c",
         label=r"\mathbb{E}_{\text{LS}} \hat{y}(x)")

plt.xlim([-5, 5])
plt.title(name)

if n == n_estimators - 1:
    plt.legend(loc=(1.1, .5))

plt.subplot(2, n_estimators, n_estimators + n + 1)
plt.plot(X_test, y_error, "r", label="$\text{error}(x)$")
plt.plot(X_test, y_bias, "b", label="$\text{bias}^2(x)$"),
plt.plot(X_test, y_var, "g", label="$\text{variance}(x)$"),
plt.plot(X_test, y_noise, "c", label="$\text{noise}(x)$")

plt.xlim([-5, 5])
plt.ylim([0, 0.1])

if n == n_estimators - 1:
    plt.legend(loc=(1.1, .5))

plt.subplots_adjust(right=.75)

```

```
plt.show()
```

Total running time of the script: (0 minutes 0.515 seconds)

Note: Click [here](#) to download the full example code

5.12.21 Plot the decision surfaces of ensembles of trees on the iris dataset

Plot the decision surfaces of forests of randomized trees trained on pairs of features of the iris dataset.

This plot compares the decision surfaces learned by a decision tree classifier (first column), by a random forest classifier (second column), by an extra-trees classifier (third column) and by an AdaBoost classifier (fourth column).

In the first row, the classifiers are built using the sepal width and the sepal length features only, on the second row using the petal length and sepal length only, and on the third row using the petal width and the petal length only.

In descending order of quality, when trained (outside of this example) on all 4 features using 30 estimators and scored using 10 fold cross validation, we see:

```
ExtraTreesClassifier()  # 0.95 score
RandomForestClassifier() # 0.94 score
AdaBoost(DecisionTree(max_depth=3)) # 0.94 score
DecisionTree(max_depth=None) # 0.94 score
```

Increasing `max_depth` for AdaBoost lowers the standard deviation of the scores (but the average score does not improve).

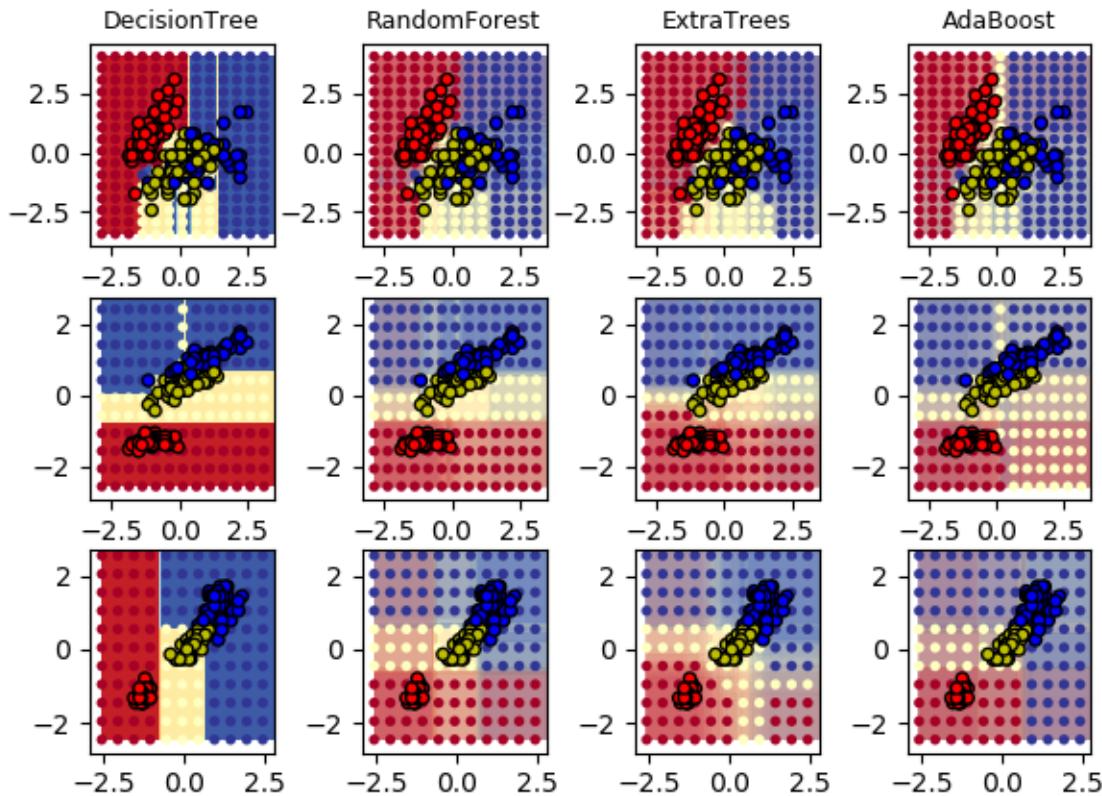
See the console's output for further details about each model.

In this example you might try to:

1. vary the `max_depth` for the `DecisionTreeClassifier` and `AdaBoostClassifier`, perhaps try `max_depth=3` for the `DecisionTreeClassifier` or `max_depth=None` for `AdaBoostClassifier`
2. vary `n_estimators`

It is worth noting that RandomForests and ExtraTrees can be fitted in parallel on many cores as each tree is built independently of the others. AdaBoost's samples are built sequentially and so do not use multiple cores.

Classifiers on feature subsets of the Iris dataset



Out:

```
DecisionTree with features [0, 1] has a score of 0.9266666666666666
RandomForest with 30 estimators with features [0, 1] has a score of 0.9266666666666666
ExtraTrees with 30 estimators with features [0, 1] has a score of 0.9266666666666666
AdaBoost with 30 estimators with features [0, 1] has a score of 0.84
DecisionTree with features [0, 2] has a score of 0.9933333333333333
RandomForest with 30 estimators with features [0, 2] has a score of 0.9933333333333333
ExtraTrees with 30 estimators with features [0, 2] has a score of 0.9933333333333333
AdaBoost with 30 estimators with features [0, 2] has a score of 0.9933333333333333
DecisionTree with features [2, 3] has a score of 0.9933333333333333
RandomForest with 30 estimators with features [2, 3] has a score of 0.9933333333333333
ExtraTrees with 30 estimators with features [2, 3] has a score of 0.9933333333333333
AdaBoost with 30 estimators with features [2, 3] has a score of 0.9933333333333333
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
```

```

from sklearn.datasets import load_iris
from sklearn.ensemble import (RandomForestClassifier, ExtraTreesClassifier,
                             AdaBoostClassifier)
from sklearn.tree import DecisionTreeClassifier

# Parameters
n_classes = 3
n_estimators = 30
cmap = plt.cm.RdYlBu
plot_step = 0.02 # fine step width for decision surface contours
plot_step_coarser = 0.5 # step widths for coarse classifier guesses
RANDOM_SEED = 13 # fix the seed on each iteration

# Load data
iris = load_iris()

plot_idx = 1

models = [DecisionTreeClassifier(max_depth=None),
          RandomForestClassifier(n_estimators=n_estimators),
          ExtraTreesClassifier(n_estimators=n_estimators),
          AdaBoostClassifier(DecisionTreeClassifier(max_depth=3),
                             n_estimators=n_estimators)]


for pair in ([0, 1], [0, 2], [2, 3]):
    for model in models:
        # We only take the two corresponding features
        X = iris.data[:, pair]
        y = iris.target

        # Shuffle
        idx = np.arange(X.shape[0])
        np.random.seed(RANDOM_SEED)
        np.random.shuffle(idx)
        X = X[idx]
        y = y[idx]

        # Standardize
        mean = X.mean(axis=0)
        std = X.std(axis=0)
        X = (X - mean) / std

        # Train
        model.fit(X, y)

        scores = model.score(X, y)
        # Create a title for each column and the console by using str() and
        # slicing away useless parts of the string
        model_title = str(type(model)).split(
            ".")[-1][-2][:len("Classifier")]

        model_details = model_title
        if hasattr(model, "estimators_"):
            model_details += " with {} estimators".format(
                len(model.estimators_))
        print(model_details + " with features", pair,
              "has a score of", scores)

```

```

plt.subplot(3, 4, plot_idx)
if plot_idx <= len(models):
    # Add a title at the top of each column
    plt.title(model_title, fontsize=9)

    # Now plot the decision boundary using a fine mesh as input to a
    # filled contour plot
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                          np.arange(y_min, y_max, plot_step))

    # Plot either a single DecisionTreeClassifier or alpha blend the
    # decision surfaces of the ensemble of classifiers
    if isinstance(model, DecisionTreeClassifier):
        Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)
        cs = plt.contourf(xx, yy, Z, cmap=cmap)
    else:
        # Choose alpha blend level with respect to the number
        # of estimators
        # that are in use (noting that AdaBoost can use fewer estimators
        # than its maximum if it achieves a good enough fit early on)
        estimator_alpha = 1.0 / len(model.estimators_)
        for tree in model.estimators_:
            Z = tree.predict(np.c_[xx.ravel(), yy.ravel()])
            Z = Z.reshape(xx.shape)
            cs = plt.contourf(xx, yy, Z, alpha=estimator_alpha, cmap=cmap)

    # Build a coarser grid to plot a set of ensemble classifications
    # to show how these are different to what we see in the decision
    # surfaces. These points are regularly space and do not have a
    # black outline
    xx_coarser, yy_coarser = np.meshgrid(
        np.arange(x_min, x_max, plot_step_coarser),
        np.arange(y_min, y_max, plot_step_coarser))
    Z_points_coarser = model.predict(np.c_[xx_coarser.ravel(),
                                            yy_coarser.ravel()])
    Z_points_coarser = Z_points_coarser.reshape(xx_coarser.shape)
    cs_points = plt.scatter(xx_coarser, yy_coarser, s=15,
                           c=Z_points_coarser, cmap=cmap,
                           edgecolors="none")

    # Plot the training points, these are clustered together and have a
    # black outline
    plt.scatter(X[:, 0], X[:, 1], c=y,
                cmap=ListedColormap(['r', 'y', 'b']),
                edgecolor='k', s=20)
    plot_idx += 1 # move on to the next plot in sequence

plt.suptitle("Classifiers on feature subsets of the Iris dataset", fontsize=12)
plt.axis("tight")
plt.tight_layout(h_pad=0.2, w_pad=0.2, pad=2.5)
plt.show()

```

Total running time of the script: (0 minutes 6.177 seconds)

5.13 Tutorial exercises

Exercises for the tutorials

Note: Click [here](#) to download the full example code

5.13.1 Digits Classification Exercise

A tutorial exercise regarding the use of classification techniques on the Digits dataset.

This exercise is used in the *Classification* part of the *Supervised learning: predicting an output variable from high-dimensional observations* section of the [A tutorial on statistical-learning for scientific data processing](#).

Out:

```
KNN score: 0.961111  
LogisticRegression score: 0.933333
```

```
print(__doc__)

from sklearn import datasets, neighbors, linear_model

digits = datasets.load_digits()
X_digits = digits.data / digits.data.max()
y_digits = digits.target

n_samples = len(X_digits)

X_train = X_digits[:int(.9 * n_samples)]
y_train = y_digits[:int(.9 * n_samples)]
X_test = X_digits[int(.9 * n_samples):]
y_test = y_digits[int(.9 * n_samples):]

knn = neighbors.KNeighborsClassifier()
logistic = linear_model.LogisticRegression(solver='lbfgs', max_iter=1000,
                                            multi_class='multinomial')

print('KNN score: %f' % knn.fit(X_train, y_train).score(X_test, y_test))
print('LogisticRegression score: %f'
      % logistic.fit(X_train, y_train).score(X_test, y_test))
```

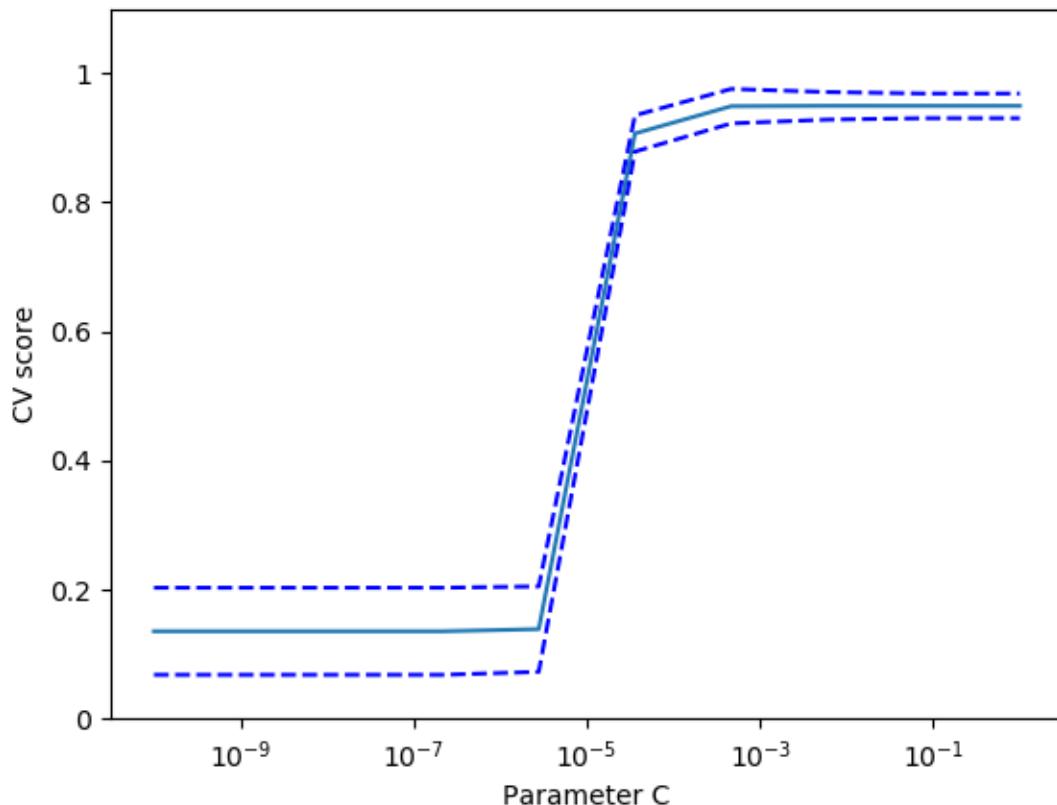
Total running time of the script: (0 minutes 0.432 seconds)

Note: Click [here](#) to download the full example code

5.13.2 Cross-validation on Digits Dataset Exercise

A tutorial exercise using Cross-validation with an SVM on the Digits dataset.

This exercise is used in the [Cross-validation generators](#) part of the [Model selection: choosing estimators and their parameters](#) section of the [A tutorial on statistical-learning for scientific data processing](#).



```
print(__doc__)

import numpy as np
from sklearn.model_selection import cross_val_score
from sklearn import datasets, svm

digits = datasets.load_digits()
X = digits.data
y = digits.target

svc = svm.SVC(kernel='linear')
Cs = np.logspace(-10, 0, 10)

scores = list()
scores_std = list()
for C in Cs:
    svc.C = C
    this_scores = cross_val_score(svc, X, y, cv=5, n_jobs=1)
    scores.append(np.mean(this_scores))
    scores_std.append(np.std(this_scores))
```

```
scores.append(np.mean(this_scores))
scores_std.append(np.std(this_scores))

# Do the plotting
import matplotlib.pyplot as plt
plt.figure()
plt.semilogx(C_s, scores)
plt.semilogx(C_s, np.array(scores) + np.array(scores_std), 'b--')
plt.semilogx(C_s, np.array(scores) - np.array(scores_std), 'b--')
locs, labels = plt.yticks()
plt.yticks(locs, list(map(lambda x: "%g" % x, locs)))
plt.ylabel('CV score')
plt.xlabel('Parameter C')
plt.ylim(0, 1.1)
plt.show()
```

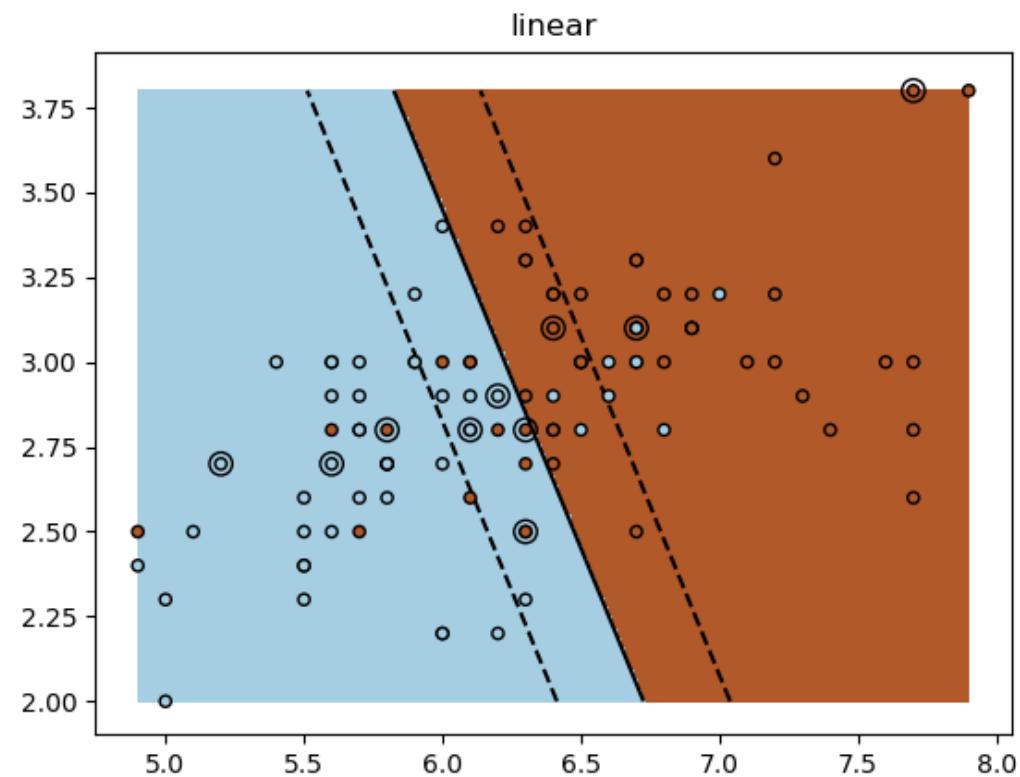
Total running time of the script: (0 minutes 8.826 seconds)

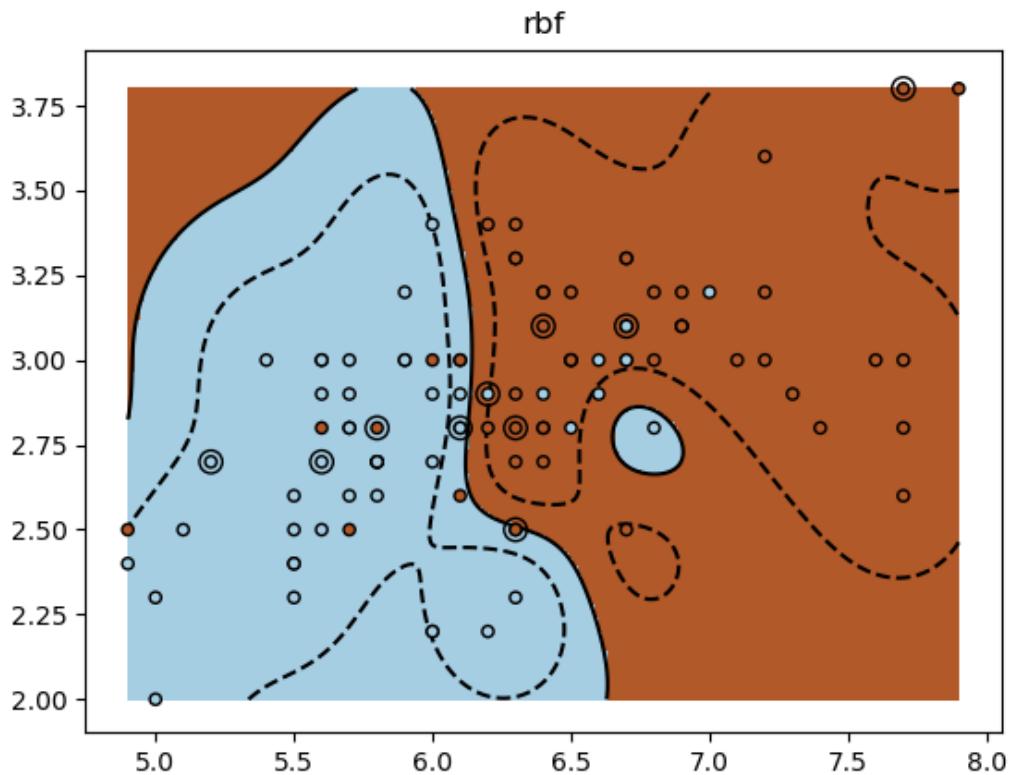
Note: Click [here](#) to download the full example code

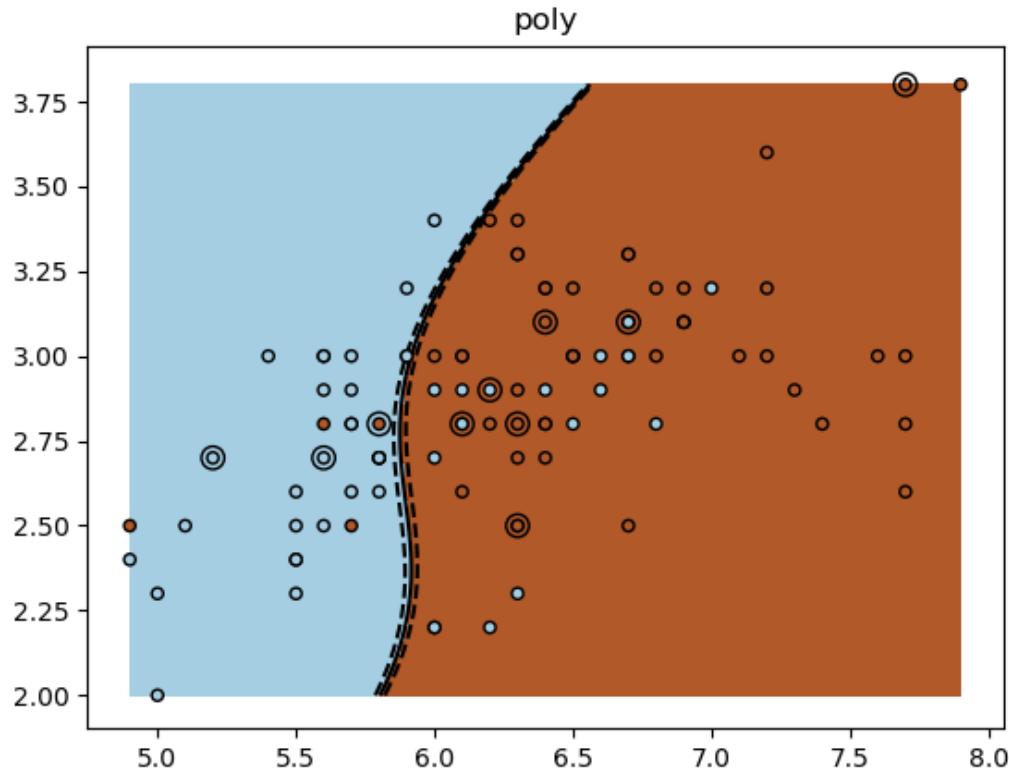
5.13.3 SVM Exercise

A tutorial exercise for using different SVM kernels.

This exercise is used in the [Using kernels](#) part of the [Supervised learning: predicting an output variable from high-dimensional observations](#) section of the [A tutorial on statistical-learning for scientific data processing](#).







```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets, svm

iris = datasets.load_iris()
X = iris.data
y = iris.target

X = X[y != 0, :2]
y = y[y != 0]

n_sample = len(X)

np.random.seed(0)
order = np.random.permutation(n_sample)
X = X[order]
y = y[order].astype(np.float)

X_train = X[:int(.9 * n_sample)]
y_train = y[:int(.9 * n_sample)]
X_test = X[int(.9 * n_sample):]
y_test = y[int(.9 * n_sample):]

# fit the model

```

```
for kernel in ('linear', 'rbf', 'poly'):
    clf = svm.SVC(kernel=kernel, gamma=10)
    clf.fit(X_train, y_train)

    plt.figure()
    plt.clf()
    plt.scatter(X[:, 0], X[:, 1], c=y, zorder=10, cmap=plt.cm.Paired,
                edgecolor='k', s=20)

    # Circle out the test data
    plt.scatter(X_test[:, 0], X_test[:, 1], s=80, facecolors='none',
                zorder=10, edgecolor='k')

    plt.axis('tight')
    x_min = X[:, 0].min()
    x_max = X[:, 0].max()
    y_min = X[:, 1].min()
    y_max = X[:, 1].max()

    XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]
    Z = clf.decision_function(np.c_[XX.ravel(), YY.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(XX.shape)
    plt.pcolormesh(XX, YY, Z > 0, cmap=plt.cm.Paired)
    plt.contour(XX, YY, Z, colors=['k', 'k', 'k'],
                linestyles=[ '--', '-', '--'], levels=[-.5, 0, .5])

    plt.title(kernel)
plt.show()
```

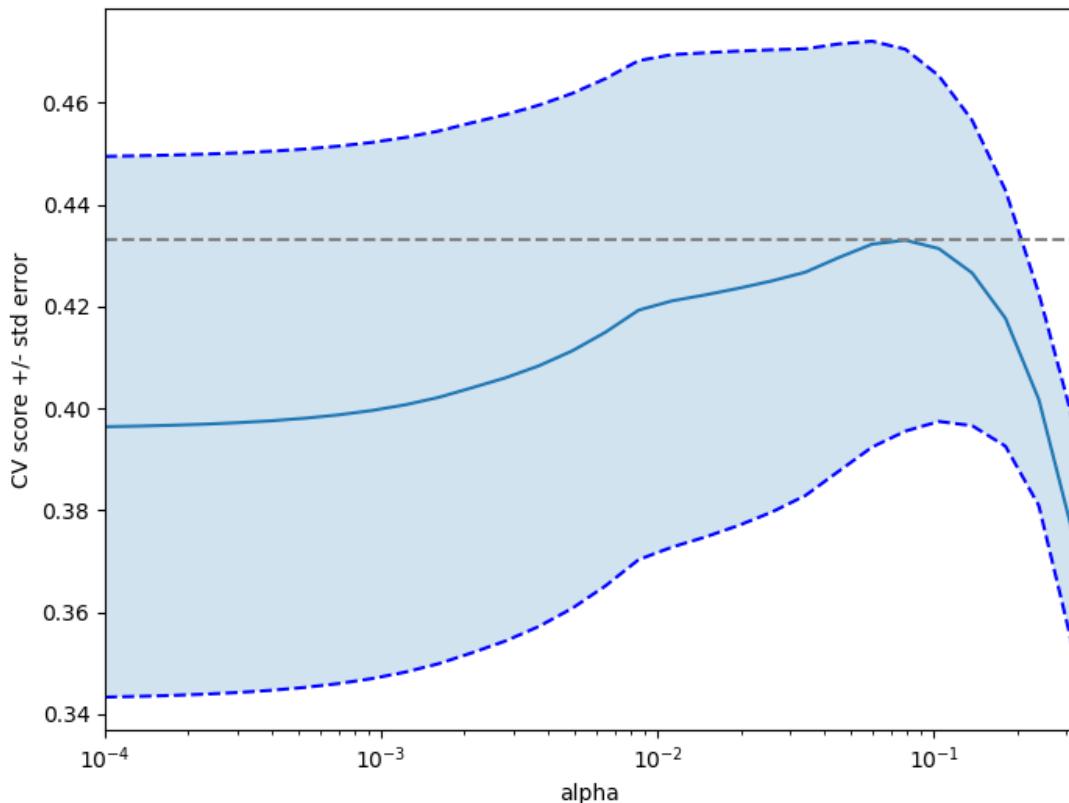
Total running time of the script: (0 minutes 5.320 seconds)

Note: Click [here](#) to download the full example code

5.13.4 Cross-validation on diabetes Dataset Exercise

A tutorial exercise which uses cross-validation with linear models.

This exercise is used in the *Cross-validated estimators* part of the *Model selection: choosing estimators and their parameters* section of the *A tutorial on statistical-learning for scientific data processing*.



Out:

Answer to the bonus question: how much can you trust the selection of alpha?

Alpha parameters maximising the generalization score on different subsets of the data:

```
[fold 0] alpha: 0.05968, score: 0.54209
[fold 1] alpha: 0.04520, score: 0.15523
[fold 2] alpha: 0.07880, score: 0.45193
```

Answer: Not very much since we obtained different alphas for different subsets of the data and moreover, the scores for these alphas differ quite substantially.

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.linear_model import LassoCV
```

```

from sklearn.linear_model import Lasso
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV

diabetes = datasets.load_diabetes()
X = diabetes.data[:150]
y = diabetes.target[:150]

lasso = Lasso(random_state=0, max_iter=10000)
alphas = np.logspace(-4, -0.5, 30)

tuned_parameters = [{alpha: alphas}]
n_folds = 5

clf = GridSearchCV(lasso, tuned_parameters, cv=n_folds, refit=False)
clf.fit(X, y)
scores = clf.cv_results_['mean_test_score']
scores_std = clf.cv_results_['std_test_score']
plt.figure().set_size_inches(8, 6)
plt.semilogx(alphas, scores)

# plot error lines showing +/- std. errors of the scores
std_error = scores_std / np.sqrt(n_folds)

plt.semilogx(alphas, scores + std_error, 'b--')
plt.semilogx(alphas, scores - std_error, 'b--')

# alpha=0.2 controls the translucency of the fill color
plt.fill_between(alphas, scores + std_error, scores - std_error, alpha=0.2)

plt.ylabel('CV score +/- std error')
plt.xlabel('alpha')
plt.axhline(np.max(scores), linestyle='--', color='.5')
plt.xlim([alphas[0], alphas[-1]])

#####
# Bonus: how much can you trust the selection of alpha?

# To answer this question we use the LassoCV object that sets its alpha
# parameter automatically from the data by internal cross-validation (i.e. it
# performs cross-validation on the training data it receives).
# We use external cross-validation to see how much the automatically obtained
# alphas differ across different cross-validation folds.
lasso_cv = LassoCV(alphas=alphas, cv=5, random_state=0, max_iter=10000)
k_fold = KFold(3)

print("Answer to the bonus question:",
      "how much can you trust the selection of alpha?")
print()
print("Alpha parameters maximising the generalization score on different")
print("subsets of the data:")
for k, (train, test) in enumerate(k_fold.split(X, y)):
    lasso_cv.fit(X[train], y[train])
    print("[fold {0}] alpha: {1:.5f}, score: {2:.5f}".
          format(k, lasso_cv.alpha_, lasso_cv.score(X[test], y[test])))
print()
print("Answer: Not very much since we obtained different alphas for different")
print("subsets of the data and moreover, the scores for these alphas differ")

```

```
print("quite substantially.")  
plt.show()
```

Total running time of the script: (0 minutes 0.294 seconds)

5.14 Feature Selection

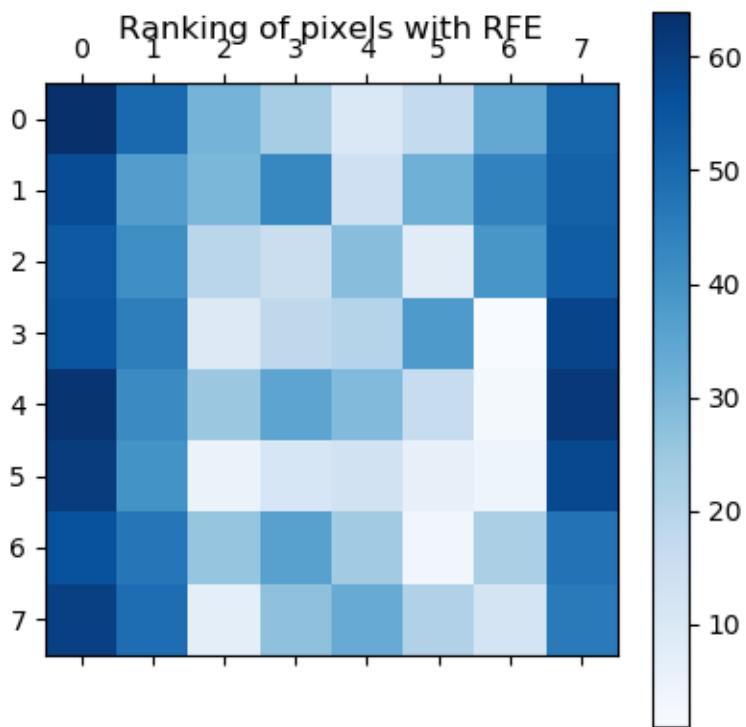
Examples concerning the `sklearn.feature_selection` module.

Note: Click [here](#) to download the full example code

5.14.1 Recursive feature elimination

A recursive feature elimination example showing the relevance of pixels in a digit classification task.

Note: See also [Recursive feature elimination with cross-validation](#)



```
print(__doc__)

from sklearn.svm import SVC
from sklearn.datasets import load_digits
from sklearn.feature_selection import RFE
import matplotlib.pyplot as plt

# Load the digits dataset
digits = load_digits()
X = digits.images.reshape((len(digits.images), -1))
y = digits.target

# Create the RFE object and rank each pixel
svc = SVC(kernel="linear", C=1)
rfe = RFE(estimator=svc, n_features_to_select=1, step=1)
rfe.fit(X, y)
ranking = rfe.ranking_.reshape(digits.images[0].shape)

# Plot pixel ranking
plt.matshow(ranking, cmap=plt.cm.Blues)
plt.colorbar()
plt.title("Ranking of pixels with RFE")
plt.show()
```

Total running time of the script: (0 minutes 3.436 seconds)

Note: Click [here](#) to download the full example code

5.14.2 Comparison of F-test and mutual information

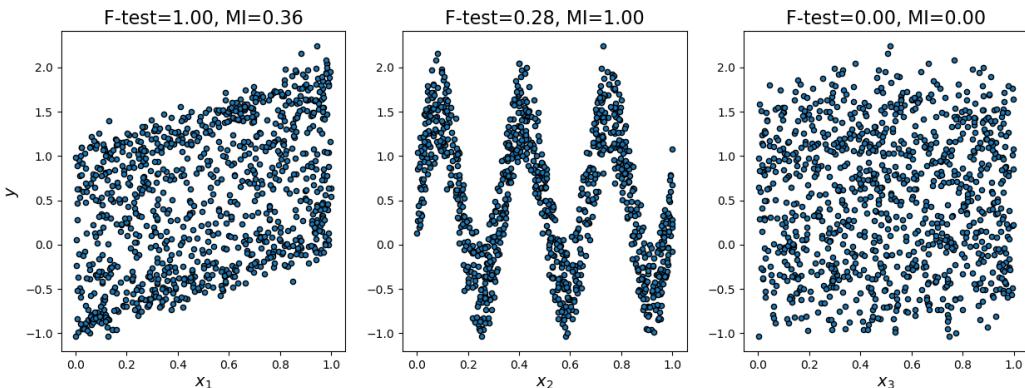
This example illustrates the differences between univariate F-test statistics and mutual information.

We consider 3 features x_{-1}, x_{-2}, x_{-3} distributed uniformly over $[0, 1]$, the target depends on them as follows:

$y = x_{-1} + \sin(6 * \pi * x_{-2}) + 0.1 * N(0, 1)$, that is the third features is completely irrelevant.

The code below plots the dependency of y against individual x_i and normalized values of univariate F-tests statistics and mutual information.

As F-test captures only linear dependency, it rates x_{-1} as the most discriminative feature. On the other hand, mutual information can capture any kind of dependency between variables and it rates x_{-2} as the most discriminative feature, which probably agrees better with our intuitive perception for this example. Both methods correctly marks x_{-3} as irrelevant.



```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.feature_selection import f_regression, mutual_info_regression

np.random.seed(0)
X = np.random.rand(1000, 3)
y = X[:, 0] + np.sin(6 * np.pi * X[:, 1]) + 0.1 * np.random.randn(1000)

f_test, _ = f_regression(X, y)
f_test /= np.max(f_test)

mi = mutual_info_regression(X, y)
mi /= np.max(mi)

plt.figure(figsize=(15, 5))
for i in range(3):
    plt.subplot(1, 3, i + 1)
    plt.scatter(X[:, i], y, edgecolor='black', s=20)
    plt.xlabel("$x_{}$".format(i + 1), fontsize=14)
    if i == 0:
        plt.ylabel("$y$", fontsize=14)
    plt.title("F-test={:.2f}, MI={:.2f}".format(f_test[i], mi[i]),
              fontsize=16)
plt.show()
```

Total running time of the script: (0 minutes 0.062 seconds)

Note: Click [here](#) to download the full example code

5.14.3 Pipeline Anova SVM

Simple usage of Pipeline that runs successively a univariate feature selection with anova and then a SVM of the selected features.

Using a sub-pipeline, the fitted coefficients can be mapped back into the original feature space.

Out:

precision	recall	f1-score	support
0	0.75	0.50	0.60
1	0.67	1.00	0.80
2	0.67	0.80	0.73
3	1.00	0.75	0.86
accuracy			0.76
macro avg	0.77	0.76	0.75
weighted avg	0.79	0.76	0.76
[[-0.23912131 0.	0.	0.	-0.3236911
0.	0.	0.	0.
0.10836648 0.	0.	0.	0.
0.	0.]	0.
[0.43878747 0.	0.	0.	-0.51415652
0.	0.	0.	0.
0.04845652 0.	0.	0.	0.
0.	0.]	0.
[-0.65382998 0.	0.	0.	0.57962856
0.	0.	0.	0.
-0.04736524 0.	0.	0.	0.
0.	0.]	0.
[0.54403412 0.	0.	0.	0.58478491
0.	0.	0.	0.
-0.11344659 0.	0.	0.	0.
0.	0.]	0.

```

from sklearn import svm
from sklearn.datasets import samples_generator
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

print(__doc__)

# import some data to play with
X, y = samples_generator.make_classification(
    n_features=20, n_informative=3, n_redundant=0, n_classes=4,
    n_clusters_per_class=2)

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

# ANOVA SVM-C
# 1) anova filter, take 3 best ranked features
anova_filter = SelectKBest(f_regression, k=3)
# 2) svm
clf = svm.LinearSVC()

anova_svm = make_pipeline(anova_filter, clf)
anova_svm.fit(X_train, y_train)
y_pred = anova_svm.predict(X_test)

```

```
print(classification_report(y_test, y_pred))

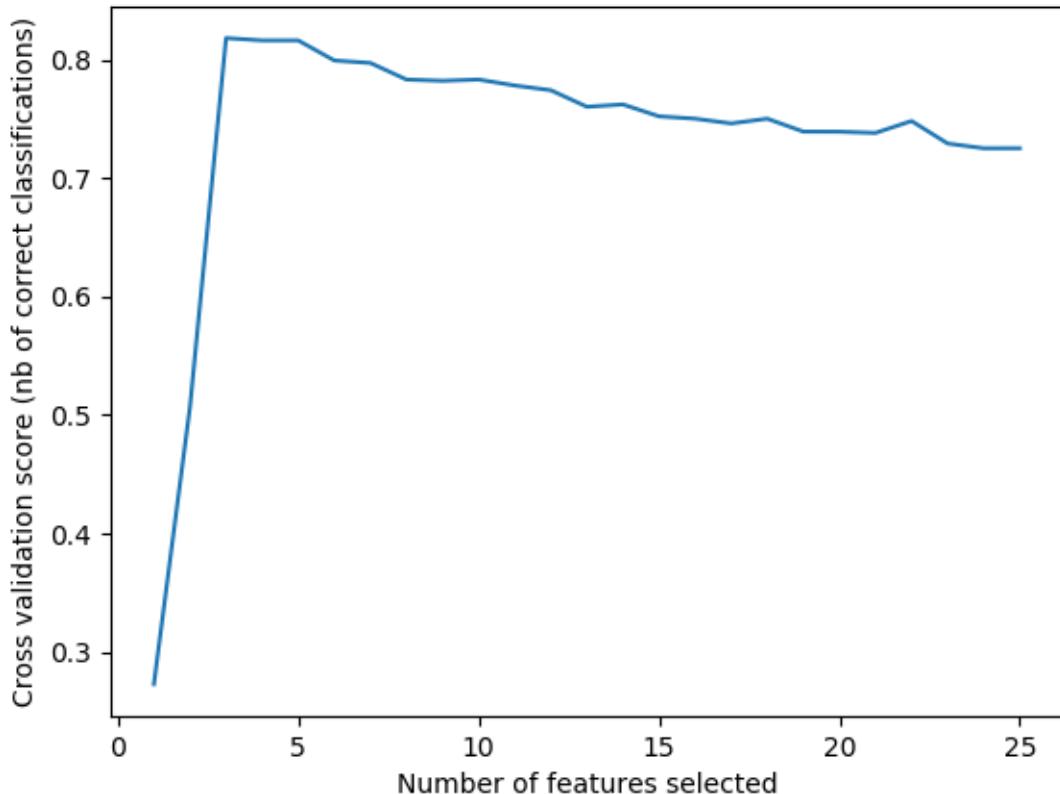
coef = anova_svm[:-1].inverse_transform(anova_svm['linearsvc'].coef_)
print(coef)
```

Total running time of the script: (0 minutes 0.008 seconds)

Note: Click [here](#) to download the full example code

5.14.4 Recursive feature elimination with cross-validation

A recursive feature elimination example with automatic tuning of the number of features selected with cross-validation.



Out:

```
Optimal number of features : 3
```

```
print(__doc__)

import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.model_selection import StratifiedKFold
from sklearn.feature_selection import RFECV
from sklearn.datasets import make_classification

# Build a classification task using 3 informative features
X, y = make_classification(n_samples=1000, n_features=25, n_informative=3,
                           n_redundant=2, n_repeated=0, n_classes=8,
                           n_clusters_per_class=1, random_state=0)

# Create the RFE object and compute a cross-validated score.
svc = SVC(kernel="linear")
# The "accuracy" scoring is proportional to the number of correct
# classifications
rfecv = RFECV(estimator=svc, step=1, cv=StratifiedKFold(2),
              scoring='accuracy')
rfecv.fit(X, y)

print("Optimal number of features : %d" % rfecv.n_features_)

# Plot number of features VS. cross-validation scores
plt.figure()
plt.xlabel("Number of features selected")
plt.ylabel("Cross validation score (nb of correct classifications)")
plt.plot(range(1, len(rfecv.grid_scores_) + 1), rfecv.grid_scores_)
plt.show()
```

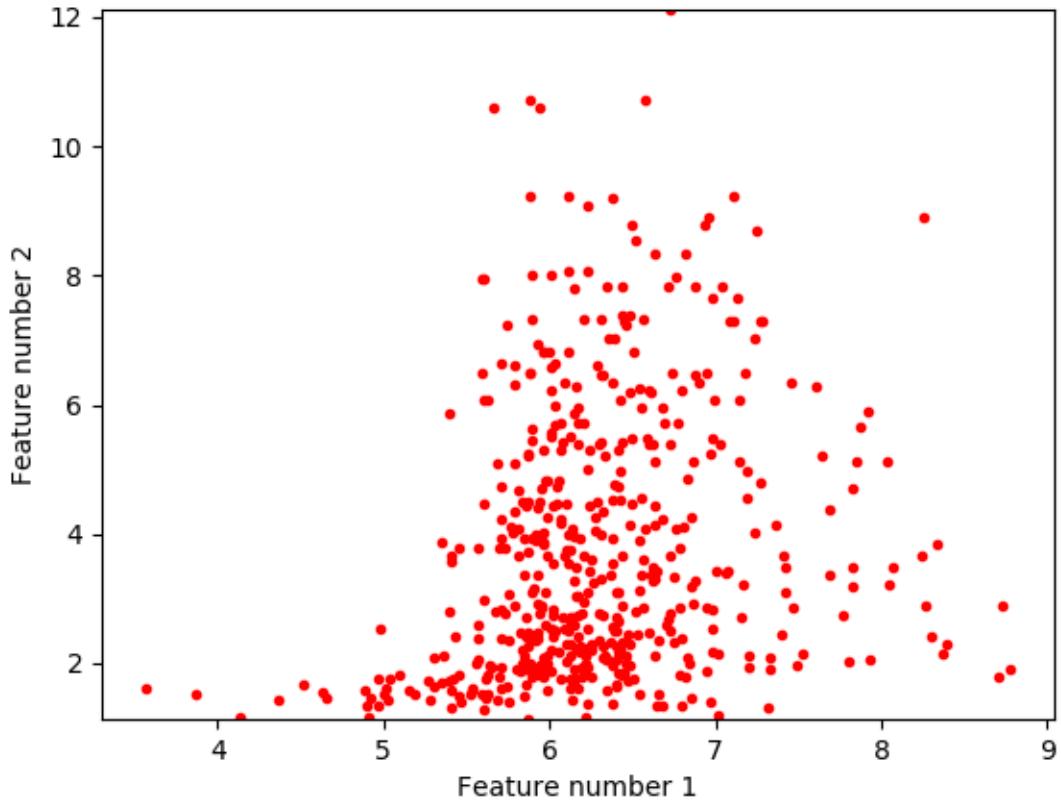
Total running time of the script: (0 minutes 1.806 seconds)

Note: Click [here](#) to download the full example code

5.14.5 Feature selection using SelectFromModel and LassoCV

Use SelectFromModel meta-transformer along with Lasso to select the best couple of features from the Boston dataset.

Features selected from Boston using SelectFromModel with threshold 0.750



```
# Author: Manoj Kumar <mks542@nyu.edu>
# License: BSD 3 clause

print(__doc__)

import matplotlib.pyplot as plt
import numpy as np

from sklearn.datasets import load_boston
from sklearn.feature_selection import SelectFromModel
from sklearn.linear_model import LassoCV

# Load the boston dataset.
boston = load_boston()
X, y = boston['data'], boston['target']

# We use the base estimator LassoCV since the L1 norm promotes sparsity of features.
clf = LassoCV(cv=5)

# Set a minimum threshold of 0.25
sfm = SelectFromModel(clf, threshold=0.25)
sfm.fit(X, y)
n_features = sfm.transform(X).shape[1]

# Reset the threshold till the number of features equals two.
# Note that the attribute can be set directly instead of repeatedly
```

```
# fitting the metatransformer.
while n_features > 2:
    sfm.threshold += 0.1
    X_transform = sfm.transform(X)
    n_features = X_transform.shape[1]

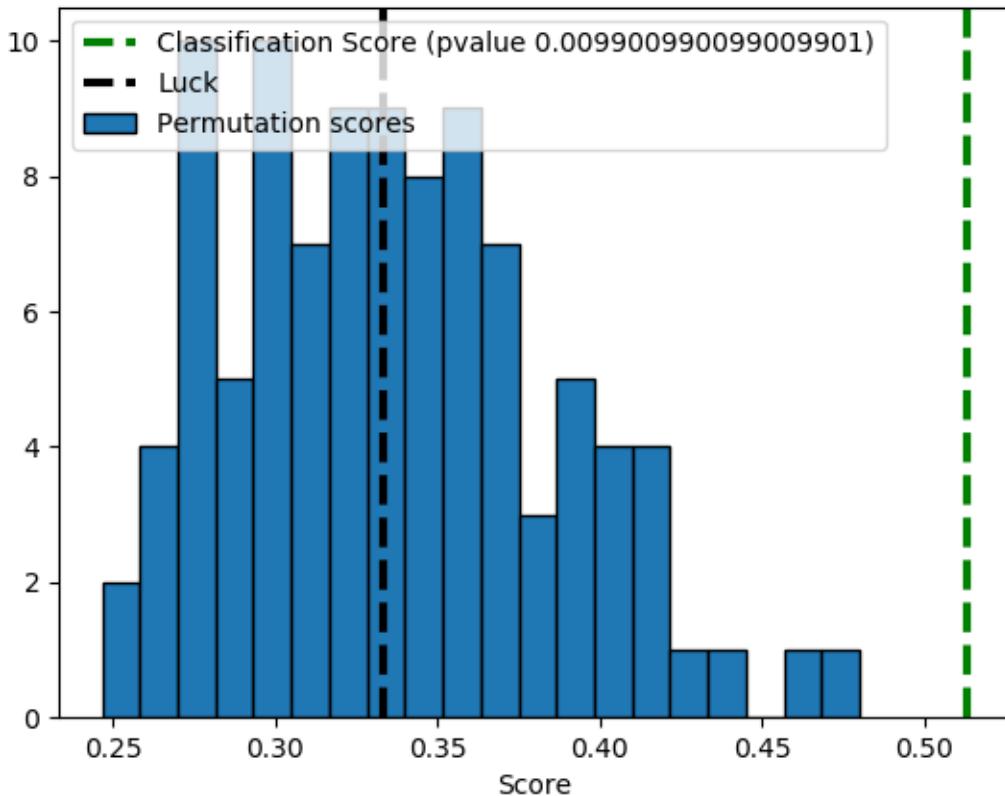
# Plot the selected two features from X.
plt.title(
    "Features selected from Boston using SelectFromModel with "
    "threshold %0.3f." % sfm.threshold)
feature1 = X_transform[:, 0]
feature2 = X_transform[:, 1]
plt.plot(feature1, feature2, 'r.')
plt.xlabel("Feature number 1")
plt.ylabel("Feature number 2")
plt.ylim([np.min(feature2), np.max(feature2)])
plt.show()
```

Total running time of the script: (0 minutes 0.056 seconds)

Note: Click [here](#) to download the full example code

5.14.6 Test with permutations the significance of a classification score

In order to test if a classification score is significative a technique in repeating the classification procedure after randomizing, permuting, the labels. The p-value is then given by the percentage of runs for which the score obtained is greater than the classification score obtained in the first place.



Out:

```
Classification score 0.5133333333333333 (pvalue : 0.009900990099009901)
```

```
# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.svm import SVC
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import permutation_test_score
from sklearn import datasets

#####
# Loading a dataset
iris = datasets.load_iris()
```

```
X = iris.data
y = iris.target
n_classes = np.unique(y).size

# Some noisy data not correlated
random = np.random.RandomState(seed=0)
E = random.normal(size=(len(X), 2200))

# Add noisy data to the informative features for make the task harder
X = np.c_[X, E]

svm = SVC(kernel='linear')
cv = StratifiedKFold(2)

score, permutation_scores, pvalue = permutation_test_score(
    svm, X, y, scoring="accuracy", cv=cv, n_permutations=100, n_jobs=1)

print("Classification score %s (pvalue : %s)" % (score, pvalue))

#####
# View histogram of permutation scores
plt.hist(permutation_scores, 20, label='Permutation scores',
         edgecolor='black')
ylim = plt.ylim()
# BUG: vlines(..., linestyle='--') fails on older versions of matplotlib
# plt.vlines(score, ylim[0], ylim[1], linestyle='--',
#             color='g', linewidth=3, label='Classification Score'
#             '(pvalue %s)' % pvalue)
# plt.vlines(1.0 / n_classes, ylim[0], ylim[1], linestyle='--',
#             color='k', linewidth=3, label='Luck')
plt.plot(2 * [score], ylim, '--g', linewidth=3,
         label='Classification Score'
         '(pvalue %s)' % pvalue)
plt.plot(2 * [1. / n_classes], ylim, '--k', linewidth=3, label='Luck')

plt.ylim(ylim)
plt.legend()
plt.xlabel('Score')
plt.show()
```

Total running time of the script: (0 minutes 7.883 seconds)

Note: Click [here](#) to download the full example code

5.14.7 Univariate Feature Selection

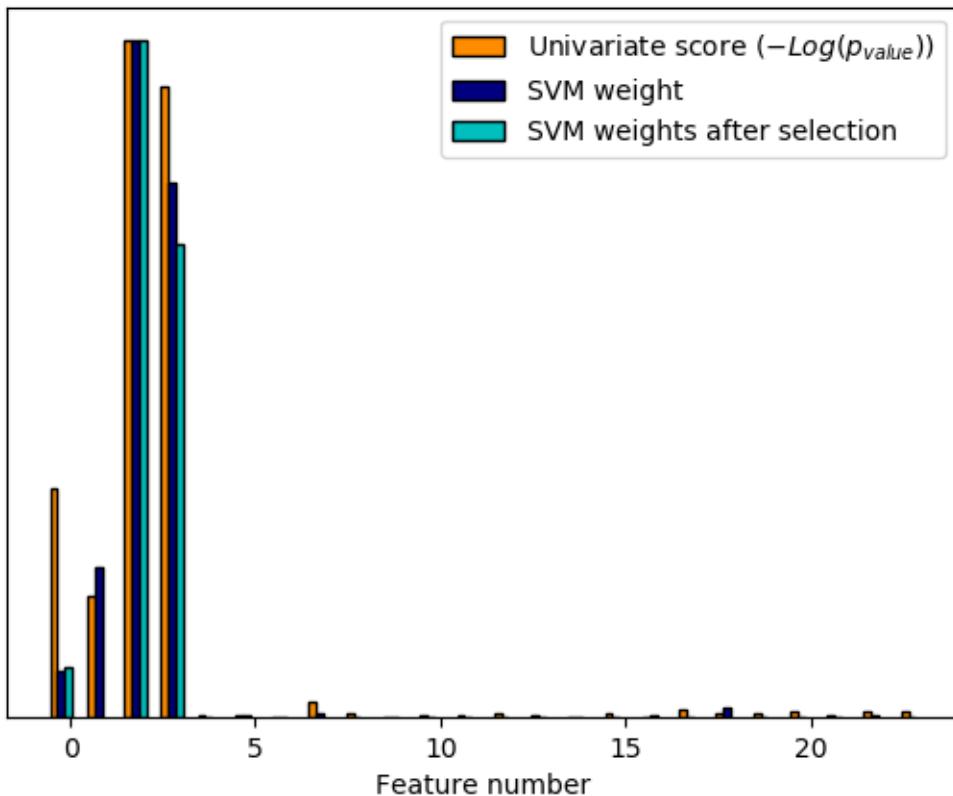
An example showing univariate feature selection.

Noisy (non informative) features are added to the iris data and univariate feature selection is applied. For each feature, we plot the p-values for the univariate feature selection and the corresponding weights of an SVM. We can see that univariate feature selection selects the informative features and that these have larger SVM weights.

In the total set of features, only the 4 first ones are significant. We can see that they have the highest score with univariate feature selection. The SVM assigns a large weight to one of these features, but also Selects many of the non-informative features. Applying univariate feature selection before the SVM increases the SVM weight attributed

to the significant features, and will thus improve classification.

Comparing feature selection



```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn import datasets, svm
from sklearn.feature_selection import SelectPercentile, f_classif

# ##### Import some data to play with

# The iris dataset
iris = datasets.load_iris()

# Some noisy data not correlated
E = np.random.uniform(0, 0.1, size=(len(iris.data), 20))

# Add the noisy data to the informative features
X = np.hstack((iris.data, E))
y = iris.target

plt.figure(1)
plt.clf()
```

```
X_indices = np.arange(X.shape[-1])

# ##### Univariate feature selection with F-test for feature scoring
# We use the default selection function: the 10% most significant features
selector = SelectPercentile(f_classif, percentile=10)
selector.fit(X, y)
scores = -np.log10(selector.pvalues_)
scores /= scores.max()
plt.bar(X_indices - .45, scores, width=.2,
        label=r'Univariate score ($-\log(p_{\text{value}})$)', color='darkorange',
        edgecolor='black')

# ##### Compare to the weights of an SVM
clf = svm.SVC(kernel='linear')
clf.fit(X, y)

svm_weights = (clf.coef_ ** 2).sum(axis=0)
svm_weights /= svm_weights.max()

plt.bar(X_indices - .25, svm_weights, width=.2, label='SVM weight',
        color='navy', edgecolor='black')

clf_selected = svm.SVC(kernel='linear')
clf_selected.fit(selector.transform(X), y)

svm_weights_selected = (clf_selected.coef_ ** 2).sum(axis=0)
svm_weights_selected /= svm_weights_selected.max()

plt.bar(X_indices[selector.get_support()] - .05, svm_weights_selected,
        width=.2, label='SVM weights after selection', color='c',
        edgecolor='black')

plt.title("Comparing feature selection")
plt.xlabel('Feature number')
plt.yticks(())
plt.axis('tight')
plt.legend(loc='upper right')
plt.show()
```

Total running time of the script: (0 minutes 0.045 seconds)

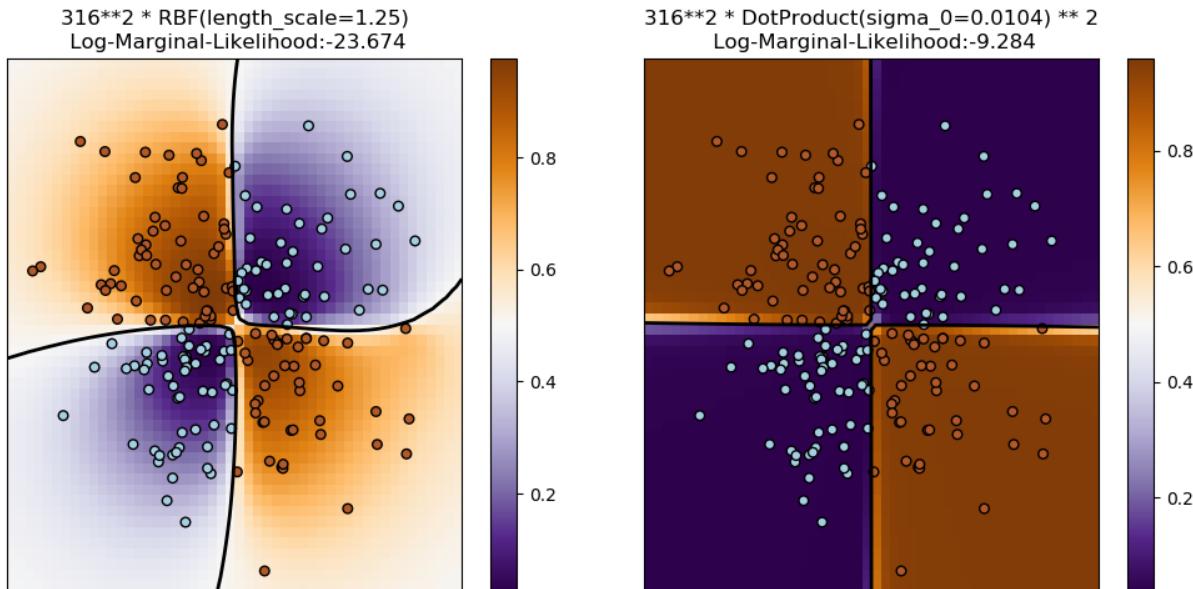
5.15 Gaussian Process for Machine Learning

Examples concerning the `sklearn.gaussian_process` module.

Note: Click [here](#) to download the full example code

5.15.1 Illustration of Gaussian process classification (GPC) on the XOR dataset

This example illustrates GPC on XOR data. Compared are a stationary, isotropic kernel (RBF) and a non-stationary kernel (DotProduct). On this particular dataset, the DotProduct kernel obtains considerably better results because the class-boundaries are linear and coincide with the coordinate axes. In general, stationary kernels often obtain better results.



```
print(__doc__)

# Authors: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
#
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF, DotProduct

xx, yy = np.meshgrid(np.linspace(-3, 3, 50),
                     np.linspace(-3, 3, 50))
rng = np.random.RandomState(0)
X = rng.randn(200, 2)
Y = np.logical_xor(X[:, 0] > 0, X[:, 1] > 0)

# fit the model
plt.figure(figsize=(10, 5))
kernels = [1.0 * RBF(length_scale=1.0), 1.0 * DotProduct(sigma_0=1.0)**2]
for i, kernel in enumerate(kernels):
    clf = GaussianProcessClassifier(kernel=kernel, warm_start=True).fit(X, Y)

    # plot the decision function for each datapoint on the grid
    Z = clf.predict_proba(np.vstack((xx.ravel(), yy.ravel())).T)[:, 1]
    Z = Z.reshape(xx.shape)
```

```

plt.subplot(1, 2, i + 1)
image = plt.imshow(Z, interpolation='nearest',
                    extent=(xx.min(), xx.max(), yy.min(), yy.max()),
                    aspect='auto', origin='lower', cmap=plt.cm.PuOr_r)
contours = plt.contour(xx, yy, Z, levels=[0.5], linewidths=2,
                      colors=['k'])
plt.scatter(X[:, 0], X[:, 1], s=30, c=Y, cmap=plt.cm.Paired,
            edgecolors=(0, 0, 0))
plt.xticks(())
plt.yticks(())
plt.axis([-3, 3, -3, 3])
plt.colorbar(image)
plt.title("%s\n Log-Marginal-Likelihood: %.3f"
          % (clf.kernel_, clf.log_marginal_likelihood(clf.kernel_.theta)),
          fontsize=12)

plt.tight_layout()
plt.show()

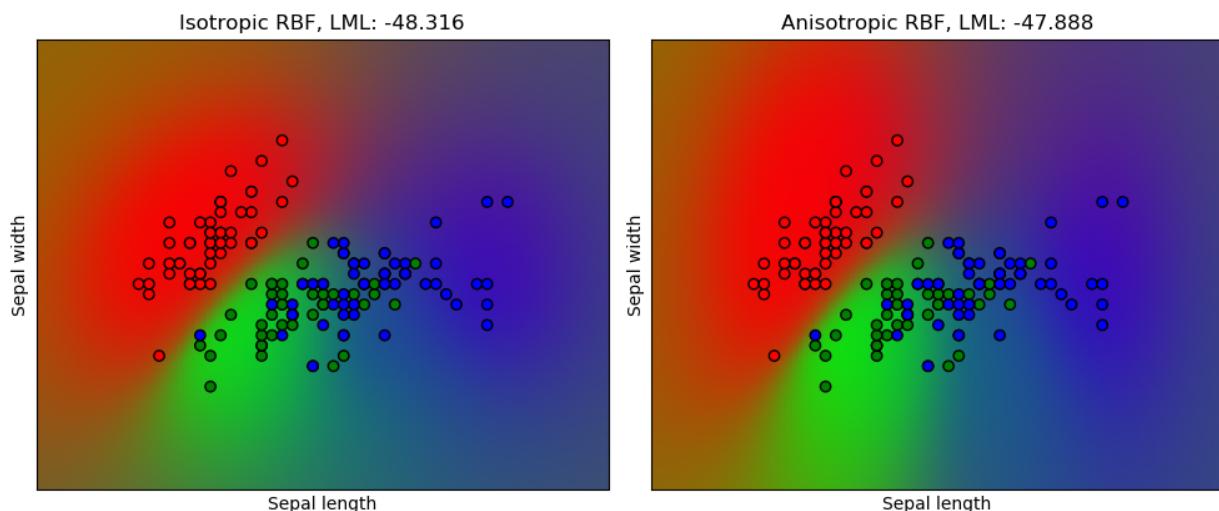
```

Total running time of the script: (0 minutes 0.686 seconds)

Note: Click [here](#) to download the full example code

5.15.2 Gaussian process classification (GPC) on iris dataset

This example illustrates the predicted probability of GPC for an isotropic and anisotropic RBF kernel on a two-dimensional version for the iris-dataset. The anisotropic RBF kernel obtains slightly higher log-marginal-likelihood by assigning different length-scales to the two feature dimensions.



```

print(__doc__)

import numpy as np

```

```

import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF

# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2] # we only take the first two features.
y = np.array(iris.target, dtype=int)

h = .02 # step size in the mesh

kernel = 1.0 * RBF([1.0])
gpc_rbf_isotropic = GaussianProcessClassifier(kernel=kernel).fit(X, y)
kernel = 1.0 * RBF([1.0, 1.0])
gpc_rbf_anisotropic = GaussianProcessClassifier(kernel=kernel).fit(X, y)

# create a mesh to plot in
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                      np.arange(y_min, y_max, h))

titles = ["Isotropic RBF", "Anisotropic RBF"]
plt.figure(figsize=(10, 5))
for i, clf in enumerate((gpc_rbf_isotropic, gpc_rbf_anisotropic)):
    # Plot the predicted probabilities. For that, we will assign a color to
    # each point in the mesh [x_min, m_max]x[y_min, y_max].
    plt.subplot(1, 2, i + 1)

    Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape((xx.shape[0], xx.shape[1], 3))
    plt.imshow(Z, extent=(x_min, x_max, y_min, y_max), origin="lower")

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=np.array(["r", "g", "b"])[y],
                edgecolors=(0, 0, 0))
    plt.xlabel('Sepal length')
    plt.ylabel('Sepal width')
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.xticks(())
    plt.yticks(())
    plt.title("%s, LML: %.3f" %
              (titles[i], clf.log_marginal_likelihood(clf.kernel_.theta)))

plt.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 4.265 seconds)

Note: Click [here](#) to download the full example code

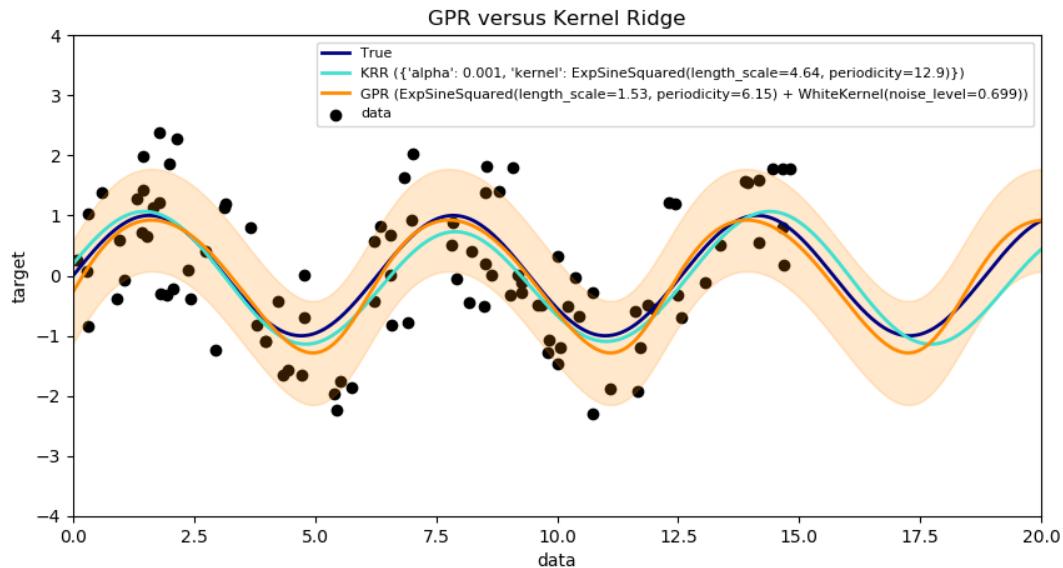
5.15.3 Comparison of kernel ridge and Gaussian process regression

Both kernel ridge regression (KRR) and Gaussian process regression (GPR) learn a target function by employing internally the “kernel trick”. KRR learns a linear function in the space induced by the respective kernel which corresponds to a non-linear function in the original space. The linear function in the kernel space is chosen based on the mean-squared error loss with ridge regularization. GPR uses the kernel to define the covariance of a prior distribution over the target functions and uses the observed training data to define a likelihood function. Based on Bayes theorem, a (Gaussian) posterior distribution over target functions is defined, whose mean is used for prediction.

A major difference is that GPR can choose the kernel’s hyperparameters based on gradient-ascent on the marginal likelihood function while KRR needs to perform a grid search on a cross-validated loss function (mean-squared error loss). A further difference is that GPR learns a generative, probabilistic model of the target function and can thus provide meaningful confidence intervals and posterior samples along with the predictions while KRR only provides predictions.

This example illustrates both methods on an artificial dataset, which consists of a sinusoidal target function and strong noise. The figure compares the learned model of KRR and GPR based on a ExpSineSquared kernel, which is suited for learning periodic functions. The kernel’s hyperparameters control the smoothness (l) and periodicity of the kernel (p). Moreover, the noise level of the data is learned explicitly by GPR by an additional WhiteKernel component in the kernel and by the regularization parameter alpha of KRR.

The figure shows that both methods learn reasonable models of the target function. GPR correctly identifies the periodicity of the function to be roughly 2π (6.28), while KRR chooses the doubled periodicity 4π . Besides that, GPR provides reasonable confidence bounds on the prediction which are not available for KRR. A major difference between the two methods is the time required for fitting and predicting: while fitting KRR is fast in principle, the grid-search for hyperparameter optimization scales exponentially with the number of hyperparameters (“curse of dimensionality”). The gradient-based optimization of the parameters in GPR does not suffer from this exponential scaling and is thus considerably faster on this example with 3-dimensional hyperparameter space. The time for predicting is similar; however, generating the variance of the predictive distribution of GPR takes considerable longer than just predicting the mean.



Out:

```
Time for KRR fitting: 3.180
Time for GPR fitting: 0.096
Time for KRR prediction: 0.009
```

```
Time for GPR prediction: 0.010
Time for GPR prediction with standard-deviation: 0.014
```

```
print(__doc__)

# Authors: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
# License: BSD 3 clause


import time

import numpy as np

import matplotlib.pyplot as plt

from sklearn.kernel_ridge import KernelRidge
from sklearn.model_selection import GridSearchCV
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import WhiteKernel, ExpSineSquared

rng = np.random.RandomState(0)

# Generate sample data
X = 15 * rng.rand(100, 1)
y = np.sin(X).ravel()
y += 3 * (0.5 - rng.rand(X.shape[0])) # add noise

# Fit KernelRidge with parameter selection based on 5-fold cross validation
param_grid = {"alpha": [1e0, 1e-1, 1e-2, 1e-3],
              "kernel": [ExpSineSquared(1, p)
                         for l in np.logspace(-2, 2, 10)
                         for p in np.logspace(0, 2, 10)]}
kr = GridSearchCV(KernelRidge(), cv=5, param_grid=param_grid)
stime = time.time()
kr.fit(X, y)
print("Time for KRR fitting: %.3f" % (time.time() - stime))

gp_kernel = ExpSineSquared(1.0, 5.0, periodicity_bounds=(1e-2, 1e1)) \
    + WhiteKernel(1e-1)
gpr = GaussianProcessRegressor(kernel=gp_kernel)
stime = time.time()
gpr.fit(X, y)
print("Time for GPR fitting: %.3f" % (time.time() - stime))

# Predict using kernel ridge
X_plot = np.linspace(0, 20, 10000)[:, None]
stime = time.time()
y_kr = kr.predict(X_plot)
print("Time for KRR prediction: %.3f" % (time.time() - stime))

# Predict using gaussian process regressor
stime = time.time()
y_gpr = gpr.predict(X_plot, return_std=False)
```

```
print("Time for GPR prediction: %.3f" % (time.time() - stime))

stime = time.time()
y_gpr, y_std = gpr.predict(X_plot, return_std=True)
print("Time for GPR prediction with standard-deviation: %.3f"
      % (time.time() - stime))

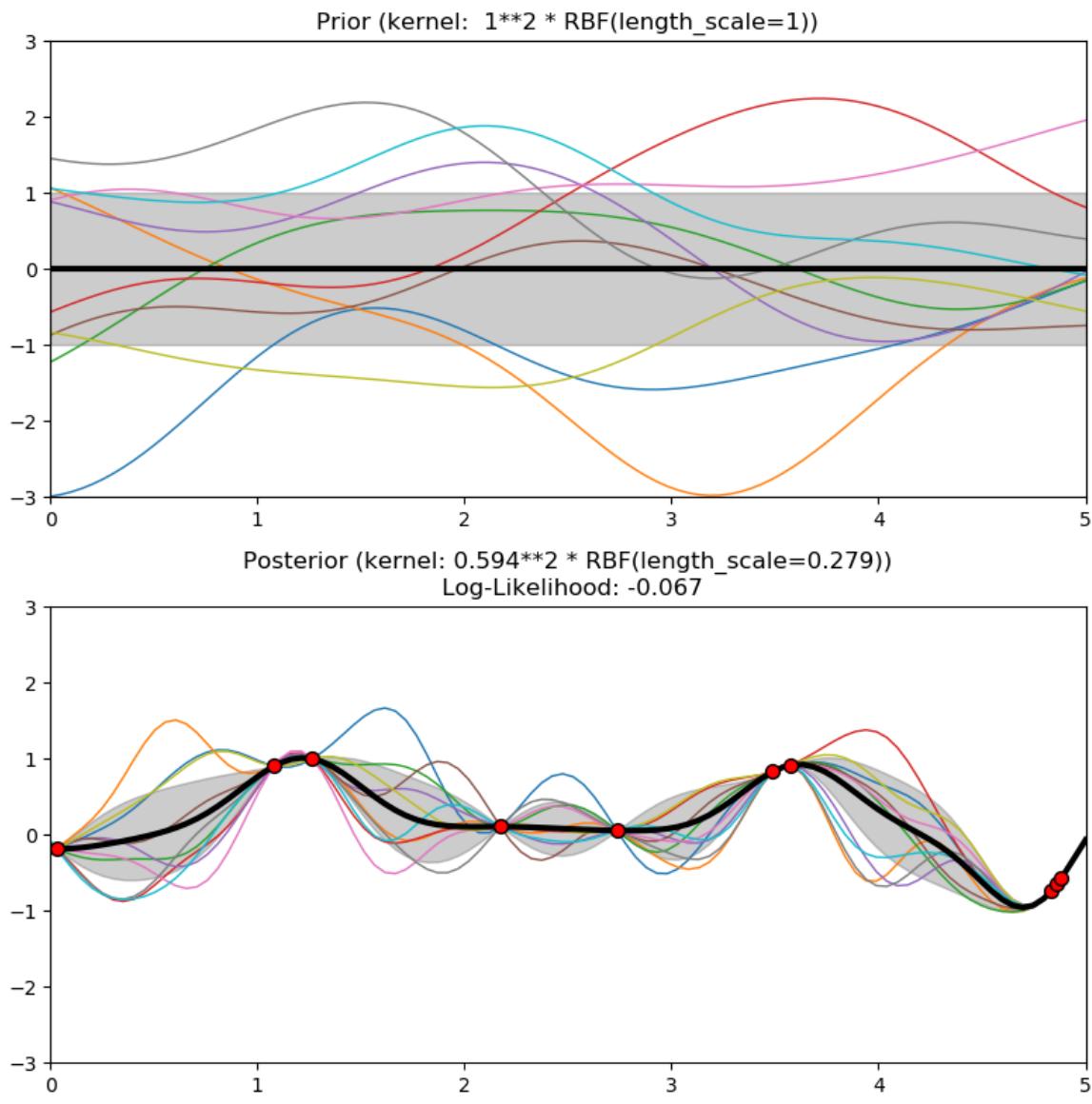
# Plot results
plt.figure(figsize=(10, 5))
lw = 2
plt.scatter(X, y, c='k', label='data')
plt.plot(X_plot, np.sin(X_plot), color='navy', lw=lw, label='True')
plt.plot(X_plot, y_kr, color='turquoise', lw=lw,
         label='KRR (%s)' % kr.best_params_)
plt.plot(X_plot, y_gpr, color='darkorange', lw=lw,
         label='GPR (%s)' % gpr.kernel_)
plt.fill_between(X_plot[:, 0], y_gpr - y_std, y_gpr + y_std, color='darkorange',
                 alpha=0.2)
plt.xlabel('data')
plt.ylabel('target')
plt.xlim(0, 20)
plt.ylim(-4, 4)
plt.title('GPR versus Kernel Ridge')
plt.legend(loc="best", scatterpoints=1, prop={'size': 8})
plt.show()
```

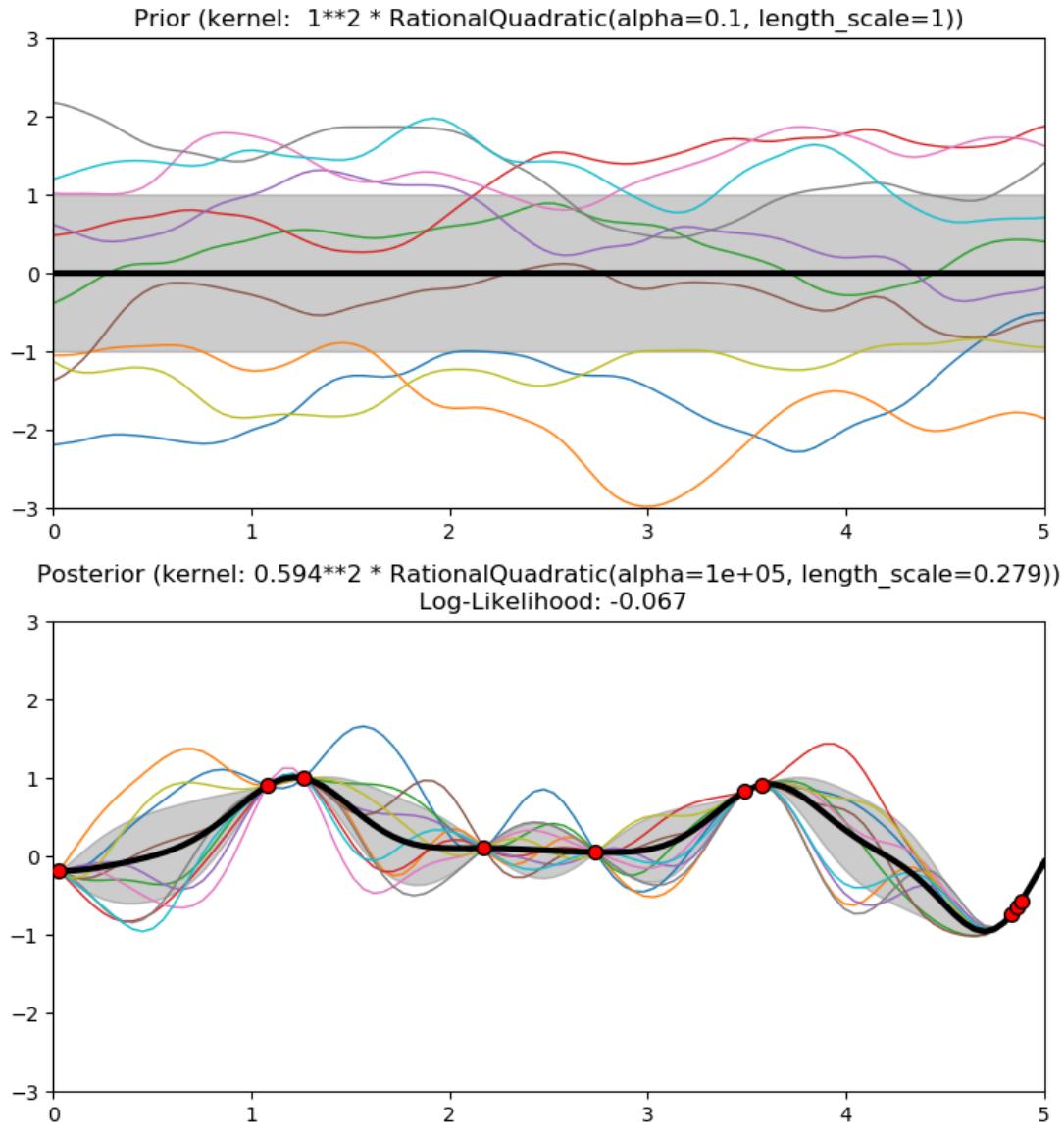
Total running time of the script: (0 minutes 3.377 seconds)

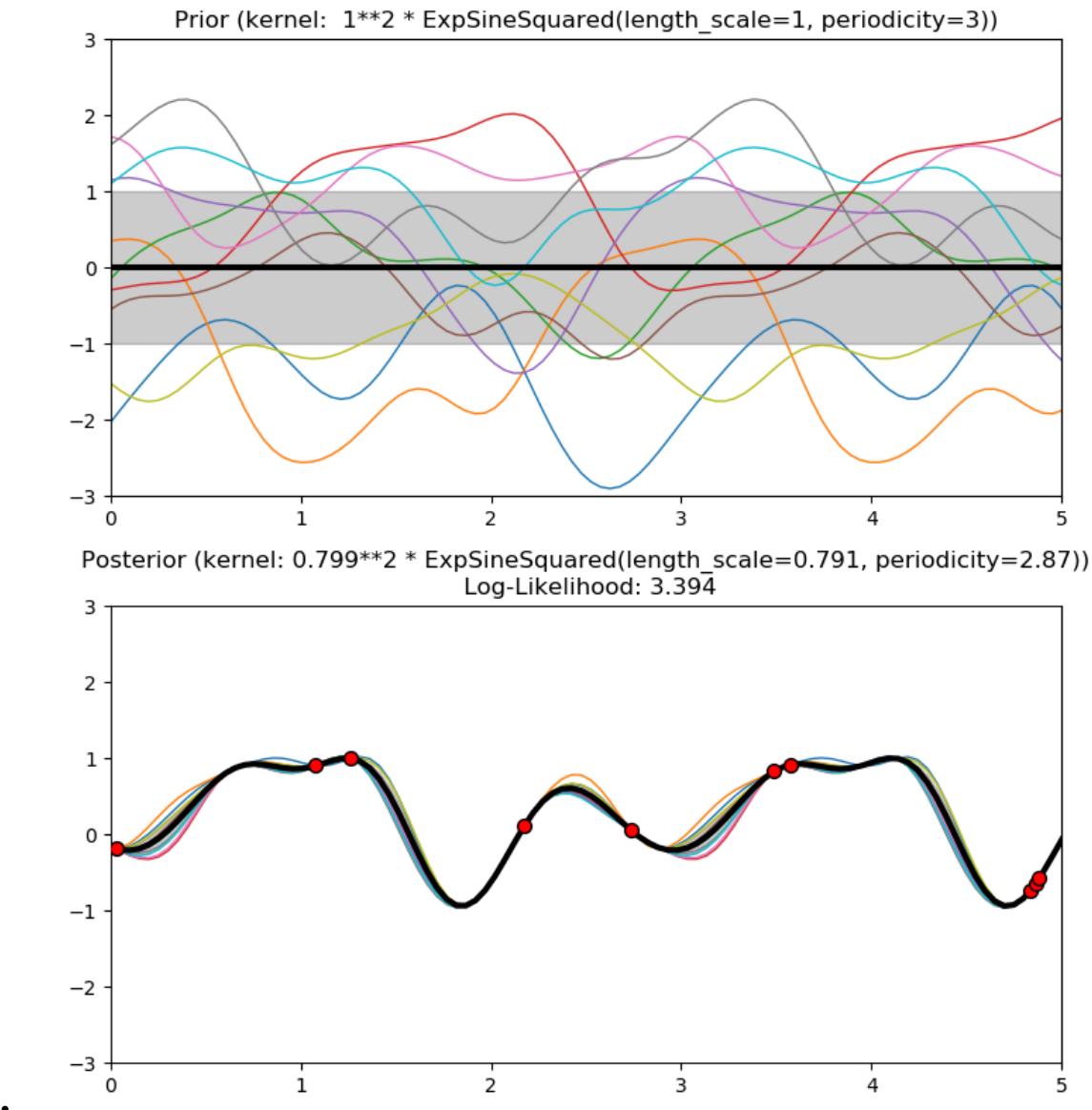
Note: Click [here](#) to download the full example code

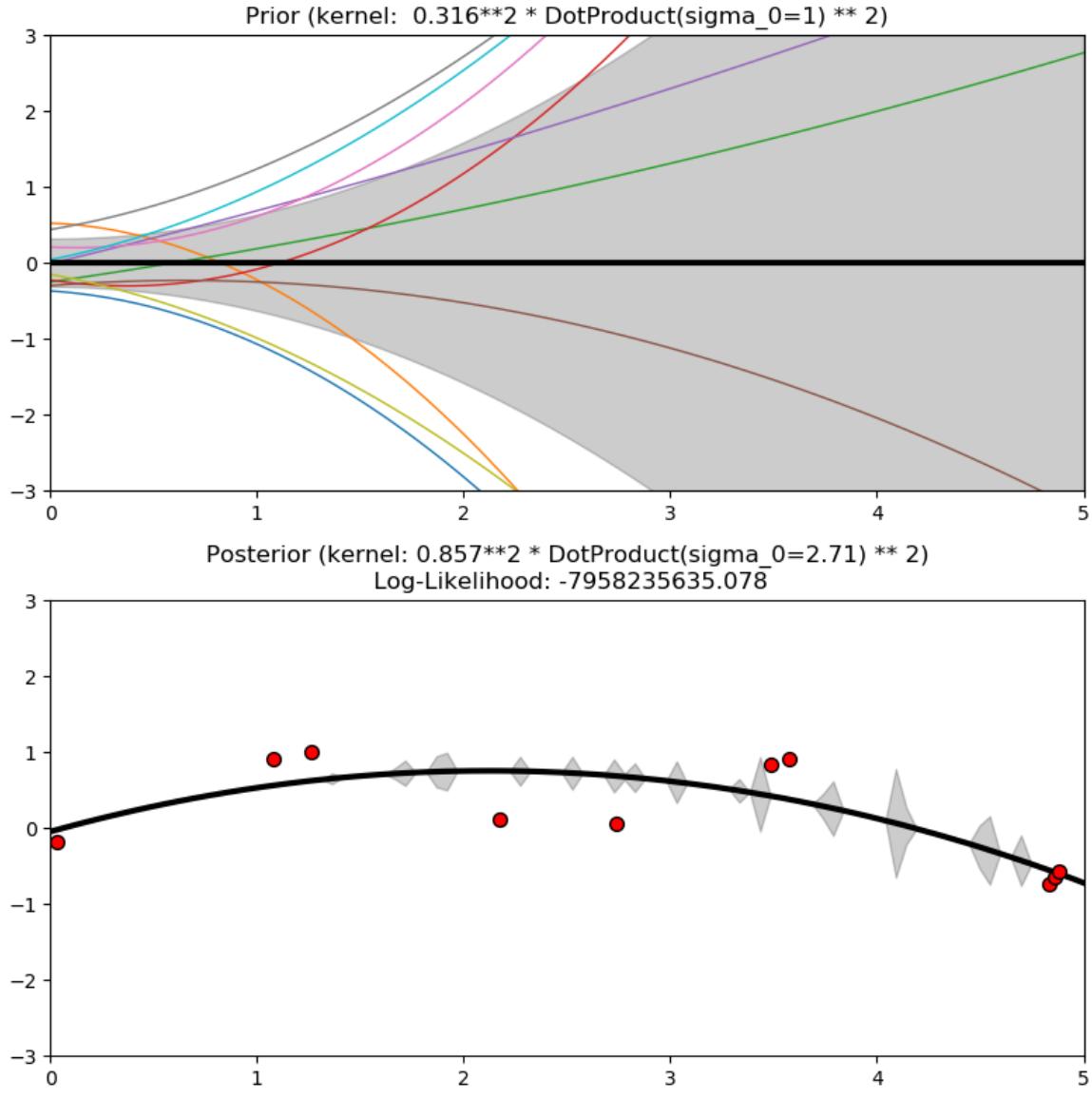
5.15.4 Illustration of prior and posterior Gaussian process for different kernels

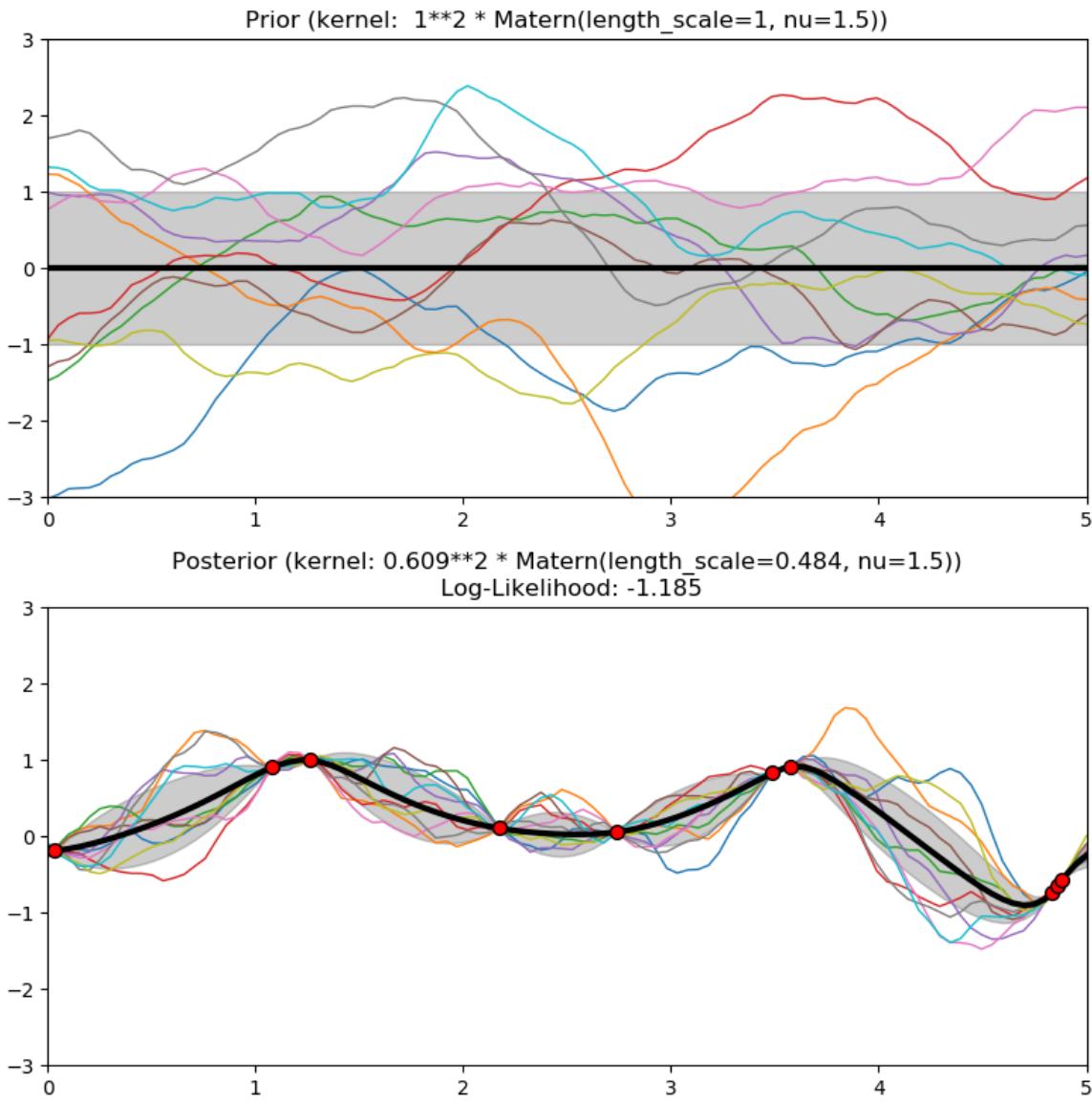
This example illustrates the prior and posterior of a GPR with different kernels. Mean, standard deviation, and 10 samples are shown for both prior and posterior.











```

print(__doc__)

# Authors: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
#
# License: BSD 3 clause

import numpy as np

from matplotlib import pyplot as plt

from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import (RBF, Matern, RationalQuadratic,
                                              ExpSineSquared, DotProduct,
                                              ConstantKernel)

kernels = [1.0 * RBF(length_scale=1.0, length_scale_bounds=(1e-1, 10.0)),
           1.0 * Matern(length_scale=1.0, length_scale_bounds=(1e-1, 10.0)),
           1.0 * RationalQuadratic(length_scale=1.0,
                                    length_scale_bounds=(1e-1, 10.0)),
           1.0 * ExpSineSquared(length_scale=1.0, period_length=1.0,
                                length_scale_bounds=(1e-1, 10.0),
                                period_length_bounds=(0.1, 10.0)),
           1.0 * DotProduct(sigma_nu=1.0, sigma_nu_bounds=(0.1, 10.0)),
           1.0 * ConstantKernel(constant_value=1.0,
                                constant_value_bounds=(0.1, 10.0))]

```

```
1.0 * RationalQuadratic(length_scale=1.0, alpha=0.1),
1.0 * ExpSineSquared(length_scale=1.0, periodicity=3.0,
                      length_scale_bounds=(0.1, 10.0),
                      periodicity_bounds=(1.0, 10.0)),
ConstantKernel(0.1, (0.01, 10.0))
    * (DotProduct(sigma_0=1.0, sigma_0_bounds=(0.1, 10.0)) ** 2),
1.0 * Matern(length_scale=1.0, length_scale_bounds=(1e-1, 10.0),
              nu=1.5)]
```

```
for kernel in kernels:
    # Specify Gaussian Process
    gp = GaussianProcessRegressor(kernel=kernel)

    # Plot prior
    plt.figure(figsize=(8, 8))
    plt.subplot(2, 1, 1)
    X_ = np.linspace(0, 5, 100)
    y_mean, y_std = gp.predict(X_[:, np.newaxis], return_std=True)
    plt.plot(X_, y_mean, 'k', lw=3, zorder=9)
    plt.fill_between(X_, y_mean - y_std, y_mean + y_std,
                     alpha=0.2, color='k')
    y_samples = gp.sample_y(X_[:, np.newaxis], 10)
    plt.plot(X_, y_samples, lw=1)
    plt.xlim(0, 5)
    plt.ylim(-3, 3)
    plt.title("Prior (kernel: %s)" % kernel, fontsize=12)

    # Generate data and fit GP
    rng = np.random.RandomState(4)
    X = rng.uniform(0, 5, 10)[:, np.newaxis]
    y = np.sin((X[:, 0] - 2.5) ** 2)
    gp.fit(X, y)

    # Plot posterior
    plt.subplot(2, 1, 2)
    X_ = np.linspace(0, 5, 100)
    y_mean, y_std = gp.predict(X_[:, np.newaxis], return_std=True)
    plt.plot(X_, y_mean, 'k', lw=3, zorder=9)
    plt.fill_between(X_, y_mean - y_std, y_mean + y_std,
                     alpha=0.2, color='k')

    y_samples = gp.sample_y(X_[:, np.newaxis], 10)
    plt.plot(X_, y_samples, lw=1)
    plt.scatter(X[:, 0], y, c='r', s=50, zorder=10, edgecolors=(0, 0, 0))
    plt.xlim(0, 5)
    plt.ylim(-3, 3)
    plt.title("Posterior (kernel: %s)\n Log-Likelihood: %.3f"
              % (gp.kernel_, gp.log_marginal_likelihood(gp.kernel_.theta)),
              fontsize=12)
    plt.tight_layout()

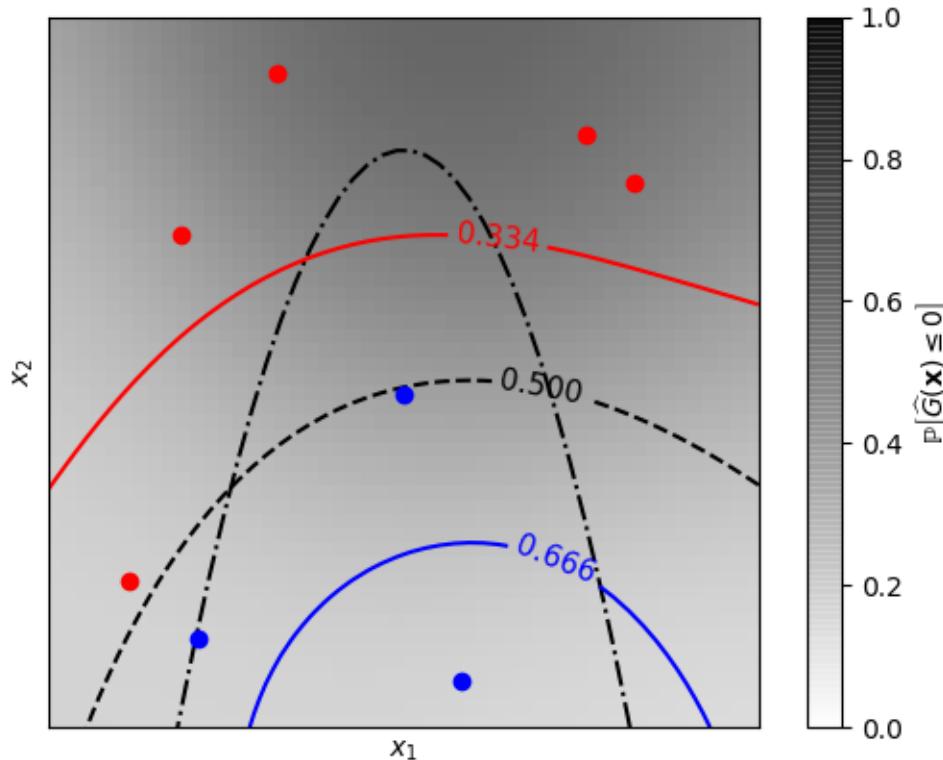
plt.show()
```

Total running time of the script: (0 minutes 1.458 seconds)

Note: Click [here](#) to download the full example code

5.15.5 Iso-probability lines for Gaussian Processes classification (GPC)

A two-dimensional classification example showing iso-probability lines for the predicted probabilities.



Out:

```
Learned kernel: 0.0256**2 * DotProduct(sigma_0=5.72) ** 2
```

```
print(__doc__)

# Author: Vincent Dubourg <vincent.dubourg@gmail.com>
# Adapted to GaussianProcessClassifier:
#         Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
# License: BSD 3 clause

import numpy as np

from matplotlib import pyplot as plt
from matplotlib import cm

from sklearn.gaussian_process import GaussianProcessClassifier
```

```

from sklearn.gaussian_process.kernels import DotProduct, ConstantKernel as C

# A few constants
lim = 8

def g(x):
    """The function to predict (classification will then consist in predicting
    whether g(x) <= 0 or not)"""
    return 5. - x[:, 1] - .5 * x[:, 0] ** 2.

# Design of experiments
X = np.array([[-4.61611719, -6.00099547],
              [4.10469096, 5.32782448],
              [0.00000000, -0.50000000],
              [-6.17289014, -4.6984743],
              [1.3109306, -6.93271427],
              [-5.03823144, 3.10584743],
              [-2.87600388, 6.74310541],
              [5.21301203, 4.26386883]])

# Observations
y = np.array(g(X) > 0, dtype=int)

# Instantiate and fit Gaussian Process Model
kernel = C(0.1, (1e-5, np.inf)) * DotProduct(sigma_0=0.1) ** 2
gp = GaussianProcessClassifier(kernel=kernel)
gp.fit(X, y)
print("Learned kernel: %s" % gp.kernel_)

# Evaluate real function and the predicted probability
res = 50
x1, x2 = np.meshgrid(np.linspace(-lim, lim, res),
                      np.linspace(-lim, lim, res))
xx = np.vstack([x1.reshape(x1.size), x2.reshape(x2.size)]).T

y_true = g(xx)
y_prob = gp.predict_proba(xx)[:, 1]
y_true = y_true.reshape((res, res))
y_prob = y_prob.reshape((res, res))

# Plot the probabilistic classification iso-values
fig = plt.figure(1)
ax = fig.gca()
ax.axes.set_aspect('equal')
plt.xticks([])
plt.yticks([])
ax.set_xticklabels([])
ax.set_yticklabels([])
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

cax = plt.imshow(y_prob, cmap=cm.gray_r, alpha=0.8,
                  extent=(-lim, lim, -lim, lim))
norm = plt.matplotlib.colors.Normalize(vmin=0., vmax=0.9)
cb = plt.colorbar(cax, ticks=[0., 0.2, 0.4, 0.6, 0.8, 1.], norm=norm)
cb.set_label(r'$\rm \left[ \widehat{G}(x) \leq 0 \right]$')
plt.clim(0, 1)

```

```

plt.plot(X[y <= 0, 0], X[y <= 0, 1], 'r.', markersize=12)
plt.plot(X[y > 0, 0], X[y > 0, 1], 'b.', markersize=12)

plt.contour(x1, x2, y_true, [0.], colors='k', linestyles='dashdot')

cs = plt.contour(x1, x2, y_prob, [0.666], colors='b',
                  linestyles='solid')
plt.clabel(cs, fontsize=11)

cs = plt.contour(x1, x2, y_prob, [0.5], colors='k',
                  linestyles='dashed')
plt.clabel(cs, fontsize=11)

cs = plt.contour(x1, x2, y_prob, [0.334], colors='r',
                  linestyles='solid')
plt.clabel(cs, fontsize=11)

plt.show()

```

Total running time of the script: (0 minutes 0.098 seconds)

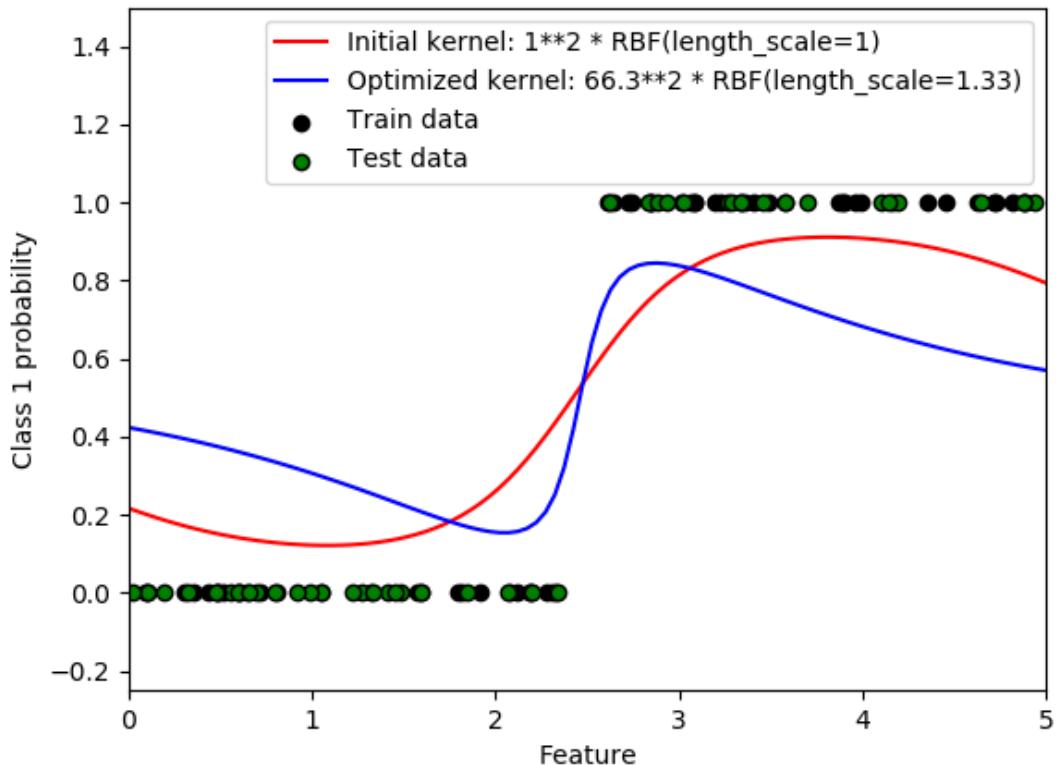
Note: Click [here](#) to download the full example code

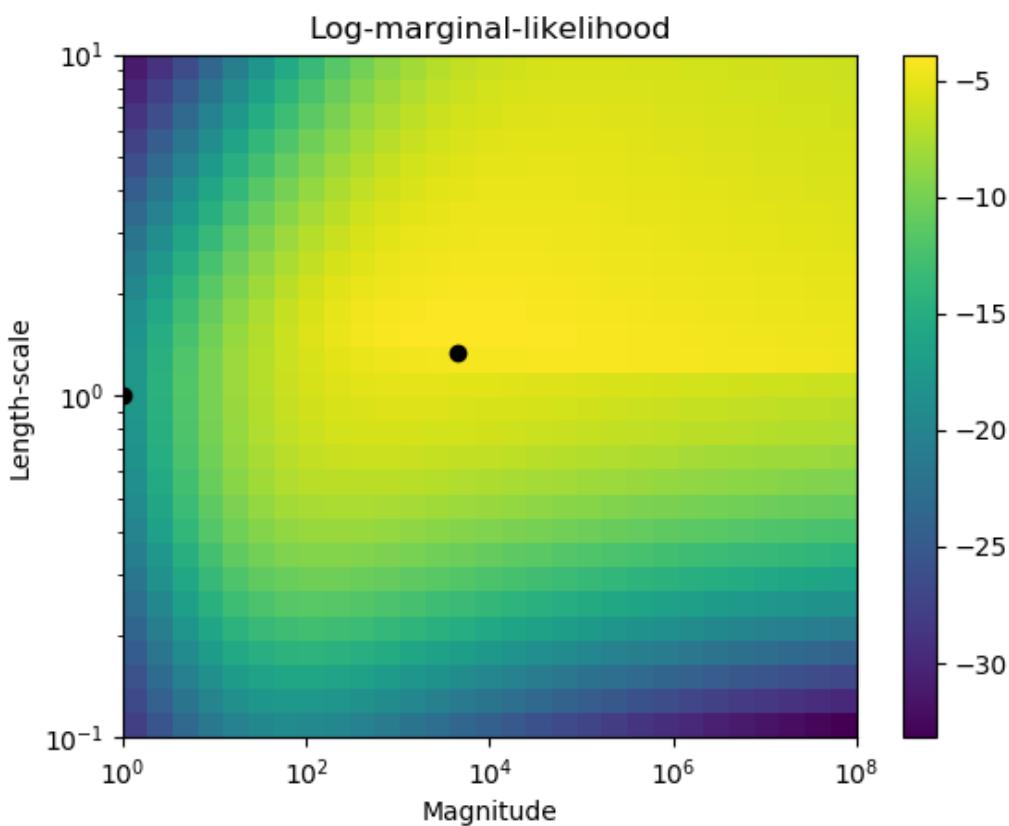
5.15.6 Probabilistic predictions with Gaussian process classification (GPC)

This example illustrates the predicted probability of GPC for an RBF kernel with different choices of the hyperparameters. The first figure shows the predicted probability of GPC with arbitrarily chosen hyperparameters and with the hyperparameters corresponding to the maximum log-marginal-likelihood (LML).

While the hyperparameters chosen by optimizing LML have a considerable larger LML, they perform slightly worse according to the log-loss on test data. The figure shows that this is because they exhibit a steep change of the class probabilities at the class boundaries (which is good) but have predicted probabilities close to 0.5 far away from the class boundaries (which is bad). This undesirable effect is caused by the Laplace approximation used internally by GPC.

The second figure shows the log-marginal-likelihood for different choices of the kernel's hyperparameters, highlighting the two choices of the hyperparameters used in the first figure by black dots.





Out:

```
Log Marginal Likelihood (initial): -17.598
Log Marginal Likelihood (optimized): -3.875
Accuracy: 1.000 (initial) 1.000 (optimized)
Log-loss: 0.214 (initial) 0.319 (optimized)
```

```
print(__doc__)

# Authors: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
#
# License: BSD 3 clause

import numpy as np

from matplotlib import pyplot as plt

from sklearn.metrics.classification import accuracy_score, log_loss
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
```

```

# Generate data
train_size = 50
rng = np.random.RandomState(0)
X = rng.uniform(0, 5, 100)[:, np.newaxis]
y = np.array(X[:, 0] > 2.5, dtype=int)

# Specify Gaussian Processes with fixed and optimized hyperparameters
gp_fix = GaussianProcessClassifier(kernel=1.0 * RBF(length_scale=1.0),
                                    optimizer=None)
gp_fix.fit(X[:train_size], y[:train_size])

gp_opt = GaussianProcessClassifier(kernel=1.0 * RBF(length_scale=1.0))
gp_opt.fit(X[:train_size], y[:train_size])

print("Log Marginal Likelihood (initial): %.3f"
      % gp_fix.log_marginal_likelihood(gp_fix.kernel_.theta))
print("Log Marginal Likelihood (optimized): %.3f"
      % gp_opt.log_marginal_likelihood(gp_opt.kernel_.theta))

print("Accuracy: %.3f (initial) %.3f (optimized)"
      % (accuracy_score(y[:train_size], gp_fix.predict(X[:train_size])),
         accuracy_score(y[:train_size], gp_opt.predict(X[:train_size]))))
print("Log-loss: %.3f (initial) %.3f (optimized)"
      % (log_loss(y[:train_size], gp_fix.predict_proba(X[:train_size])[:, 1]),
         log_loss(y[:train_size], gp_opt.predict_proba(X[:train_size])[:, 1])))

# Plot posteriors
plt.figure()
plt.scatter(X[:train_size, 0], y[:train_size], c='k', label="Train data",
            edgecolors=(0, 0, 0))
plt.scatter(X[train_size:, 0], y[train_size:], c='g', label="Test data",
            edgecolors=(0, 0, 0))
X_ = np.linspace(0, 5, 100)
plt.plot(X_, gp_fix.predict_proba(X_[:, np.newaxis])[:, 1], 'r',
          label="Initial kernel: %s" % gp_fix.kernel_)
plt.plot(X_, gp_opt.predict_proba(X_[:, np.newaxis])[:, 1], 'b',
          label="Optimized kernel: %s" % gp_opt.kernel_)
plt.xlabel("Feature")
plt.ylabel("Class 1 probability")
plt.xlim(0, 5)
plt.ylim(-0.25, 1.5)
plt.legend(loc="best")

# Plot LML landscape
plt.figure()
theta0 = np.logspace(0, 8, 30)
theta1 = np.logspace(-1, 1, 29)
Theta0, Theta1 = np.meshgrid(theta0, theta1)
LML = [[gp_opt.log_marginal_likelihood(np.log([Theta0[i, j], Theta1[i, j]]))]
        for i in range(Theta0.shape[0])] for j in range(Theta0.shape[1])]
LML = np.array(LML).T
plt.plot(np.exp(gp_fix.kernel_.theta)[0], np.exp(gp_fix.kernel_.theta)[1],
        'ko', zorder=10)
plt.plot(np.exp(gp_opt.kernel_.theta)[0], np.exp(gp_opt.kernel_.theta)[1],
        'ko', zorder=10)
plt.pcolor(Theta0, Theta1, LML)
plt.xscale("log")

```

```

plt.yscale("log")
plt.colorbar()
plt.xlabel("Magnitude")
plt.ylabel("Length-scale")
plt.title("Log-marginal-likelihood")

plt.show()

```

Total running time of the script: (0 minutes 2.514 seconds)

Note: Click [here](#) to download the full example code

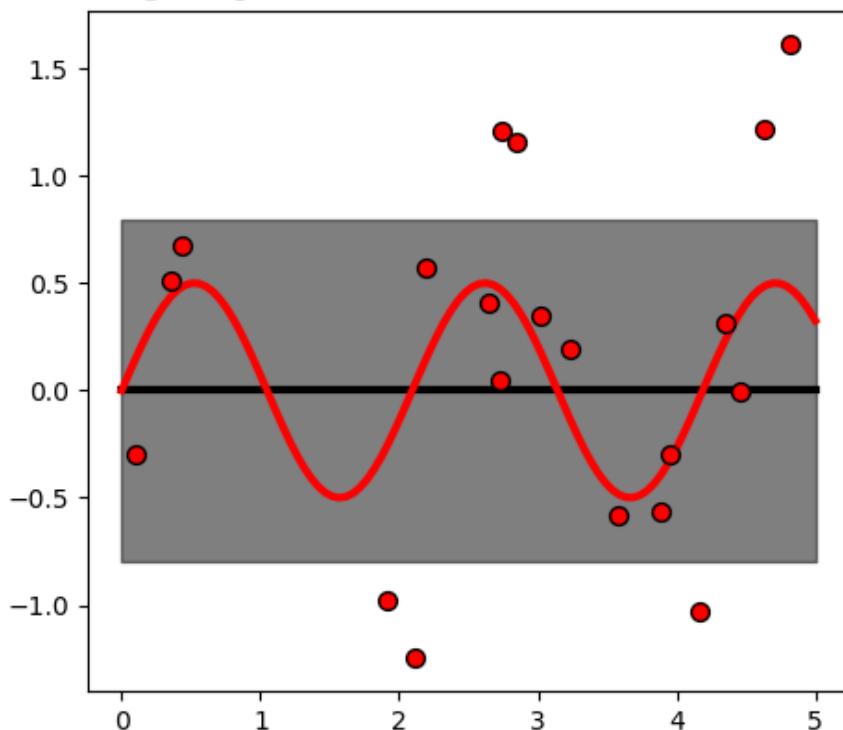
5.15.7 Gaussian process regression (GPR) with noise-level estimation

This example illustrates that GPR with a sum-kernel including a WhiteKernel can estimate the noise level of data. An illustration of the log-marginal-likelihood (LML) landscape shows that there exist two local maxima of LML. The first corresponds to a model with a high noise level and a large length scale, which explains all variations in the data by noise. The second one has a smaller noise level and shorter length scale, which explains most of the variation by the noise-free functional relationship. The second model has a higher likelihood; however, depending on the initial value for the hyperparameters, the gradient-based optimization might also converge to the high-noise solution. It is thus important to repeat the optimization several times for different initializations.

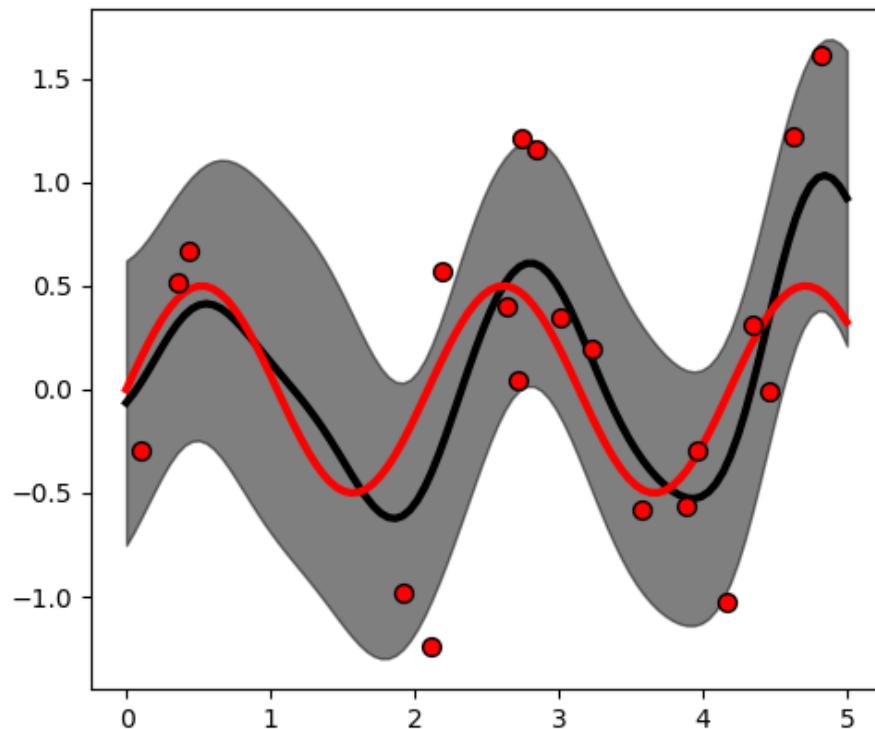
```

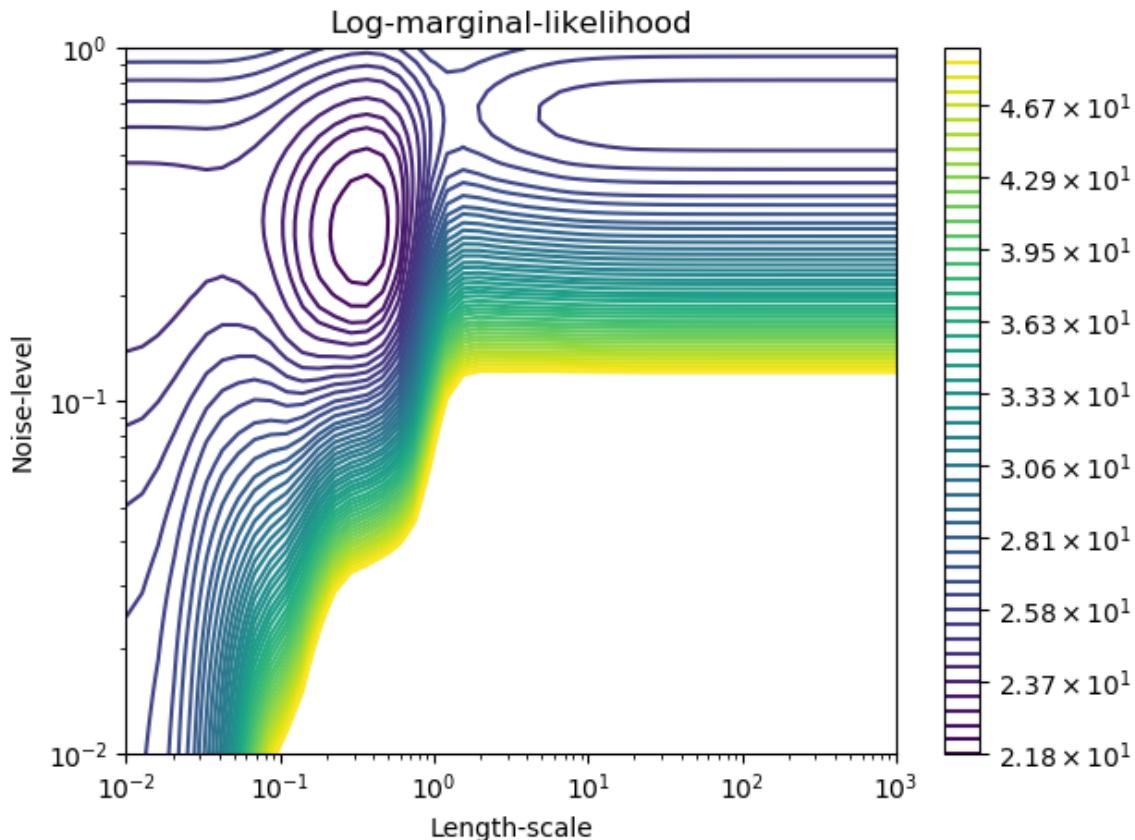
Initial: 1**2 * RBF(length_scale=100) + WhiteKernel(noise_level=1)
Optimum: 0.00316**2 * RBF(length_scale=109) + WhiteKernel(noise_level=0.6)
Log-Marginal-Likelihood: -23.87233736198489

```



```
Initial: 1**2 * RBF(length_scale=1) + WhiteKernel(noise_level=1e-05)
Optimum: 0.64**2 * RBF(length_scale=0.365) + WhiteKernel(noise_level=0.29
Log-Marginal-Likelihood: -21.805090890162035
```





```

print(__doc__)

# Authors: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
#
# License: BSD 3 clause

import numpy as np

from matplotlib import pyplot as plt
from matplotlib.colors import LogNorm

from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel


rng = np.random.RandomState(0)
X = rng.uniform(0, 5, 20)[:, np.newaxis]
y = 0.5 * np.sin(3 * X[:, 0]) + rng.normal(0, 0.5, X.shape[0])

# First run
plt.figure()
kernel = 1.0 * RBF(length_scale=100.0, length_scale_bounds=(1e-2, 1e3)) \
    + WhiteKernel(noise_level=1, noise_level_bounds=(1e-10, 1e+1))
gp = GaussianProcessRegressor(kernel=kernel,
                             alpha=0.0).fit(X, y)
X_ = np.linspace(0, 5, 100)
y_mean, y_cov = gp.predict(X_[:, np.newaxis], return_cov=True)

```

```

plt.plot(X_, y_mean, 'k', lw=3, zorder=9)
plt.fill_between(X_, y_mean - np.sqrt(np.diag(y_cov)),
                 y_mean + np.sqrt(np.diag(y_cov)),
                 alpha=0.5, color='k')
plt.plot(X_, 0.5*np.sin(3*X_), 'r', lw=3, zorder=9)
plt.scatter(X[:, 0], y, c='r', s=50, zorder=10, edgecolors=(0, 0, 0))
plt.title("Initial: %s\nOptimum: %s\nLog-Marginal-Likelihood: %s"
          % (kernel, gp.kernel_,
             gp.log_marginal_likelihood(gp.kernel_.theta)))
plt.tight_layout()

# Second run
plt.figure()
kernel = 1.0 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e3)) \
    + WhiteKernel(noise_level=1e-5, noise_level_bounds=(1e-10, 1e+1))
gp = GaussianProcessRegressor(kernel=kernel,
                               alpha=0.0).fit(X, y)
X_ = np.linspace(0, 5, 100)
y_mean, y_cov = gp.predict(X_[ :, np.newaxis], return_cov=True)
plt.plot(X_, y_mean, 'k', lw=3, zorder=9)
plt.fill_between(X_, y_mean - np.sqrt(np.diag(y_cov)),
                 y_mean + np.sqrt(np.diag(y_cov)),
                 alpha=0.5, color='k')
plt.plot(X_, 0.5*np.sin(3*X_), 'r', lw=3, zorder=9)
plt.scatter(X[:, 0], y, c='r', s=50, zorder=10, edgecolors=(0, 0, 0))
plt.title("Initial: %s\nOptimum: %s\nLog-Marginal-Likelihood: %s"
          % (kernel, gp.kernel_,
             gp.log_marginal_likelihood(gp.kernel_.theta)))
plt.tight_layout()

# Plot LML landscape
plt.figure()
theta0 = np.logspace(-2, 3, 49)
theta1 = np.logspace(-2, 0, 50)
Theta0, Theta1 = np.meshgrid(theta0, theta1)
LML = [[gp.log_marginal_likelihood(np.log([0.36, Theta0[i, j], Theta1[i, j]]))
         for i in range(Theta0.shape[0])] for j in range(Theta0.shape[1])]
LML = np.array(LML).T

vmin, vmax = (-LML).min(), (-LML).max()
vmax = 50
level = np.around(np.logspace(np.log10(vmin), np.log10(vmax), 50), decimals=1)
plt.contour(Theta0, Theta1, -LML,
            levels=level, norm=LogNorm(vmin=vmin, vmax=vmax))
plt.colorbar()
plt.xscale("log")
plt.yscale("log")
plt.xlabel("Length-scale")
plt.ylabel("Noise-level")
plt.title("Log-marginal-likelihood")
plt.tight_layout()

plt.show()

```

Total running time of the script: (0 minutes 2.874 seconds)

Note: Click [here](#) to download the full example code

5.15.8 Gaussian Processes regression: basic introductory example

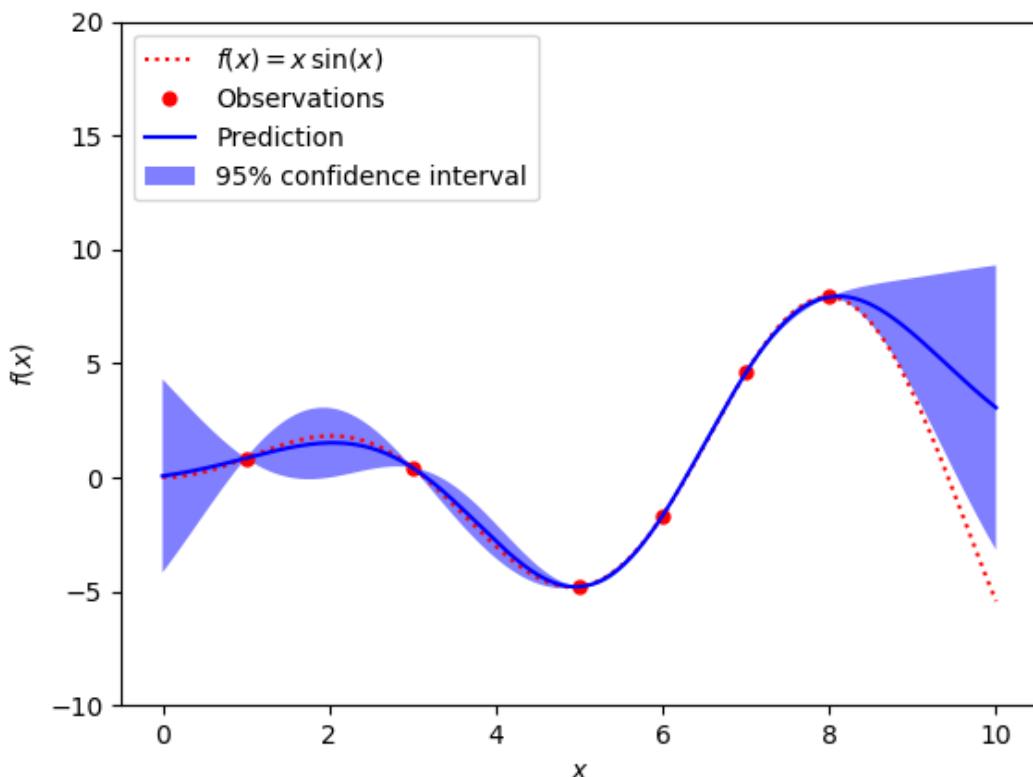
A simple one-dimensional regression example computed in two different ways:

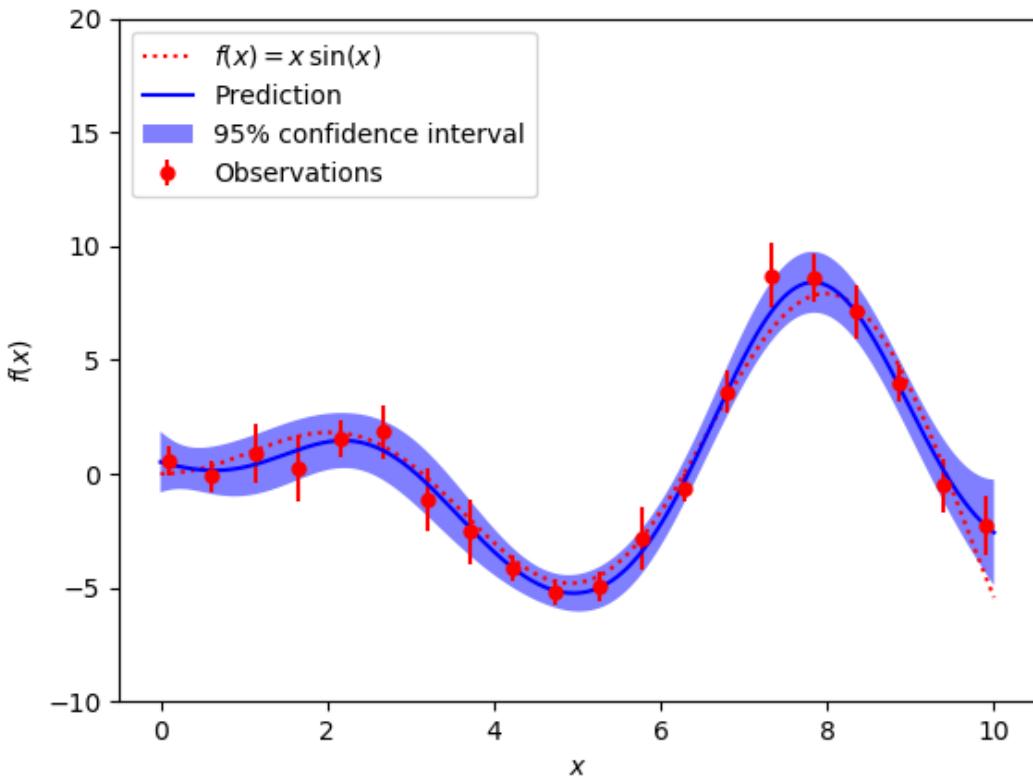
1. A noise-free case
2. A noisy case with known noise-level per datapoint

In both cases, the kernel's parameters are estimated using the maximum likelihood principle.

The figures illustrate the interpolating property of the Gaussian Process model as well as its probabilistic nature in the form of a pointwise 95% confidence interval.

Note that the parameter `alpha` is applied as a Tikhonov regularization of the assumed covariance between the training points.





```

print(__doc__)

# Author: Vincent Dubourg <vincent.dubourg@gmail.com>
#         Jake Vanderplas <jakevdp@astro.washington.edu>
#         Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
# License: BSD 3 clause

import numpy as np
from matplotlib import pyplot as plt

from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, ConstantKernel as C

np.random.seed(1)

def f(x):
    """The function to predict."""
    return x * np.sin(x)

# -----
# First the noiseless case
X = np.atleast_2d([1., 3., 5., 6., 7., 8.]).T

# Observations
y = f(X).ravel()

```

```

# Mesh the input space for evaluations of the real function, the prediction and
# its MSE
x = np.atleast_2d(np.linspace(0, 10, 1000)).T

# Instantiate a Gaussian Process model
kernel = C(1.0, (1e-3, 1e3)) * RBF(10, (1e-2, 1e2))
gp = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)

# Fit to data using Maximum Likelihood Estimation of the parameters
gp.fit(X, y)

# Make the prediction on the meshed x-axis (ask for MSE as well)
y_pred, sigma = gp.predict(x, return_std=True)

# Plot the function, the prediction and the 95% confidence interval based on
# the MSE
plt.figure()
plt.plot(x, f(x), 'r:', label=r'$f(x) = x\backslash,\sin(x)$')
plt.plot(X, y, 'r.', markersize=10, label='Observations')
plt.plot(x, y_pred, 'b-', label='Prediction')
plt.fill(np.concatenate([x, x[::-1]]),
         np.concatenate([y_pred - 1.9600 * sigma,
                        (y_pred + 1.9600 * sigma)[::-1]]),
         alpha=.5, fc='b', ec='None', label='95% confidence interval')
plt.xlabel('$x$')
plt.ylabel('$f(x)$')
plt.ylim(-10, 20)
plt.legend(loc='upper left')

# -----
# now the noisy case
X = np.linspace(0.1, 9.9, 20)
X = np.atleast_2d(X).T

# Observations and noise
y = f(X).ravel()
dy = 0.5 + 1.0 * np.random.random(y.shape)
noise = np.random.normal(0, dy)
y += noise

# Instantiate a Gaussian Process model
gp = GaussianProcessRegressor(kernel=kernel, alpha=dy ** 2,
                               n_restarts_optimizer=10)

# Fit to data using Maximum Likelihood Estimation of the parameters
gp.fit(X, y)

# Make the prediction on the meshed x-axis (ask for MSE as well)
y_pred, sigma = gp.predict(x, return_std=True)

# Plot the function, the prediction and the 95% confidence interval based on
# the MSE
plt.figure()
plt.plot(x, f(x), 'r:', label=r'$f(x) = x\backslash,\sin(x)$')
plt.errorbar(X.ravel(), y, dy, fmt='r.', markersize=10, label='Observations')
plt.plot(x, y_pred, 'b-', label='Prediction')
plt.fill(np.concatenate([x, x[::-1]]),
         np.concatenate([y_pred - 1.9600 * sigma,
                        (y_pred + 1.9600 * sigma)[::-1]]),
         alpha=.5, fc='b', ec='None', label='95% confidence interval')

```

```

        (y_pred + 1.9600 * sigma) [:-1]]),
alpha=.5, fc='b', ec='None', label='95% confidence interval')
plt.xlabel('$x$')
plt.ylabel('$f(x)$')
plt.ylim(-10, 20)
plt.legend(loc='upper left')

plt.show()

```

Total running time of the script: (0 minutes 0.284 seconds)

Note: Click [here](#) to download the full example code

5.15.9 Gaussian process regression (GPR) on Mauna Loa CO2 data.

This example is based on Section 5.4.3 of “Gaussian Processes for Machine Learning” [RW2006]. It illustrates an example of complex kernel engineering and hyperparameter optimization using gradient ascent on the log-marginal-likelihood. The data consists of the monthly average atmospheric CO₂ concentrations (in parts per million by volume (ppmv)) collected at the Mauna Loa Observatory in Hawaii, between 1958 and 2001. The objective is to model the CO₂ concentration as a function of the time t.

The kernel is composed of several terms that are responsible for explaining different properties of the signal:

- a long term, smooth rising trend is to be explained by an RBF kernel. The RBF kernel with a large length-scale enforces this component to be smooth; it is not enforced that the trend is rising which leaves this choice to the GP. The specific length-scale and the amplitude are free hyperparameters.
- a seasonal component, which is to be explained by the periodic ExpSineSquared kernel with a fixed periodicity of 1 year. The length-scale of this periodic component, controlling its smoothness, is a free parameter. In order to allow decaying away from exact periodicity, the product with an RBF kernel is taken. The length-scale of this RBF component controls the decay time and is a further free parameter.
- smaller, medium term irregularities are to be explained by a RationalQuadratic kernel component, whose length-scale and alpha parameter, which determines the diffuseness of the length-scales, are to be determined. According to [RW2006], these irregularities can better be explained by a RationalQuadratic than an RBF kernel component, probably because it can accommodate several length-scales.
- a “noise” term, consisting of an RBF kernel contribution, which shall explain the correlated noise components such as local weather phenomena, and a WhiteKernel contribution for the white noise. The relative amplitudes and the RBF’s length scale are further free parameters.

Maximizing the log-marginal-likelihood after subtracting the target’s mean yields the following kernel with an LML of -83.214:

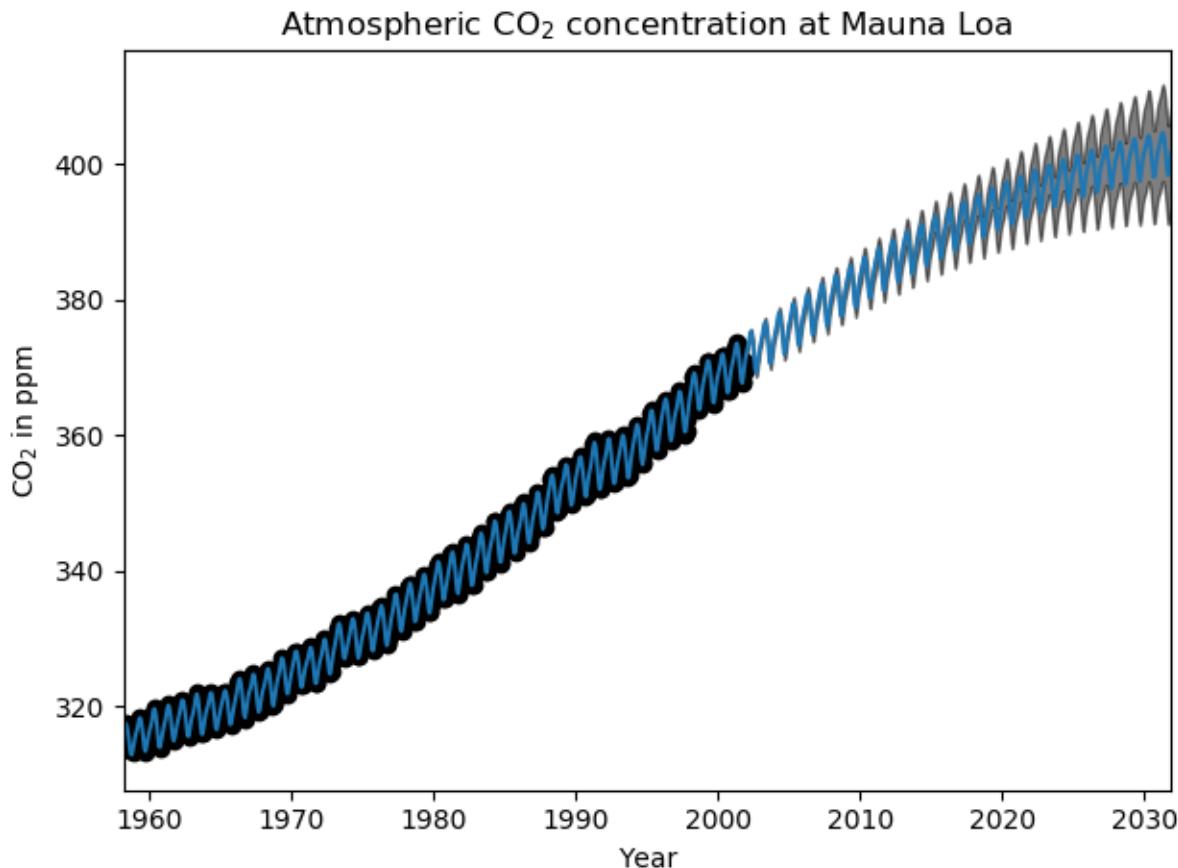
```

34.4**2 * RBF(length_scale=41.8)
+ 3.27**2 * RBF(length_scale=180) * ExpSineSquared(length_scale=1.44,
                                                       periodicity=1)
+ 0.446**2 * RationalQuadratic(alpha=17.7, length_scale=0.957)
+ 0.197**2 * RBF(length_scale=0.138) + WhiteKernel(noise_level=0.0336)

```

Thus, most of the target signal (34.4ppm) is explained by a long-term rising trend (length-scale 41.8 years). The periodic component has an amplitude of 3.27ppm, a decay time of 180 years and a length-scale of 1.44. The long decay time indicates that we have a locally very close to periodic seasonal component. The correlated noise has an amplitude of 0.197ppm with a length scale of 0.138 years and a white-noise contribution of 0.197ppm. Thus, the

overall noise level is very small, indicating that the data can be very well explained by the model. The figure shows also that the model makes very confident predictions until around 2015.



Out:

```
GPLM kernel: 66**2 * RBF(length_scale=67) + 2.4**2 * RBF(length_scale=90) *_
↪ExpSineSquared(length_scale=1.3, periodicity=1) + 0.66**2 *_
↪RationalQuadratic(alpha=0.78, length_scale=1.2) + 0.18**2 * RBF(length_scale=0.134)_
↪+ WhiteKernel(noise_level=0.0361)
Log-marginal-likelihood: -117.023

Learned kernel: 44.8**2 * RBF(length_scale=51.6) + 2.64**2 * RBF(length_scale=91.5) *_
↪ExpSineSquared(length_scale=1.48, periodicity=1) + 0.536**2 *_
↪RationalQuadratic(alpha=2.89, length_scale=0.968) + 0.188**2 * RBF(length_scale=0.122) +
↪WhiteKernel(noise_level=0.0367)
Log-marginal-likelihood: -115.050
```

```
# Authors: Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
#
# License: BSD 3 clause
```

```

import numpy as np

from matplotlib import pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels \
    import RBF, WhiteKernel, RationalQuadratic, ExpSineSquared

print(__doc__)

def load_mauna_loa_atmospheric_co2():
    ml_data = fetch_openml(data_id=41187)
    months = []
    ppmv_sums = []
    counts = []

    y = ml_data.data[:, 0]
    m = ml_data.data[:, 1]
    month_float = y + (m - 1) / 12
    ppmvs = ml_data.target

    for month, ppmv in zip(month_float, ppmvs):
        if not months or month != months[-1]:
            months.append(month)
            ppmv_sums.append(ppmv)
            counts.append(1)
        else:
            # aggregate monthly sum to produce average
            ppmv_sums[-1] += ppmv
            counts[-1] += 1

    months = np.asarray(months).reshape(-1, 1)
    avg_ppmvs = np.asarray(ppmv_sums) / counts
    return months, avg_ppmvs

X, y = load_mauna_loa_atmospheric_co2()

# Kernel with parameters given in GPML book
k1 = 66.0**2 * RBF(length_scale=67.0) # long term smooth rising trend
k2 = 2.4**2 * RBF(length_scale=90.0) \
    * ExpSineSquared(length_scale=1.3, periodicity=1.0) # seasonal component
# medium term irregularity
k3 = 0.66**2 \
    * RationalQuadratic(length_scale=1.2, alpha=0.78)
k4 = 0.18**2 * RBF(length_scale=0.134) \
    + WhiteKernel(noise_level=0.19**2) # noise terms
kernel_gpml = k1 + k2 + k3 + k4

gp = GaussianProcessRegressor(kernel=kernel_gpml, alpha=0,
                             optimizer=None, normalize_y=True)
gp.fit(X, y)

print("GPML kernel: %s" % gp.kernel_)
print("Log-marginal-likelihood: %.3f"
      % gp.log_marginal_likelihood(gp.kernel_.theta))

```

```

# Kernel with optimized parameters
k1 = 50.0**2 * RBF(length_scale=50.0) # long term smooth rising trend
k2 = 2.0**2 * RBF(length_scale=100.0) \
    * ExpSineSquared(length_scale=1.0, periodicity=1.0,
                      periodicity_bounds="fixed") # seasonal component
# medium term irregularities
k3 = 0.5**2 * RationalQuadratic(length_scale=1.0, alpha=1.0)
k4 = 0.1**2 * RBF(length_scale=0.1) \
    + WhiteKernel(noise_level=0.1**2,
                  noise_level_bounds=(1e-3, np.inf)) # noise terms
kernel = k1 + k2 + k3 + k4

gp = GaussianProcessRegressor(kernel=kernel, alpha=0,
                               normalize_y=True)
gp.fit(X, y)

print("\nLearned kernel: %s" % gp.kernel_)
print("Log-marginal-likelihood: %.3f"
      % gp.log_marginal_likelihood(gp.kernel_.theta))

X_ = np.linspace(X.min(), X.max() + 30, 1000)[:, np.newaxis]
y_pred, y_std = gp.predict(X_, return_std=True)

# Illustration
plt.scatter(X, y, c='k')
plt.plot(X_, y_pred)
plt.fill_between(X_[:, 0], y_pred - y_std, y_pred + y_std,
                 alpha=0.5, color='k')
plt.xlim(X_.min(), X_.max())
plt.xlabel("Year")
plt.ylabel(r"CO2 in ppm")
plt.title(r"Atmospheric CO2 concentration at Mauna Loa")
plt.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 11.550 seconds)

5.16 Missing Value Imputation

Examples concerning the `sklearn.impute` module.

Note: Click [here](#) to download the full example code

5.16.1 Imputing missing values with variants of IterativeImputer

The `sklearn.impute.IterativeImputer` class is very flexible - it can be used with a variety of estimators to do round-robin regression, treating every variable as an output in turn.

In this example we compare some estimators for the purpose of missing feature imputation with `sklearn.impute.IterativeImputer`:

- `BayesianRidge`: regularized linear regression

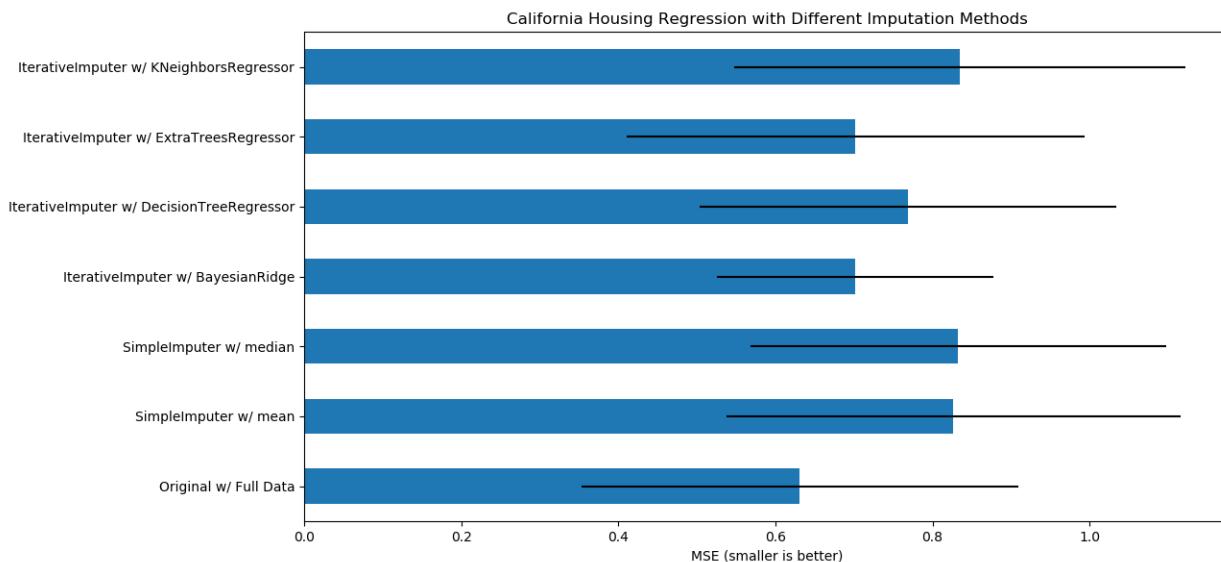
- `DecisionTreeRegressor`: non-linear regression
- `ExtraTreesRegressor`: similar to missForest in R
- `KNeighborsRegressor`: comparable to other KNN imputation approaches

Of particular interest is the ability of `sklearn.impute.IterativeImputer` to mimic the behavior of missForest, a popular imputation package for R. In this example, we have chosen to use `sklearn.ensemble.ExtraTreesRegressor` instead of `sklearn.ensemble.RandomForestRegressor` (as in missForest) due to its increased speed.

Note that `sklearn.neighbors.KNeighborsRegressor` is different from KNN imputation, which learns from samples with missing values by using a distance metric that accounts for missing values, rather than imputing them.

The goal is to compare different estimators to see which one is best for the `sklearn.impute.IterativeImputer` when using a `sklearn.linear_model.BayesianRidge` estimator on the California housing dataset with a single value randomly removed from each row.

For this particular pattern of missing values we see that `sklearn.ensemble.ExtraTreesRegressor` and `sklearn.linear_model.BayesianRidge` give the best results.



```
print(__doc__)

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

# To use this experimental feature, we need to explicitly ask for it:
from sklearn.experimental import enable_iterative_imputer # noqa
from sklearn.datasets import fetch_california_housing
from sklearn.impute import SimpleImputer
from sklearn.impute import IterativeImputer
from sklearn.linear_model import BayesianRidge
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score
```

```

N_SPLITS = 5

rng = np.random.RandomState(0)

X_full, y_full = fetch_california_housing(return_X_y=True)
# ~2k samples is enough for the purpose of the example.
# Remove the following two lines for a slower run with different error bars.
X_full = X_full[::10]
y_full = y_full[::10]
n_samples, n_features = X_full.shape

# Estimate the score on the entire dataset, with no missing values
br_estimator = BayesianRidge()
score_full_data = pd.DataFrame(
    cross_val_score(
        br_estimator, X_full, y_full, scoring='neg_mean_squared_error',
        cv=N_SPLITS
    ),
    columns=['Full Data']
)

# Add a single missing value to each row
X_missing = X_full.copy()
y_missing = y_full
missing_samples = np.arange(n_samples)
missing_features = rng.choice(n_features, n_samples, replace=True)
X_missing[missing_samples, missing_features] = np.nan

# Estimate the score after imputation (mean and median strategies)
score_simple_imputer = pd.DataFrame()
for strategy in ('mean', 'median'):
    estimator = make_pipeline(
        SimpleImputer(missing_values=np.nan, strategy=strategy),
        br_estimator
    )
    score_simple_imputer[strategy] = cross_val_score(
        estimator, X_missing, y_missing, scoring='neg_mean_squared_error',
        cv=N_SPLITS
    )

# Estimate the score after iterative imputation of the missing values
# with different estimators
estimators = [
    BayesianRidge(),
    DecisionTreeRegressor(max_features='sqrt', random_state=0),
    ExtraTreesRegressor(n_estimators=10, random_state=0),
    KNeighborsRegressor(n_neighbors=15)
]
score_iterative_imputer = pd.DataFrame()
for impute_estimator in estimators:
    estimator = make_pipeline(
        IterativeImputer(random_state=0, estimator=impute_estimator),
        br_estimator
    )
    score_iterative_imputer[impute_estimator.__class__.__name__] = \
        cross_val_score(
            estimator, X_missing, y_missing, scoring='neg_mean_squared_error',
            cv=N_SPLITS
        )

```

```
)  
  
scores = pd.concat(  
    [score_full_data, score_simple_imputer, score_iterative_imputer],  
    keys=['Original', 'SimpleImputer', 'IterativeImputer'], axis=1  
)  
  
# plot boston results  
fig, ax = plt.subplots(figsize=(13, 6))  
means = -scores.mean()  
errors = scores.std()  
means.plot.barh(xerr=errors, ax=ax)  
ax.set_title('California Housing Regression with Different Imputation Methods')  
ax.set_xlabel('MSE (smaller is better)')  
ax.set_yticks(np.arange(means.shape[0]))  
ax.set_yticklabels([" w/ ".join(label) for label in means.index.get_values()])  
plt.tight_layout(pad=1)  
plt.show()
```

Total running time of the script: (0 minutes 19.017 seconds)

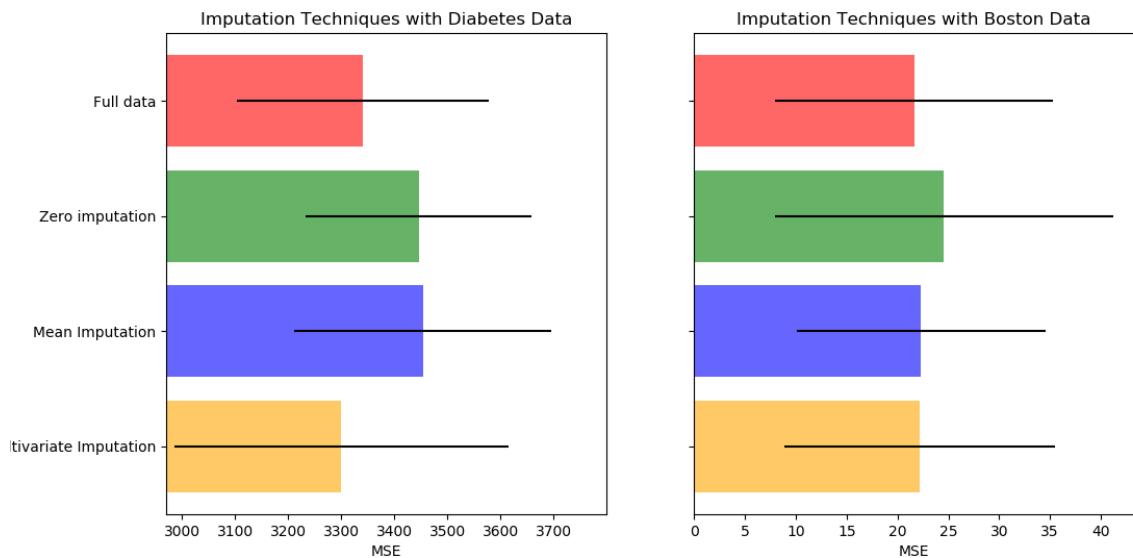
Note: Click [here](#) to download the full example code

5.16.2 Imputing missing values before building an estimator

Missing values can be replaced by the mean, the median or the most frequent value using the basic `sklearn.impute.SimpleImputer`. The median is a more robust estimator for data with high magnitude variables which could dominate results (otherwise known as a ‘long tail’).

Another option is the `sklearn.impute.IterativeImputer`. This uses round-robin linear regression, treating every variable as an output in turn. The version implemented assumes Gaussian (output) variables. If your features are obviously non-Normal, consider transforming them to look more Normal so as to potentially improve performance.

In addition of using an imputing method, we can also keep an indication of the missing information using `sklearn.impute.MissingIndicator` which might carry some information.



```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

# To use the experimental IterativeImputer, we need to explicitly ask for it:
from sklearn.experimental import enable_iterative_imputer # noqa
from sklearn.datasets import load_diabetes
from sklearn.datasets import load_boston
from sklearn.ensemble import RandomForestRegressor
from sklearn.pipeline import make_pipeline, make_union
from sklearn.impute import SimpleImputer, IterativeImputer, MissingIndicator
from sklearn.model_selection import cross_val_score

rng = np.random.RandomState(0)

N_SPLITS = 5
REGRESSOR = RandomForestRegressor(random_state=0, n_estimators=100)

def get_scores_for_imputer(imputer, X_missing, y_missing):
    estimator = make_pipeline(
        make_union(imputer, MissingIndicator(missing_values=0)),
        REGRESSOR)
    impute_scores = cross_val_score(estimator, X_missing, y_missing,
                                    scoring='neg_mean_squared_error',
                                    cv=N_SPLITS)
    return impute_scores

def get_results(dataset):
    X_full, y_full = dataset.data, dataset.target
    n_samples = X_full.shape[0]
    n_features = X_full.shape[1]

    # Estimate the score on the entire dataset, with no missing values
```

```

full_scores = cross_val_score(REGRESSOR, X_full, y_full,
                             scoring='neg_mean_squared_error',
                             cv=N_SPLITS)

# Add missing values in 75% of the lines
missing_rate = 0.75
n_missing_samples = int(np.floor(n_samples * missing_rate))
missing_samples = np.hstack((np.zeros(n_samples - n_missing_samples,
                                      dtype=np.bool),
                             np.ones(n_missing_samples,
                                      dtype=np.bool)))
rng.shuffle(missing_samples)
missing_features = rng.randint(0, n_features, n_missing_samples)
X_missing = X_full.copy()
X_missing[np.where(missing_samples)[0], missing_features] = 0
y_missing = y_full.copy()

# Estimate the score after replacing missing values by 0
imputer = SimpleImputer(missing_values=0,
                         strategy='constant',
                         fill_value=0)
zero_impute_scores = get_scores_for_imputer(imputer, X_missing, y_missing)

# Estimate the score after imputation (mean strategy) of the missing values
imputer = SimpleImputer(missing_values=0, strategy="mean")
mean_impute_scores = get_scores_for_imputer(imputer, X_missing, y_missing)

# Estimate the score after iterative imputation of the missing values
imputer = IterativeImputer(missing_values=0,
                           random_state=0,
                           n_nearest_features=5)
iterative_impute_scores = get_scores_for_imputer(imputer,
                                                 X_missing,
                                                 y_missing)

return ((full_scores.mean(), full_scores.std()),
         (zero_impute_scores.mean(), zero_impute_scores.std()),
         (mean_impute_scores.mean(), mean_impute_scores.std()),
         (iterative_impute_scores.mean(), iterative_impute_scores.std()))

results_diabetes = np.array(get_results(load_diabetes()))
mses_diabetes = results_diabetes[:, 0] * -1
stds_diabetes = results_diabetes[:, 1]

results_boston = np.array(get_results(load_boston()))
mses_boston = results_boston[:, 0] * -1
stds_boston = results_boston[:, 1]

n_bars = len(mses_diabetes)
xval = np.arange(n_bars)

x_labels = ['Full data',
            'Zero imputation',
            'Mean Imputation',
            'Multivariate Imputation']
colors = ['r', 'g', 'b', 'orange']

```

```
# plot diabetes results
plt.figure(figsize=(12, 6))
ax1 = plt.subplot(121)
for j in xval:
    ax1.barh(j, mses_diabetes[j], xerr=stds_diabetes[j],
              color=colors[j], alpha=0.6, align='center')

ax1.set_title('Imputation Techniques with Diabetes Data')
ax1.set_xlim(left=np.min(mses_diabetes) * 0.9,
             right=np.max(mses_diabetes) * 1.1)
ax1.set_yticks(xval)
ax1.set_xlabel('MSE')
ax1.invert_yaxis()
ax1.set_yticklabels(x_labels)

# plot boston results
ax2 = plt.subplot(122)
for j in xval:
    ax2.barh(j, mses_boston[j], xerr=stds_boston[j],
              color=colors[j], alpha=0.6, align='center')

ax2.set_title('Imputation Techniques with Boston Data')
ax2.set_yticks(xval)
ax2.set_xlabel('MSE')
ax2.invert_yaxis()
ax2.set_yticklabels([''] * n_bars)

plt.show()
```

Total running time of the script: (0 minutes 11.569 seconds)

5.17 Inspection

Examples related to the `sklearn.inspection` module.

Note: Click [here](#) to download the full example code

5.17.1 Partial Dependence Plots

Partial dependence plots show the dependence between the target function² and a set of ‘target’ features, marginalizing over the values of all other features (the complement features). Due to the limits of human perception the size of the target feature set must be small (usually, one or two) thus the target features are usually chosen among the most important features.

This example shows how to obtain partial dependence plots from a `MLPRegressor` and a `GradientBoostingRegressor` trained on the California housing dataset. The example is taken from¹.

The plots show four 1-way and two 1-way partial dependence plots (ommitted for `MLPRegressor` due to computation time). The target variables for the one-way PDP are: median income (`MedInc`), average occupants per household (`AvgOccup`), median house age (`HouseAge`), and average rooms per household (`AveRooms`).

² For classification you can think of it as the regression score before the link function.

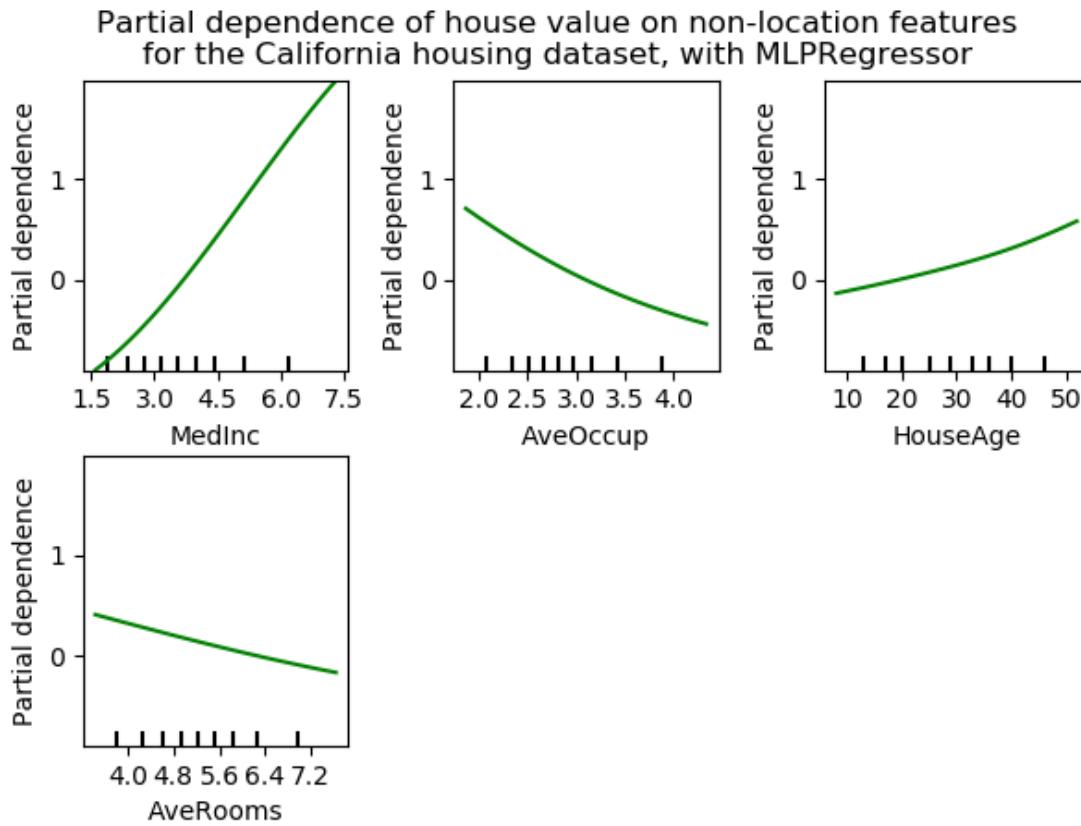
¹ T. Hastie, R. Tibshirani and J. Friedman, “Elements of Statistical Learning Ed. 2”, Springer, 2009.

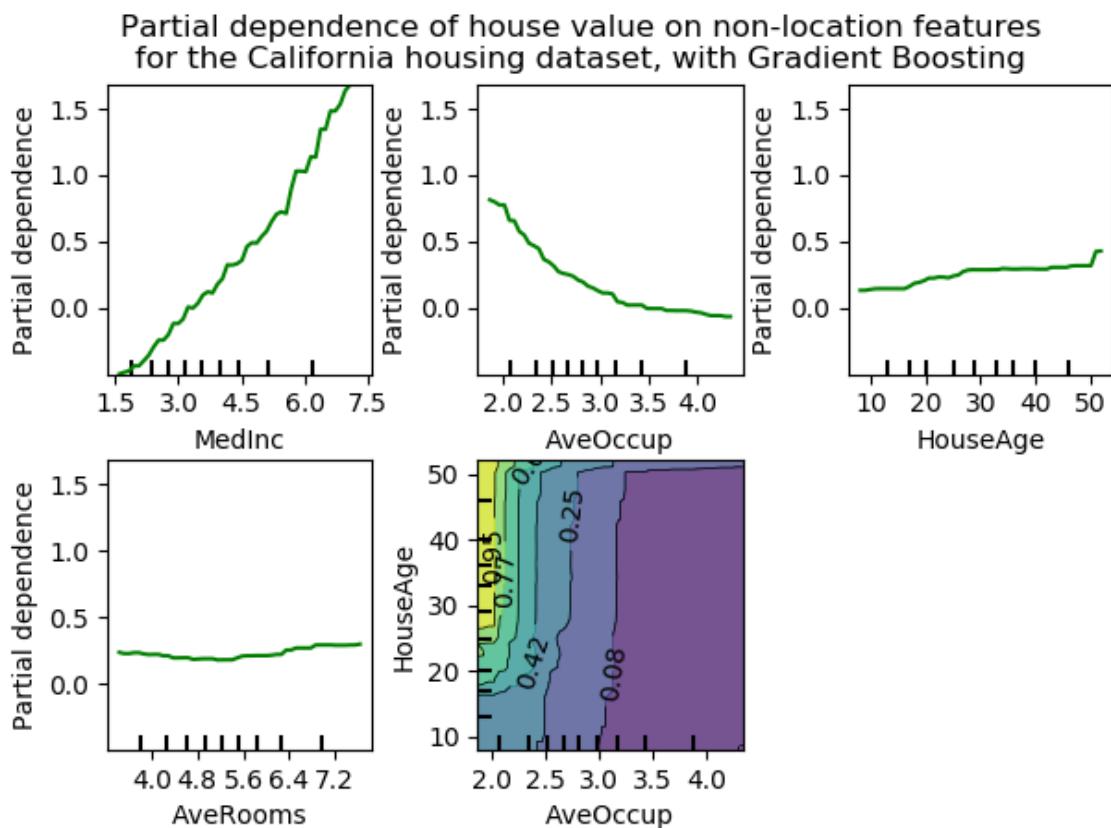
We can clearly see that the median house price shows a linear relationship with the median income (top left) and that the house price drops when the average occupants per household increases (top middle). The top right plot shows that the house age in a district does not have a strong influence on the (median) house price; so does the average rooms per household. The tick marks on the x-axis represent the deciles of the feature values in the training data.

We also observe that `MLPRegressor` has much smoother predictions than `GradientBoostingRegressor`. For the plots to be comparable, it is necessary to subtract the average value of the target y : The ‘recursion’ method, used by default for `GradientBoostingRegressor`, does not account for the initial predictor (in our case the average target). Setting the target average to 0 avoids this bias.

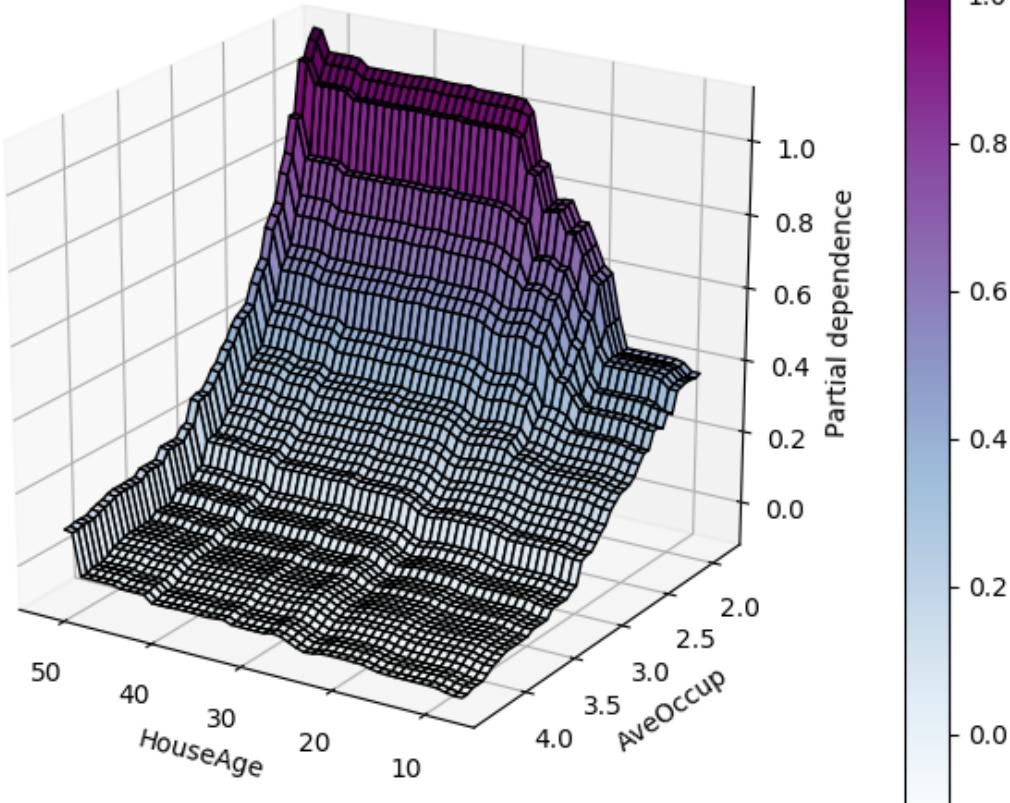
Partial dependence plots with two target features enable us to visualize interactions among them. The two-way partial dependence plot shows the dependence of median house price on joint values of house age and average occupants per household. We can clearly see an interaction between the two features: for an average occupancy greater than two, the house price is nearly independent of the house age, whereas for values less than two there is a strong dependence on age.

On a third figure, we have plotted the same partial dependence plot, this time in 3 dimensions.





Partial dependence of house value on median age and average occupancy, with Gradient Boosting



Out:

```
Training MLPRegressor...
Computing partial dependence plots...
Training GradientBoostingRegressor...
Computing partial dependence plots...
Custom 3d plot via ``partial_dependence``
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from sklearn.inspection import partial_dependence
from sklearn.inspection import plot_partial_dependence
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.neural_network import MLPRegressor
from sklearn.datasets.california_housing import fetch_california_housing

def main():
```

```

cal_housing = fetch_california_housing()

X, y = cal_housing.data, cal_housing.target
names = cal_housing.feature_names

# Center target to avoid gradient boosting init bias: gradient boosting
# with the 'recursion' method does not account for the initial estimator
# (here the average target, by default)
y -= y.mean()

print("Training MLPRegressor...")
est = MLPRegressor(activation='logistic')
est.fit(X, y)
print('Computing partial dependence plots...')
# We don't compute the 2-way PDP (5, 1) here, because it is a lot slower
# with the brute method.
features = [0, 5, 1, 2]
plot_partial_dependence(est, X, features, feature_names=names,
                        n_jobs=3, grid_resolution=50)
fig = plt.gcf()
fig.suptitle('Partial dependence of house value on non-location features\n'
             'for the California housing dataset, with MLPRegressor')
plt.subplots_adjust(top=0.9) # tight_layout causes overlap with suptitle

print("Training GradientBoostingRegressor...")
est = GradientBoostingRegressor(n_estimators=100, max_depth=4,
                                 learning_rate=0.1, loss='huber',
                                 random_state=1)
est.fit(X, y)
print('Computing partial dependence plots...')
features = [0, 5, 1, 2, (5, 1)]
plot_partial_dependence(est, X, features, feature_names=names,
                        n_jobs=3, grid_resolution=50)
fig = plt.gcf()
fig.suptitle('Partial dependence of house value on non-location features\n'
             'for the California housing dataset, with Gradient Boosting')
plt.subplots_adjust(top=0.9)

print('Custom 3d plot via ``partial_dependence``')
fig = plt.figure()

target_feature = (1, 5)
pdp, axes = partial_dependence(est, X, target_feature,
                               grid_resolution=50)
XX, YY = np.meshgrid(axes[0], axes[1])
Z = pdp[0].T
ax = Axes3D(fig)
surf = ax.plot_surface(XX, YY, Z, rstride=1, cstride=1,
                      cmap=plt.cm.BuPu, edgecolor='k')
ax.set_xlabel(names[target_feature[0]])
ax.set_ylabel(names[target_feature[1]])
ax.set_zlabel('Partial dependence')
# pretty init view
ax.view_init(elev=22, azim=122)
plt.colorbar(surf)
plt.suptitle('Partial dependence of house value on median\n'
             'age and average occupancy, with Gradient Boosting')
plt.subplots_adjust(top=0.9)

```

```
plt.show()

# Needed on Windows because plot_partial_dependence uses multiprocessing
if __name__ == '__main__':
    main()
```

Total running time of the script: (0 minutes 24.838 seconds)

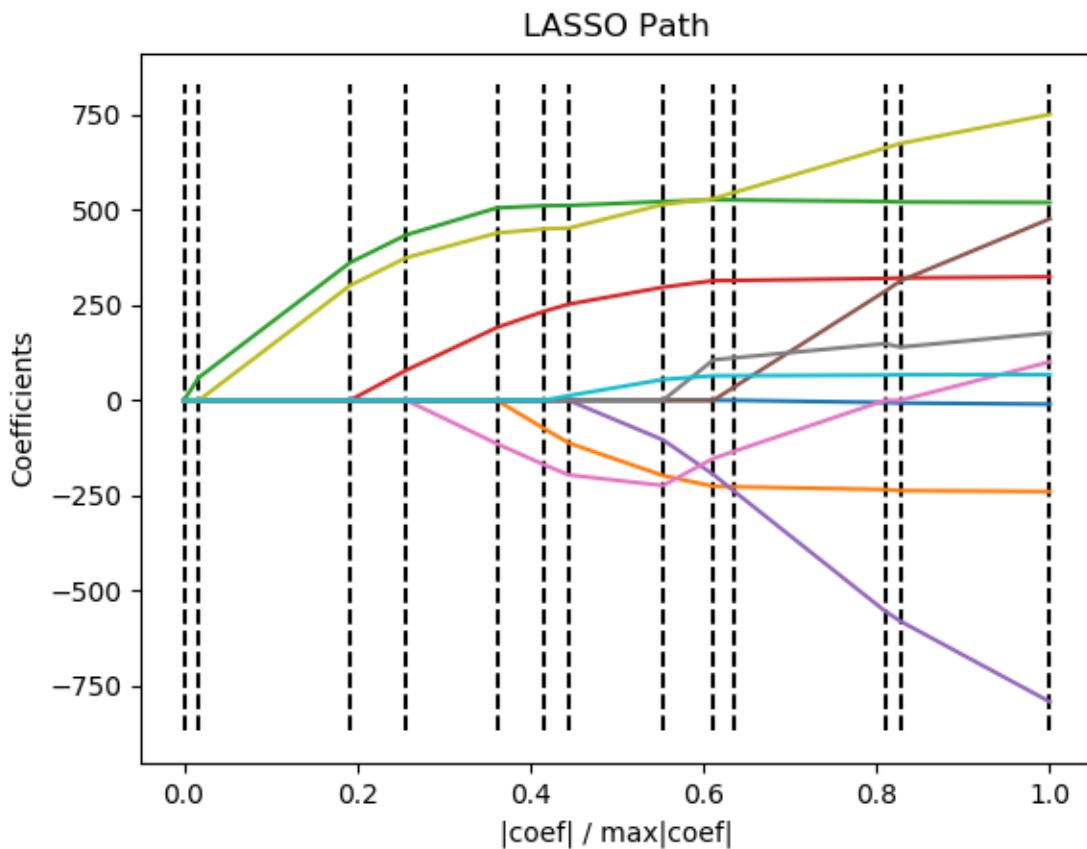
5.18 Generalized Linear Models

Examples concerning the `sklearn.linear_model` module.

Note: Click [here](#) to download the full example code

5.18.1 Lasso path using LARS

Computes Lasso Path along the regularization parameter using the LARS algorithm on the diabetes dataset. Each color represents a different feature of the coefficient vector, and this is displayed as a function of the regularization parameter.



Out:

```
Computing regularization path using the LARS ...
.
```

```
print(__doc__)

# Author: Fabian Pedregosa <fabian.pedregosa@inria.fr>
#         Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn import linear_model
from sklearn import datasets

diabetes = datasets.load_diabetes()
X = diabetes.data
y = diabetes.target
```

```
print("Computing regularization path using the LARS ...")
_, coefs = linear_model.lars_path(X, y, method='lasso', verbose=True)

xx = np.sum(np.abs(coefs.T), axis=1)
xx /= xx[-1]

plt.plot(xx, coefs.T)
ymin, ymax = plt.ylim()
plt.vlines(xx, ymin, ymax, linestyle='dashed')
plt.xlabel('|coef| / max|coef|')
plt.ylabel('Coefficients')
plt.title('LASSO Path')
plt.axis('tight')
plt.show()
```

Total running time of the script: (0 minutes 0.025 seconds)

Note: Click [here](#) to download the full example code

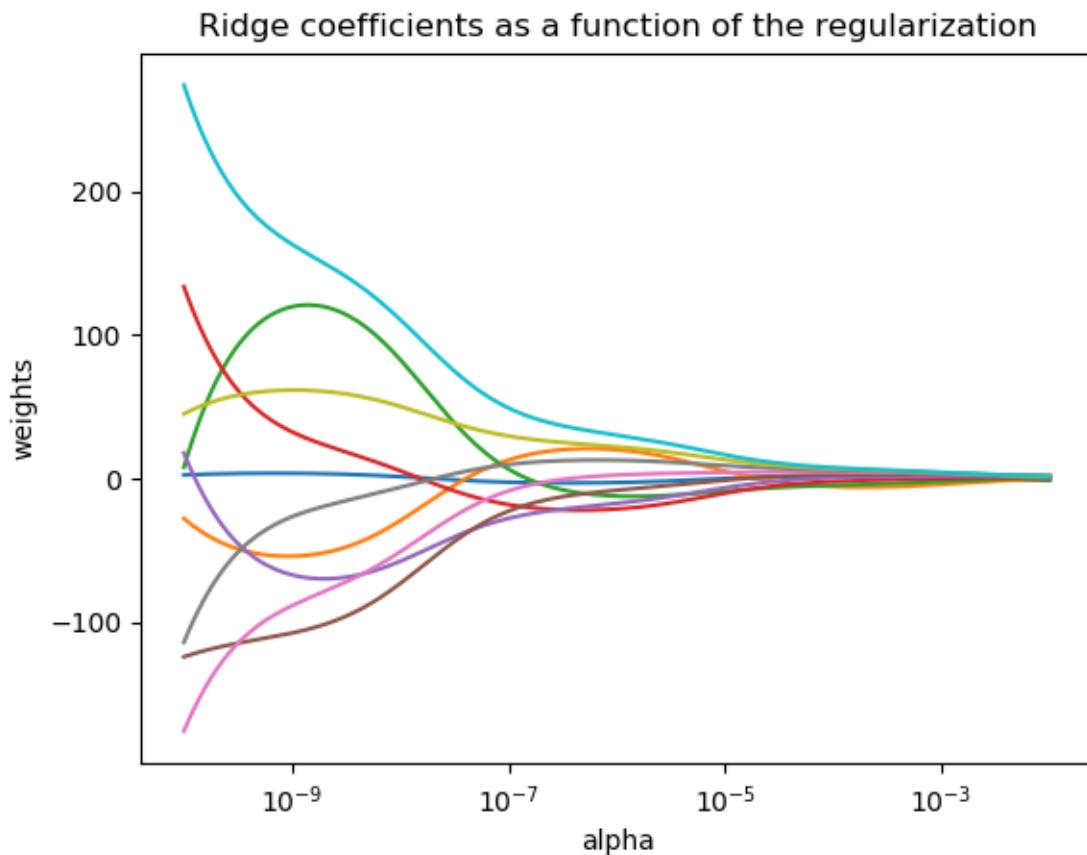
5.18.2 Plot Ridge coefficients as a function of the regularization

Shows the effect of collinearity in the coefficients of an estimator.

Ridge Regression is the estimator used in this example. Each color represents a different feature of the coefficient vector, and this is displayed as a function of the regularization parameter.

This example also shows the usefulness of applying Ridge regression to highly ill-conditioned matrices. For such matrices, a slight change in the target variable can cause huge variances in the calculated weights. In such cases, it is useful to set a certain regularization (alpha) to reduce this variation (noise).

When alpha is very large, the regularization effect dominates the squared loss function and the coefficients tend to zero. At the end of the path, as alpha tends toward zero and the solution tends towards the ordinary least squares, coefficients exhibit big oscillations. In practise it is necessary to tune alpha in such a way that a balance is maintained between both.



```
# Author: Fabian Pedregosa -- <fabian.pedregosa@inria.fr>
# License: BSD 3 clause

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model

# X is the 10x10 Hilbert matrix
X = 1. / (np.arange(1, 11) + np.arange(0, 10)[:, np.newaxis])
y = np.ones(10)

#####
# Compute paths

n_alphas = 200
alphas = np.logspace(-10, -2, n_alphas)

coefs = []
for a in alphas:
    ridge = linear_model.Ridge(alpha=a, fit_intercept=False)
    ridge.fit(X, y)
    coefs.append(ridge.coef_)

#####
```

```
# Display results
ax = plt.gca()

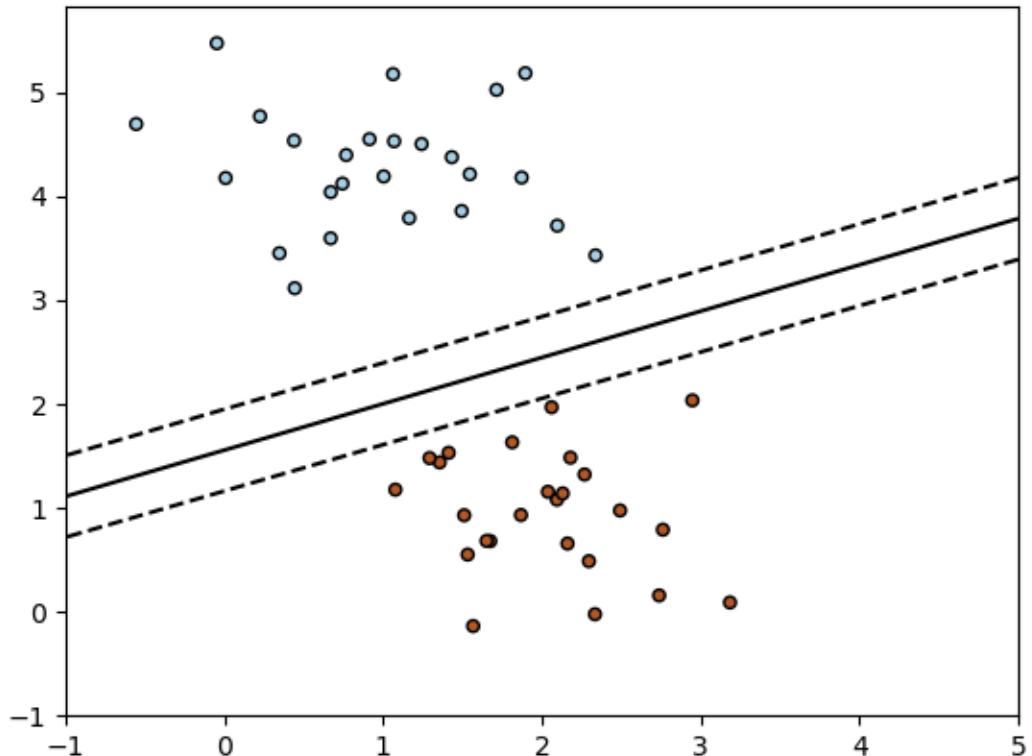
ax.plot(alphas, coefs)
ax.set_xscale('log')
ax.set_xlim(ax.get_xlim()[::-1]) # reverse axis
plt.xlabel('alpha')
plt.ylabel('weights')
plt.title('Ridge coefficients as a function of the regularization')
plt.axis('tight')
plt.show()
```

Total running time of the script: (0 minutes 0.161 seconds)

Note: Click [here](#) to download the full example code

5.18.3 SGD: Maximum margin separating hyperplane

Plot the maximum margin separating hyperplane within a two-class separable dataset using a linear Support Vector Machines classifier trained using SGD.



```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDClassifier
from sklearn.datasets.samples_generator import make_blobs

# we create 50 separable points
X, Y = make_blobs(n_samples=50, centers=2, random_state=0, cluster_std=0.60)

# fit the model
clf = SGDClassifier(loss="hinge", alpha=0.01, max_iter=200,
                     fit_intercept=True, tol=1e-3)
clf.fit(X, Y)

# plot the line, the points, and the nearest vectors to the plane
xx = np.linspace(-1, 5, 10)
yy = np.linspace(-1, 5, 10)

X1, X2 = np.meshgrid(xx, yy)
Z = np.empty(X1.shape)
for (i, j), val in np.ndenumerate(X1):
    x1 = val[0]
    x2 = X2[i, j]
    p = clf.decision_function([[x1, x2]])
    Z[i, j] = p[0]
levels = [-1.0, 0.0, 1.0]
linestyles = ['dashed', 'solid', 'dashed']
colors = 'k'
plt.contour(X1, X2, Z, levels, colors=colors, linestyles=linestyles)
plt.scatter(X[:, 0], X[:, 1], c=Y, cmap=plt.cm.Paired,
            edgecolor='black', s=20)

plt.axis('tight')
plt.show()

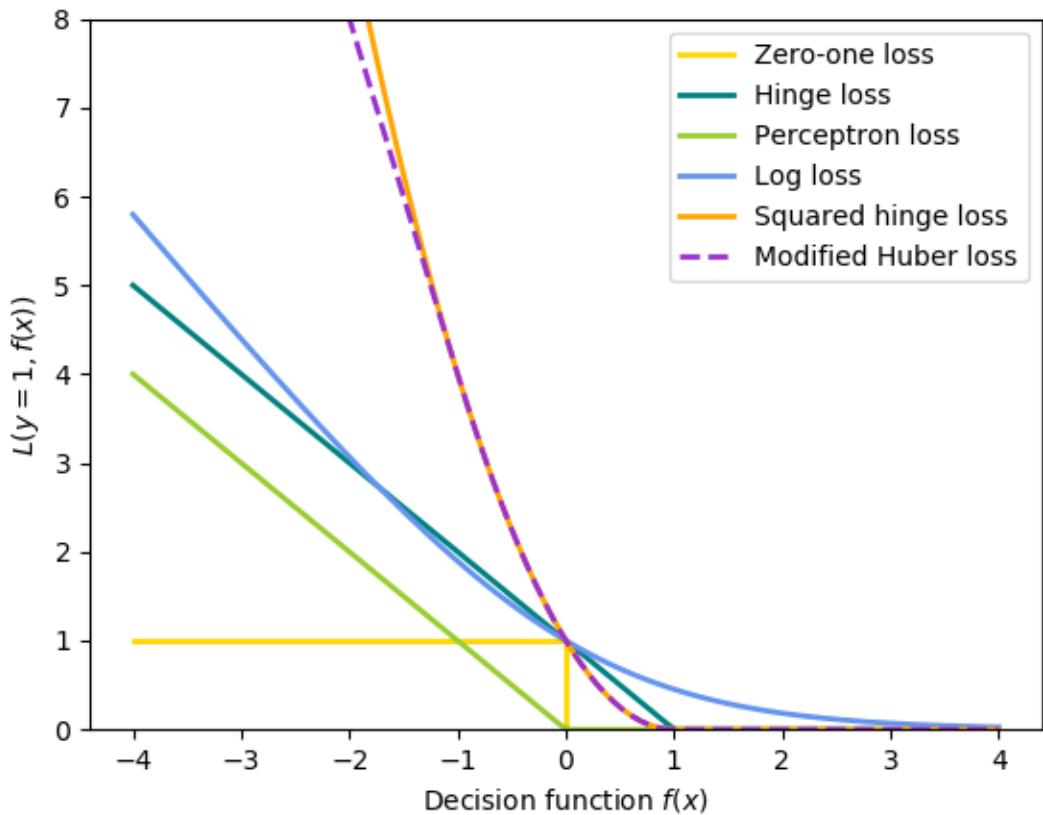
```

Total running time of the script: (0 minutes 0.020 seconds)

Note: Click [here](#) to download the full example code

5.18.4 SGD: convex loss functions

A plot that compares the various convex loss functions supported by `sklearn.linear_model.SGDClassifier`.



```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

def modified_huber_loss(y_true, y_pred):
    z = y_pred * y_true
    loss = -4 * z
    loss[z >= -1] = (1 - z[z >= -1]) ** 2
    loss[z >= 1.] = 0
    return loss

xmin, xmax = -4, 4
xx = np.linspace(xmin, xmax, 100)
lw = 2
plt.plot([xmin, 0, 0, xmax], [1, 1, 0, 0], color='gold', lw=lw,
          label="Zero-one loss")
plt.plot(xx, np.where(xx < 1, 1 - xx, 0), color='teal', lw=lw,
          label="Hinge loss")
plt.plot(xx, -np.minimum(xx, 0), color='yellowgreen', lw=lw,
          label="Perceptron loss")
plt.plot(xx, np.log2(1 + np.exp(-xx)), color='cornflowerblue', lw=lw,
          label="Log loss")
plt.plot(xx, np.where(xx < 1, 1 - xx, 0) ** 2, color='orange', lw=lw,
          label="Squared hinge loss")
plt.plot(xx, modified_huber_loss(np.ones_like(xx), xx), color='purple', lw=lw,
          label="Modified Huber loss")

```

```

label="Squared hinge loss")
plt.plot(xx, modified_huber_loss(xx, 1), color='darkorchid', lw=lw,
         linestyle='--', label="Modified Huber loss")
plt.ylim((0, 8))
plt.legend(loc="upper right")
plt.xlabel(r"Decision function $f(x)$")
plt.ylabel("$L(y=1, f(x))$")
plt.show()

```

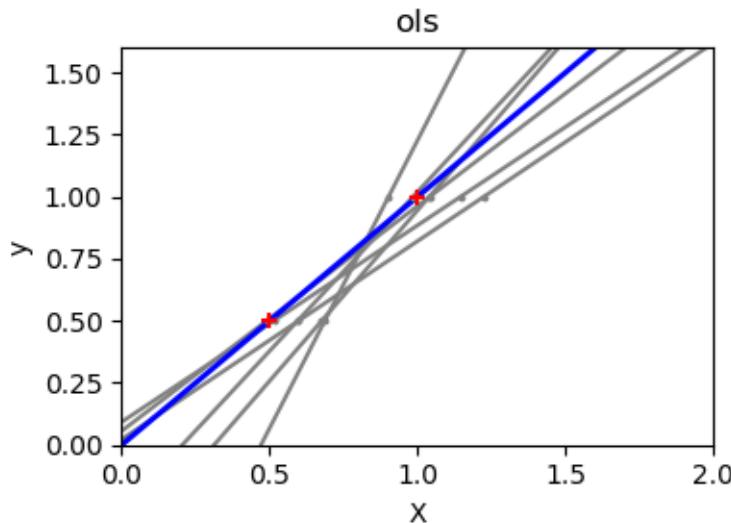
Total running time of the script: (0 minutes 0.019 seconds)

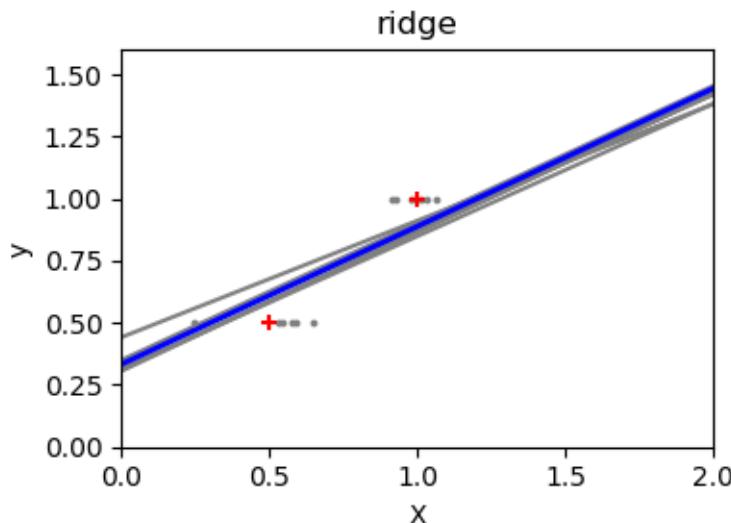
Note: Click [here](#) to download the full example code

5.18.5 Ordinary Least Squares and Ridge Regression Variance

Due to the few points in each dimension and the straight line that linear regression uses to follow these points as well as it can, noise on the observations will cause great variance as shown in the first plot. Every line's slope can vary quite a bit for each prediction due to the noise induced in the observations.

Ridge regression is basically minimizing a penalised version of the least-squared function. The penalising shrinks the value of the regression coefficients. Despite the few data points in each dimension, the slope of the prediction is much more stable and the variance in the line itself is greatly reduced, in comparison to that of the standard linear regression





```

print(__doc__)

# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause


import numpy as np
import matplotlib.pyplot as plt

from sklearn import linear_model

X_train = np.c_[.5, 1].T
y_train = [.5, 1]
X_test = np.c_[0, 2].T

np.random.seed(0)

classifiers = dict(ols=linear_model.LinearRegression(),
                   ridge=linear_model.Ridge(alpha=.1))

for name, clf in classifiers.items():
    fig, ax = plt.subplots(figsize=(4, 3))

    for _ in range(6):
        this_X = .1 * np.random.normal(size=(2, 1)) + X_train
        clf.fit(this_X, y_train)

        ax.plot(X_test, clf.predict(X_test), color='gray')
        ax.scatter(this_X, y_train, s=3, c='gray', marker='o', zorder=10)

    clf.fit(X_train, y_train)
    ax.plot(X_test, clf.predict(X_test), linewidth=2, color='blue')
    ax.scatter(X_train, y_train, s=30, c='red', marker='+', zorder=10)

    ax.set_title(name)
    ax.set_xlim(0, 2)

```

```

ax.set_xlim((0, 1.6))
ax.set_xlabel('X')
ax.set_ylabel('y')

fig.tight_layout()

plt.show()

```

Total running time of the script: (0 minutes 0.130 seconds)

Note: Click [here](#) to download the full example code

5.18.6 Plot Ridge coefficients as a function of the L2 regularization

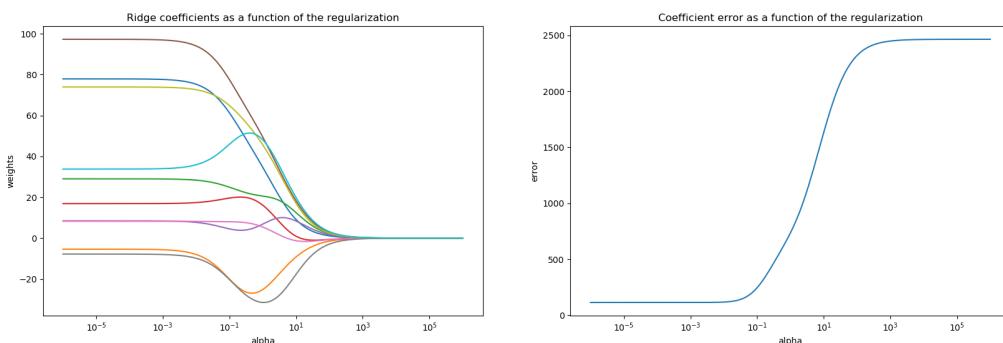
Ridge Regression is the estimator used in this example. Each color in the left plot represents one different dimension of the coefficient vector, and this is displayed as a function of the regularization parameter. The right plot shows how exact the solution is. This example illustrates how a well defined solution is found by Ridge regression and how regularization affects the coefficients and their values. The plot on the right shows how the difference of the coefficients from the estimator changes as a function of regularization.

In this example the dependent variable Y is set as a function of the input features: $y = X^*w + c$. The coefficient vector w is randomly sampled from a normal distribution, whereas the bias term c is set to a constant.

As alpha tends toward zero the coefficients found by Ridge regression stabilize towards the randomly sampled vector w. For big alpha (strong regularisation) the coefficients are smaller (eventually converging at 0) leading to a simpler and biased solution. These dependencies can be observed on the left plot.

The right plot shows the mean squared error between the coefficients found by the model and the chosen vector w. Less regularised models retrieve the exact coefficients (error is equal to 0), stronger regularised models increase the error.

Please note that in this example the data is non-noisy, hence it is possible to extract the exact coefficients.



```

# Author: Kornel Kielczewski -- <kornel.k@plusnet.pl>

print(__doc__)

import matplotlib.pyplot as plt
import numpy as np

from sklearn.datasets import make_regression
from sklearn.linear_model import Ridge

```

```
from sklearn.metrics import mean_squared_error

clf = Ridge()

X, y, w = make_regression(n_samples=10, n_features=10, coef=True,
                           random_state=1, bias=3.5)

coefs = []
errors = []

alphas = np.logspace(-6, 6, 200)

# Train the model with different regularisation strengths
for a in alphas:
    clf.set_params(alpha=a)
    clf.fit(X, y)
    coefs.append(clf.coef_)
    errors.append(mean_squared_error(clf.coef_, w))

# Display results
plt.figure(figsize=(20, 6))

plt.subplot(121)
ax = plt.gca()
ax.plot(alphas, coefs)
ax.set_xscale('log')
plt.xlabel('alpha')
plt.ylabel('weights')
plt.title('Ridge coefficients as a function of the regularization')
plt.axis('tight')

plt.subplot(122)
ax = plt.gca()
ax.plot(alphas, errors)
ax.set_xscale('log')
plt.xlabel('alpha')
plt.ylabel('error')
plt.title('Coefficient error as a function of the regularization')
plt.axis('tight')

plt.show()
```

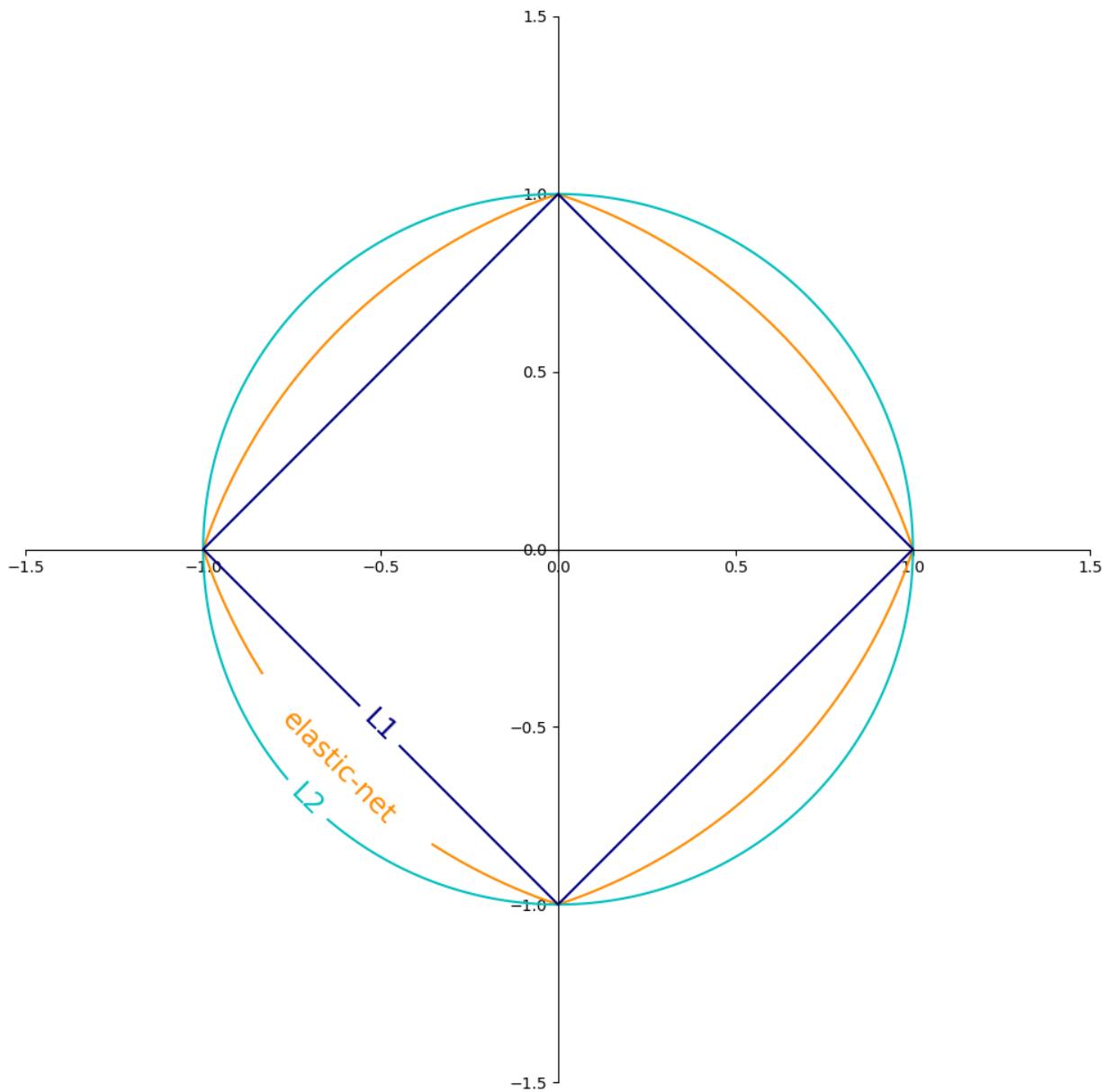
Total running time of the script: (0 minutes 0.126 seconds)

Note: Click [here](#) to download the full example code

5.18.7 SGD: Penalties

Contours of where the penalty is equal to 1 for the three penalties L1, L2 and elastic-net.

All of the above are supported by `sklearn.linear_model.stochastic_gradient`.



```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

l1_color = "navy"
l2_color = "c"
elastic_net_color = "darkorange"

line = np.linspace(-1.5, 1.5, 1001)
xx, yy = np.meshgrid(line, line)

l2 = xx ** 2 + yy ** 2
l1 = np.abs(xx) + np.abs(yy)
rho = 0.5
```

```
elastic_net = rho * 11 + (1 - rho) * 12

plt.figure(figsize=(10, 10), dpi=100)
ax = plt.gca()

elastic_net_contour = plt.contour(xx, yy, elastic_net, levels=[1],
                                 colors=elastic_net_color)
l2_contour = plt.contour(xx, yy, l2, levels=[1], colors=l2_color)
l1_contour = plt.contour(xx, yy, l1, levels=[1], colors=l1_color)
ax.set_aspect("equal")
ax.spines['left'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position('center')
ax.spines['top'].set_color('none')

plt.clabel(elastic_net_contour, inline=1, fontsize=18,
           fmt={1.0: 'elastic-net'}, manual=[(-1, -1)])
plt.clabel(l2_contour, inline=1, fontsize=18,
           fmt={1.0: 'L2'}, manual=[(-1, -1)])
plt.clabel(l1_contour, inline=1, fontsize=18,
           fmt={1.0: 'L1'}, manual=[(-1, -1)])

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.247 seconds)

Note: Click [here](#) to download the full example code

5.18.8 Regularization path of L1- Logistic Regression

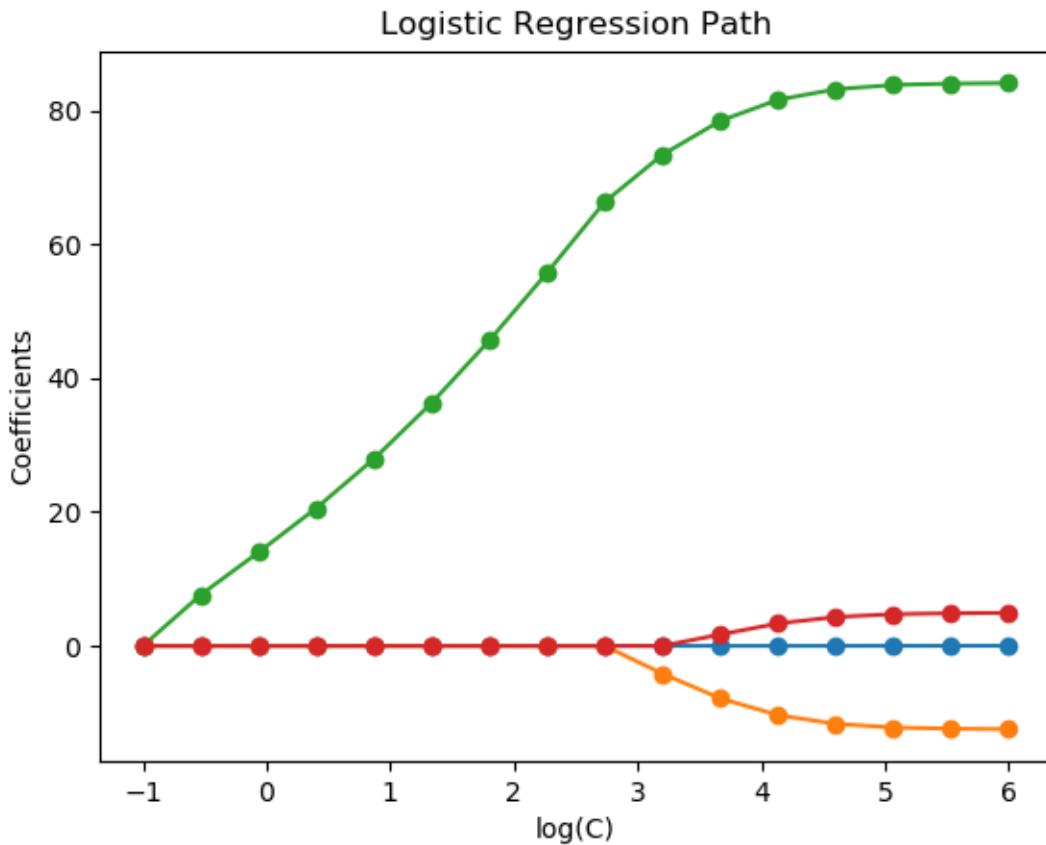
Train l1-penalized logistic regression models on a binary classification problem derived from the Iris dataset.

The models are ordered from strongest regularized to least regularized. The 4 coefficients of the models are collected and plotted as a “regularization path”: on the left-hand side of the figure (strong regularizers), all the coefficients are exactly 0. When regularization gets progressively looser, coefficients can get non-zero values one after the other.

Here we choose the SAGA solver because it can efficiently optimize for the Logistic Regression loss with a non-smooth, sparsity inducing l1 penalty.

Also note that we set a low value for the tolerance to make sure that the model has converged before collecting the coefficients.

We also use warm_start=True which means that the coefficients of the models are reused to initialize the next model fit to speed-up the computation of the full-path.



Out:

```
Computing regularization path ...
This took 2.008s
```

```
print(__doc__)

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause

from time import time
import numpy as np
import matplotlib.pyplot as plt

from sklearn import linear_model
from sklearn import datasets
from sklearn.svm import l1_min_c

iris = datasets.load_iris()
X = iris.data
y = iris.target
```

```
X = X[y != 2]
y = y[y != 2]

X /= X.max()    # Normalize X to speed-up convergence

# ##### Demo path functions

cs = np.logspace(0, 7, 16)

print("Computing regularization path ...")
start = time()
clf = linear_model.LogisticRegression(penalty='l1', solver='saga',
                                       tol=1e-6, max_iter=int(1e6),
                                       warm_start=True)
coefs_ = []
for c in cs:
    clf.set_params(C=c)
    clf.fit(X, y)
    coefs_.append(clf.coef_.ravel().copy())
print("This took %0.3fs" % (time() - start))

coefs_ = np.array(coefs_)
plt.plot(np.log10(cs), coefs_, marker='o')
ymin, ymax = plt.ylim()
plt.xlabel('log(C)')
plt.ylabel('Coefficients')
plt.title('Logistic Regression Path')
plt.axis('tight')
plt.show()
```

Total running time of the script: (0 minutes 2.022 seconds)

Note: Click [here](#) to download the full example code

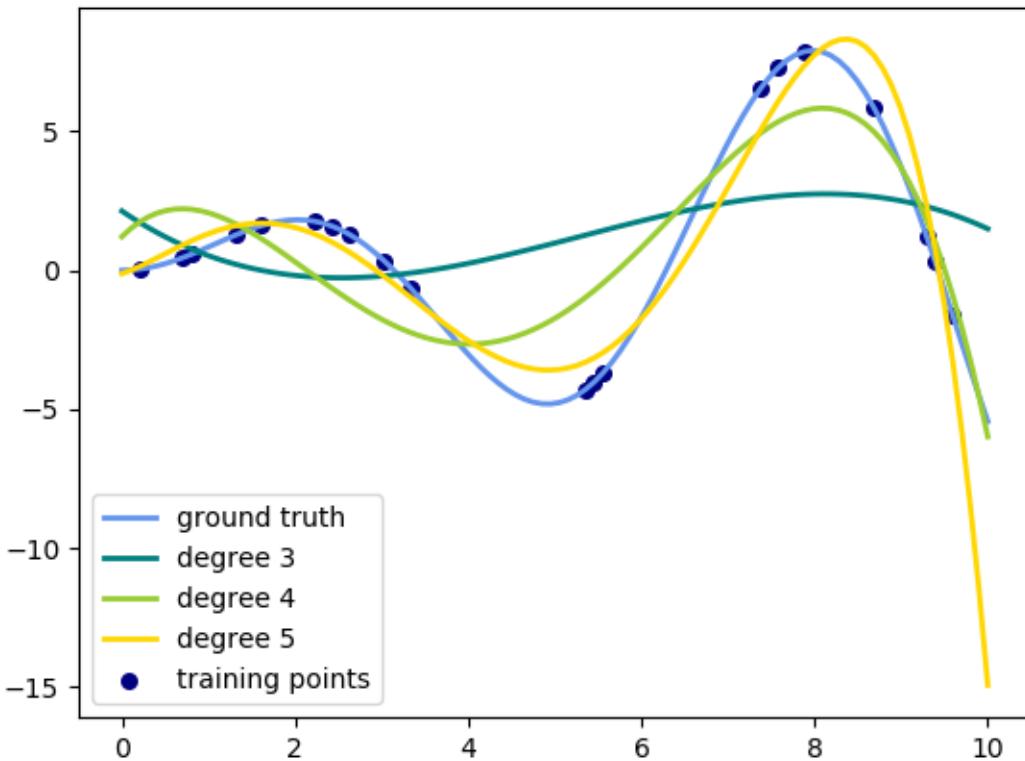
5.18.9 Polynomial interpolation

This example demonstrates how to approximate a function with a polynomial of degree n_degree by using ridge regression. Concretely, from n_samples 1d points, it suffices to build the Vandermonde matrix, which is n_samples x n_degree+1 and has the following form:

```
[[1, x_1, x_1 ** 2, x_1 ** 3, ...], [1, x_2, x_2 ** 2, x_2 ** 3, ...], ...]
```

Intuitively, this matrix can be interpreted as a matrix of pseudo features (the points raised to some power). The matrix is akin to (but different from) the matrix induced by a polynomial kernel.

This example shows that you can do non-linear regression with a linear model, using a pipeline to add non-linear features. Kernel methods extend this idea and can induce very high (even infinite) dimensional feature spaces.



```

print(__doc__)

# Author: Mathieu Blondel
#          Jake Vanderplas
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import Ridge
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

def f(x):
    """ function to approximate by polynomial interpolation"""
    return x * np.sin(x)

# generate points used to plot
x_plot = np.linspace(0, 10, 100)

# generate points and keep a subset of them
x = np.linspace(0, 10, 100)
rng = np.random.RandomState(0)
rng.shuffle(x)

```

```
x = np.sort(x[:20])
y = f(x)

# create matrix versions of these arrays
X = x[:, np.newaxis]
X_plot = X_plot[:, np.newaxis]

colors = ['teal', 'yellowgreen', 'gold']
lw = 2
plt.plot(X_plot, f(X_plot), color='cornflowerblue', linewidth=lw,
          label="ground truth")
plt.scatter(x, y, color='navy', s=30, marker='o', label="training points")

for count, degree in enumerate([3, 4, 5]):
    model = make_pipeline(PolynomialFeatures(degree), Ridge())
    model.fit(X, y)
    y_plot = model.predict(X_plot)
    plt.plot(X_plot, y_plot, color=colors[count], linewidth=lw,
              label="degree %d" % degree)

plt.legend(loc='lower left')

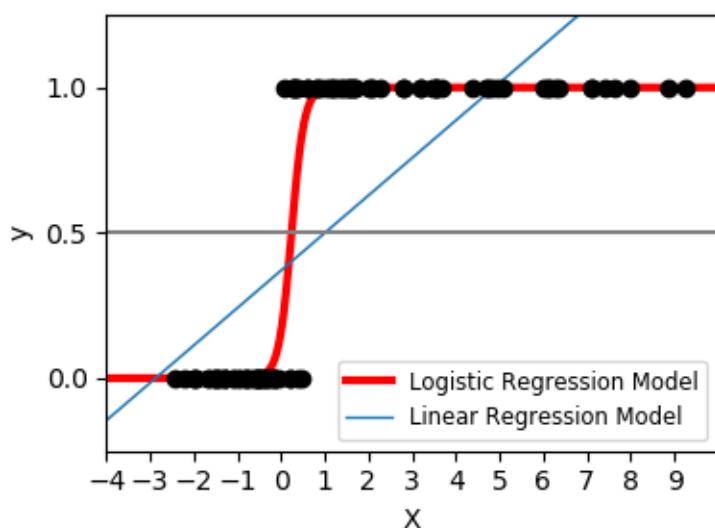
plt.show()
```

Total running time of the script: (0 minutes 0.020 seconds)

Note: Click [here](#) to download the full example code

5.18.10 Logistic function

Shown in the plot is how the logistic regression would, in this synthetic dataset, classify values as either 0 or 1, i.e. class one or two, using the logistic curve.



```

print(__doc__)

# Code source: Gael Varoquaux
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn import linear_model
from scipy.special import expit

# General a toy dataset: it's just a straight line with some Gaussian noise:
xmin, xmax = -5, 5
n_samples = 100
np.random.seed(0)
X = np.random.normal(size=n_samples)
y = (X > 0).astype(np.float)
X[X > 0] *= 4
X += .3 * np.random.normal(size=n_samples)

X = X[:, np.newaxis]

# Fit the classifier
clf = linear_model.LogisticRegression(C=1e5, solver='lbfgs')
clf.fit(X, y)

# and plot the result
plt.figure(1, figsize=(4, 3))
plt.clf()
plt.scatter(X.ravel(), y, color='black', zorder=20)
X_test = np.linspace(-5, 10, 300)

loss = expit(X_test * clf.coef_ + clf.intercept_).ravel()
plt.plot(X_test, loss, color='red', linewidth=3)

ols = linear_model.LinearRegression()
ols.fit(X, y)
plt.plot(X_test, ols.coef_ * X_test + ols.intercept_, linewidth=1)
plt.axhline(.5, color='.5')

plt.ylabel('y')
plt.xlabel('X')
plt.xticks(range(-5, 10))
plt.yticks([0, 0.5, 1])
plt.ylim(-.25, 1.25)
plt.xlim(-4, 10)
plt.legend(['Logistic Regression Model', 'Linear Regression Model'],
          loc="lower right", fontsize='small')
plt.tight_layout()
plt.show()

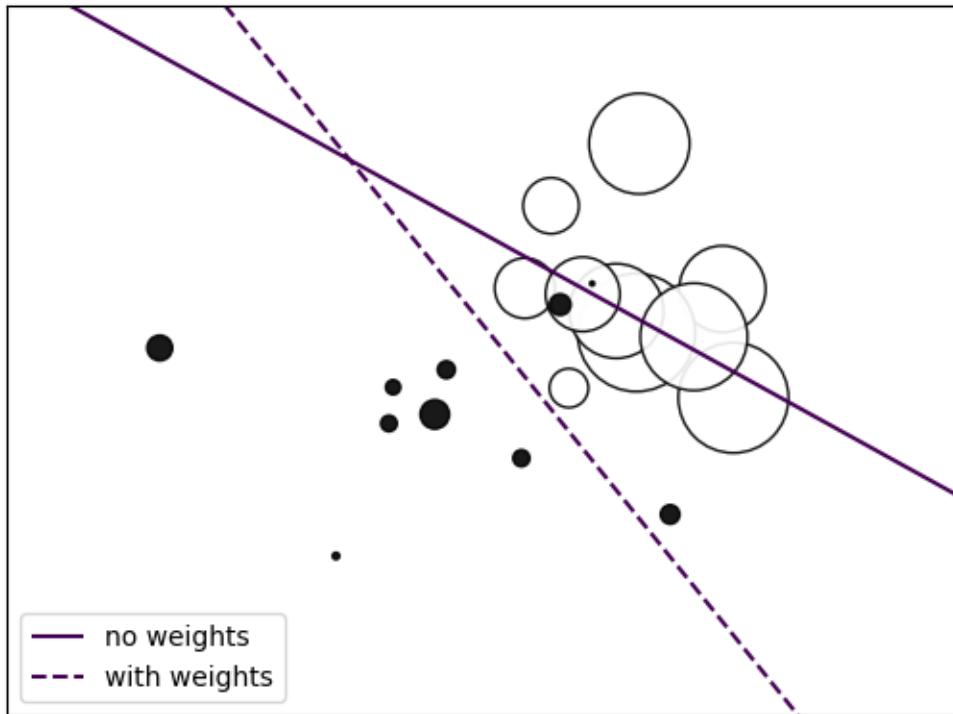
```

Total running time of the script: (0 minutes 0.046 seconds)

Note: Click [here](#) to download the full example code

5.18.11 SGD: Weighted samples

Plot decision function of a weighted dataset, where the size of points is proportional to its weight.



```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model

# we create 20 points
np.random.seed(0)
X = np.r_[np.random.randn(10, 2) + [1, 1], np.random.randn(10, 2)]
y = [1] * 10 + [-1] * 10
sample_weight = 100 * np.abs(np.random.randn(20))
# and assign a bigger weight to the last 10 samples
sample_weight[:10] *= 10

# plot the weighted data points
xx, yy = np.meshgrid(np.linspace(-4, 5, 500), np.linspace(-4, 5, 500))
plt.figure()
plt.scatter(X[:, 0], X[:, 1], c=y, s=sample_weight, alpha=0.9,
            cmap=plt.cm.bone, edgecolor='black')

# fit the unweighted model
clf = linear_model.SGDClassifier(alpha=0.01, max_iter=100, tol=1e-3)
```

```

clf.fit(X, y)
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
no_weights = plt.contour(xx, yy, Z, levels=[0], linestyles=['solid'])

# fit the weighted model
clf = linear_model.SGDClassifier(alpha=0.01, max_iter=100, tol=1e-3)
clf.fit(X, y, sample_weight=sample_weight)
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
samples_weights = plt.contour(xx, yy, Z, levels=[0], linestyles=['dashed'])

plt.legend([no_weights.collections[0], samples_weights.collections[0]],
           ["no weights", "with weights"], loc="lower left")

plt.xticks(())
plt.yticks(())
plt.show()

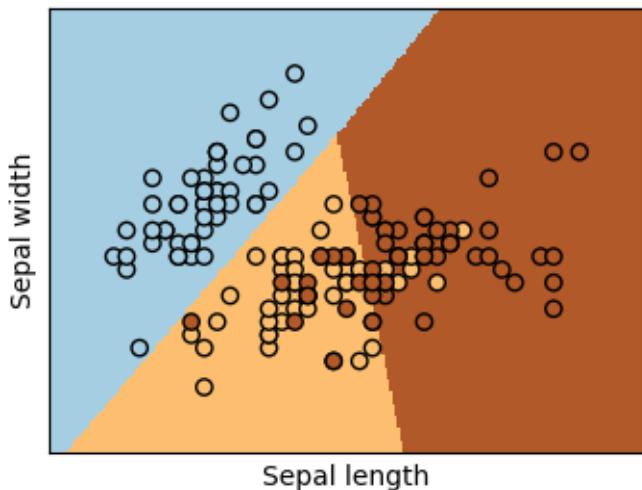
```

Total running time of the script: (0 minutes 0.086 seconds)

Note: Click [here](#) to download the full example code

5.18.12 Logistic Regression 3-class Classifier

Show below is a logistic-regression classifiers decision boundaries on the first two dimensions (sepal length and width) of the `iris` dataset. The datapoints are colored according to their labels.



```

print(__doc__)

# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

```

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn import datasets

# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2] # we only take the first two features.
Y = iris.target

logreg = LogisticRegression(C=1e5, solver='lbfgs', multi_class='multinomial')

# Create an instance of Logistic Regression Classifier and fit the data.
logreg.fit(X, Y)

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
h = .02 # step size in the mesh
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = logreg.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure(1, figsize=(4, 3))
plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)

# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=Y, edgecolors='k', cmap=plt.cm.Paired)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')

plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.xticks(())
plt.yticks(())

plt.show()
```

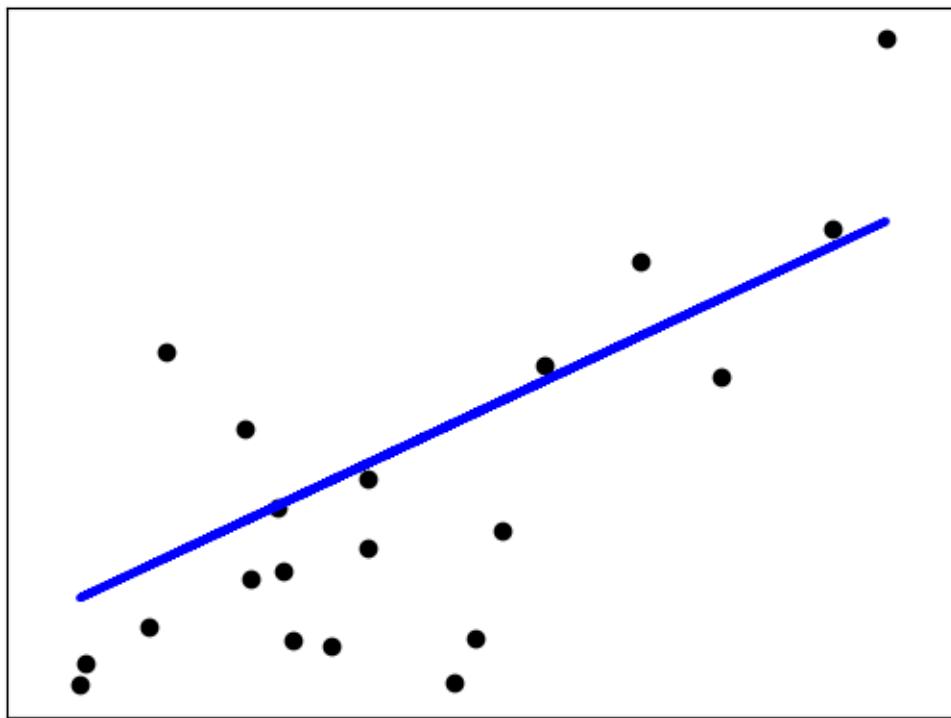
Total running time of the script: (0 minutes 0.083 seconds)

Note: Click [here](#) to download the full example code

5.18.13 Linear Regression Example

This example uses the only the first feature of the `diabetes` dataset, in order to illustrate a two-dimensional plot of this regression technique. The straight line can be seen in the plot, showing how linear regression attempts to draw a straight line that will best minimize the residual sum of squares between the observed responses in the dataset, and the responses predicted by the linear approximation.

The coefficients, the residual sum of squares and the variance score are also calculated.



Out:

```
Coefficients:  
[ 938.23786125]  
Mean squared error: 2548.07  
Variance score: 0.47
```

```
print(__doc__)

# Code source: Jaques Grobler
# License: BSD 3 clause

import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model
from sklearn.metrics import mean_squared_error, r2_score

# Load the diabetes dataset
diabetes = datasets.load_diabetes()
```

```
# Use only one feature
diabetes_X = diabetes.data[:, np.newaxis, 2]

# Split the data into training/testing sets
diabetes_X_train = diabetes_X[:-20]
diabetes_X_test = diabetes_X[-20:]

# Split the targets into training/testing sets
diabetes_y_train = diabetes.target[:-20]
diabetes_y_test = diabetes.target[-20:]

# Create linear regression object
regr = linear_model.LinearRegression()

# Train the model using the training sets
regr.fit(diabetes_X_train, diabetes_y_train)

# Make predictions using the testing set
diabetes_y_pred = regr.predict(diabetes_X_test)

# The coefficients
print('Coefficients: \n', regr.coef_)
# The mean squared error
print("Mean squared error: %.2f"
      % mean_squared_error(diabetes_y_test, diabetes_y_pred))
# Explained variance score: 1 is perfect prediction
print('Variance score: %.2f' % r2_score(diabetes_y_test, diabetes_y_pred))

# Plot outputs
plt.scatter(diabetes_X_test, diabetes_y_test, color='black')
plt.plot(diabetes_X_test, diabetes_y_pred, color='blue', linewidth=3)

plt.xticks(())
plt.yticks(())

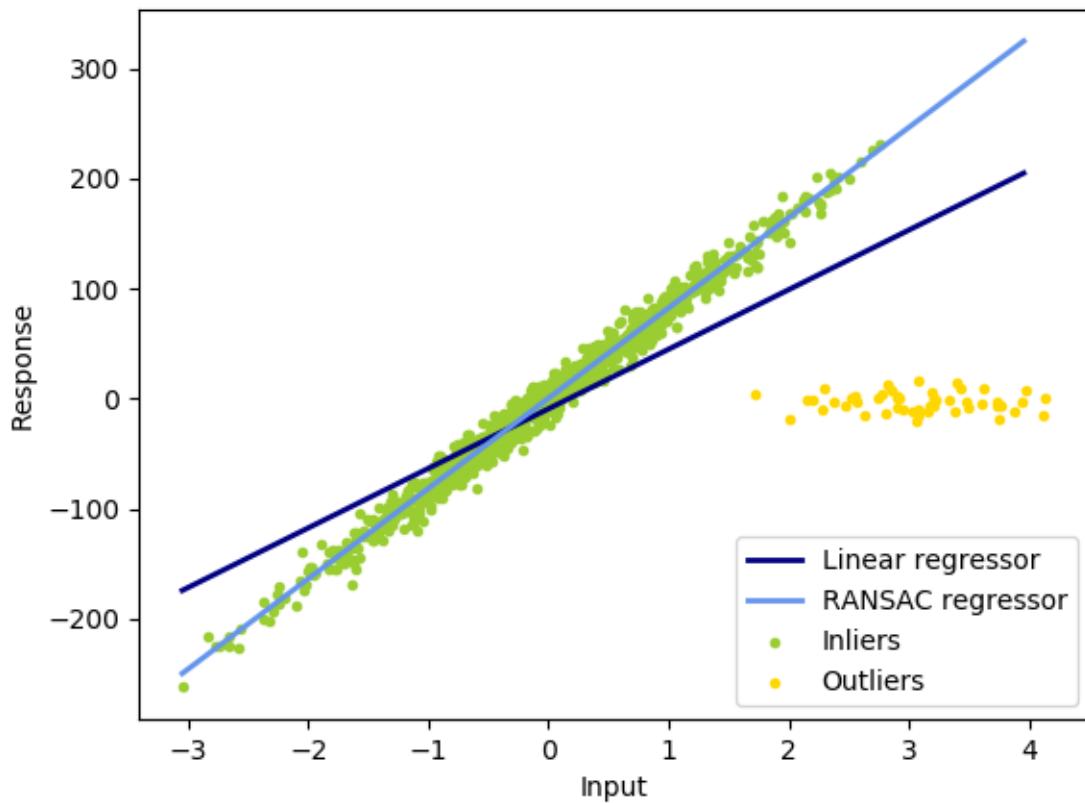
plt.show()
```

Total running time of the script: (0 minutes 0.175 seconds)

Note: Click [here](#) to download the full example code

5.18.14 Robust linear model estimation using RANSAC

In this example we see how to robustly fit a linear model to faulty data using the RANSAC algorithm.



Out:

```
Estimated coefficients (true, linear regression, RANSAC):
82.1903908407869 [54.17236387] [82.08533159]
```

```
import numpy as np
from matplotlib import pyplot as plt

from sklearn import linear_model, datasets

n_samples = 1000
n_outliers = 50

X, y, coef = datasets.make_regression(n_samples=n_samples, n_features=1,
                                       n_informative=1, noise=10,
                                       coef=True, random_state=0)

# Add outlier data
np.random.seed(0)
```

```
X[:n_outliers] = 3 + 0.5 * np.random.normal(size=(n_outliers, 1))
y[:n_outliers] = -3 + 10 * np.random.normal(size=n_outliers)

# Fit line using all data
lr = linear_model.LinearRegression()
lr.fit(X, y)

# Robustly fit linear model with RANSAC algorithm
ransac = linear_model.RANSACRegressor()
ransac.fit(X, y)
inlier_mask = ransac.inlier_mask_
outlier_mask = np.logical_not(inlier_mask)

# Predict data of estimated models
line_X = np.arange(X.min(), X.max())[:, np.newaxis]
line_y = lr.predict(line_X)
line_y_ransac = ransac.predict(line_X)

# Compare estimated coefficients
print("Estimated coefficients (true, linear regression, RANSAC):")
print(coef, lr.coef_, ransac.estimator_.coef_)

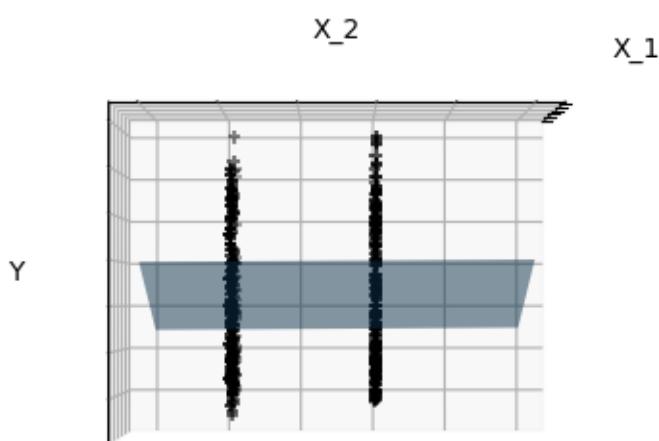
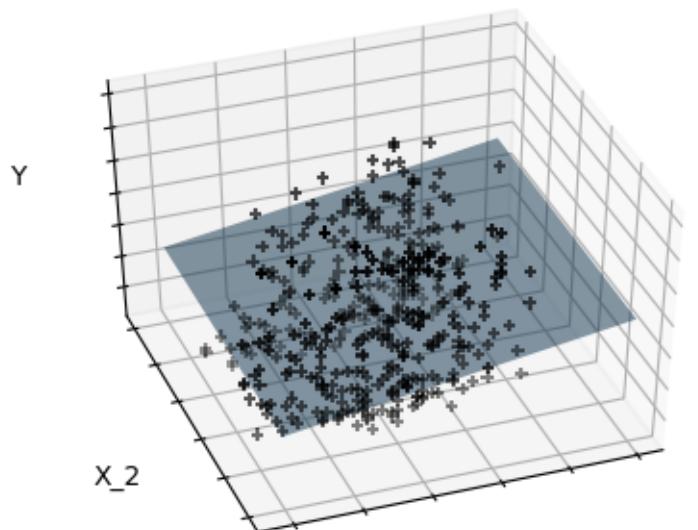
lw = 2
plt.scatter(X[inlier_mask], y[inlier_mask], color='yellowgreen', marker='.',
            label='Inliers')
plt.scatter(X[outlier_mask], y[outlier_mask], color='gold', marker='.',
            label='Outliers')
plt.plot(line_X, line_y, color='navy', linewidth=lw, label='Linear regressor')
plt.plot(line_X, line_y_ransac, color='cornflowerblue', linewidth=lw,
         label='RANSAC regressor')
plt.legend(loc='lower right')
plt.xlabel("Input")
plt.ylabel("Response")
plt.show()
```

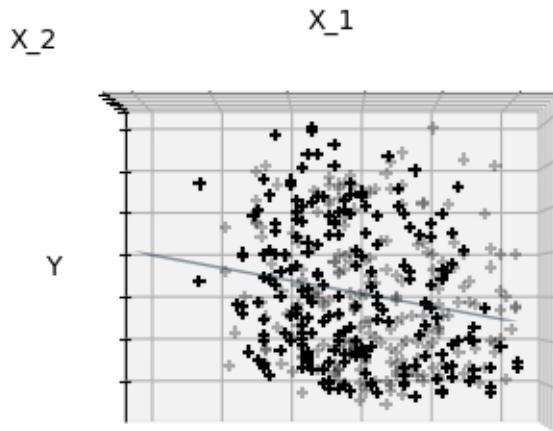
Total running time of the script: (0 minutes 0.028 seconds)

Note: Click [here](#) to download the full example code

5.18.15 Sparsity Example: Fitting only features 1 and 2

Features 1 and 2 of the diabetes-dataset are fitted and plotted below. It illustrates that although feature 2 has a strong coefficient on the full model, it does not give us much regarding `y` when compared to just feature 1





```

print(__doc__)

# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

from sklearn import datasets, linear_model

diabetes = datasets.load_diabetes()
indices = (0, 1)

X_train = diabetes.data[:-20, indices]
X_test = diabetes.data[-20:, indices]
y_train = diabetes.target[:-20]
y_test = diabetes.target[-20:]

ols = linear_model.LinearRegression()
ols.fit(X_train, y_train)

# ######
# Plot the figure
def plot_figs(fig_num, elev, azim, X_train, clf):
    fig = plt.figure(fig_num, figsize=(4, 3))
    plt.clf()
    ax = Axes3D(fig, elev=elev, azim=azim)

    ax.scatter(X_train[:, 0], X_train[:, 1], y_train, c='k', marker='+')
    ax.plot_surface(np.array([[-.1, -.1], [.15, .15]]),
                    np.array([[-.1, .15], [-.1, .15]]),
                    clf.predict(np.array([[-.1, -.1, .15, .15],
                                         [-.1, .15, -.1, .15]]).T
                               .reshape((2, 2))),

```

```

        alpha=.5)
ax.set_xlabel('X_1')
ax.set_ylabel('X_2')
ax.set_zlabel('Y')
ax.w_xaxis.set_ticklabels([])
ax.w_yaxis.set_ticklabels([])
ax.w_zaxis.set_ticklabels([])

#Generate the three different figures from different views
elev = 43.5
azim = -110
plot_figs(1, elev, azim, X_train, ols)

elev = -.5
azim = 0
plot_figs(2, elev, azim, X_train, ols)

elev = -.5
azim = 90
plot_figs(3, elev, azim, X_train, ols)

plt.show()

```

Total running time of the script: (0 minutes 0.197 seconds)

Note: Click [here](#) to download the full example code

5.18.16 Lasso on dense and sparse data

We show that linear_model.Lasso provides the same results for dense and sparse data and that in the case of sparse data the speed is improved.

Out:

```

--- Dense matrices
Sparse Lasso done in 0.186067s
Dense Lasso done in 0.034536s
Distance between coefficients : 9.043732562018544e-14
--- Sparse matrices
Matrix density : 0.6263000000000001 %
Sparse Lasso done in 0.284597s
Dense Lasso done in 0.955330s
Distance between coefficients : 7.344760355532163e-12

```

```

print(__doc__)

from time import time
from scipy import sparse
from scipy import linalg

```

```
from sklearn.datasets.samples_generator import make_regression
from sklearn.linear_model import Lasso

# ##########
# The two Lasso implementations on Dense data
print("--- Dense matrices")

X, y = make_regression(n_samples=200, n_features=5000, random_state=0)
X_sp = sparse.coo_matrix(X)

alpha = 1
sparse_lasso = Lasso(alpha=alpha, fit_intercept=False, max_iter=1000)
dense_lasso = Lasso(alpha=alpha, fit_intercept=False, max_iter=1000)

t0 = time()
sparse_lasso.fit(X_sp, y)
print("Sparse Lasso done in %fs" % (time() - t0))

t0 = time()
dense_lasso.fit(X, y)
print("Dense Lasso done in %fs" % (time() - t0))

print("Distance between coefficients : %s"
      % linalg.norm(sparse_lasso.coef_ - dense_lasso.coef_))

# ##########
# The two Lasso implementations on Sparse data
print("--- Sparse matrices")

Xs = X.copy()
Xs[Xs < 2.5] = 0.0
Xs = sparse.coo_matrix(Xs)
Xs = Xs.tocsc()

print("Matrix density : %s %%" % (Xs.nnz / float(X.size) * 100))

alpha = 0.1
sparse_lasso = Lasso(alpha=alpha, fit_intercept=False, max_iter=10000)
dense_lasso = Lasso(alpha=alpha, fit_intercept=False, max_iter=10000)

t0 = time()
sparse_lasso.fit(Xs, y)
print("Sparse Lasso done in %fs" % (time() - t0))

t0 = time()
dense_lasso.fit(Xs.toarray(), y)
print("Dense Lasso done in %fs" % (time() - t0))

print("Distance between coefficients : %s"
      % linalg.norm(sparse_lasso.coef_ - dense_lasso.coef_))
```

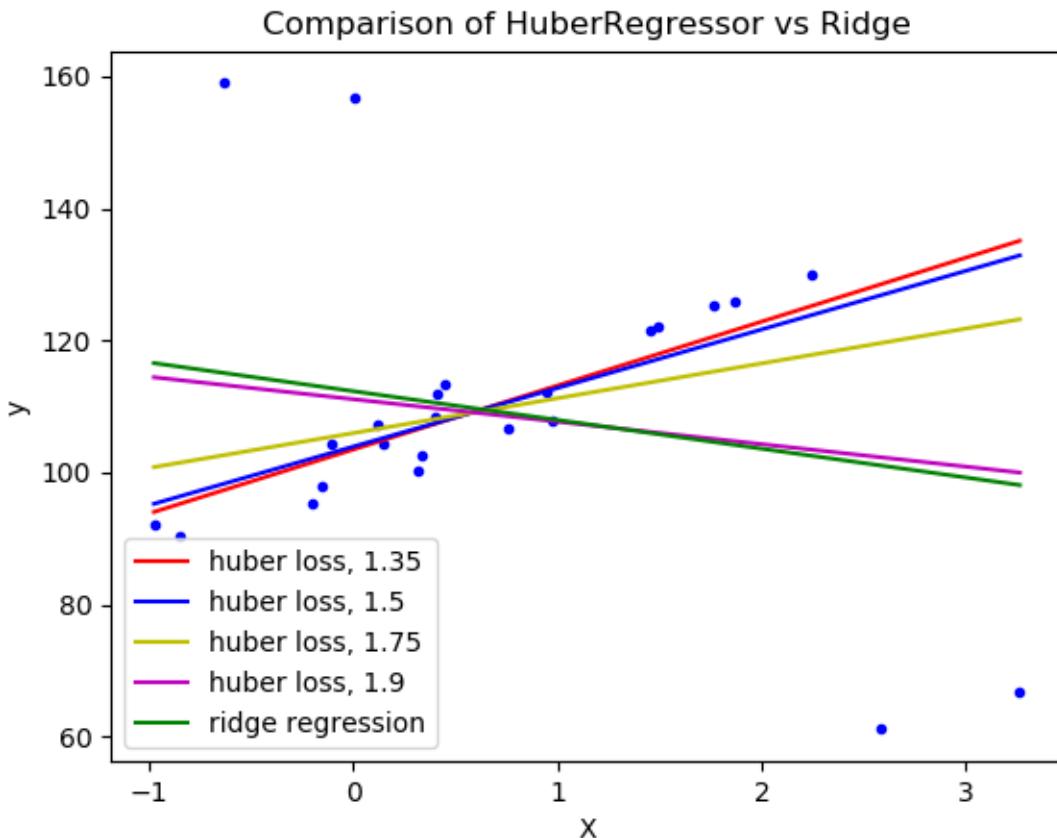
Total running time of the script: (0 minutes 1.565 seconds)

Note: Click [here](#) to download the full example code

5.18.17 HuberRegressor vs Ridge on dataset with strong outliers

Fit Ridge and HuberRegressor on a dataset with outliers.

The example shows that the predictions in ridge are strongly influenced by the outliers present in the dataset. The Huber regressor is less influenced by the outliers since the model uses the linear loss for these. As the parameter epsilon is increased for the Huber regressor, the decision function approaches that of the ridge.



```
# Authors: Manoj Kumar mks542@nyu.edu
# License: BSD 3 clause

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_regression
from sklearn.linear_model import HuberRegressor, Ridge

# Generate toy data.
rng = np.random.RandomState(0)
X, y = make_regression(n_samples=20, n_features=1, random_state=0, noise=4.0,
                      bias=100.0)

# Add four strong outliers to the dataset.
X_outliers = rng.normal(0, 0.5, size=(4, 1))
```

```
y_outliers = rng.normal(0, 2.0, size=4)
X_outliers[:2, :] += X.max() + X.mean() / 4.
X_outliers[2:, :] += X.min() - X.mean() / 4.
y_outliers[:2] += y.min() - y.mean() / 4.
y_outliers[2:] += y.max() + y.mean() / 4.
X = np.vstack((X, X_outliers))
y = np.concatenate((y, y_outliers))
plt.plot(X, y, 'b.')

# Fit the huber regressor over a series of epsilon values.
colors = ['r-', 'b-', 'y-', 'm-']

x = np.linspace(X.min(), X.max(), 7)
epsilon_values = [1.35, 1.5, 1.75, 1.9]
for k, epsilon in enumerate(epsilon_values):
    huber = HuberRegressor(fit_intercept=True, alpha=0.0, max_iter=100,
                           epsilon=epsilon)
    huber.fit(X, y)
    coef_ = huber.coef_ * x + huber.intercept_
    plt.plot(x, coef_, colors[k], label="huber loss, %s" % epsilon)

# Fit a ridge regressor to compare it to huber regressor.
ridge = Ridge(fit_intercept=True, alpha=0.0, random_state=0, normalize=True)
ridge.fit(X, y)
coef_ridge = ridge.coef_
coef_ = ridge.coef_ * x + ridge.intercept_
plt.plot(x, coef_, 'g-', label="ridge regression")

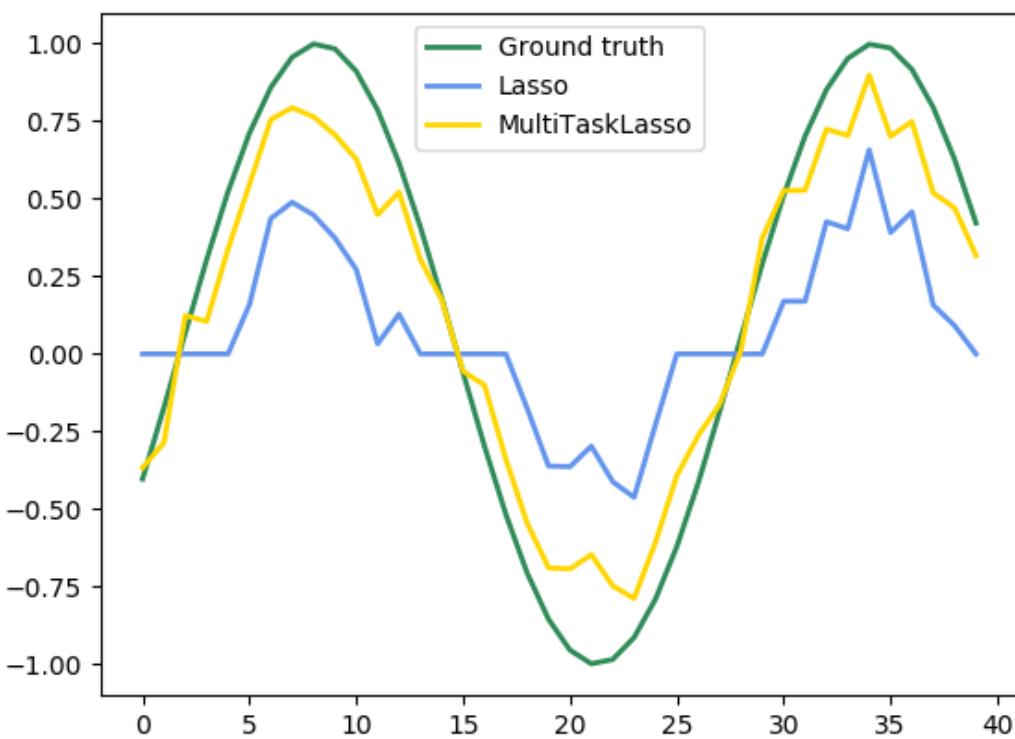
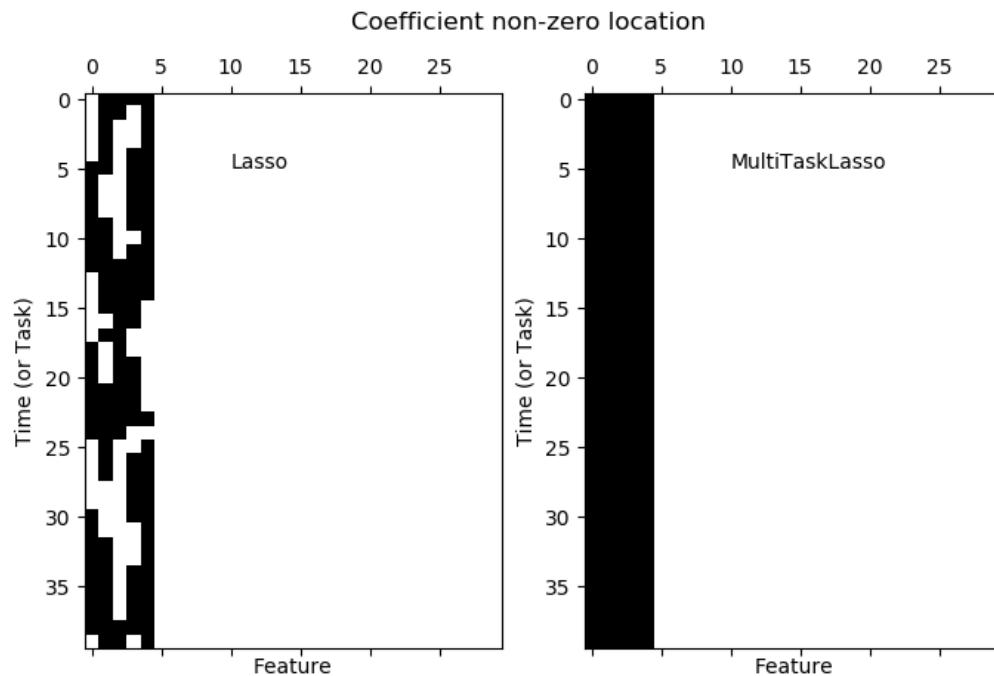
plt.title("Comparison of HuberRegressor vs Ridge")
plt.xlabel("X")
plt.ylabel("y")
plt.legend(loc=0)
plt.show()
```

Total running time of the script: (0 minutes 0.032 seconds)

Note: Click [here](#) to download the full example code

5.18.18 Joint feature selection with multi-task Lasso

The multi-task lasso allows to fit multiple regression problems jointly enforcing the selected features to be the same across tasks. This example simulates sequential measurements, each task is a time instant, and the relevant features vary in amplitude over time while being the same. The multi-task lasso imposes that features that are selected at one time point are select for all time point. This makes feature selection by the Lasso more stable.



```

print(__doc__)

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause

import matplotlib.pyplot as plt
import numpy as np

from sklearn.linear_model import MultiTaskLasso, Lasso

rng = np.random.RandomState(42)

# Generate some 2D coefficients with sine waves with random frequency and phase
n_samples, n_features, n_tasks = 100, 30, 40
n_relevant_features = 5
coef = np.zeros((n_tasks, n_features))
times = np.linspace(0, 2 * np.pi, n_tasks)
for k in range(n_relevant_features):
    coef[:, k] = np.sin((1. + rng.randn(1)) * times + 3 * rng.randn(1))

X = rng.randn(n_samples, n_features)
Y = np.dot(X, coef.T) + rng.randn(n_samples, n_tasks)

coef_lasso_ = np.array([Lasso(alpha=0.5).fit(X, y).coef_ for y in Y.T])
coef_multi_task_lasso_ = MultiTaskLasso(alpha=1.).fit(X, Y).coef_

#####
# Plot support and time series
fig = plt.figure(figsize=(8, 5))
plt.subplot(1, 2, 1)
plt.spy(coef_lasso_)
plt.xlabel('Feature')
plt.ylabel('Time (or Task)')
plt.text(10, 5, 'Lasso')
plt.subplot(1, 2, 2)
plt.spy(coef_multi_task_lasso_)
plt.xlabel('Feature')
plt.ylabel('Time (or Task)')
plt.text(10, 5, 'MultiTaskLasso')
fig.suptitle('Coefficient non-zero location')

feature_to_plot = 0
plt.figure()
lw = 2
plt.plot(coef[:, feature_to_plot], color='seagreen', linewidth=lw,
         label='Ground truth')
plt.plot(coef_lasso_[:, feature_to_plot], color='cornflowerblue', linewidth=lw,
         label='Lasso')
plt.plot(coef_multi_task_lasso_[:, feature_to_plot], color='gold', linewidth=lw,
         label='MultiTaskLasso')
plt.legend(loc='upper center')
plt.axis('tight')
plt.ylim([-1.1, 1.1])
plt.show()

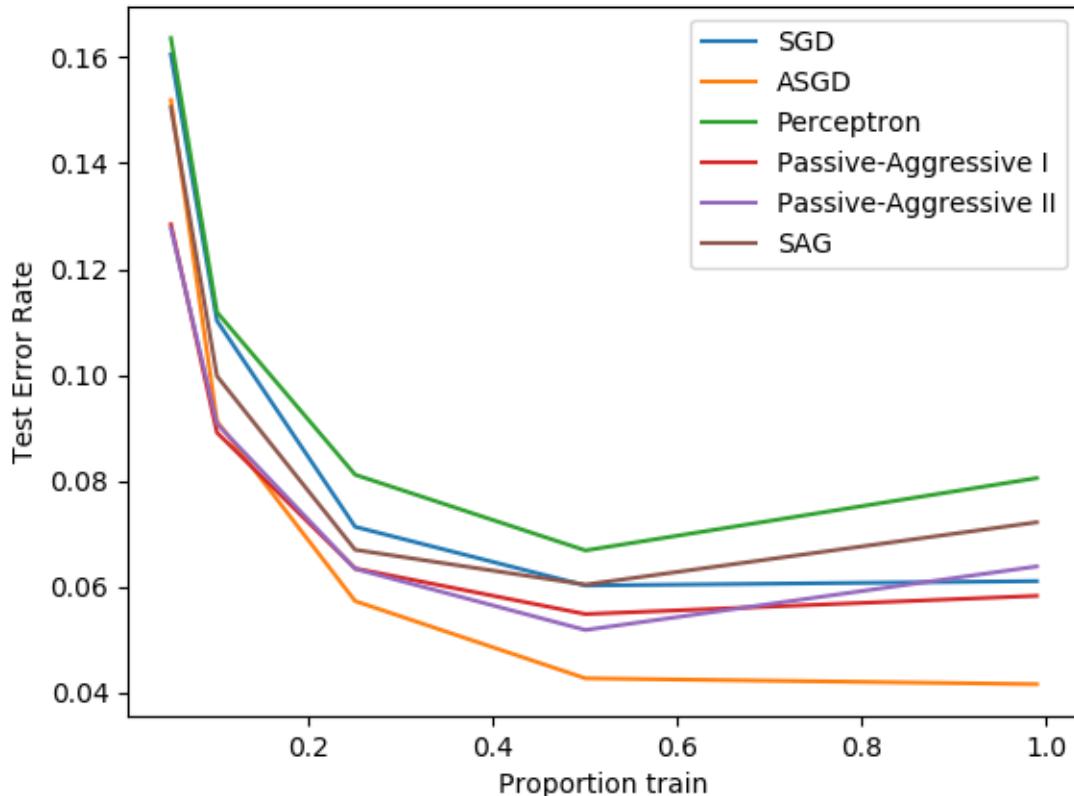
```

Total running time of the script: (0 minutes 0.061 seconds)

Note: Click [here](#) to download the full example code

5.18.19 Comparing various online solvers

An example showing how different online solvers perform on the hand-written digits dataset.



Out:

```
training SGD
training ASGD
training Perceptron
training Passive-Aggressive I
training Passive-Aggressive II
training SAG
```

```
# Author: Rob Zinkov <rob at zinkov dot com>
# License: BSD 3 clause
```

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets

from sklearn.model_selection import train_test_split
from sklearn.linear_model import SGDClassifier, Perceptron
from sklearn.linear_model import PassiveAggressiveClassifier
from sklearn.linear_model import LogisticRegression

heldout = [0.95, 0.90, 0.75, 0.50, 0.01]
rounds = 20
digits = datasets.load_digits()
X, y = digits.data, digits.target

classifiers = [
    ("SGD", SGDClassifier(max_iter=100, tol=1e-3)),
    ("ASGD", SGDClassifier(average=True, max_iter=1000, tol=1e-3)),
    ("Perceptron", Perceptron(tol=1e-3)),
    ("Passive-Aggressive I", PassiveAggressiveClassifier(loss='hinge',
                                                          C=1.0, tol=1e-4)),
    ("Passive-Aggressive II", PassiveAggressiveClassifier(loss='squared_hinge',
                                                          C=1.0, tol=1e-4)),
    ("SAG", LogisticRegression(solver='sag', tol=1e-1, C=1.e4 / X.shape[0],
                               multi_class='auto'))
]

xx = 1. - np.array(heldout)

for name, clf in classifiers:
    print("training %s" % name)
    rng = np.random.RandomState(42)
    yy = []
    for i in heldout:
        yy_ = []
        for r in range(rounds):
            X_train, X_test, y_train, y_test = \
                train_test_split(X, y, test_size=i, random_state=rng)
            clf.fit(X_train, y_train)
            y_pred = clf.predict(X_test)
            yy_.append(1 - np.mean(y_pred == y_test))
        yy.append(np.mean(yy_))
    plt.plot(xx, yy, label=name)

plt.legend(loc="upper right")
plt.xlabel("Proportion train")
plt.ylabel("Test Error Rate")
plt.show()
```

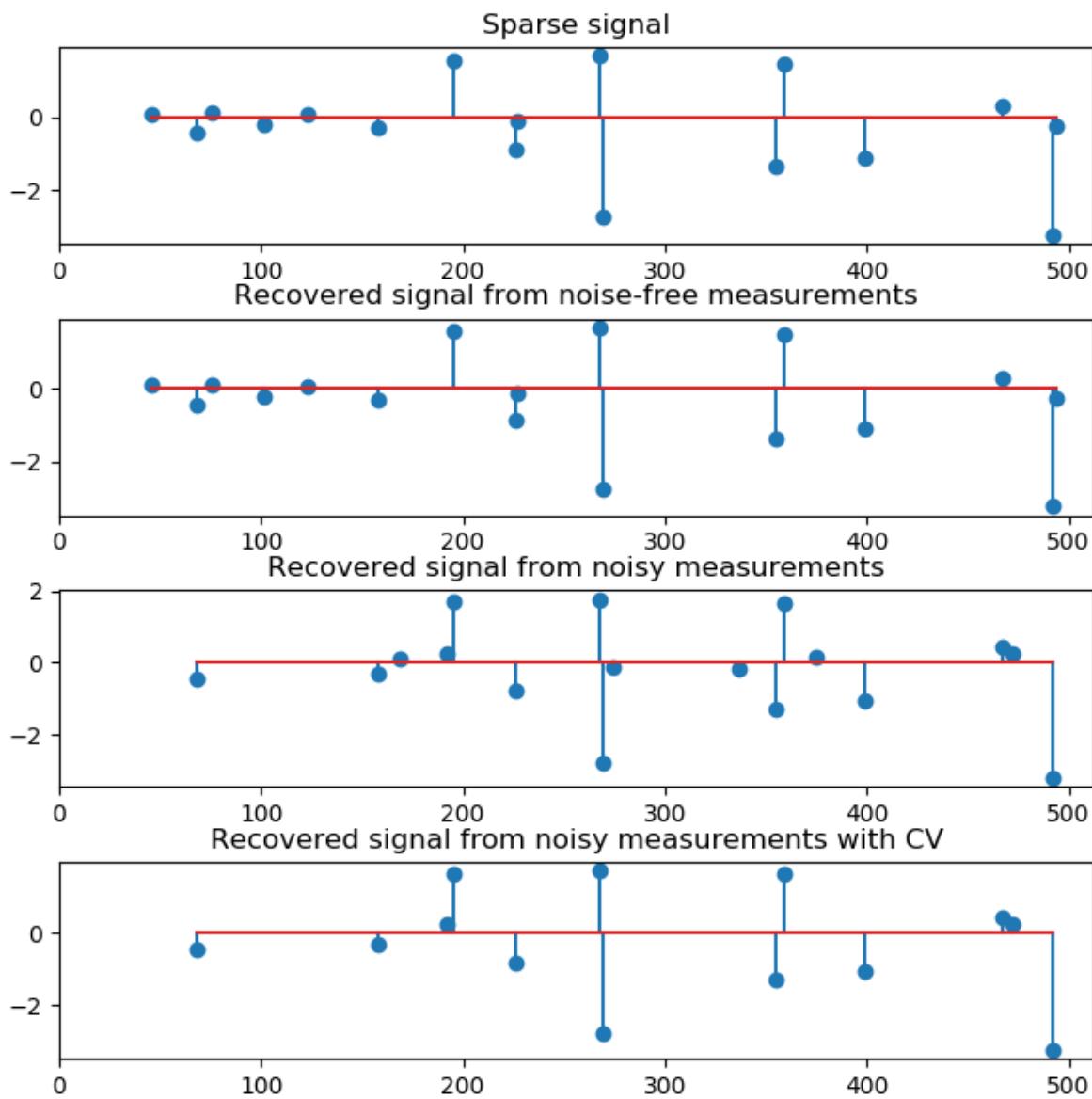
Total running time of the script: (0 minutes 24.266 seconds)

Note: Click [here](#) to download the full example code

5.18.20 Orthogonal Matching Pursuit

Using orthogonal matching pursuit for recovering a sparse signal from a noisy measurement encoded with a dictionary

Sparse signal recovery with Orthogonal Matching Pursuit



```
print(__doc__)

import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import OrthogonalMatchingPursuit
from sklearn.linear_model import OrthogonalMatchingPursuitCV
from sklearn.datasets import make_sparse_coded_signal

n_components, n_features = 512, 100
n_nonzero_coefs = 17

# generate the data

#  $y = Xw$ 
#  $\|x\|_0 = n_{\text{nonzero\_coefs}}$ 
```

```
y, X, w = make_sparse_coded_signal(n_samples=1,
                                    n_components=n_components,
                                    n_features=n_features,
                                    n_nonzero_coefs=n_nonzero_coefs,
                                    random_state=0)

idx, = w.nonzero()

# distort the clean signal
y_noisy = y + 0.05 * np.random.randn(len(y))

# plot the sparse signal
plt.figure(figsize=(7, 7))
plt.subplot(4, 1, 1)
plt.xlim(0, 512)
plt.title("Sparse signal")
plt.stem(idx, w[idx])

# plot the noise-free reconstruction
omp = OrthogonalMatchingPursuit(n_nonzero_coefs=n_nonzero_coefs)
omp.fit(X, y)
coef = omp.coef_
idx_r, = coef.nonzero()
plt.subplot(4, 1, 2)
plt.xlim(0, 512)
plt.title("Recovered signal from noise-free measurements")
plt.stem(idx_r, coef[idx_r])

# plot the noisy reconstruction
omp.fit(X, y_noisy)
coef = omp.coef_
idx_r, = coef.nonzero()
plt.subplot(4, 1, 3)
plt.xlim(0, 512)
plt.title("Recovered signal from noisy measurements")
plt.stem(idx_r, coef[idx_r])

# plot the noisy reconstruction with number of non-zeros set by CV
omp_cv = OrthogonalMatchingPursuitCV(cv=5)
omp_cv.fit(X, y_noisy)
coef = omp_cv.coef_
idx_r, = coef.nonzero()
plt.subplot(4, 1, 4)
plt.xlim(0, 512)
plt.title("Recovered signal from noisy measurements with CV")
plt.stem(idx_r, coef[idx_r])

plt.subplots_adjust(0.06, 0.04, 0.94, 0.90, 0.20, 0.38)
plt.suptitle('Sparse signal recovery with Orthogonal Matching Pursuit',
             fontsize=16)
plt.show()
```

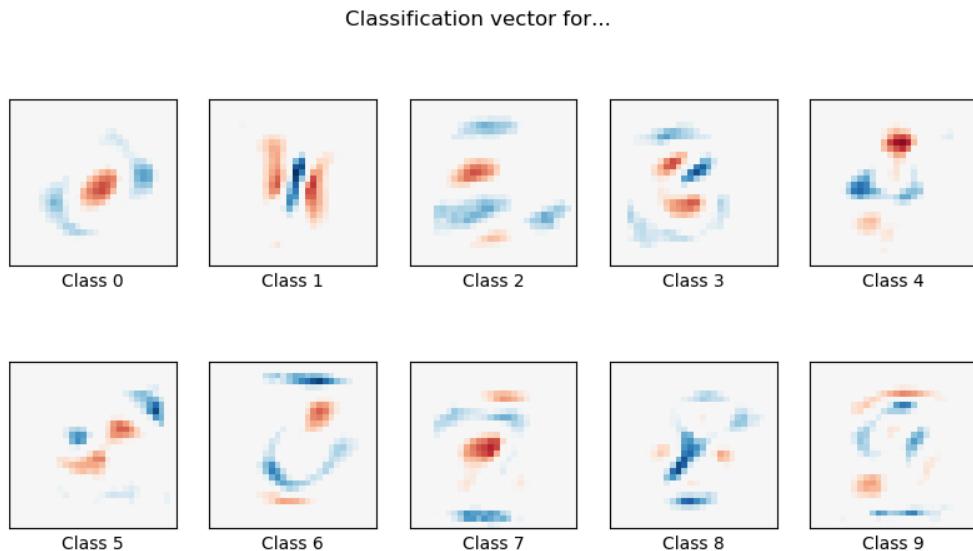
Total running time of the script: (0 minutes 0.499 seconds)

Note: Click [here](#) to download the full example code

5.18.21 MNIST classification using multinomial logistic + L1

Here we fit a multinomial logistic regression with L1 penalty on a subset of the MNIST digits classification task. We use the SAGA algorithm for this purpose: this a solver that is fast when the number of samples is significantly larger than the number of features and is able to finely optimize non-smooth objective functions which is the case with the l1-penalty. Test accuracy reaches > 0.8, while weight vectors remains *sparse* and therefore more easily *interpretable*.

Note that this accuracy of this l1-penalized linear model is significantly below what can be reached by an l2-penalized linear model or a non-linear multi-layer perceptron model on this dataset.



Out:

```
Sparsity with L1 penalty: 80.80%
Test score with L1 penalty: 0.8351
Example run in 28.654 s
```

```
import time
import matplotlib.pyplot as plt
import numpy as np

from sklearn.datasets import fetch_openml
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.utils import check_random_state

print(__doc__)

# Author: Arthur Mensch <arthur.mensch@m4x.org>
# License: BSD 3 clause
```

```
# Turn down for faster convergence
t0 = time.time()
train_samples = 5000

# Load data from https://www.openml.org/d/554
X, y = fetch_openml('mnist_784', version=1, return_X_y=True)

random_state = check_random_state(0)
permutation = random_state.permutation(X.shape[0])
X = X[permutation]
y = y[permutation]
X = X.reshape((X.shape[0], -1))

X_train, X_test, y_train, y_test = train_test_split(
    X, y, train_size=train_samples, test_size=10000)

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

# Turn up tolerance for faster convergence
clf = LogisticRegression(C=50. / train_samples,
                         multi_class='multinomial',
                         penalty='l1', solver='saga', tol=0.1)
clf.fit(X_train, y_train)
sparsity = np.mean(clf.coef_.ravel() == 0) * 100
score = clf.score(X_test, y_test)
# print('Best C %.4f' % clf.C_)
print("Sparsity with L1 penalty: %.2f%%" % sparsity)
print("Test score with L1 penalty: %.4f" % score)

coef = clf.coef_.copy()
plt.figure(figsize=(10, 5))
scale = np.abs(coef).max()
for i in range(10):
    l1_plot = plt.subplot(2, 5, i + 1)
    l1_plot.imshow(coef[i].reshape(28, 28), interpolation='nearest',
                  cmap=plt.cm.RdBu, vmin=-scale, vmax=scale)
    l1_plot.set_xticks(())
    l1_plot.set_yticks(())
    l1_plot.set_xlabel('Class %i' % i)
plt.suptitle('Classification vector for...')

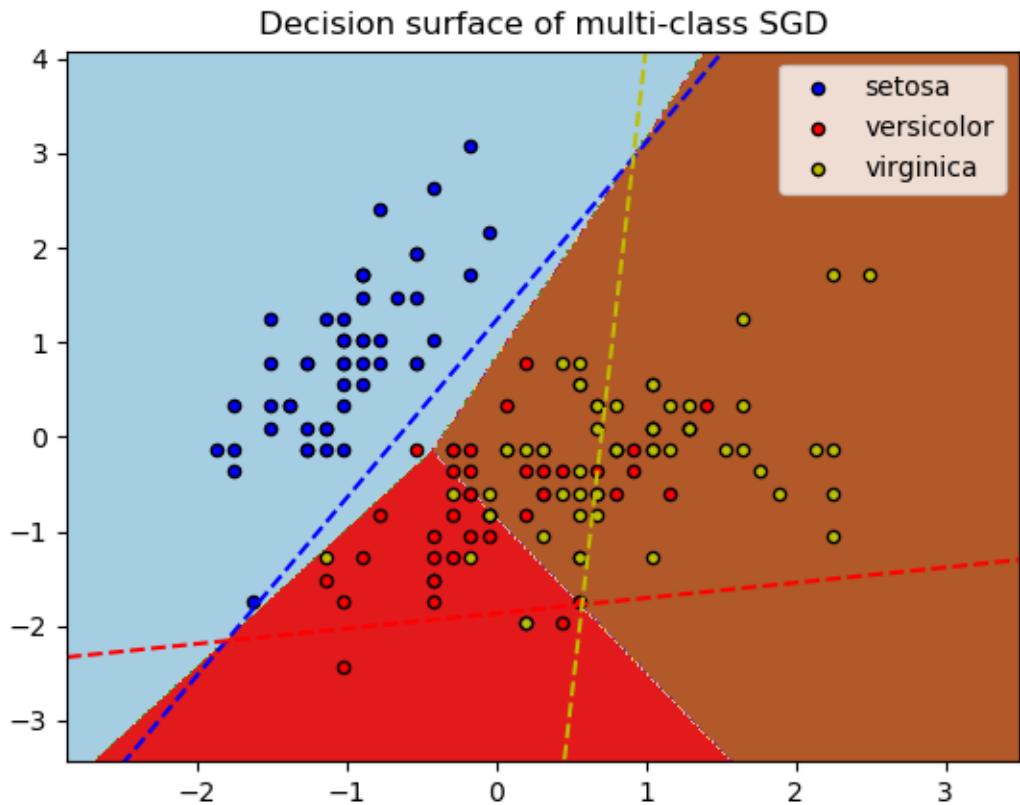
run_time = time.time() - t0
print('Example run in %.3f s' % run_time)
plt.show()
```

Total running time of the script: (0 minutes 28.655 seconds)

Note: Click [here](#) to download the full example code

5.18.22 Plot multi-class SGD on the iris dataset

Plot decision surface of multi-class SGD on iris dataset. The hyperplanes corresponding to the three one-versus-all (OVA) classifiers are represented by the dashed lines.



```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.linear_model import SGDClassifier

# import some data to play with
iris = datasets.load_iris()

# we only take the first two features. We could
# avoid this ugly slicing by using a two-dim dataset
X = iris.data[:, :2]
y = iris.target
colors = "bry"

# shuffle
idx = np.arange(X.shape[0])
np.random.seed(13)
np.random.shuffle(idx)
X = X[idx]
y = y[idx]

# standardize
mean = X.mean(axis=0)
std = X.std(axis=0)

```

```
X = (X - mean) / std

h = .02 # step size in the mesh

clf = SGDClassifier(alpha=0.001, max_iter=100, tol=1e-3).fit(X, y)

# create a mesh to plot in
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                      np.arange(y_min, y_max, h))

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
# Put the result into a color plot
Z = Z.reshape(xx.shape)
cs = plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
plt.axis('tight')

# Plot also the training points
for i, color in zip(clf.classes_, colors):
    idx = np.where(y == i)
    plt.scatter(X[idx, 0], X[idx, 1], c=color, label=iris.target_names[i],
                cmap=plt.cm.Paired, edgecolor='black', s=20)
plt.title("Decision surface of multi-class SGD")
plt.axis('tight')

# Plot the three one-against-all classifiers
xmin, xmax = plt.xlim()
ymin, ymax = plt.ylim()
coef = clf.coef_
intercept = clf.intercept_

def plot_hyperplane(c, color):
    def line(x0):
        return (-(x0 * coef[c, 0]) - intercept[c]) / coef[c, 1]

    plt.plot([xmin, xmax], [line(xmin), line(xmax)],
             ls="--", color=color)

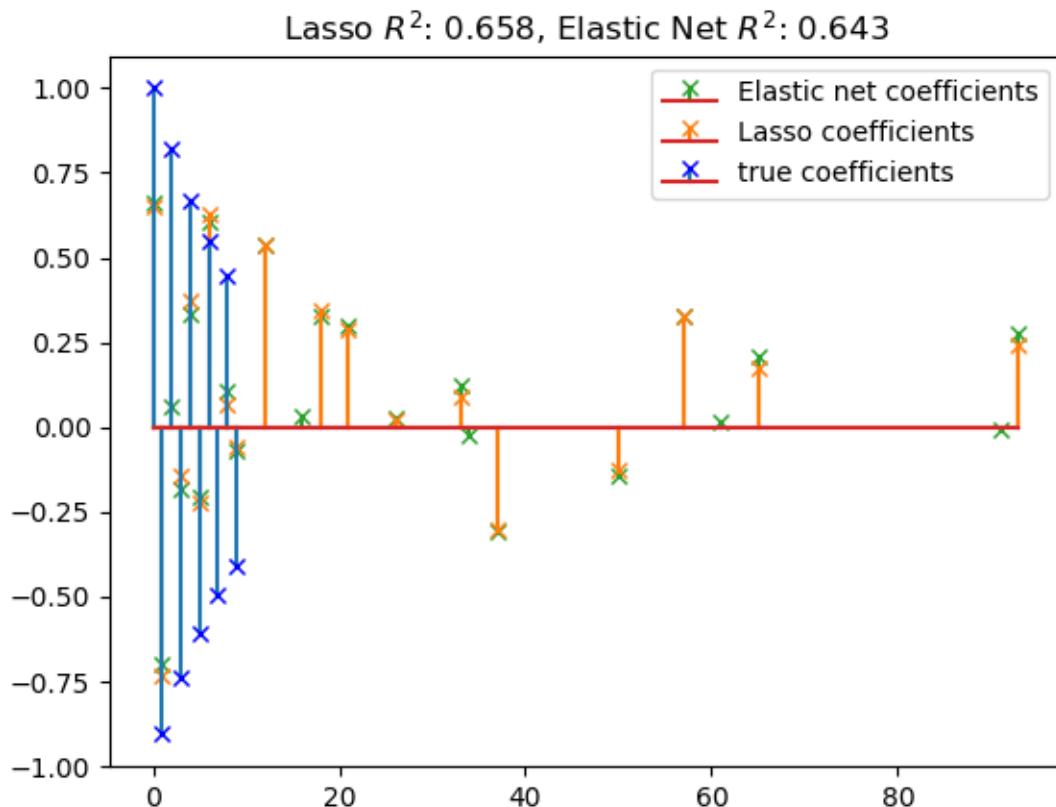
for i, color in zip(clf.classes_, colors):
    plot_hyperplane(i, color)
plt.legend()
plt.show()
```

Total running time of the script: (0 minutes 0.050 seconds)

Note: Click [here](#) to download the full example code

5.18.23 Lasso and Elastic Net for Sparse Signals

Estimates Lasso and Elastic-Net regression models on a manually generated sparse signal corrupted with an additive noise. Estimated coefficients are compared with the ground-truth.



Out:

```
Lasso(alpha=0.1)
r^2 on test data : 0.658064
ElasticNet(alpha=0.1, l1_ratio=0.7)
r^2 on test data : 0.642515
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.metrics import r2_score

# ##########
# Generate some sparse data to play with
```

```

np.random.seed(42)

n_samples, n_features = 50, 100
X = np.random.randn(n_samples, n_features)

# Decreasing coef w. alternated signs for visualization
idx = np.arange(n_features)
coef = (-1) ** idx * np.exp(-idx / 10)
coef[10:] = 0 # sparsify coef
y = np.dot(X, coef)

# Add noise
y += 0.01 * np.random.normal(size=n_samples)

# Split data in train set and test set
n_samples = X.shape[0]
X_train, y_train = X[:n_samples // 2], y[:n_samples // 2]
X_test, y_test = X[n_samples // 2:], y[n_samples // 2:]

#####
# Lasso
from sklearn.linear_model import Lasso

alpha = 0.1
lasso = Lasso(alpha=alpha)

y_pred_lasso = lasso.fit(X_train, y_train).predict(X_test)
r2_score_lasso = r2_score(y_test, y_pred_lasso)
print(lasso)
print("r^2 on test data : %f" % r2_score_lasso)

#####
# ElasticNet
from sklearn.linear_model import ElasticNet

enet = ElasticNet(alpha=alpha, l1_ratio=0.7)

y_pred_enet = enet.fit(X_train, y_train).predict(X_test)
r2_score_enet = r2_score(y_test, y_pred_enet)
print(enet)
print("r^2 on test data : %f" % r2_score_enet)

m, s, _ = plt.stem(np.where(enet.coef_) [0], enet.coef_[enet.coef_ != 0],
                   markerfmt='x', label='Elastic net coefficients')
plt.setp([m, s], color="#2ca02c")
m, s, _ = plt.stem(np.where(lasso.coef_) [0], lasso.coef_[lasso.coef_ != 0],
                   markerfmt='x', label='Lasso coefficients')
plt.setp([m, s], color='#ff7f0e')
plt.stem(np.where(coef)[0], coef[coef != 0], label='true coefficients',
        markerfmt='bx')

plt.legend(loc='best')
plt.title("Lasso $R^2$: %.3f, Elastic Net $R^2$: %.3f"
          % (r2_score_lasso, r2_score_enet))
plt.show()

```

Total running time of the script: (0 minutes 0.071 seconds)

Note: Click [here](#) to download the full example code

5.18.24 Theil-Sen Regression

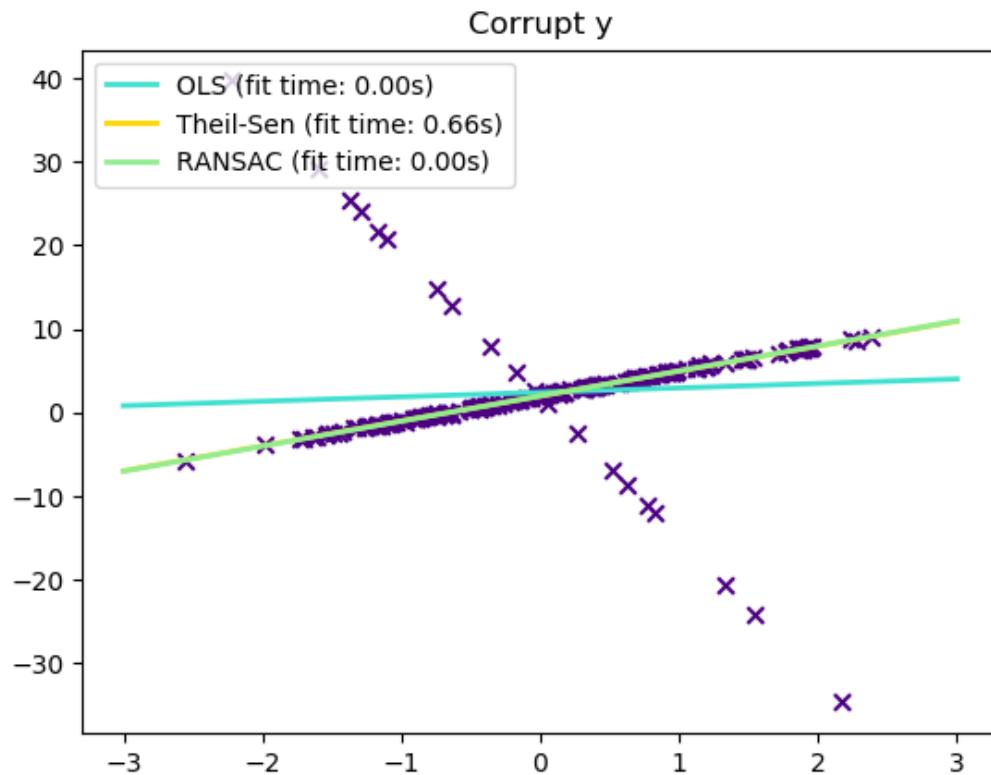
Computes a Theil-Sen Regression on a synthetic dataset.

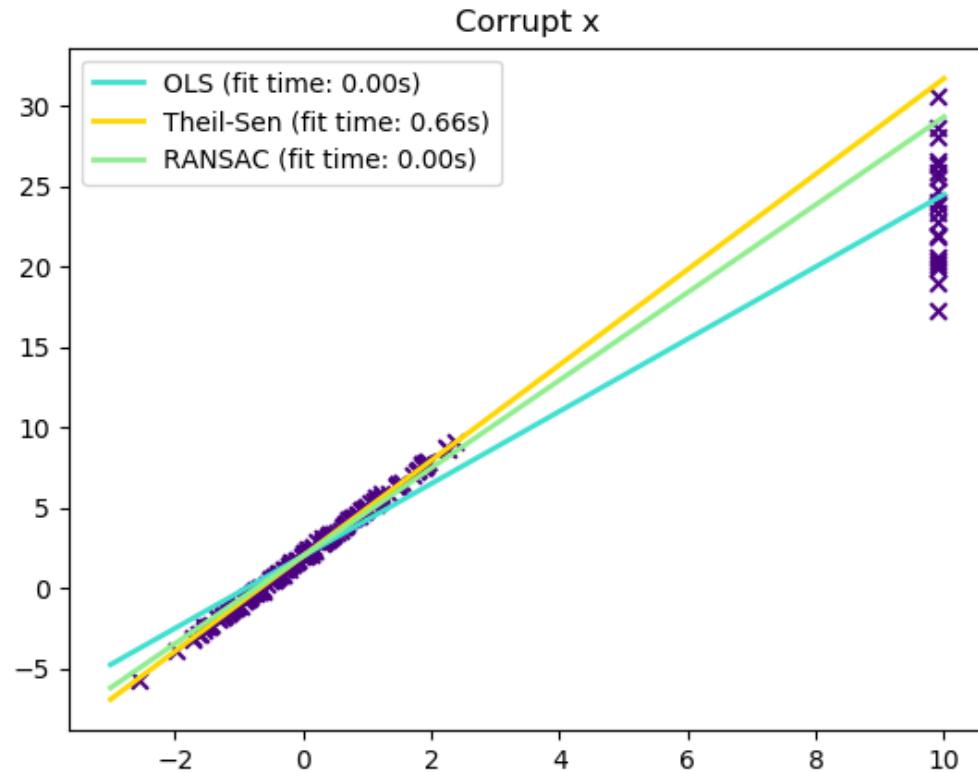
See [Theil-Sen estimator: generalized-median-based estimator](#) for more information on the regressor.

Compared to the OLS (ordinary least squares) estimator, the Theil-Sen estimator is robust against outliers. It has a breakdown point of about 29.3% in case of a simple linear regression which means that it can tolerate arbitrary corrupted data (outliers) of up to 29.3% in the two-dimensional case.

The estimation of the model is done by calculating the slopes and intercepts of a subpopulation of all possible combinations of p subsample points. If an intercept is fitted, p must be greater than or equal to $n_features + 1$. The final slope and intercept is then defined as the spatial median of these slopes and intercepts.

In certain cases Theil-Sen performs better than [RANSAC](#) which is also a robust method. This is illustrated in the second example below where outliers with respect to the x-axis perturb RANSAC. Tuning the `residual_threshold` parameter of RANSAC remedies this but in general a priori knowledge about the data and the nature of the outliers is needed. Due to the computational complexity of Theil-Sen it is recommended to use it only for small problems in terms of number of samples and features. For larger problems the `max_subpopulation` parameter restricts the magnitude of all possible combinations of p subsample points to a randomly chosen subset and therefore also limits the runtime. Therefore, Theil-Sen is applicable to larger problems with the drawback of losing some of its mathematical properties since it then works on a random subset.





```
# Author: Florian Wilhelm -- <florian.wilhelm@gmail.com>
# License: BSD 3 clause

import time
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, TheilSenRegressor
from sklearn.linear_model import RANSACRegressor

print(__doc__)

estimators = [('OLS', LinearRegression()),
              ('Theil-Sen', TheilSenRegressor(random_state=42)),
              ('RANSAC', RANSACRegressor(random_state=42))]

colors = {'OLS': 'turquoise', 'Theil-Sen': 'gold', 'RANSAC': 'lightgreen'}
lw = 2

#####
# Outliers only in the y direction

np.random.seed(0)
n_samples = 200
# Linear model y = 3*x + N(2, 0.1**2)
x = np.random.randn(n_samples)
w = 3.
c = 2.
noise = 0.1 * np.random.randn(n_samples)
```

```

y = w * x + c + noise
# 10% outliers
y[-20:] += -20 * x[-20:]
X = x[:, np.newaxis]

plt.scatter(x, y, color='indigo', marker='x', s=40)
line_x = np.array([-3, 3])
for name, estimator in estimators:
    t0 = time.time()
    estimator.fit(X, y)
    elapsed_time = time.time() - t0
    y_pred = estimator.predict(line_x.reshape(2, 1))
    plt.plot(line_x, y_pred, color=colors[name], linewidth=lw,
              label='%s (fit time: %.2fs)' % (name, elapsed_time))

plt.axis('tight')
plt.legend(loc='upper left')
plt.title("Corrupt y")

#####
# Outliers in the X direction

np.random.seed(0)
# Linear model y = 3*x + N(2, 0.1**2)
x = np.random.randn(n_samples)
noise = 0.1 * np.random.randn(n_samples)
y = 3 * x + 2 + noise
# 10% outliers
x[-20:] = 9.9
y[-20:] += 22
X = x[:, np.newaxis]

plt.figure()
plt.scatter(x, y, color='indigo', marker='x', s=40)

line_x = np.array([-3, 10])
for name, estimator in estimators:
    t0 = time.time()
    estimator.fit(X, y)
    elapsed_time = time.time() - t0
    y_pred = estimator.predict(line_x.reshape(2, 1))
    plt.plot(line_x, y_pred, color=colors[name], linewidth=lw,
              label='%s (fit time: %.2fs)' % (name, elapsed_time))

plt.axis('tight')
plt.legend(loc='upper left')
plt.title("Corrupt x")
plt.show()

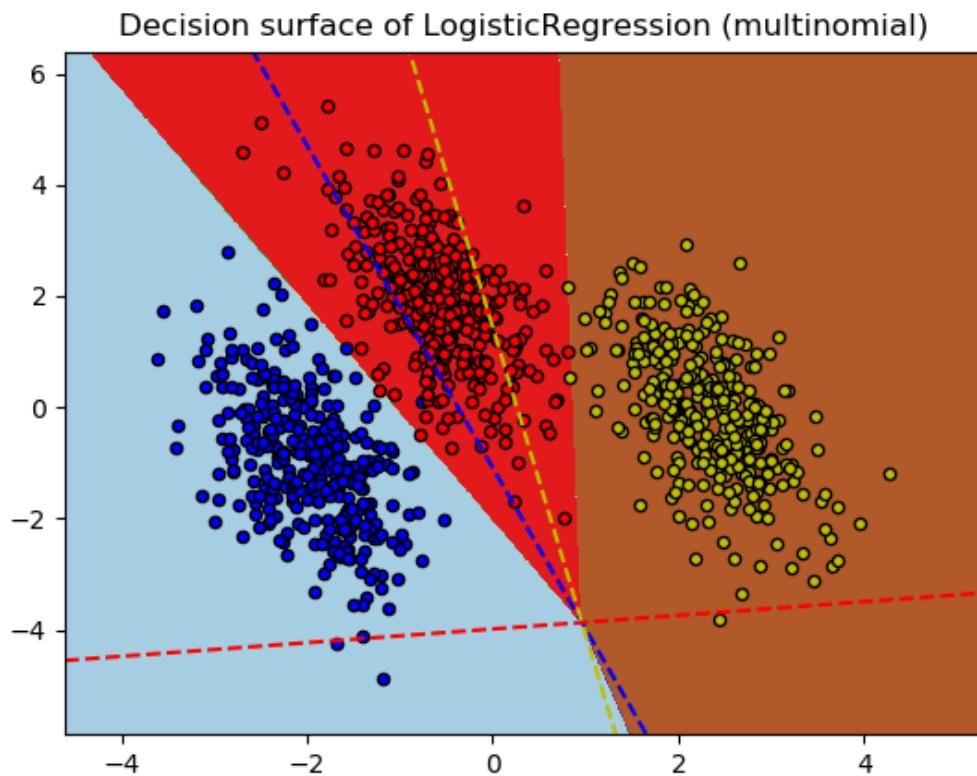
```

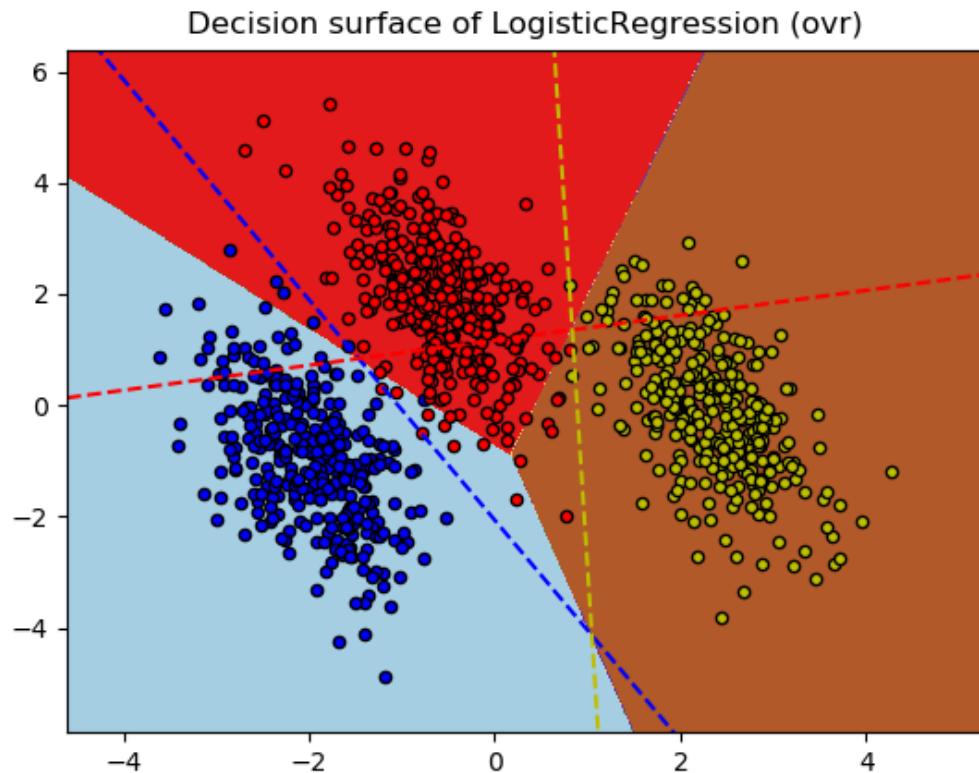
Total running time of the script: (0 minutes 1.359 seconds)

Note: Click [here](#) to download the full example code

5.18.25 Plot multinomial and One-vs-Rest Logistic Regression

Plot decision surface of multinomial and One-vs-Rest Logistic Regression. The hyperplanes corresponding to the three One-vs-Rest (OVR) classifiers are represented by the dashed lines.





Out:

```
training score : 0.995 (multinomial)
training score : 0.976 (ovr)
```

```
print(__doc__)
# Authors: Tom Dupre la Tour <tom.dupre-la-tour@m4x.org>
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.linear_model import LogisticRegression

# make 3-class dataset for classification
centers = [[-5, 0], [0, 1.5], [5, -1]]
X, y = make_blobs(n_samples=1000, centers=centers, random_state=40)
transformation = [[0.4, 0.2], [-0.4, 1.2]]
X = np.dot(X, transformation)

for multi_class in ('multinomial', 'ovr'):
    clf = LogisticRegression(solver='sag', max_iter=100, random_state=42,
```

```
multi_class=multi_class).fit(X, y)

# print the training scores
print("training score : %.3f (%s)" % (clf.score(X, y), multi_class))

# create a mesh to plot in
h = .02 # step size in the mesh
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                      np.arange(y_min, y_max, h))

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure()
plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
plt.title("Decision surface of LogisticRegression (%s)" % multi_class)
plt.axis('tight')

# Plot also the training points
colors = "bry"
for i, color in zip(clf.classes_, colors):
    idx = np.where(y == i)
    plt.scatter(X[idx, 0], X[idx, 1], c=color, cmap=plt.cm.Paired,
                edgecolor='black', s=20)

# Plot the three one-against-all classifiers
xmin, xmax = plt.xlim()
ymin, ymax = plt.ylim()
coef = clf.coef_
intercept = clf.intercept_

def plot_hyperplane(c, color):
    def line(x0):
        return -(x0 * coef[c, 0]) - intercept[c] / coef[c, 1]
    plt.plot([xmin, xmax], [line(xmin), line(xmax)],
             ls="--", color=color)

    for i, color in zip(clf.classes_, colors):
        plot_hyperplane(i, color)

plt.show()
```

Total running time of the script: (0 minutes 0.262 seconds)

Note: Click [here](#) to download the full example code

5.18.26 Robust linear estimator fitting

Here a sine function is fit with a polynomial of order 3, for values close to zero.

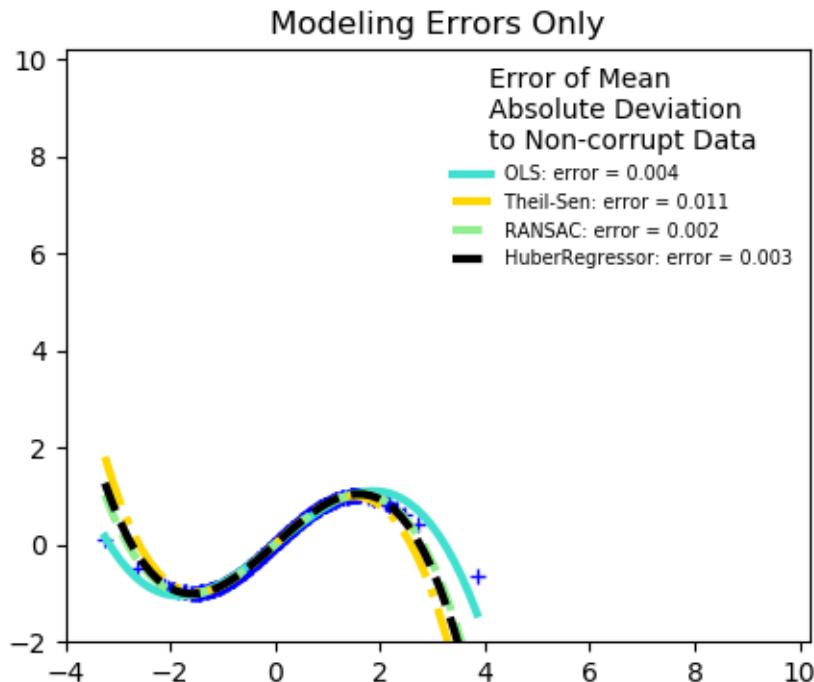
Robust fitting is demoed in different situations:

- No measurement errors, only modelling errors (fitting a sine with a polynomial)
- Measurement errors in X
- Measurement errors in y

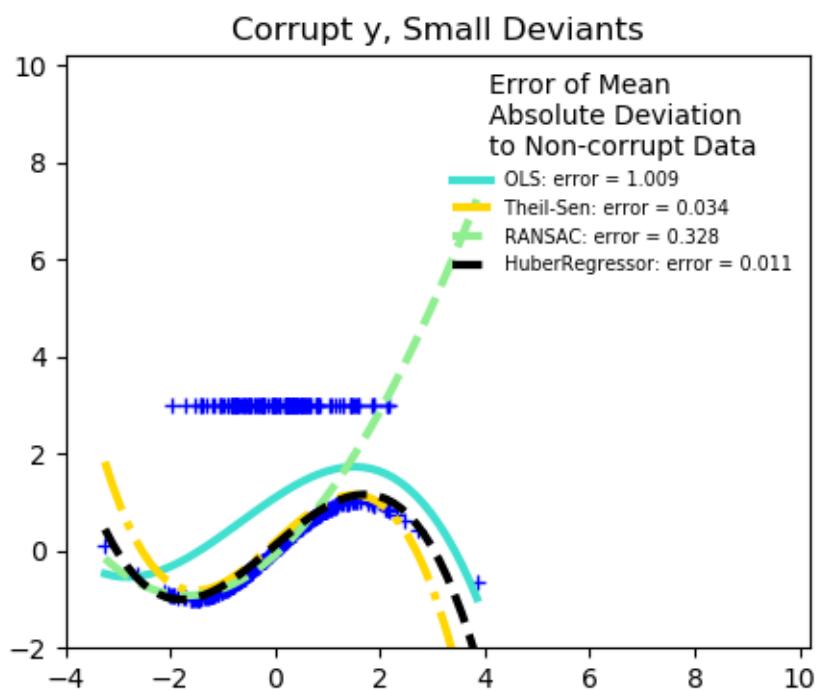
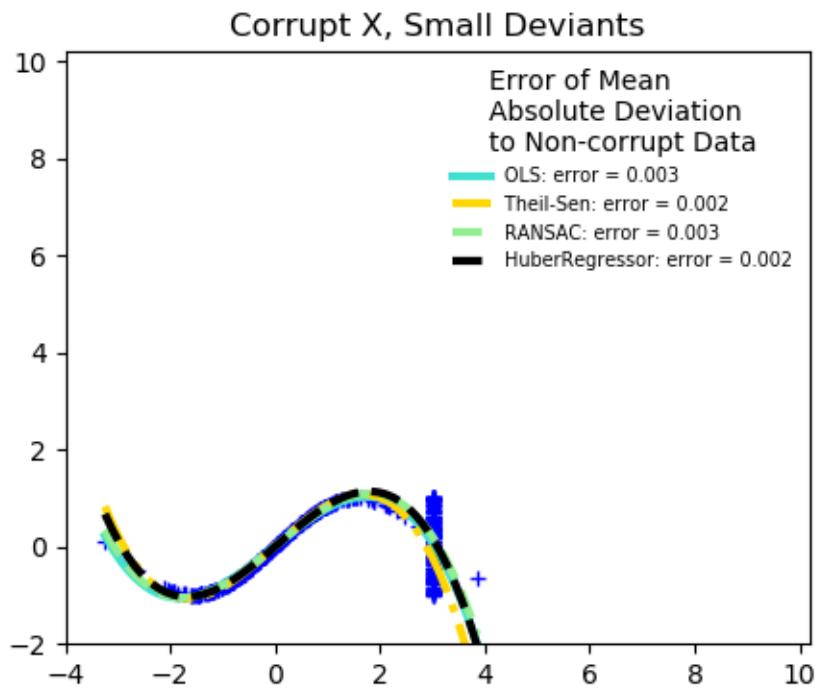
The median absolute deviation to non corrupt new data is used to judge the quality of the prediction.

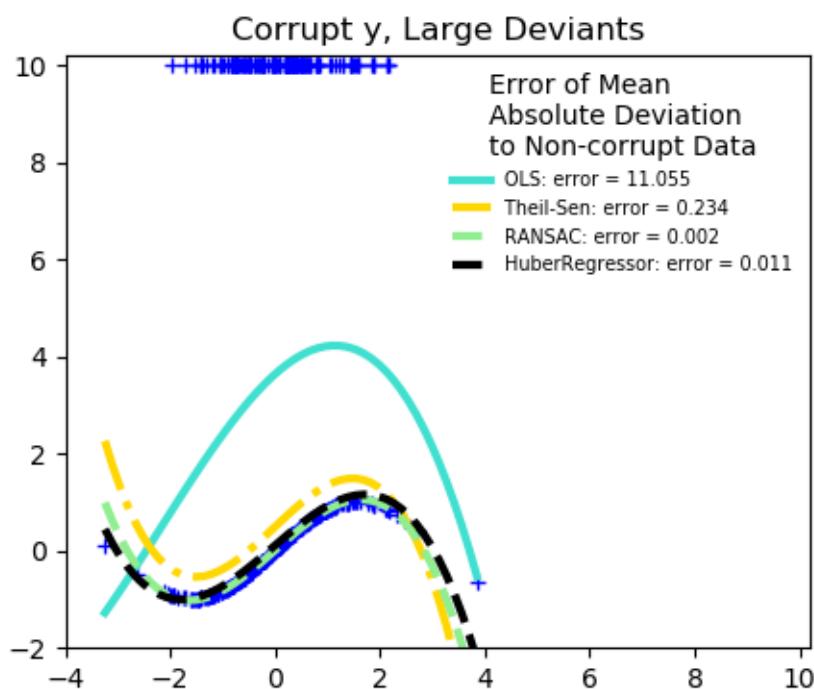
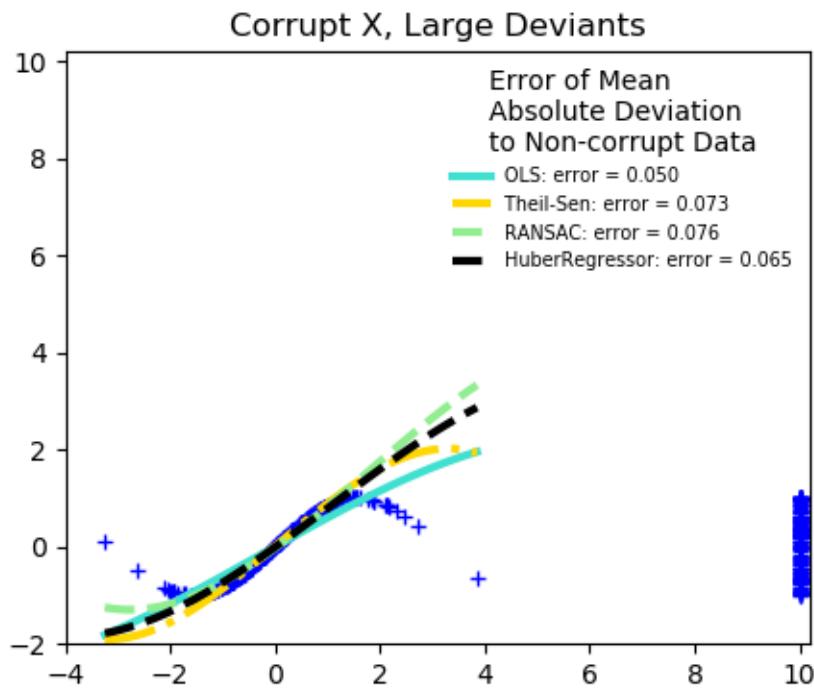
What we can see that:

- RANSAC is good for strong outliers in the y direction
- TheilSen is good for small outliers, both in direction X and y, but has a break point above which it performs worse than OLS.
- The scores of HuberRegressor may not be compared directly to both TheilSen and RANSAC because it does not attempt to completely filter the outliers but lessen their effect.



.





```
from matplotlib import pyplot as plt
```

```
import numpy as np
```

```
from sklearn.linear_model import (
```

```

    LinearRegression, TheilSenRegressor, RANSACRegressor, HuberRegressor)
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

np.random.seed(42)

X = np.random.normal(size=400)
y = np.sin(X)
# Make sure that it X is 2D
X = X[:, np.newaxis]

X_test = np.random.normal(size=200)
y_test = np.sin(X_test)
X_test = X_test[:, np.newaxis]

y_errors = y.copy()
y_errors[::3] = 3

X_errors = X.copy()
X_errors[::3] = 3

y_errors_large = y.copy()
y_errors_large[::3] = 10

X_errors_large = X.copy()
X_errors_large[::3] = 10

estimators = [('OLS', LinearRegression()),
               ('Theil-Sen', TheilSenRegressor(random_state=42)),
               ('RANSAC', RANSACRegressor(random_state=42)),
               ('HuberRegressor', HuberRegressor())]
colors = {'OLS': 'turquoise', 'Theil-Sen': 'gold', 'RANSAC': 'lightgreen',
          'HuberRegressor': 'black'}
linestyle = {'OLS': '-', 'Theil-Sen': '-.', 'RANSAC': '--', 'HuberRegressor': '--'}
lw = 3

x_plot = np.linspace(X.min(), X.max())
for title, this_X, this_y in [
    ('Modeling Errors Only', X, y),
    ('Corrupt X, Small Deviants', X_errors, y),
    ('Corrupt y, Small Deviants', X, y_errors),
    ('Corrupt X, Large Deviants', X_errors_large, y),
    ('Corrupt y, Large Deviants', X, y_errors_large)]:
    plt.figure(figsize=(5, 4))
    plt.plot(this_X[:, 0], this_y, 'b+')

    for name, estimator in estimators:
        model = make_pipeline(PolynomialFeatures(3), estimator)
        model.fit(this_X, this_y)
        mse = mean_squared_error(model.predict(X_test), y_test)
        y_plot = model.predict(x_plot[:, np.newaxis])
        plt.plot(x_plot, y_plot, color=colors[name], linestyle=linestyle[name],
                  linewidth=lw, label='{}: error = {:.3f}'.format(name, mse))

legend_title = 'Error of Mean\nAbsolute Deviation\ninto Non-corrupt Data'
legend = plt.legend(loc='upper right', frameon=False, title=legend_title,
                    prop=dict(size='x-small'))

```

```

plt.xlim(-4, 10.2)
plt.ylim(-2, 10.2)
plt.title(title)
plt.show()

```

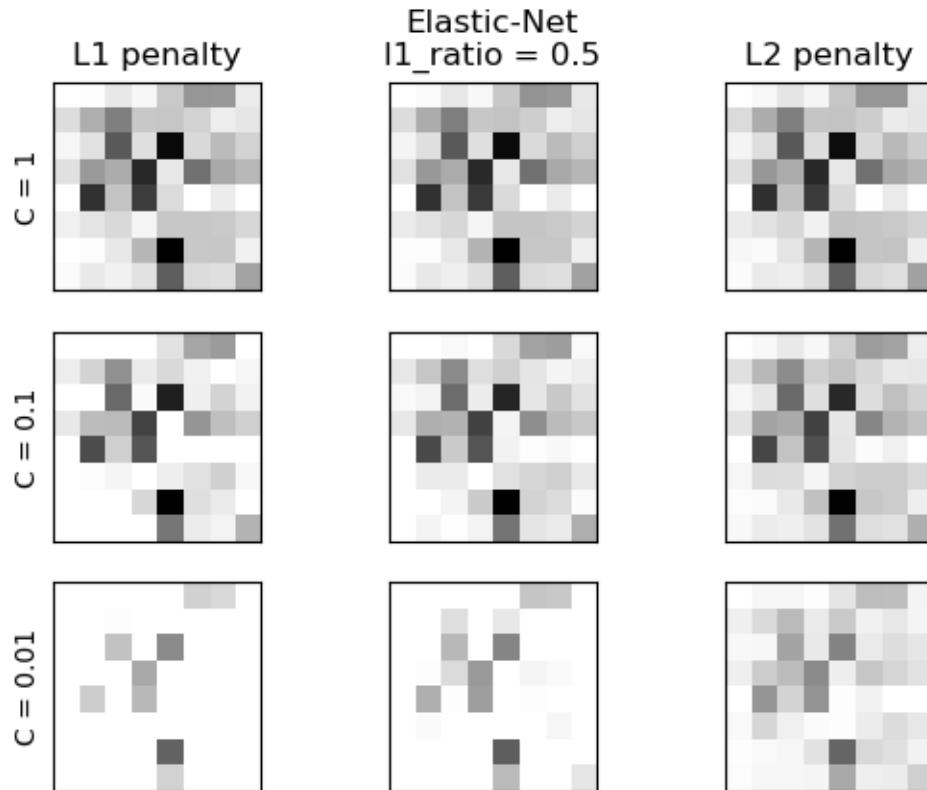
Total running time of the script: (0 minutes 3.950 seconds)

Note: Click [here](#) to download the full example code

5.18.27 L1 Penalty and Sparsity in Logistic Regression

Comparison of the sparsity (percentage of zero coefficients) of solutions when L1, L2 and Elastic-Net penalty are used for different values of C. We can see that large values of C give more freedom to the model. Conversely, smaller values of C constrain the model more. In the L1 penalty case, this leads to sparser solutions. As expected, the Elastic-Net sparsity is between that of L1 and L2.

We classify 8x8 images of digits into two classes: 0-4 against 5-9. The visualization shows coefficients of the models for varying C.



Out:

C=1.00	
Sparsity with L1 penalty:	6.25%

Sparsity with Elastic-Net penalty:	4.69%
Sparsity with L2 penalty:	4.69%
Score with L1 penalty:	0.90
Score with Elastic-Net penalty:	0.90
Score with L2 penalty:	0.90
C=0.10	
Sparsity with L1 penalty:	29.69%
Sparsity with Elastic-Net penalty:	12.50%
Sparsity with L2 penalty:	4.69%
Score with L1 penalty:	0.90
Score with Elastic-Net penalty:	0.90
Score with L2 penalty:	0.90
C=0.01	
Sparsity with L1 penalty:	84.38%
Sparsity with Elastic-Net penalty:	68.75%
Sparsity with L2 penalty:	4.69%
Score with L1 penalty:	0.86
Score with Elastic-Net penalty:	0.88
Score with L2 penalty:	0.89

```

print(__doc__)

# Authors: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#          Mathieu Blondel <mathieu@mblondel.org>
#          Andreas Mueller <amueller@ais.uni-bonn.de>
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

digits = datasets.load_digits()

X, y = digits.data, digits.target
X = StandardScaler().fit_transform(X)

# classify small against large digits
y = (y > 4).astype(np.int)

l1_ratio = 0.5 # L1 weight in the Elastic-Net regularization

fig, axes = plt.subplots(3, 3)

# Set regularization parameter
for i, (C, axes_row) in enumerate(zip((1, 0.1, 0.01), axes)):
    # turn down tolerance for short training time
    clf_l1_LR = LogisticRegression(C=C, penalty='l1', tol=0.01, solver='saga')
    clf_l2_LR = LogisticRegression(C=C, penalty='l2', tol=0.01, solver='saga')
    clf_en_LR = LogisticRegression(C=C, penalty='elasticnet', solver='saga',
                                   l1_ratio=l1_ratio, tol=0.01)

```

```

clf_l1_LR.fit(X, y)
clf_l2_LR.fit(X, y)
clf_en_LR.fit(X, y)

coef_l1_LR = clf_l1_LR.coef_.ravel()
coef_l2_LR = clf_l2_LR.coef_.ravel()
coef_en_LR = clf_en_LR.coef_.ravel()

# coef_l1_LR contains zeros due to the
# L1 sparsity inducing norm

sparsity_l1_LR = np.mean(coef_l1_LR == 0) * 100
sparsity_l2_LR = np.mean(coef_l2_LR == 0) * 100
sparsity_en_LR = np.mean(coef_en_LR == 0) * 100

print("C=% .2f" % C)
print("{:<40} {:.2f}%".format("Sparsity with L1 penalty:", sparsity_l1_LR))
print("{:<40} {:.2f}%".format("Sparsity with Elastic-Net penalty:",
                             sparsity_en_LR))
print("{:<40} {:.2f}%".format("Sparsity with L2 penalty:", sparsity_l2_LR))
print("{:<40} {:.2f}%".format("Score with L1 penalty:",
                             clf_l1_LR.score(X, y)))
print("{:<40} {:.2f}%".format("Score with Elastic-Net penalty:",
                             clf_en_LR.score(X, y)))
print("{:<40} {:.2f}%".format("Score with L2 penalty:",
                             clf_l2_LR.score(X, y)))

if i == 0:
    axes_row[0].set_title("L1 penalty")
    axes_row[1].set_title("Elastic-Net\nn1_ratio = %s" % l1_ratio)
    axes_row[2].set_title("L2 penalty")

for ax, coefs in zip(axes_row, [coef_l1_LR, coef_en_LR, coef_l2_LR]):
    ax.imshow(np.abs(coefs.reshape(8, 8)), interpolation='nearest',
              cmap='binary', vmax=1, vmin=0)
    ax.set_xticks(())
    ax.set_yticks(())

axes_row[0].set_ylabel('C = %s' % C)

plt.show()

```

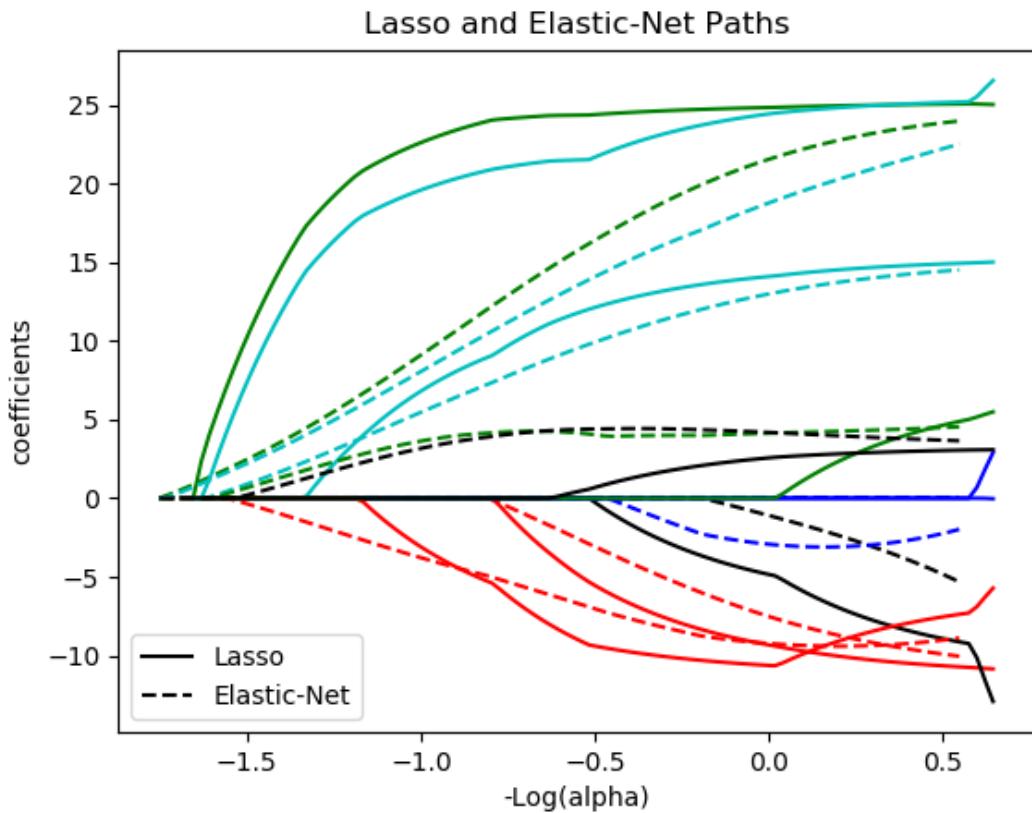
Total running time of the script: (0 minutes 0.659 seconds)

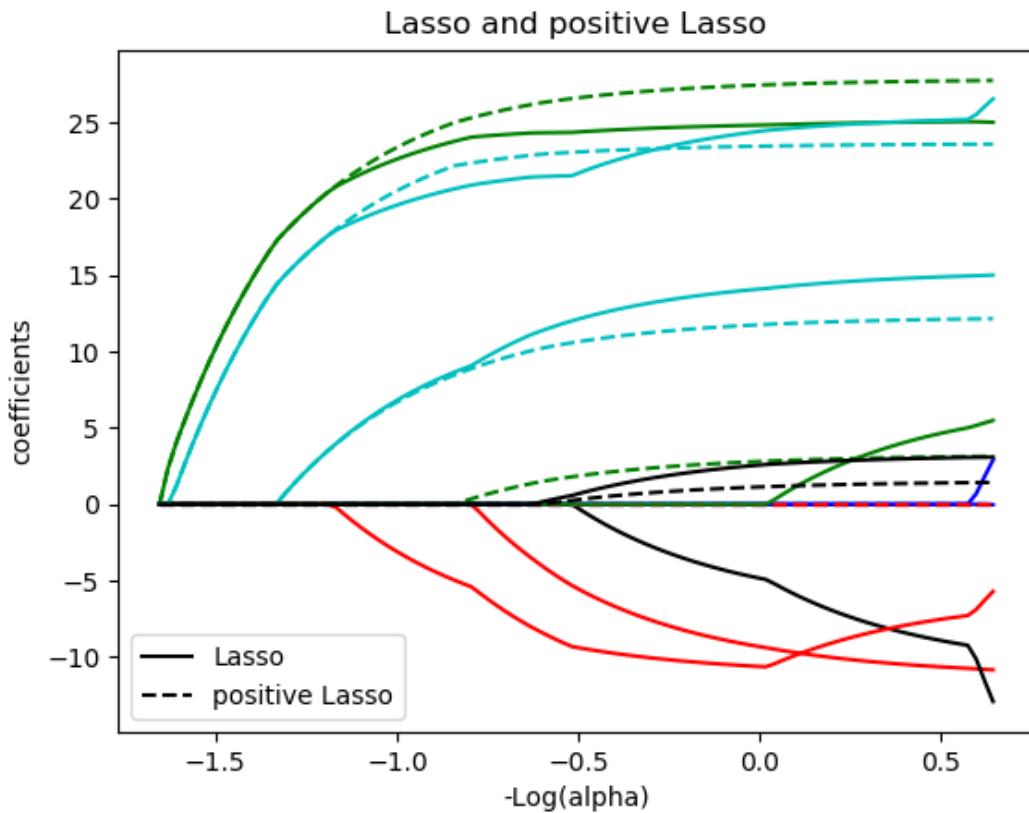
Note: Click [here](#) to download the full example code

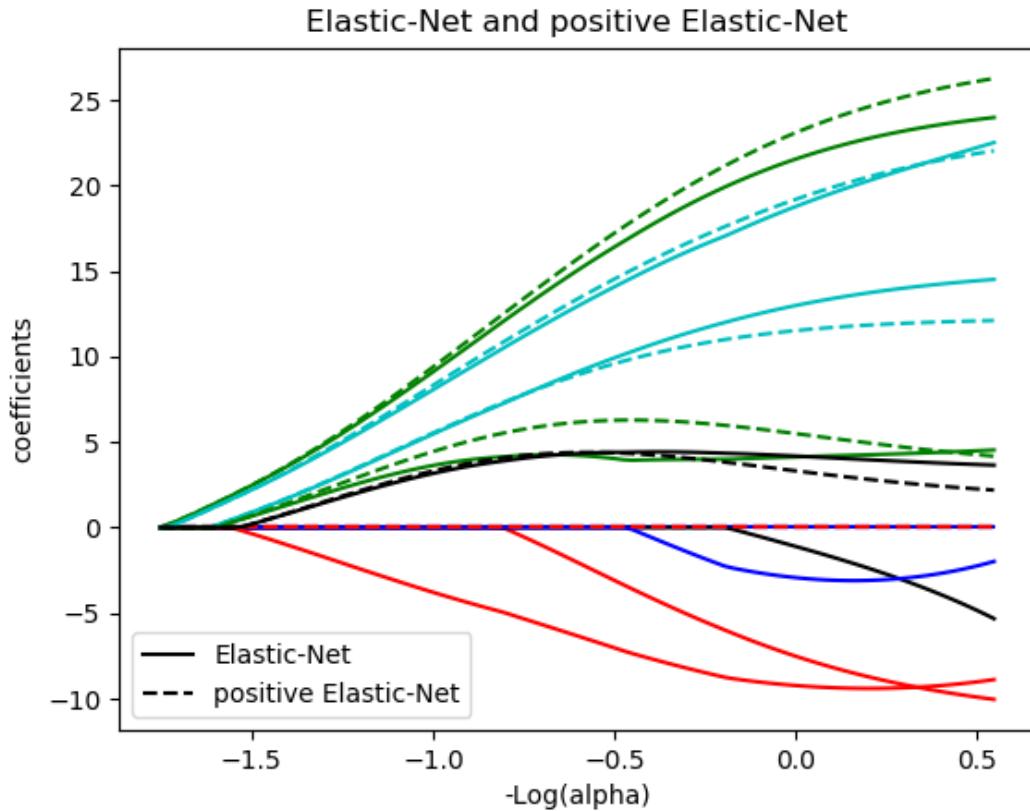
5.18.28 Lasso and Elastic Net

Lasso and elastic net (L1 and L2 penalisation) implemented using a coordinate descent.

The coefficients can be forced to be positive.







Out:

```
Computing regularization path using the lasso...
Computing regularization path using the positive lasso...
Computing regularization path using the elastic net...
Computing regularization path using the positive elastic net...
```

```
print(__doc__)

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause

from itertools import cycle
import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import lasso_path, enet_path
from sklearn import datasets

diabetes = datasets.load_diabetes()
X = diabetes.data
y = diabetes.target
```

```

X /= X.std(axis=0) # Standardize data (easier to set the l1_ratio parameter)

# Compute paths

eps = 5e-3 # the smaller it is the longer is the path

print("Computing regularization path using the lasso...")
alphas_lasso, coefs_lasso, _ = lasso_path(X, y, eps, fit_intercept=False)

print("Computing regularization path using the positive lasso...")
alphas_positive_lasso, coefs_positive_lasso, _ = lasso_path(
    X, y, eps, positive=True, fit_intercept=False)
print("Computing regularization path using the elastic net...")
alphas_enet, coefs_enet, _ = enet_path(
    X, y, eps=eps, l1_ratio=0.8, fit_intercept=False)

print("Computing regularization path using the positive elastic net...")
alphas_positive_enet, coefs_positive_enet, _ = enet_path(
    X, y, eps=eps, l1_ratio=0.8, positive=True, fit_intercept=False)

# Display results

plt.figure(1)
colors = cycle(['b', 'r', 'g', 'c', 'k'])
neg_log_alphas_lasso = -np.log10(alphas_lasso)
neg_log_alphas_enet = -np.log10(alphas_enet)
for coef_l, coef_e, c in zip(coefs_lasso, coefs_enet, colors):
    l1 = plt.plot(neg_log_alphas_lasso, coef_l, c=c)
    l2 = plt.plot(neg_log_alphas_enet, coef_e, linestyle='--', c=c)

plt.xlabel('-Log(alpha)')
plt.ylabel('coefficients')
plt.title('Lasso and Elastic-Net Paths')
plt.legend((l1[-1], l2[-1]), ('Lasso', 'Elastic-Net'), loc='lower left')
plt.axis('tight')


plt.figure(2)
neg_log_alphas_positive_lasso = -np.log10(alphas_positive_lasso)
for coef_l, coef_pl, c in zip(coefs_lasso, coefs_positive_lasso, colors):
    l1 = plt.plot(neg_log_alphas_lasso, coef_l, c=c)
    l2 = plt.plot(neg_log_alphas_positive_lasso, coef_pl, linestyle='--', c=c)

plt.xlabel('-Log(alpha)')
plt.ylabel('coefficients')
plt.title('Lasso and positive Lasso')
plt.legend((l1[-1], l2[-1]), ('Lasso', 'positive Lasso'), loc='lower left')
plt.axis('tight')


plt.figure(3)
neg_log_alphas_positive_enet = -np.log10(alphas_positive_enet)
for (coef_e, coef_pe, c) in zip(coefs_enet, coefs_positive_enet, colors):
    l1 = plt.plot(neg_log_alphas_enet, coef_e, c=c)
    l2 = plt.plot(neg_log_alphas_positive_enet, coef_pe, linestyle='--', c=c)

plt.xlabel('-Log(alpha)')

```

```
plt.ylabel('coefficients')
plt.title('Elastic-Net and positive Elastic-Net')
plt.legend((l1[-1], l2[-1]), ('Elastic-Net', 'positive Elastic-Net'),
           loc='lower left')
plt.axis('tight')
plt.show()
```

Total running time of the script: (0 minutes 0.232 seconds)

Note: Click [here](#) to download the full example code

5.18.29 Automatic Relevance Determination Regression (ARD)

Fit regression model with Bayesian Ridge Regression.

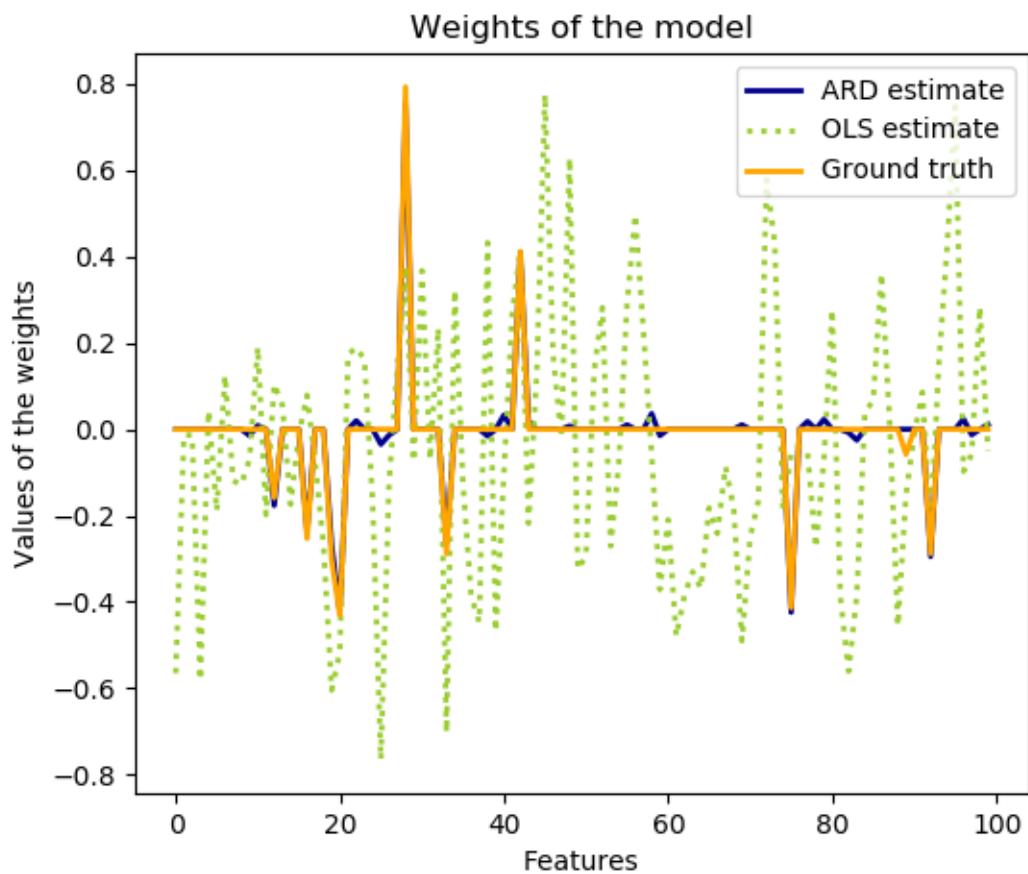
See [Bayesian Ridge Regression](#) for more information on the regressor.

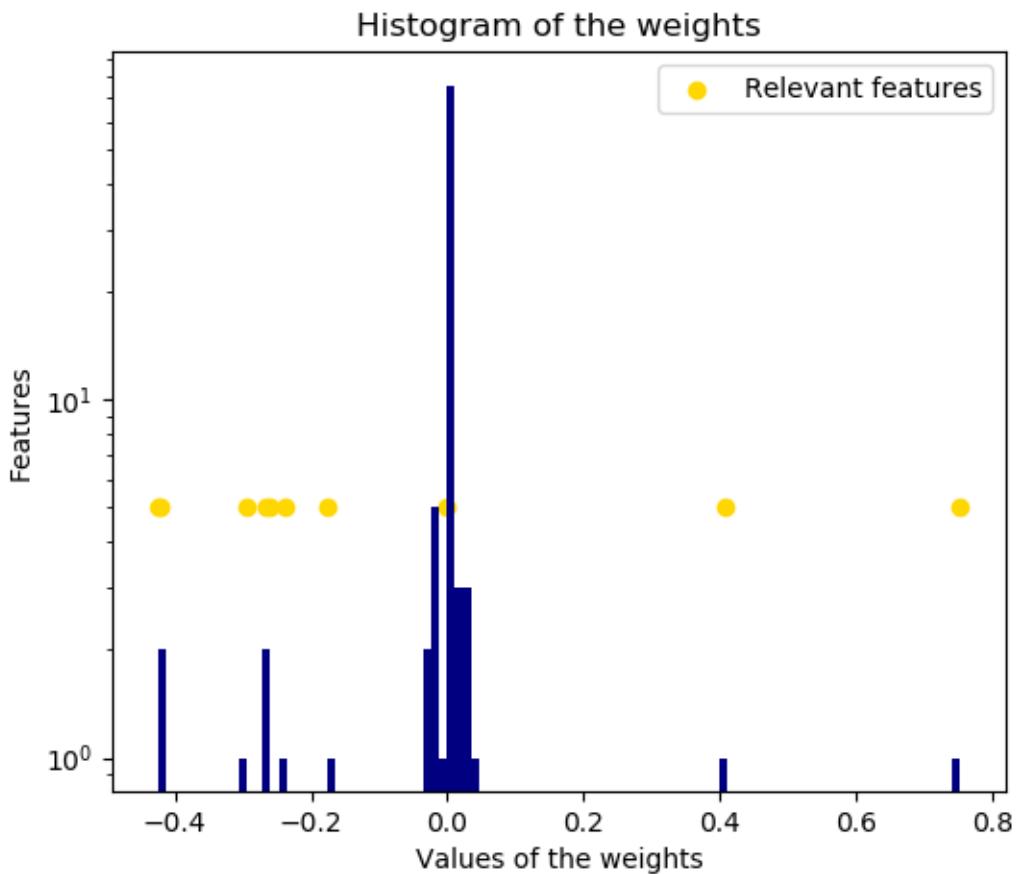
Compared to the OLS (ordinary least squares) estimator, the coefficient weights are slightly shifted toward zeros, which stabilises them.

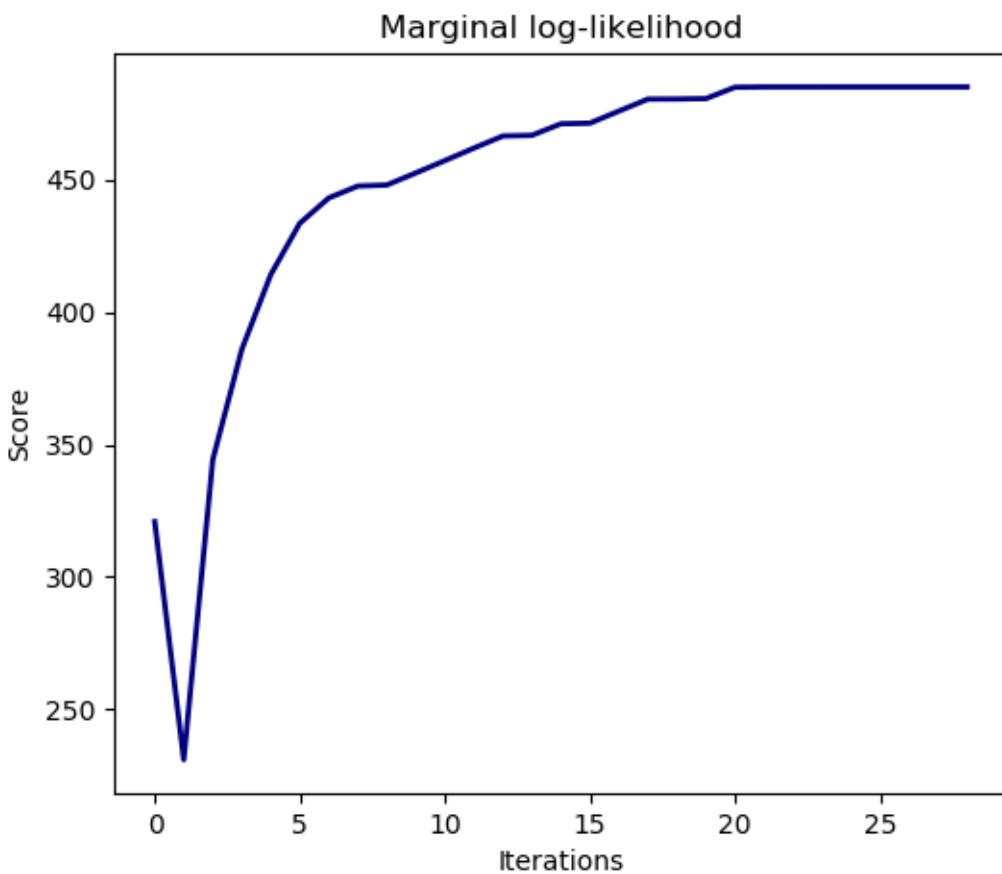
The histogram of the estimated weights is very peaked, as a sparsity-inducing prior is implied on the weights.

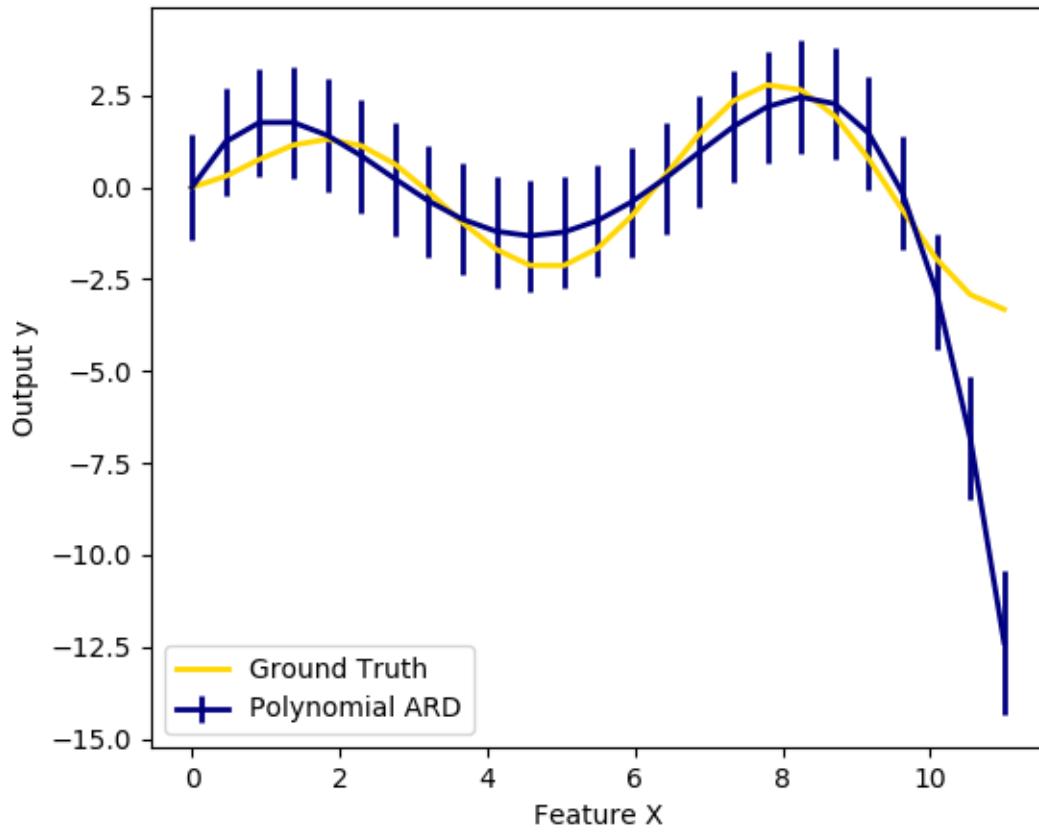
The estimation of the model is done by iteratively maximizing the marginal log-likelihood of the observations.

We also plot predictions and uncertainties for ARD for one dimensional regression using polynomial feature expansion. Note the uncertainty starts going up on the right side of the plot. This is because these test samples are outside of the range of the training samples.









```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

from sklearn.linear_model import ARDRegression, LinearRegression

# ##### Generating simulated data with Gaussian weights

# Parameters of the example
np.random.seed(0)
n_samples, n_features = 100, 100
# Create Gaussian data
X = np.random.randn(n_samples, n_features)
# Create weights with a precision lambda_ of 4.
lambda_ = 4.
w = np.zeros(n_features)
# Only keep 10 weights of interest
relevant_features = np.random.randint(0, n_features, 10)
for i in relevant_features:
    w[i] = stats.norm.rvs(loc=0, scale=1. / np.sqrt(lambda_))
# Create noise with a precision alpha of 50.
alpha_ = 50.

```

```

noise = stats.norm.rvs(loc=0, scale=1. / np.sqrt(alpha_), size=n_samples)
# Create the target
y = np.dot(X, w) + noise

# ##### Fit the ARD Regression #####
# Fit the ARD Regression
clf = ARDRegression(compute_score=True)
clf.fit(X, y)

ols = LinearRegression()
ols.fit(X, y)

# Plot the true weights, the estimated weights, the histogram of the
# weights, and predictions with standard deviations
plt.figure(figsize=(6, 5))
plt.title("Weights of the model")
plt.plot(clf.coef_, color='darkblue', linestyle='-', linewidth=2,
         label="ARD estimate")
plt.plot(ols.coef_, color='yellowgreen', linestyle=':', linewidth=2,
         label="OLS estimate")
plt.plot(w, color='orange', linestyle='-', linewidth=2, label="Ground truth")
plt.xlabel("Features")
plt.ylabel("Values of the weights")
plt.legend(loc=1)

plt.figure(figsize=(6, 5))
plt.title("Histogram of the weights")
plt.hist(clf.coef_, bins=n_features, color='navy', log=True)
plt.scatter(clf.coef_[relevant_features], np.full(len(relevant_features), 5.),
            color='gold', marker='o', label="Relevant features")
plt.xlabel("Features")
plt.ylabel("Values of the weights")
plt.legend(loc=1)

plt.figure(figsize=(6, 5))
plt.title("Marginal log-likelihood")
plt.plot(clf.scores_, color='navy', linewidth=2)
plt.ylabel("Score")
plt.xlabel("Iterations")

# Plotting some predictions for polynomial regression
def f(x, noise_amount):
    y = np.sqrt(x) * np.sin(x)
    noise = np.random.normal(0, 1, len(x))
    return y + noise_amount * noise

degree = 10
X = np.linspace(0, 10, 100)
y = f(X, noise_amount=1)
clf_poly = ARDRegression(threshold_lambda=1e5)
clf_poly.fit(np.vander(X, degree), y)

X_plot = np.linspace(0, 11, 25)
y_plot = f(X_plot, noise_amount=0)
y_mean, y_std = clf_poly.predict(np.vander(X_plot, degree), return_std=True)

```

```
plt.figure(figsize=(6, 5))
plt.errorbar(X_plot, y_mean, y_std, color='navy',
             label="Polynomial ARD", linewidth=2)
plt.plot(X_plot, y_plot, color='gold', linewidth=2,
         label="Ground Truth")
plt.ylabel("Output y")
plt.xlabel("Feature X")
plt.legend(loc="lower left")
plt.show()
```

Total running time of the script: (0 minutes 0.293 seconds)

Note: Click [here](#) to download the full example code

5.18.30 Bayesian Ridge Regression

Computes a Bayesian Ridge Regression on a synthetic dataset.

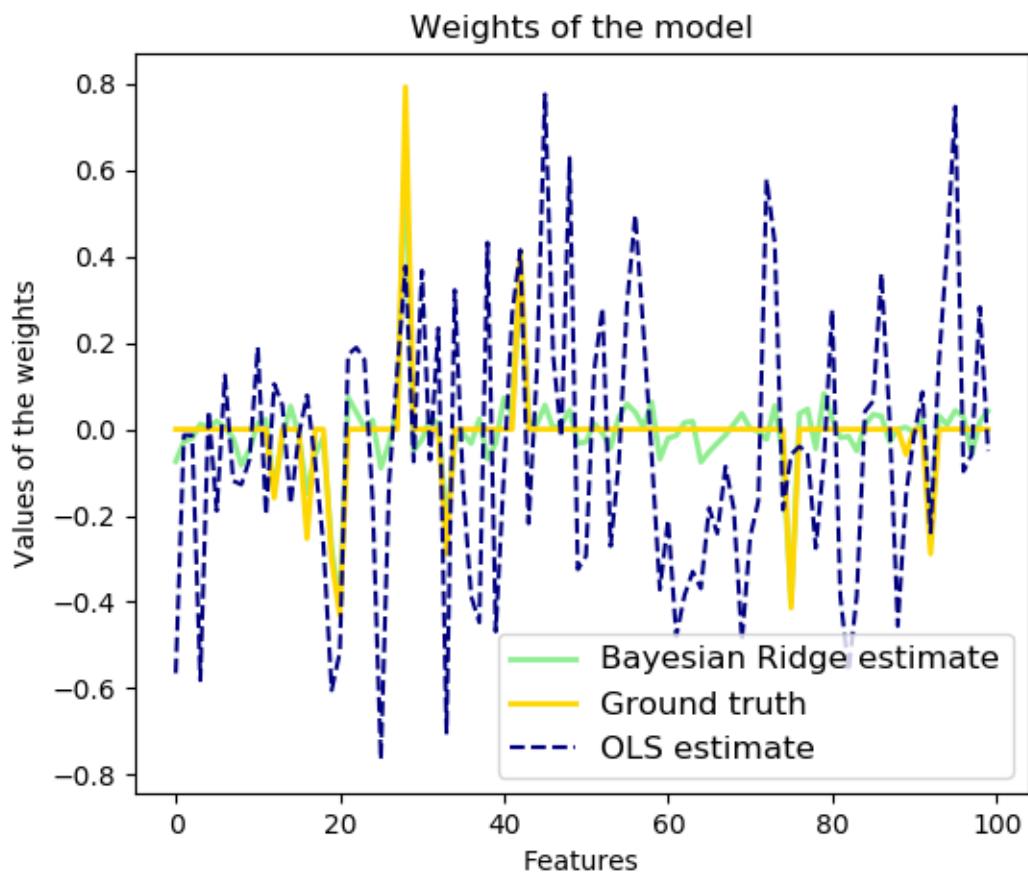
See [Bayesian Ridge Regression](#) for more information on the regressor.

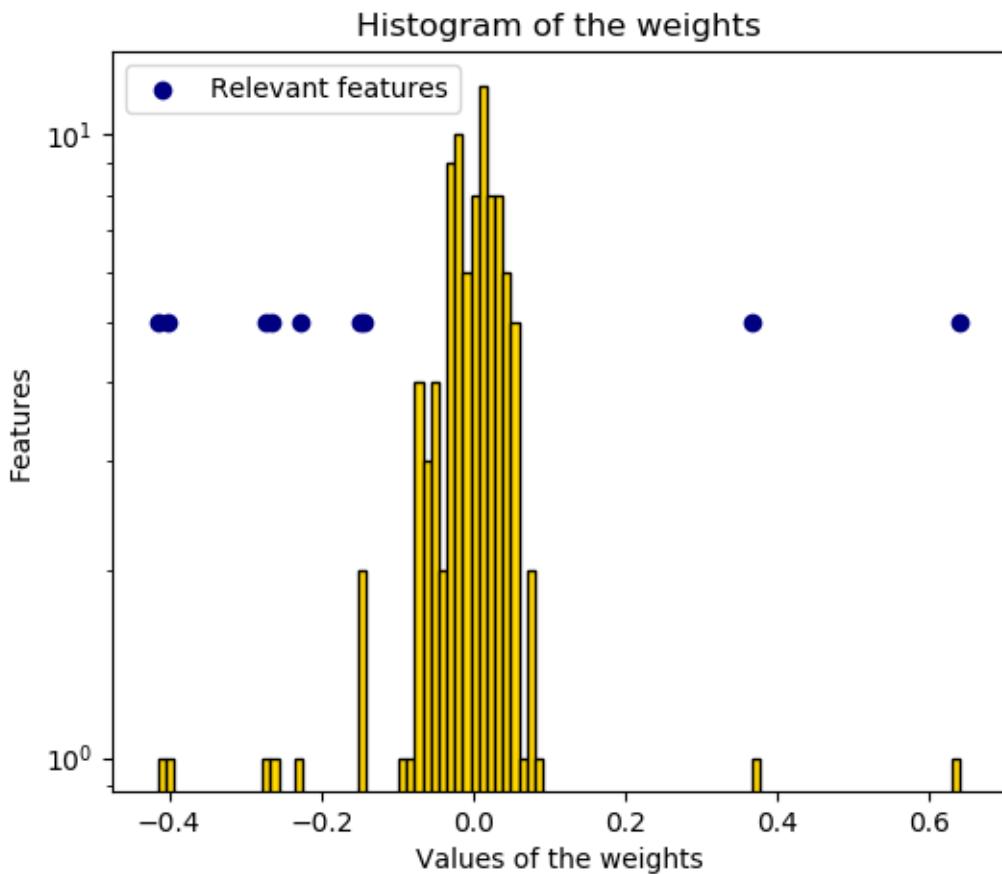
Compared to the OLS (ordinary least squares) estimator, the coefficient weights are slightly shifted toward zeros, which stabilises them.

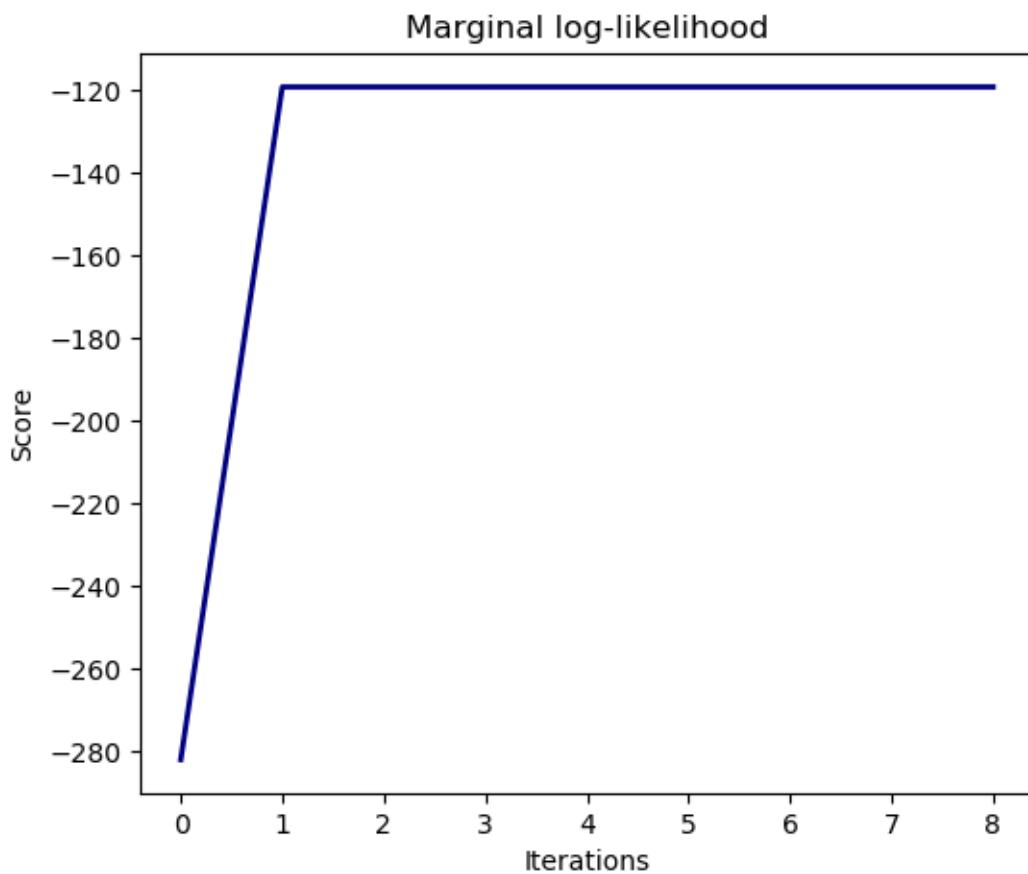
As the prior on the weights is a Gaussian prior, the histogram of the estimated weights is Gaussian.

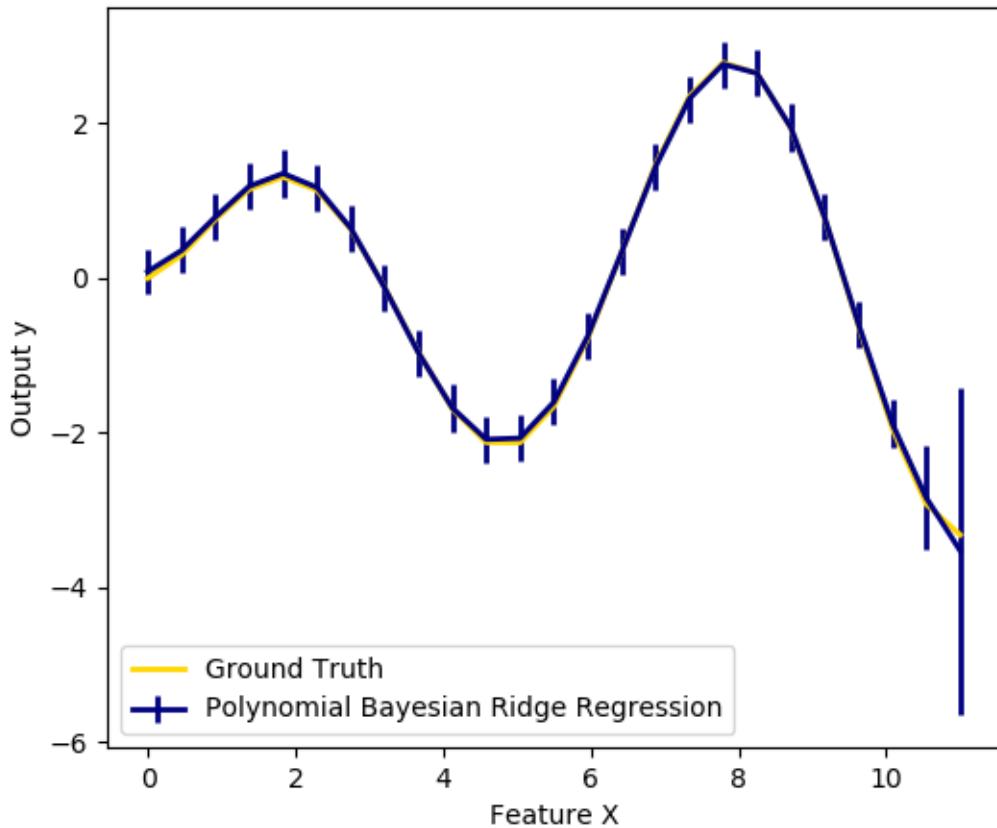
The estimation of the model is done by iteratively maximizing the marginal log-likelihood of the observations.

We also plot predictions and uncertainties for Bayesian Ridge Regression for one dimensional regression using polynomial feature expansion. Note the uncertainty starts going up on the right side of the plot. This is because these test samples are outside of the range of the training samples.









```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

from sklearn.linear_model import BayesianRidge, LinearRegression

# ##### Generating simulated data with Gaussian weights #####
# Generating simulated data with Gaussian weights
np.random.seed(0)
n_samples, n_features = 100, 100
X = np.random.randn(n_samples, n_features) # Create Gaussian data
# Create weights with a precision lambda_ of 4.
lambda_ = 4.
w = np.zeros(n_features)
# Only keep 10 weights of interest
relevant_features = np.random.randint(0, n_features, 10)
for i in relevant_features:
    w[i] = stats.norm.rvs(loc=0, scale=1. / np.sqrt(lambda_))
# Create noise with a precision alpha of 50.
alpha_ = 50.
noise = stats.norm.rvs(loc=0, scale=1. / np.sqrt(alpha_), size=n_samples)
# Create the target
y = np.dot(X, w) + noise

```

```

# ##### Fit the Bayesian Ridge Regression and an OLS for comparison
# Fit the Bayesian Ridge Regression and an OLS for comparison
clf = BayesianRidge(compute_score=True)
clf.fit(X, y)

ols = LinearRegression()
ols.fit(X, y)

# Plot true weights, estimated weights, histogram of the weights, and
# predictions with standard deviations
lw = 2
plt.figure(figsize=(6, 5))
plt.title("Weights of the model")
plt.plot(clf.coef_, color='lightgreen', linewidth=lw,
         label="Bayesian Ridge estimate")
plt.plot(w, color='gold', linewidth=lw, label="Ground truth")
plt.plot(ols.coef_, color='navy', linestyle='--', label="OLS estimate")
plt.xlabel("Features")
plt.ylabel("Values of the weights")
plt.legend(loc="best", prop=dict(size=12))

plt.figure(figsize=(6, 5))
plt.title("Histogram of the weights")
plt.hist(clf.coef_, bins=n_features, color='gold', log=True,
         edgecolor='black')
plt.scatter(clf.coef_[relevant_features], np.full(len(relevant_features), 5.),
            color='navy', label="Relevant features")
plt.xlabel("Features")
plt.ylabel("Values of the weights")
plt.legend(loc="upper left")

plt.figure(figsize=(6, 5))
plt.title("Marginal log-likelihood")
plt.plot(clf.scores_, color='navy', linewidth=lw)
plt.ylabel("Score")
plt.xlabel("Iterations")

# Plotting some predictions for polynomial regression
def f(x, noise_amount):
    y = np.sqrt(x) * np.sin(x)
    noise = np.random.normal(0, 1, len(x))
    return y + noise_amount * noise

degree = 10
X = np.linspace(0, 10, 100)
y = f(X, noise_amount=0.1)
clf_poly = BayesianRidge()
clf_poly.fit(np.vander(X, degree), y)

X_plot = np.linspace(0, 11, 25)
y_plot = f(X_plot, noise_amount=0)
y_mean, y_std = clf_poly.predict(np.vander(X_plot, degree), return_std=True)
plt.figure(figsize=(6, 5))
plt.errorbar(X_plot, y_mean, y_std, color='navy',

```

```
label="Polynomial Bayesian Ridge Regression", linewidth=lw)
plt.plot(X_plot, y_plot, color='gold', linewidth=lw,
          label="Ground Truth")
plt.ylabel("Output y")
plt.xlabel("Feature X")
plt.legend(loc="lower left")
plt.show()
```

Total running time of the script: (0 minutes 0.143 seconds)

Note: Click [here](#) to download the full example code

5.18.31 Lasso model selection: Cross-Validation / AIC / BIC

Use the Akaike information criterion (AIC), the Bayes Information criterion (BIC) and cross-validation to select an optimal value of the regularization parameter alpha of the *Lasso* estimator.

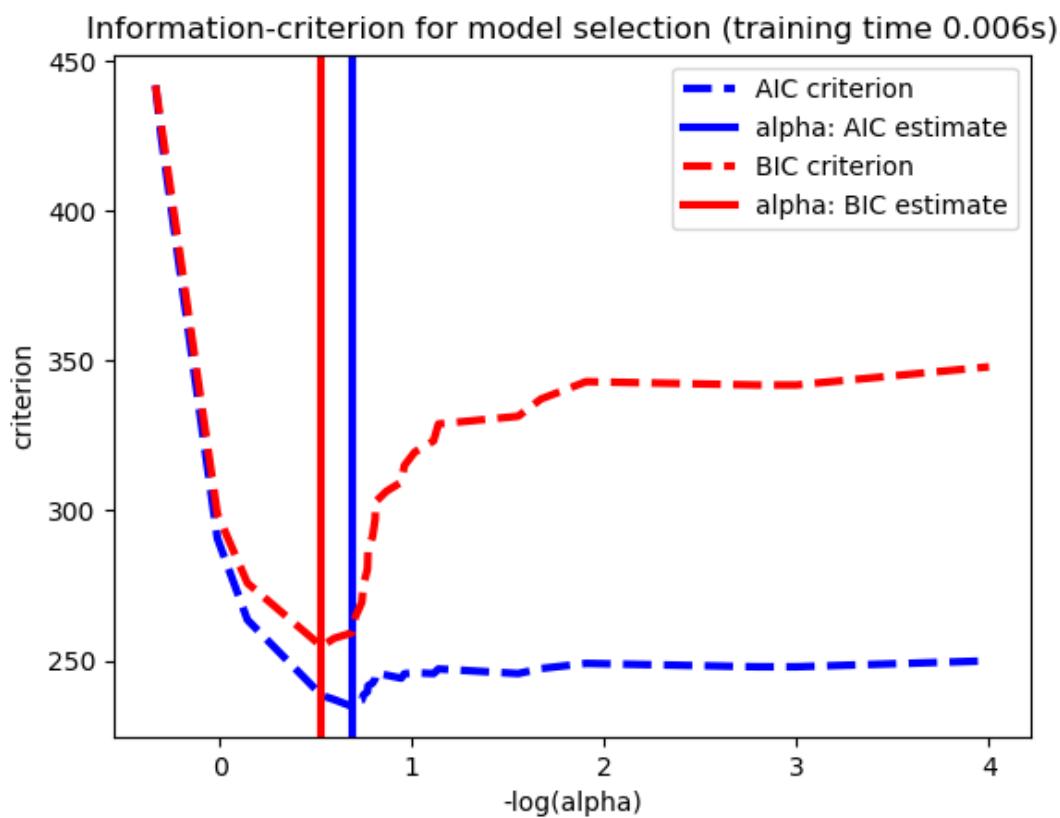
Results obtained with LassoLarsIC are based on AIC/BIC criteria.

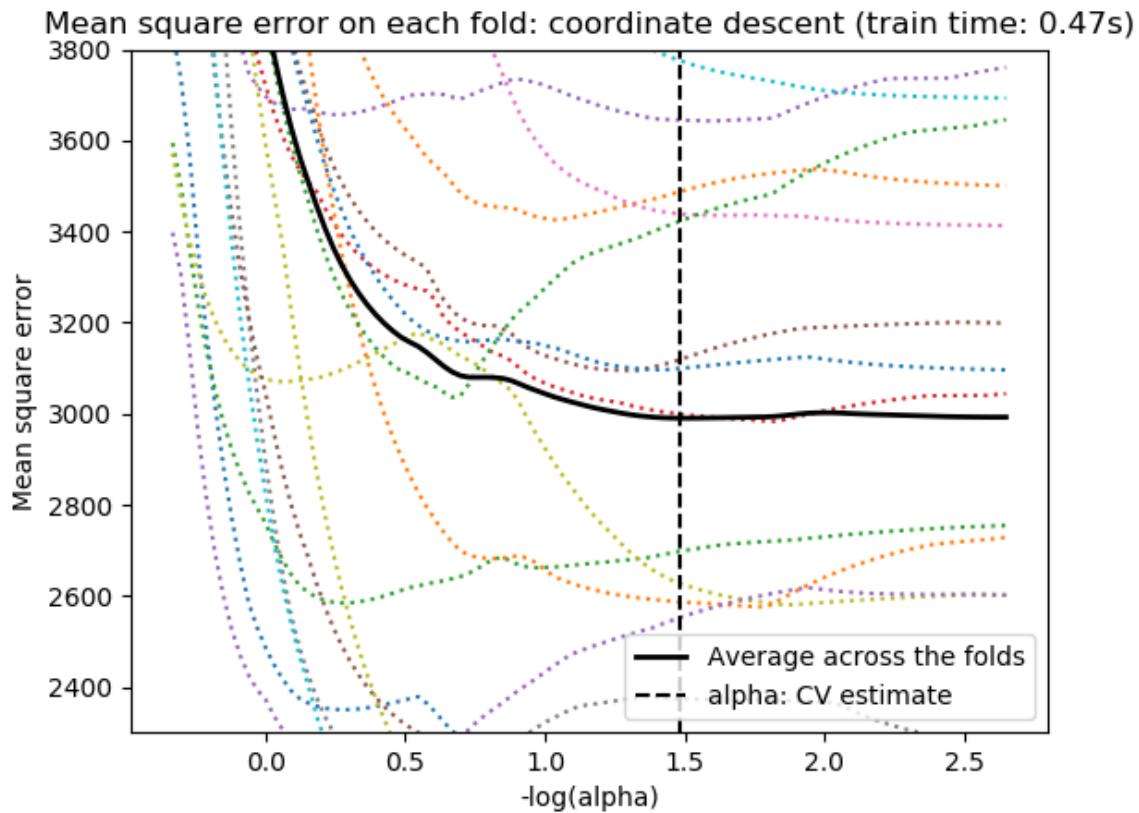
Information-criterion based model selection is very fast, but it relies on a proper estimation of degrees of freedom, are derived for large samples (asymptotic results) and assume the model is correct, i.e. that the data are actually generated by this model. They also tend to break when the problem is badly conditioned (more features than samples).

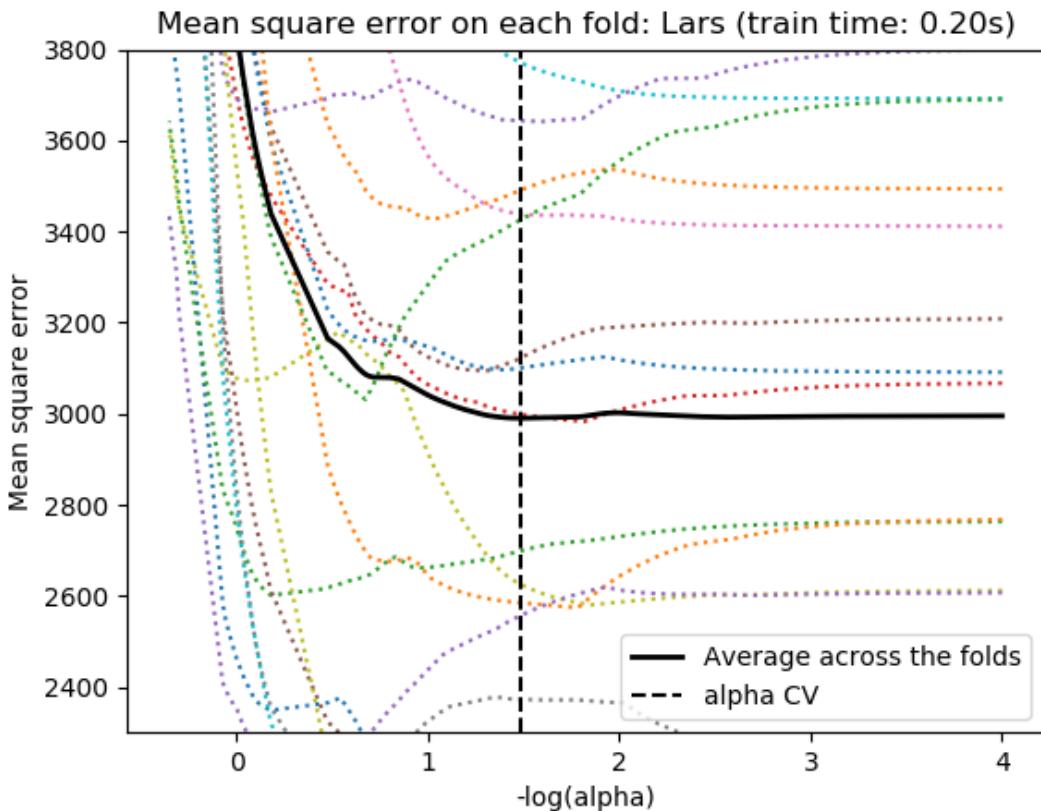
For cross-validation, we use 20-fold with 2 algorithms to compute the Lasso path: coordinate descent, as implemented by the LassoCV class, and Lars (least angle regression) as implemented by the LassoLarsCV class. Both algorithms give roughly the same results. They differ with regards to their execution speed and sources of numerical errors.

Lars computes a path solution only for each kink in the path. As a result, it is very efficient when there are only few kinks, which is the case if there are few features or samples. Also, it is able to compute the full path without setting any meta parameter. On the opposite, coordinate descent compute the path points on a pre-specified grid (here we use the default). Thus it is more efficient if the number of grid points is smaller than the number of kinks in the path. Such a strategy can be interesting if the number of features is really large and there are enough samples to select a large amount. In terms of numerical errors, for heavily correlated variables, Lars will accumulate more errors, while the coordinate descent algorithm will only sample the path on a grid.

Note how the optimal value of alpha varies for each fold. This illustrates why nested-cross validation is necessary when trying to evaluate the performance of a method for which a parameter is chosen by cross-validation: this choice of parameter may not be optimal for unseen data.







Out:

```
Computing regularization path using the coordinate descent lasso...
Computing regularization path using the Lars lasso...
```

```
print(__doc__)

# Author: Olivier Grisel, Gael Varoquaux, Alexandre Gramfort
# License: BSD 3 clause

import time

import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import LassoCV, LassoLarsCV, LassoLarsIC
from sklearn import datasets

# This is to avoid division by zero while doing np.log10
EPSILON = 1e-4

diabetes = datasets.load_diabetes()
```

```

X = diabetes.data
y = diabetes.target

rng = np.random.RandomState(42)
X = np.c_[X, rng.randn(X.shape[0], 14)] # add some bad features

# normalize data as done by Lars to allow for comparison
X /= np.sqrt(np.sum(X ** 2, axis=0))

# ##### LassoLarsIC: least angle regression with BIC/AIC criterion #####
# LassoLarsIC: least angle regression with BIC/AIC criterion

model_bic = LassoLarsIC(criterion='bic')
t1 = time.time()
model_bic.fit(X, y)
t_bic = time.time() - t1
alpha_bic_ = model_bic.alpha_

model_aic = LassoLarsIC(criterion='aic')
model_aic.fit(X, y)
alpha_aic_ = model_aic.alpha_


def plot_ic_criterion(model, name, color):
    alpha_ = model.alpha_ + EPSILON
    alphas_ = model.alphas_ + EPSILON
    criterion_ = model.criterion_
    plt.plot(-np.log10(alphas_), criterion_, '--', color=color,
             linewidth=3, label='%s criterion' % name)
    plt.axvline(-np.log10(alpha_), color=color, linewidth=3,
                label='alpha: %s estimate' % name)
    plt.xlabel('-log(alpha)')
    plt.ylabel('criterion')

    plt.figure()
    plot_ic_criterion(model_aic, 'AIC', 'b')
    plot_ic_criterion(model_bic, 'BIC', 'r')
    plt.legend()
    plt.title('Information-criterion for model selection (training time %.3fs)' %
              t_bic)

# ##### LassoCV: coordinate descent #####
# LassoCV: coordinate descent

# Compute paths
print("Computing regularization path using the coordinate descent lasso...")
t1 = time.time()
model = LassoCV(cv=20).fit(X, y)
t_lasso_cv = time.time() - t1

# Display results
m_log_alphas = -np.log10(model.alphas_ + EPSILON)

plt.figure()
ymin, ymax = 2300, 3800
plt.plot(m_log_alphas, model.mse_path_, ':')
plt.plot(m_log_alphas, model.mse_path_.mean(axis=-1), 'k',
         label='Average across the folds', linewidth=2)

```

```

plt.axvline(-np.log10(model.alpha_ + EPSILON), linestyle='--', color='k',
            label='alpha: CV estimate')

plt.legend()

plt.xlabel('-log(alpha)')
plt.ylabel('Mean square error')
plt.title('Mean square error on each fold: coordinate descent '
          '(train time: %.2fs)' % t_lasso_cv)
plt.axis('tight')
plt.ylim(ymin, ymax)

# ##### LassoLarsCV: least angle regression

# Compute paths
print("Computing regularization path using the Lars lasso...")
t1 = time.time()
model = LassoLarsCV(cv=20).fit(X, y)
t_lasso_lars_cv = time.time() - t1

# Display results
m_log_alphas = -np.log10(model.cv_alphas_ + EPSILON)

plt.figure()
plt.plot(m_log_alphas, model.mse_path_, ':')
plt.plot(m_log_alphas, model.mse_path_.mean(axis=-1), 'k',
         label='Average across the folds', linewidth=2)
plt.axvline(-np.log10(model.alpha_), linestyle='--', color='k',
            label='alpha CV')
plt.legend()

plt.xlabel('-log(alpha)')
plt.ylabel('Mean square error')
plt.title('Mean square error on each fold: Lars (train time: %.2fs) '
          '% t_lasso_lars_cv')
plt.axis('tight')
plt.ylim(ymin, ymax)

plt.show()

```

Total running time of the script: (0 minutes 0.811 seconds)

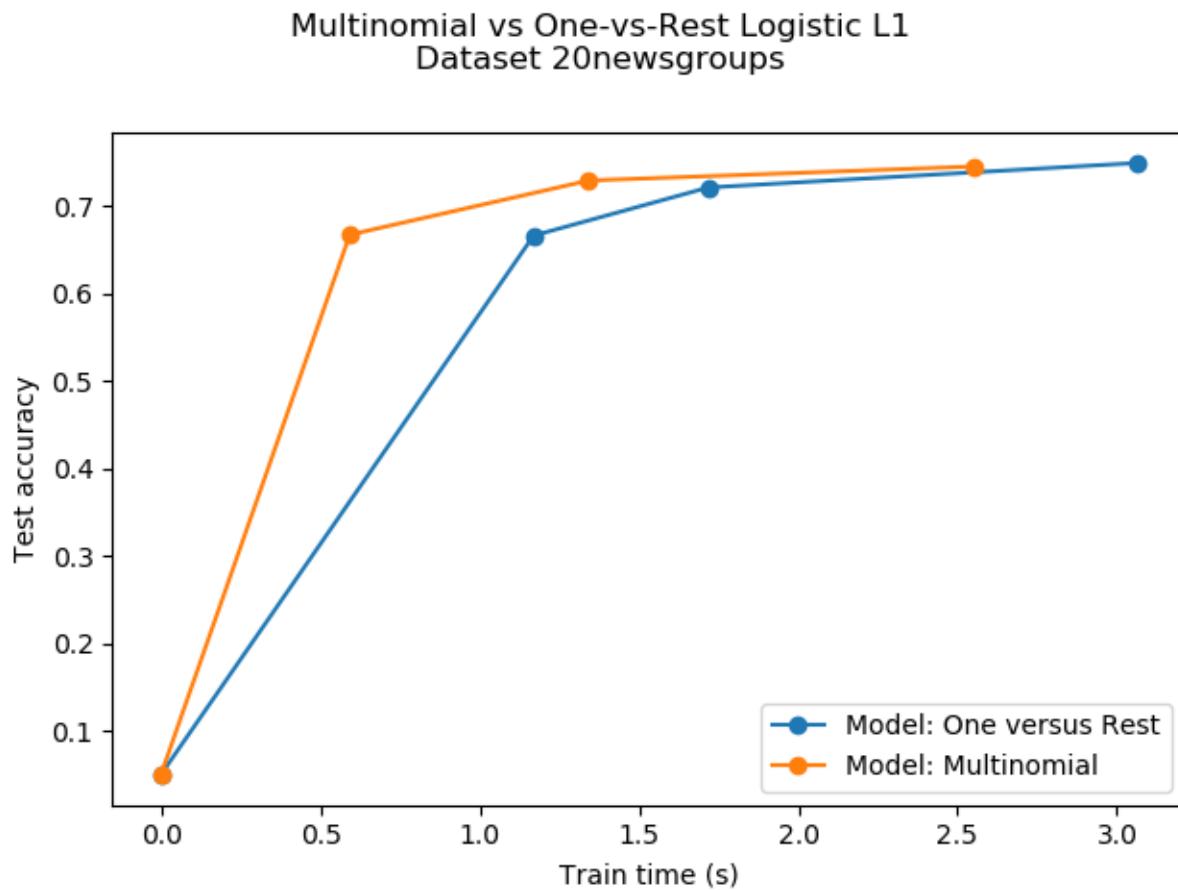
Note: Click [here](#) to download the full example code

5.18.32 Multiclass sparse logistic regression on newgroups20

Comparison of multinomial logistic L1 vs one-versus-rest L1 logistic regression to classify documents from the newgroups20 dataset. Multinomial logistic regression yields more accurate results and is faster to train on the larger scale dataset.

Here we use the l1 sparsity that trims the weights of not informative features to zero. This is good if the goal is to extract the strongly discriminative vocabulary of each class. If the goal is to get the best predictive accuracy, it is better to use the non sparsity-inducing l2 penalty instead.

A more traditional (and possibly better) way to predict on a sparse subset of input features would be to use univariate feature selection followed by a traditional (l2-penalised) logistic regression model.



Out:

```
Dataset 20newsgroup, train_samples=9000, n_features=130107, n_classes=20
[model=One versus Rest, solver=saga] Number of epochs: 1
[model=One versus Rest, solver=saga] Number of epochs: 2
[model=One versus Rest, solver=saga] Number of epochs: 4
Test accuracy for model ovr: 0.7490
% non-zero coefficients for model ovr, per class:
[0.31743104 0.36815852 0.4181174 0.46115889 0.24595141 0.41350581
 0.31281945 0.27054655 0.58720899 0.32972861 0.4158116 0.3312658
 0.41888599 0.41120001 0.59643217 0.31666244 0.34279478 0.28130692
 0.35278655 0.24748861]
Run time (4 epochs) for model ovr: 3.06
[model=Multinomial, solver=saga] Number of epochs: 1
[model=Multinomial, solver=saga] Number of epochs: 3
[model=Multinomial, solver=saga] Number of epochs: 7
Test accuracy for model multinomial: 0.7450
% non-zero coefficients for model multinomial, per class:
[0.13219888 0.11452112 0.13066169 0.13681047 0.12066991 0.15909982
 0.13450468 0.09146318 0.07916561 0.12143851 0.13911627 0.10760374
 0.18984374 0.12143851 0.17524038 0.22289346 0.11605832 0.07916561
 0.07301682 0.15141384]
Run time (7 epochs) for model multinomial: 2.55
Example run in 11.262 s
```

```

import timeit
import warnings

import matplotlib.pyplot as plt
import numpy as np

from sklearn.datasets import fetch_20newsgroups_vectorized
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.exceptions import ConvergenceWarning

print(__doc__)
# Author: Arthur Mensch

warnings.filterwarnings("ignore", category=ConvergenceWarning,
                       module="sklearn")
t0 = timeit.default_timer()

# We use SAGA solver
solver = 'saga'

# Turn down for faster run time
n_samples = 10000

# Memorized fetch_rcv1 for faster access
dataset = fetch_20newsgroups_vectorized('all')
X = dataset.data
y = dataset.target
X = X[:n_samples]
y = y[:n_samples]

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    random_state=42,
                                                    stratify=y,
                                                    test_size=0.1)
train_samples, n_features = X_train.shape
n_classes = np.unique(y).shape[0]

print('Dataset 20newsgroup, train_samples=%i, n_features=%i, n_classes=%i'
      % (train_samples, n_features, n_classes))

models = {'ovr': {'name': 'One versus Rest', 'iters': [1, 2, 4]},
          'multinomial': {'name': 'Multinomial', 'iters': [1, 3, 7]}}

for model in models:
    # Add initial chance-level values for plotting purpose
    accuracies = [1 / n_classes]
    times = [0]
    densities = [1]

    model_params = models[model]

```

```
# Small number of epochs for fast runtime
for this_max_iter in model_params['iters']:
    print('[model=%s, solver=%s] Number of epochs: %s' %
          (model_params['name'], solver, this_max_iter))
    lr = LogisticRegression(solver=solver,
                            multi_class=model,
                            C=1,
                            penalty='l1',
                            fit_intercept=True,
                            max_iter=this_max_iter,
                            random_state=42,
                            )
    t1 = timeit.default_timer()
    lr.fit(X_train, y_train)
    train_time = timeit.default_timer() - t1

    y_pred = lr.predict(X_test)
    accuracy = np.sum(y_pred == y_test) / y_test.shape[0]
    density = np.mean(np.abs(lr.coef_) != 0, axis=1) * 100
    accuracies.append(accuracy)
    densities.append(density)
    times.append(train_time)
models[model]['times'] = times
models[model]['densities'] = densities
models[model]['accuracies'] = accuracies
print('Test accuracy for model %s: %.4f' % (model, accuracies[-1]))
print('%% non-zero coefficients for model %s, '
      'per class:\n %s' % (model, densities[-1]))
print('Run time (%i epochs) for model %s:'
      '%.2f' % (model_params['iters'][-1], model, times[-1]))

fig = plt.figure()
ax = fig.add_subplot(111)

for model in models:
    name = models[model]['name']
    times = models[model]['times']
    accuracies = models[model]['accuracies']
    ax.plot(times, accuracies, marker='o',
            label='Model: %s' % name)
    ax.set_xlabel('Train time (s)')
    ax.set_ylabel('Test accuracy')
ax.legend()
fig.suptitle('Multinomial vs One-vs-Rest Logistic L1\n'
             'Dataset %s' % '20newsgroups')
fig.tight_layout()
fig.subplots_adjust(top=0.85)
run_time = timeit.default_timer() - t0
print('Example run in %.3f s' % run_time)
plt.show()
```

Total running time of the script: (0 minutes 11.263 seconds)

Note: Click [here](#) to download the full example code

5.18.33 Early stopping of Stochastic Gradient Descent

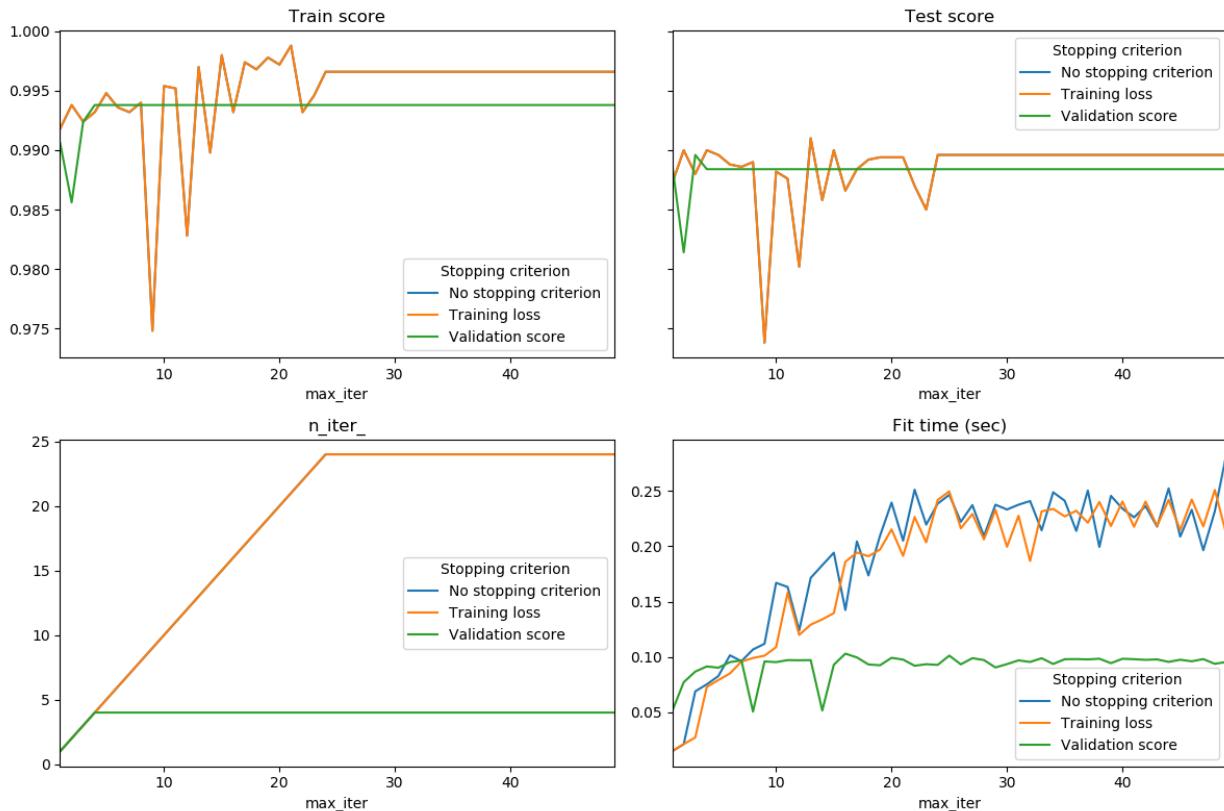
Stochastic Gradient Descent is an optimization technique which minimizes a loss function in a stochastic fashion, performing a gradient descent step sample by sample. In particular, it is a very efficient method to fit linear models.

As a stochastic method, the loss function is not necessarily decreasing at each iteration, and convergence is only guaranteed in expectation. For this reason, monitoring the convergence on the loss function can be difficult.

Another approach is to monitor convergence on a validation score. In this case, the input data is split into a training set and a validation set. The model is then fitted on the training set and the stopping criterion is based on the prediction score computed on the validation set. This enables us to find the least number of iterations which is sufficient to build a model that generalizes well to unseen data and reduces the chance of over-fitting the training data.

This early stopping strategy is activated if `early_stopping=True`; otherwise the stopping criterion only uses the training loss on the entire input data. To better control the early stopping strategy, we can specify a parameter `validation_fraction` which set the fraction of the input dataset that we keep aside to compute the validation score. The optimization will continue until the validation score did not improve by at least `tol` during the last `n_iter_no_change` iterations. The actual number of iterations is available at the attribute `n_iter_`.

This example illustrates how the early stopping can be used in the `sklearn.linear_model.SGDClassifier` model to achieve almost the same accuracy as compared to a model built without early stopping. This can significantly reduce training time. Note that scores differ between the stopping criteria even from early iterations because some of the training data is held out with the validation stopping criterion.



Out:

```
No stopping criterion: .....
Training loss: .....
Validation score: .....
```

```
# Authors: Tom Dupre la Tour
#
# License: BSD 3 clause
import time
import sys

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

from sklearn import linear_model
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.utils.testing import ignore_warnings
from sklearn.exceptions import ConvergenceWarning
from sklearn.utils import shuffle

print(__doc__)

def load_mnist(n_samples=None, class_0='0', class_1='8'):
    """Load MNIST, select two classes, shuffle and return only n_samples."""
    # Load data from http://openml.org/d/554
    mnist = fetch_openml('mnist_784', version=1)

    # take only two classes for binary classification
    mask = np.logical_or(mnist.target == class_0, mnist.target == class_1)

    X, y = shuffle(mnist.data[mask], mnist.target[mask], random_state=42)
    if n_samples is not None:
        X, y = X[:n_samples], y[:n_samples]
    return X, y

@ignore_warnings(category=ConvergenceWarning)
def fit_and_score(estimator, max_iter, X_train, X_test, y_train, y_test):
    """Fit the estimator on the train set and score it on both sets"""
    estimator.set_params(max_iter=max_iter)
    estimator.set_params(random_state=0)

    start = time.time()
    estimator.fit(X_train, y_train)

    fit_time = time.time() - start
    n_iter = estimator.n_iter_
    train_score = estimator.score(X_train, y_train)
    test_score = estimator.score(X_test, y_test)

    return fit_time, n_iter, train_score, test_score

# Define the estimators to compare
estimator_dict = {
    'No stopping criterion':
        linear_model.SGDClassifier(tol=1e-3, n_iter_no_change=3),
```

```

'Training loss':
linear_model.SGDClassifier(early_stopping=False, n_iter_no_change=3,
                           tol=0.1),
'Validation score':
linear_model.SGDClassifier(early_stopping=True, n_iter_no_change=3,
                           tol=0.0001, validation_fraction=0.2)
}

# Load the dataset
X, y = load_mnist(n_samples=10000)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5,
                                                    random_state=0)

results = []
for estimator_name, estimator in estimator_dict.items():
    print(estimator_name + ': ', end=' ')
    for max_iter in range(1, 50):
        print('.', end=' ')
        sys.stdout.flush()

        fit_time, n_iter, train_score, test_score = fit_and_score(
            estimator, max_iter, X_train, X_test, y_train, y_test)

        results.append((estimator_name, max_iter, fit_time, n_iter,
                        train_score, test_score))
    print('')

# Transform the results in a pandas dataframe for easy plotting
columns = [
    'Stopping criterion', 'max_iter', 'Fit time (sec)', 'n_iter_',
    'Train score', 'Test score'
]
results_df = pd.DataFrame(results, columns=columns)

# Define what to plot (x_axis, y_axis)
lines = 'Stopping criterion'
plot_list = [
    ('max_iter', 'Train score'),
    ('max_iter', 'Test score'),
    ('max_iter', 'n_iter_'),
    ('max_iter', 'Fit time (sec)'),
]
]

nrows = 2
ncols = int(np.ceil(len(plot_list) / 2.))
fig, axes = plt.subplots(nrows=nrows, ncols=ncols, figsize=(6 * ncols,
                                                          4 * nrows))
axes[0, 0].get_shared_y_axes().join(axes[0, 0], axes[0, 1])

for ax, (x_axis, y_axis) in zip(axes.ravel(), plot_list):
    for criterion, group_df in results_df.groupby(lines):
        group_df.plot(x=x_axis, y=y_axis, label=criterion, ax=ax)
    ax.set_title(y_axis)
    ax.legend(title=lines)

fig.tight_layout()
plt.show()

```

Total running time of the script: (0 minutes 45.461 seconds)

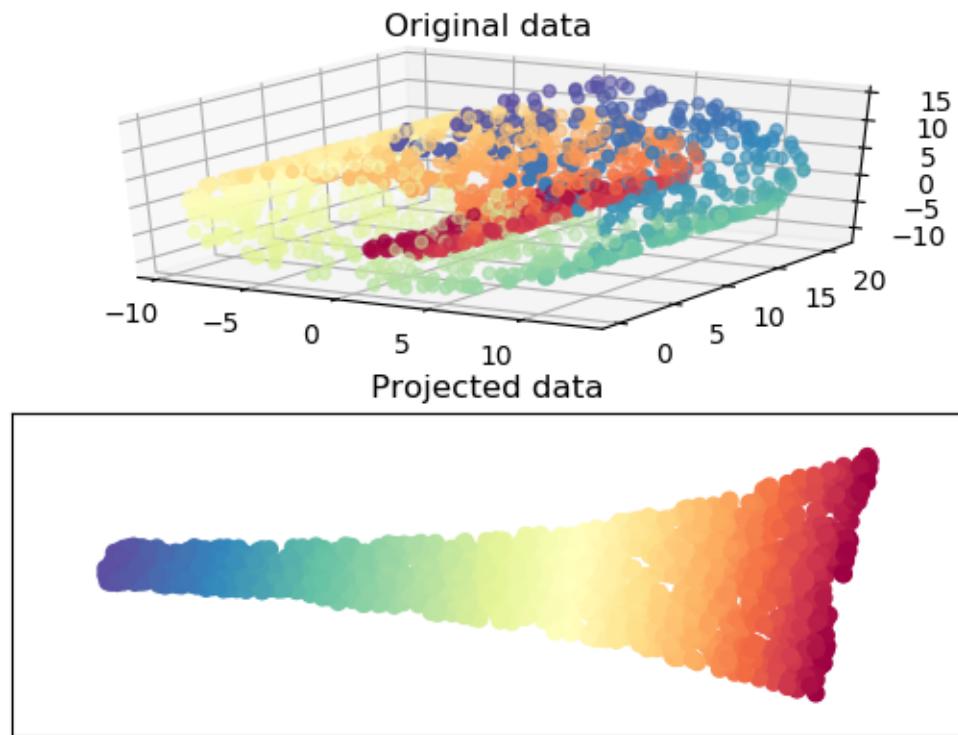
5.19 Manifold learning

Examples concerning the `sklearn.manifold` module.

Note: Click [here](#) to download the full example code

5.19.1 Swiss Roll reduction with LLE

An illustration of Swiss Roll reduction with locally linear embedding



Out:

```
Computing LLE embedding
Done. Reconstruction error: 7.32714e-08
```

```

# Author: Fabian Pedregosa -- <fabian.pedregosa@inria.fr>
# License: BSD 3 clause (C) INRIA 2011

print(__doc__)

import matplotlib.pyplot as plt

# This import is needed to modify the way figure behaves
from mpl_toolkits.mplot3d import Axes3D
Axes3D

#-----
# Locally linear embedding of the swiss roll

from sklearn import manifold, datasets
X, color = datasets.samples_generator.make_swiss_roll(n_samples=1500)

print("Computing LLE embedding")
X_r, err = manifold.locally_linear_embedding(X, n_neighbors=12,
                                              n_components=2)
print("Done. Reconstruction error: %g" % err)

#-----
# Plot result

fig = plt.figure()

ax = fig.add_subplot(211, projection='3d')
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=color, cmap=plt.cm.Spectral)

ax.set_title("Original data")
ax = fig.add_subplot(212)
ax.scatter(X_r[:, 0], X_r[:, 1], c=color, cmap=plt.cm.Spectral)
plt.axis('tight')
plt.xticks([]), plt.yticks([])
plt.title('Projected data')
plt.show()

```

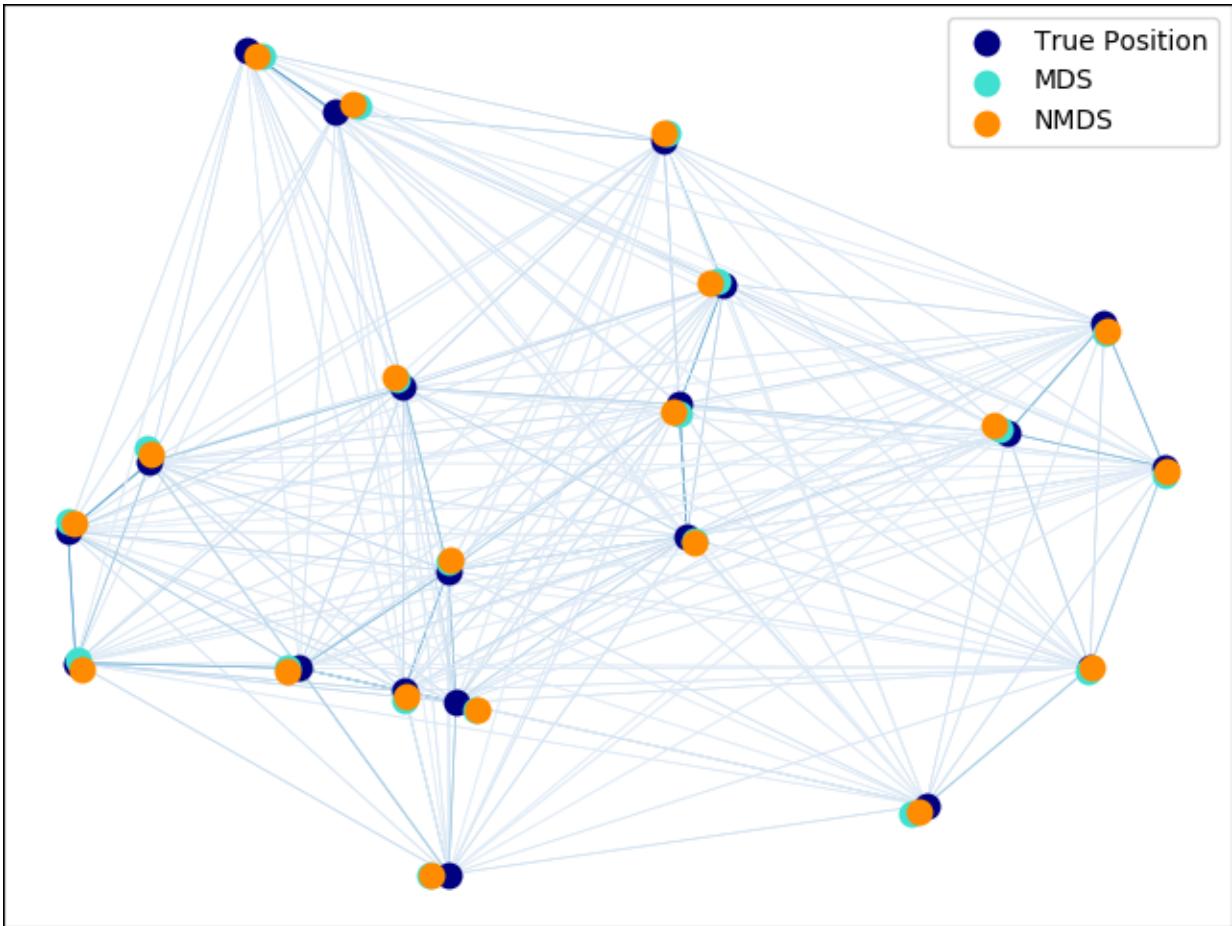
Total running time of the script: (0 minutes 0.182 seconds)

Note: Click [here](#) to download the full example code

5.19.2 Multi-dimensional scaling

An illustration of the metric and non-metric MDS on generated noisy data.

The reconstructed points using the metric MDS and non metric MDS are slightly shifted to avoid overlapping.



```
# Author: Nelle Varoquaux <nelle.varoquaux@gmail.com>
# License: BSD

print(__doc__)
import numpy as np

from matplotlib import pyplot as plt
from matplotlib.collections import LineCollection

from sklearn import manifold
from sklearn.metrics import euclidean_distances
from sklearn.decomposition import PCA

n_samples = 20
seed = np.random.RandomState(seed=3)
X_true = seed.randint(0, 20, 2 * n_samples).astype(np.float)
X_true = X_true.reshape((n_samples, 2))
# Center the data
X_true -= X_true.mean()

similarities = euclidean_distances(X_true)

# Add noise to the similarities
noise = np.random.rand(n_samples, n_samples)
noise = noise + noise.T
noise[np.arange(noise.shape[0]), np.arange(noise.shape[0])] = 0
```

```

similarities += noise

mds = manifold.MDS(n_components=2, max_iter=3000, eps=1e-9, random_state=seed,
                    dissimilarity="precomputed", n_jobs=1)
pos = mds.fit(similarities).embedding_

nmds = manifold.MDS(n_components=2, metric=False, max_iter=3000, eps=1e-12,
                     dissimilarity="precomputed", random_state=seed, n_jobs=1,
                     n_init=1)
npos = nmds.fit_transform(similarities, init=pos)

# Rescale the data
pos *= np.sqrt((X_true ** 2).sum()) / np.sqrt((pos ** 2).sum())
npos *= np.sqrt((X_true ** 2).sum()) / np.sqrt((npos ** 2).sum())

# Rotate the data
clf = PCA(n_components=2)
X_true = clf.fit_transform(X_true)

pos = clf.fit_transform(pos)

npos = clf.fit_transform(npos)

fig = plt.figure(1)
ax = plt.axes([0., 0., 1., 1.])

s = 100
plt.scatter(X_true[:, 0], X_true[:, 1], color='navy', s=s, lw=0,
            label='True Position')
plt.scatter(pos[:, 0], pos[:, 1], color='turquoise', s=s, lw=0, label='MDS')
plt.scatter(npos[:, 0], npos[:, 1], color='darkorange', s=s, lw=0, label='NMDS')
plt.legend(scatterpoints=1, loc='best', shadow=False)

similarities = similarities.max() / similarities * 100
similarities[np.isinf(similarities)] = 0

# Plot the edges
start_idx, end_idx = np.where(pos)
# a sequence of (*line0*, *line1*, *line2*), where::
#      linen = (x0, y0), (x1, y1), ... (xm, ym)
segments = [[X_true[i, :], X_true[j, :]]
            for i in range(len(pos)) for j in range(len(pos))]
values = np.abs(similarities)
lc = LineCollection(segments,
                     zorder=0, cmap=plt.cm.Blues,
                     norm=plt.Normalize(0, values.max()))
lc.set_array(similarities.flatten())
lc.set_linewidths(np.full(len(segments), 0.5))
ax.add_collection(lc)

plt.show()

```

Total running time of the script: (0 minutes 0.063 seconds)

Note: Click [here](#) to download the full example code

5.19.3 t-SNE: The effect of various perplexity values on the shape

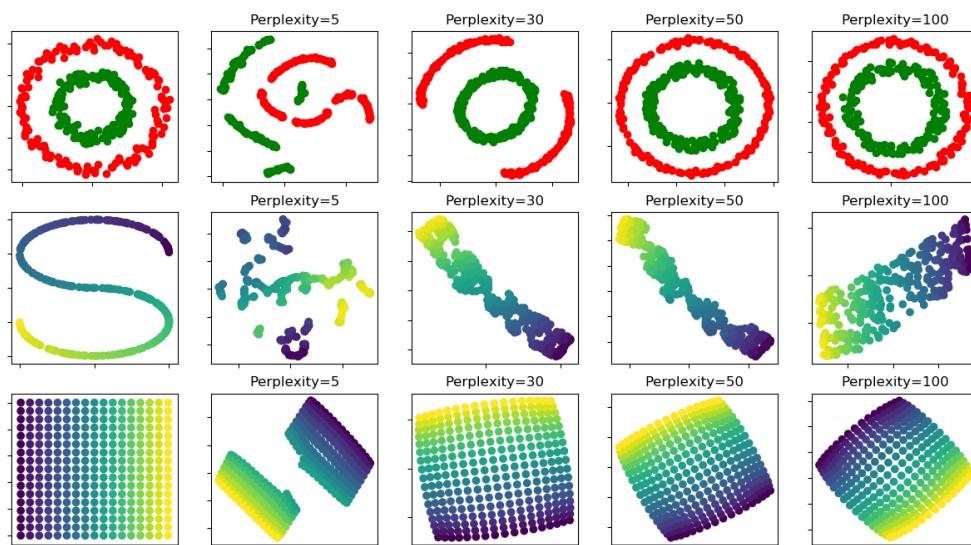
An illustration of t-SNE on the two concentric circles and the S-curve datasets for different perplexity values.

We observe a tendency towards clearer shapes as the perplexity value increases.

The size, the distance and the shape of clusters may vary upon initialization, perplexity values and does not always convey a meaning.

As shown below, t-SNE for higher perplexities finds meaningful topology of two concentric circles, however the size and the distance of the circles varies slightly from the original. Contrary to the two circles dataset, the shapes visually diverge from S-curve topology on the S-curve dataset even for larger perplexity values.

For further details, “How to Use t-SNE Effectively” <https://distill.pub/2016/misread-tsne/> provides a good discussion of the effects of various parameters, as well as interactive plots to explore those effects.



Out:

```

circles, perplexity=5 in 0.89 sec
circles, perplexity=30 in 1.2 sec
circles, perplexity=50 in 1.3 sec
circles, perplexity=100 in 1.8 sec
S-curve, perplexity=5 in 0.92 sec
S-curve, perplexity=30 in 1.2 sec
S-curve, perplexity=50 in 1.4 sec
S-curve, perplexity=100 in 1.9 sec
uniform grid, perplexity=5 in 0.88 sec
uniform grid, perplexity=30 in 1.1 sec
uniform grid, perplexity=50 in 1.1 sec
uniform grid, perplexity=100 in 1.7 sec

```

```

# Author: Narine Kokhlikyan <narine@slice.com>
# License: BSD

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from matplotlib.ticker import NullFormatter
from sklearn import manifold, datasets
from time import time

n_samples = 300
n_components = 2
(fig, subplots) = plt.subplots(3, 5, figsize=(15, 8))
perplexities = [5, 30, 50, 100]

X, y = datasets.make_circles(n_samples=n_samples, factor=.5, noise=.05)

red = y == 0
green = y == 1

ax = subplots[0][0]
ax.scatter(X[red, 0], X[red, 1], c="r")
ax.scatter(X[green, 0], X[green, 1], c="g")
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

for i, perplexity in enumerate(perplexities):
    ax = subplots[0][i + 1]

    t0 = time()
    tsne = manifold.TSNE(n_components=n_components, init='random',
                          random_state=0, perplexity=perplexity)
    Y = tsne.fit_transform(X)
    t1 = time()
    print("circles, perplexity=%d in %.2g sec" % (perplexity, t1 - t0))
    ax.set_title("Perplexity=%d" % perplexity)
    ax.scatter(Y[red, 0], Y[red, 1], c="r")
    ax.scatter(Y[green, 0], Y[green, 1], c="g")
    ax.xaxis.set_major_formatter(NullFormatter())
    ax.yaxis.set_major_formatter(NullFormatter())
    ax.axis('tight')

# Another example using s-curve
X, color = datasets.samples_generator.make_s_curve(n_samples, random_state=0)

ax = subplots[1][0]
ax.scatter(X[:, 0], X[:, 2], c=color)
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())

for i, perplexity in enumerate(perplexities):
    ax = subplots[1][i + 1]

    t0 = time()
    tsne = manifold.TSNE(n_components=n_components, init='random',

```

```
random_state=0, perplexity=perplexity)
Y = tsne.fit_transform(X)
t1 = time()
print("S-curve, perplexity=%d in %.2g sec" % (perplexity, t1 - t0))

ax.set_title("Perplexity=%d" % perplexity)
ax.scatter(Y[:, 0], Y[:, 1], c=color)
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
ax.axis('tight')

# Another example using a 2D uniform grid
x = np.linspace(0, 1, int(np.sqrt(n_samples)))
xx, yy = np.meshgrid(x, x)
X = np.hstack([
    xx.ravel().reshape(-1, 1),
    yy.ravel().reshape(-1, 1),
])
color = xx.ravel()
ax = subplots[2][0]
ax.scatter(X[:, 0], X[:, 1], c=color)
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())

for i, perplexity in enumerate(perplexities):
    ax = subplots[2][i + 1]

    t0 = time()
    tsne = manifold.TSNE(n_components=n_components, init='random',
                          random_state=0, perplexity=perplexity)
    Y = tsne.fit_transform(X)
    t1 = time()
    print("uniform grid, perplexity=%d in %.2g sec" % (perplexity, t1 - t0))

    ax.set_title("Perplexity=%d" % perplexity)
    ax.scatter(Y[:, 0], Y[:, 1], c=color)
    ax.xaxis.set_major_formatter(NullFormatter())
    ax.yaxis.set_major_formatter(NullFormatter())
    ax.axis('tight')

plt.show()
```

Total running time of the script: (0 minutes 15.568 seconds)

Note: Click [here](#) to download the full example code

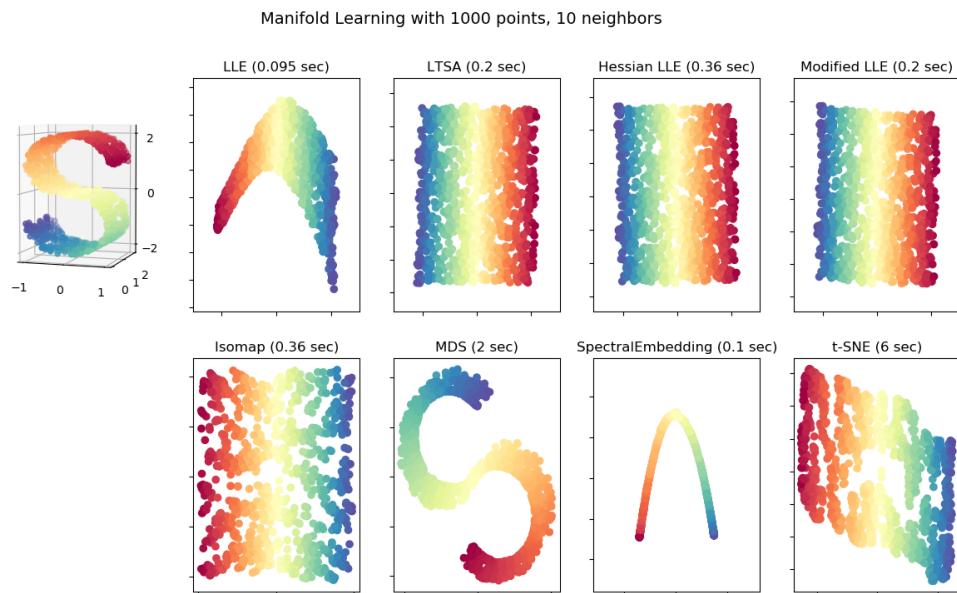
5.19.4 Comparison of Manifold Learning methods

An illustration of dimensionality reduction on the S-curve dataset with various manifold learning methods.

For a discussion and comparison of these algorithms, see the [manifold module page](#)

For a similar example, where the methods are applied to a sphere dataset, see [Manifold Learning methods on a severed sphere](#)

Note that the purpose of the MDS is to find a low-dimensional representation of the data (here 2D) in which the distances respect well the distances in the original high-dimensional space, unlike other manifold-learning algorithms, it does not seek an isotropic representation of the data in the low-dimensional space.



Out:

```
standard: 0.095 sec
ltsa: 0.2 sec
hessian: 0.36 sec
modified: 0.2 sec
Isomap: 0.36 sec
MDS: 2 sec
SpectralEmbedding: 0.1 sec
t-SNE: 6 sec
```

```
# Author: Jake Vanderplas -- <vanderplas@astro.washington.edu>

print(__doc__)

from time import time

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import NullFormatter

from sklearn import manifold, datasets

# Next line to silence pyflakes. This import is needed.
Axes3D
```

```

n_points = 1000
X, color = datasets.samples_generator.make_s_curve(n_points, random_state=0)
n_neighbors = 10
n_components = 2

fig = plt.figure(figsize=(15, 8))
plt.suptitle("Manifold Learning with %i points, %i neighbors"
             % (1000, n_neighbors), fontsize=14)

ax = fig.add_subplot(251, projection='3d')
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=color, cmap=plt.cm.Spectral)
ax.view_init(4, -72)

methods = ['standard', 'ltsa', 'hessian', 'modified']
labels = ['LLE', 'LTSA', 'Hessian LLE', 'Modified LLE']

for i, method in enumerate(methods):
    t0 = time()
    Y = manifold.LocallyLinearEmbedding(n_neighbors, n_components,
                                         eigen_solver='auto',
                                         method=method).fit_transform(X)
    t1 = time()
    print("%s: %.2g sec" % (methods[i], t1 - t0))

    ax = fig.add_subplot(252 + i)
    plt.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)
    plt.title("%s (%.2g sec)" % (labels[i], t1 - t0))
    ax.xaxis.set_major_formatter(NullFormatter())
    ax.yaxis.set_major_formatter(NullFormatter())
    plt.axis('tight')

t0 = time()
Y = manifold.Isomap(n_neighbors, n_components).fit_transform(X)
t1 = time()
print("Isomap: %.2g sec" % (t1 - t0))
ax = fig.add_subplot(257)
plt.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)
plt.title("Isomap (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

t0 = time()
mds = manifold.MDS(n_components, max_iter=100, n_init=1)
Y = mds.fit_transform(X)
t1 = time()
print("MDS: %.2g sec" % (t1 - t0))
ax = fig.add_subplot(258)
plt.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)
plt.title("MDS (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

t0 = time()

```

```

se = manifold.SpectralEmbedding(n_components=n_components,
                                 n_neighbors=n_neighbors)
Y = se.fit_transform(X)
t1 = time()
print("SpectralEmbedding: %.2g sec" % (t1 - t0))
ax = fig.add_subplot(259)
plt.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)
plt.title("SpectralEmbedding (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

t0 = time()
tsne = manifold.TSNE(n_components=n_components, init='pca', random_state=0)
Y = tsne.fit_transform(X)
t1 = time()
print("t-SNE: %.2g sec" % (t1 - t0))
ax = fig.add_subplot(2, 5, 10)
plt.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)
plt.title("t-SNE (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

plt.show()

```

Total running time of the script: (0 minutes 9.515 seconds)

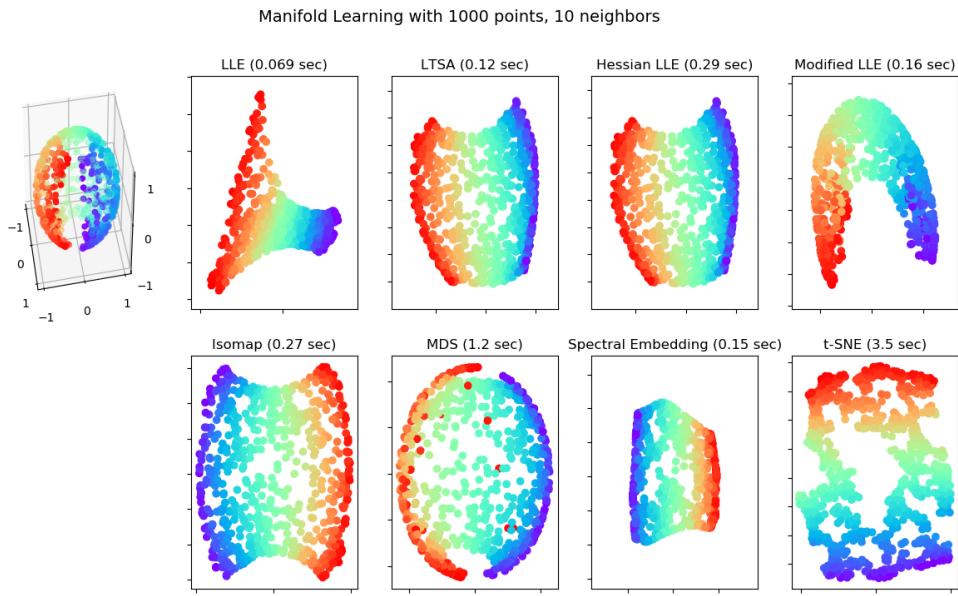
Note: Click [here](#) to download the full example code

5.19.5 Manifold Learning methods on a severed sphere

An application of the different *Manifold learning* techniques on a spherical data-set. Here one can see the use of dimensionality reduction in order to gain some intuition regarding the manifold learning methods. Regarding the dataset, the poles are cut from the sphere, as well as a thin slice down its side. This enables the manifold learning techniques to ‘spread it open’ whilst projecting it onto two dimensions.

For a similar example, where the methods are applied to the S-curve dataset, see [Comparison of Manifold Learning methods](#)

Note that the purpose of the *MDS* is to find a low-dimensional representation of the data (here 2D) in which the distances respect well the distances in the original high-dimensional space, unlike other manifold-learning algorithms, it does not seek an isotropic representation of the data in the low-dimensional space. Here the manifold problem matches fairly that of representing a flat map of the Earth, as with map projection



Out:

```
standard: 0.069 sec
ltsa: 0.12 sec
hessian: 0.29 sec
modified: 0.16 sec
ISO: 0.27 sec
MDS: 1.2 sec
Spectral Embedding: 0.15 sec
t-SNE: 3.5 sec
```

```
# Author: Jaques Grobler <jaques.grobler@inria.fr>
# License: BSD 3 clause

print(__doc__)

from time import time

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import NullFormatter

from sklearn import manifold
from sklearn.utils import check_random_state

# Next line to silence pyflakes.
Axes3D

# Variables for manifold learning.
```

```

n_neighbors = 10
n_samples = 1000

# Create our sphere.
random_state = check_random_state(0)
p = random_state.rand(n_samples) * (2 * np.pi - 0.55)
t = random_state.rand(n_samples) * np.pi

# Sever the poles from the sphere.
indices = ((t < (np.pi - (np.pi / 8))) & (t > ((np.pi / 8))))
colors = p[indices]
x, y, z = np.sin(t[indices]) * np.cos(p[indices]), \
    np.sin(t[indices]) * np.sin(p[indices]), \
    np.cos(t[indices])

# Plot our dataset.
fig = plt.figure(figsize=(15, 8))
plt.suptitle("Manifold Learning with %i points, %i neighbors"
             % (1000, n_neighbors), fontsize=14)

ax = fig.add_subplot(251, projection='3d')
ax.scatter(x, y, z, c=colors, cmap=plt.cm.rainbow)
ax.view_init(40, -10)

sphere_data = np.array([x, y, z]).T

# Perform Locally Linear Embedding Manifold learning
methods = ['standard', 'ltsa', 'hessian', 'modified']
labels = ['LLE', 'LTSA', 'Hessian LLE', 'Modified LLE']

for i, method in enumerate(methods):
    t0 = time()
    trans_data = manifold\
        .LocallyLinearEmbedding(n_neighbors, 2,
                               method=method).fit_transform(sphere_data).T
    t1 = time()
    print("%s: %.2g sec" % (methods[i], t1 - t0))

    ax = fig.add_subplot(252 + i)
    plt.scatter(trans_data[0], trans_data[1], c=colors, cmap=plt.cm.rainbow)
    plt.title("%s (%.2g sec)" % (labels[i], t1 - t0))
    ax.xaxis.set_major_formatter(NullFormatter())
    ax.yaxis.set_major_formatter(NullFormatter())
    plt.axis('tight')

# Perform Isomap Manifold learning.
t0 = time()
trans_data = manifold.Isomap(n_neighbors, n_components=2) \
    .fit_transform(sphere_data).T
t1 = time()
print("%s: %.2g sec" % ('ISO', t1 - t0))

ax = fig.add_subplot(257)
plt.scatter(trans_data[0], trans_data[1], c=colors, cmap=plt.cm.rainbow)
plt.title("%s (%.2g sec)" % ('Isomap', t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

```

```
# Perform Multi-dimensional scaling.
t0 = time()
mds = manifold.MDS(2, max_iter=100, n_init=1)
trans_data = mds.fit_transform(sphere_data).T
t1 = time()
print("MDS: %.2g sec" % (t1 - t0))

ax = fig.add_subplot(258)
plt.scatter(trans_data[0], trans_data[1], c=colors, cmap=plt.cm.rainbow)
plt.title("MDS (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

# Perform Spectral Embedding.
t0 = time()
se = manifold.SpectralEmbedding(n_components=2,
                                 n_neighbors=n_neighbors)
trans_data = se.fit_transform(sphere_data).T
t1 = time()
print("Spectral Embedding: %.2g sec" % (t1 - t0))

ax = fig.add_subplot(259)
plt.scatter(trans_data[0], trans_data[1], c=colors, cmap=plt.cm.rainbow)
plt.title("Spectral Embedding (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

# Perform t-distributed stochastic neighbor embedding.
t0 = time()
tsne = manifold.TSNE(n_components=2, init='pca', random_state=0)
trans_data = tsne.fit_transform(sphere_data).T
t1 = time()
print("t-SNE: %.2g sec" % (t1 - t0))

ax = fig.add_subplot(2, 5, 10)
plt.scatter(trans_data[0], trans_data[1], c=colors, cmap=plt.cm.rainbow)
plt.title("t-SNE (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
plt.axis('tight')

plt.show()
```

Total running time of the script: (0 minutes 5.954 seconds)

Note: Click [here](#) to download the full example code

5.19.6 Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...

An illustration of various embeddings on the digits dataset.

The RandomTreesEmbedding, from the `sklearn.ensemble` module, is not technically a manifold embedding method, as it learn a high-dimensional representation on which we apply a dimensionality reduction method. However, it is often useful to cast a dataset into a representation in which the classes are linearly-separable.

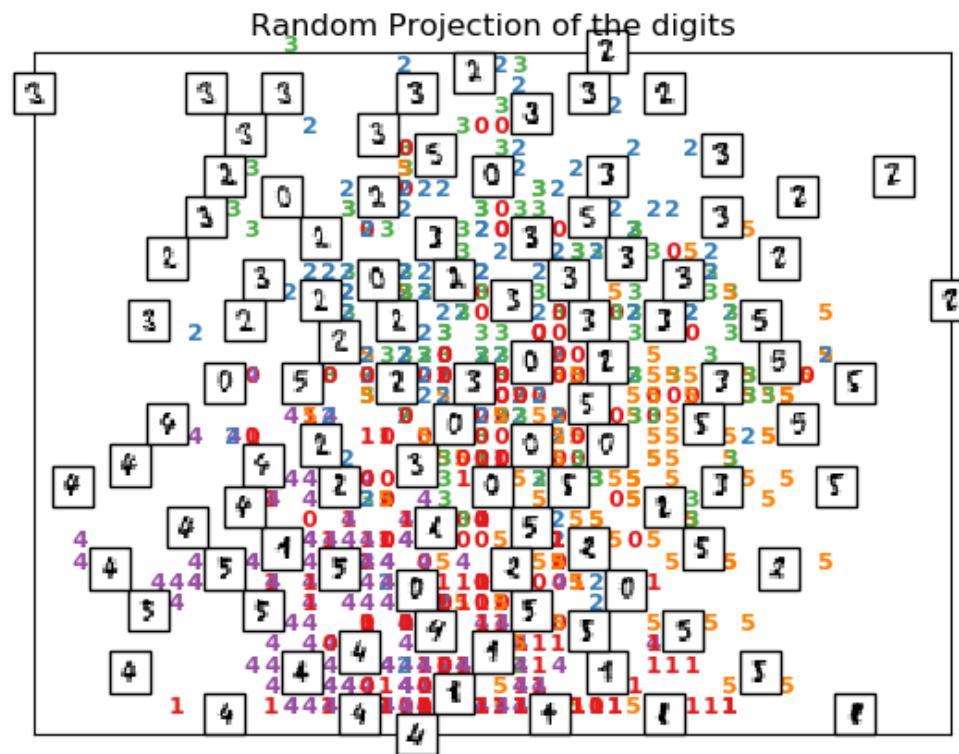
t-SNE will be initialized with the embedding that is generated by PCA in this example, which is not the default setting. It ensures global stability of the embedding, i.e., the embedding does not depend on random initialization.

Linear Discriminant Analysis, from the `sklearn.discriminant_analysis` module, and Neighborhood Components Analysis, from the `sklearn.neighbors` module, are supervised dimensionality reduction method, i.e. they make use of the provided labels, contrary to other methods.

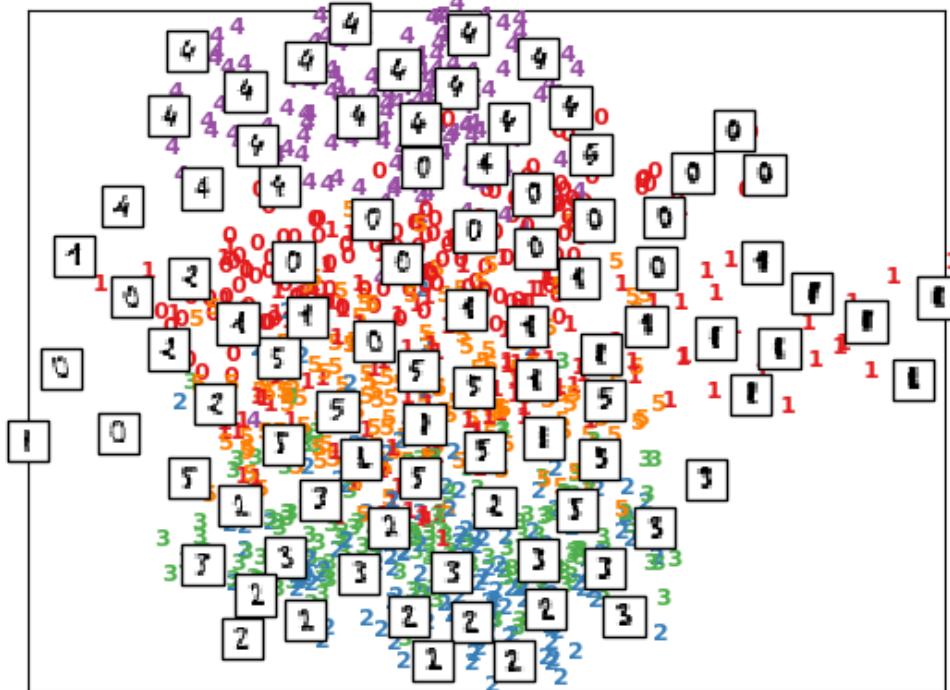
A selection from the 64-dimensional digits dataset

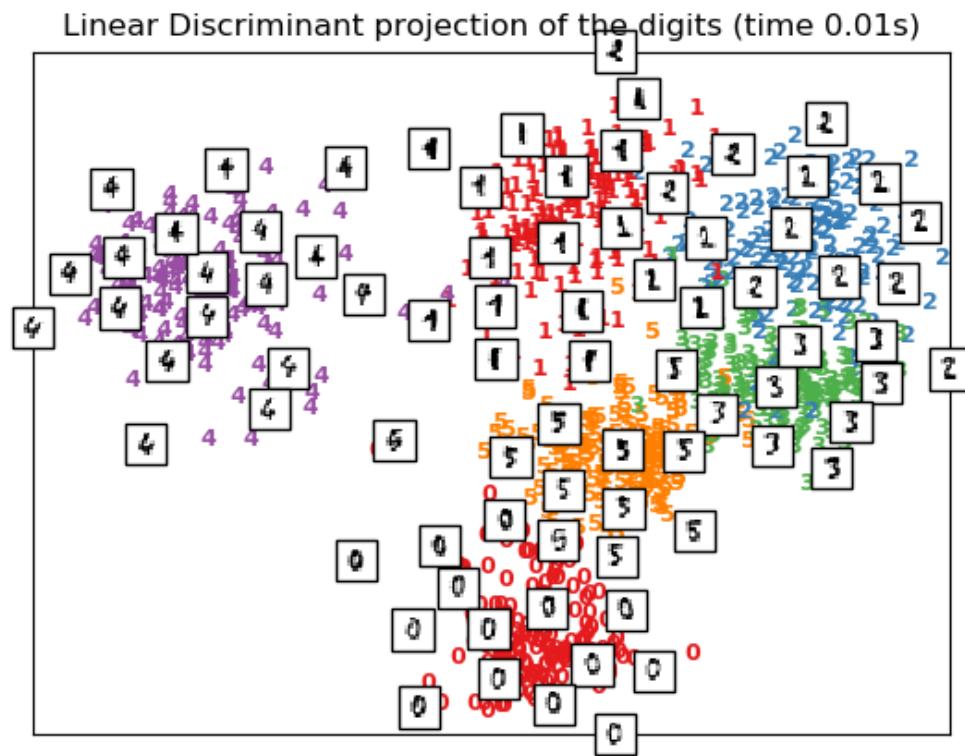
0	1	2	3	4	5	0	1	3	4	5	0	1	2	3	4	5	0	5
5	5	0	4	1	3	5	1	0	0	2	2	2	0	1	2	3	3	3
4	4	1	5	0	5	2	2	0	0	1	3	2	1	4	3	1	3	1
3	4	4	0	5	3	1	5	4	4	2	2	5	3	4	4	0	0	1
2	3	4	5	0	1	2	3	4	3	0	1	2	3	4	5	0	5	5
0	4	1	3	5	4	1	0	0	2	2	1	0	1	2	3	3	3	4
4	5	0	5	2	2	1	0	0	1	3	2	1	3	1	4	3	1	4
0	5	3	2	4	4	1	1	5	5	4	4	0	0	1	2	3	4	
5	0	4	2	3	4	5	0	4	2	3	4	5	0	5	5	5	0	1
3	5	1	0	0	2	2	2	0	1	2	3	3	3	3	4	4	1	5
5	2	2	0	0	4	3	2	4	3	4	3	1	4	3	1	4	0	5
3	3	5	4	4	2	2	2	5	5	4	4	0	3	0	1	2	3	4
0	1	2	3	4	5	0	1	2	3	4	5	0	5	5	5	0	4	1
5	1	0	0	1	2	2	0	1	2	3	3	3	3	4	4	1	5	0
1	2	0	0	1	3	2	1	4	3	1	3	1	4	3	1	4	0	5
1	5	4	4	2	1	2	5	5	4	4	0	0	1	2	3	4	0	1
2	3	4	5	0	1	2	3	4	5	0	5	5	5	0	4	1	3	5
0	0	2	1	2	0	1	3	3	3	3	4	4	4	5	0	5	1	2
0	0	1	1	1	1	4	3	1	3	1	4	3	1	4	0	5	3	1
4	4	2	2	1	5	5	4	4	0	0	1	2	3	4	5	0	1	2

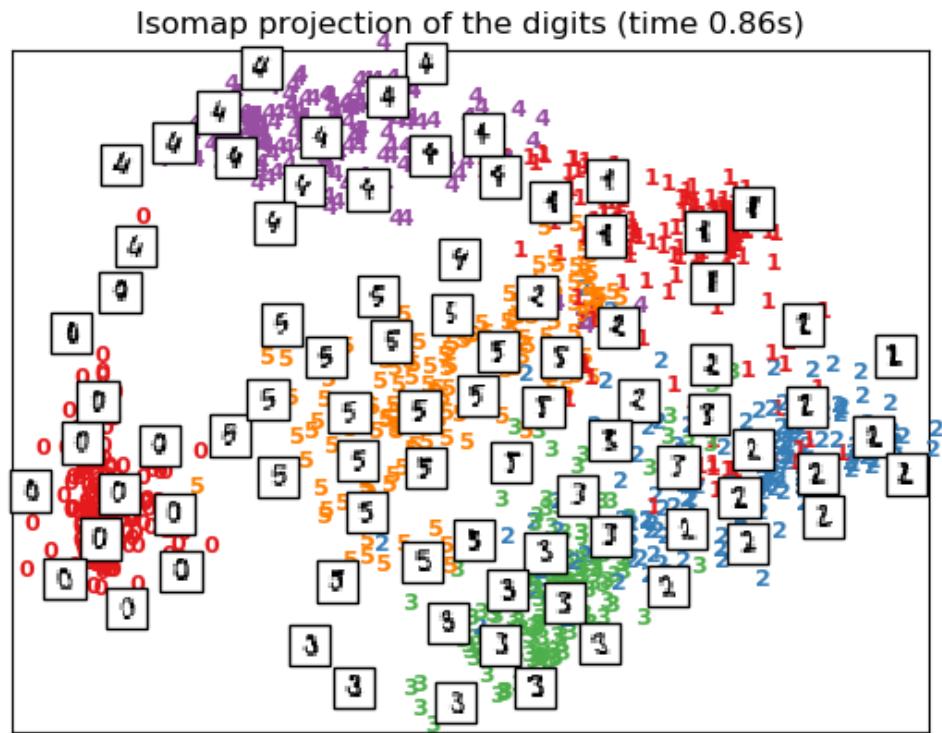
.



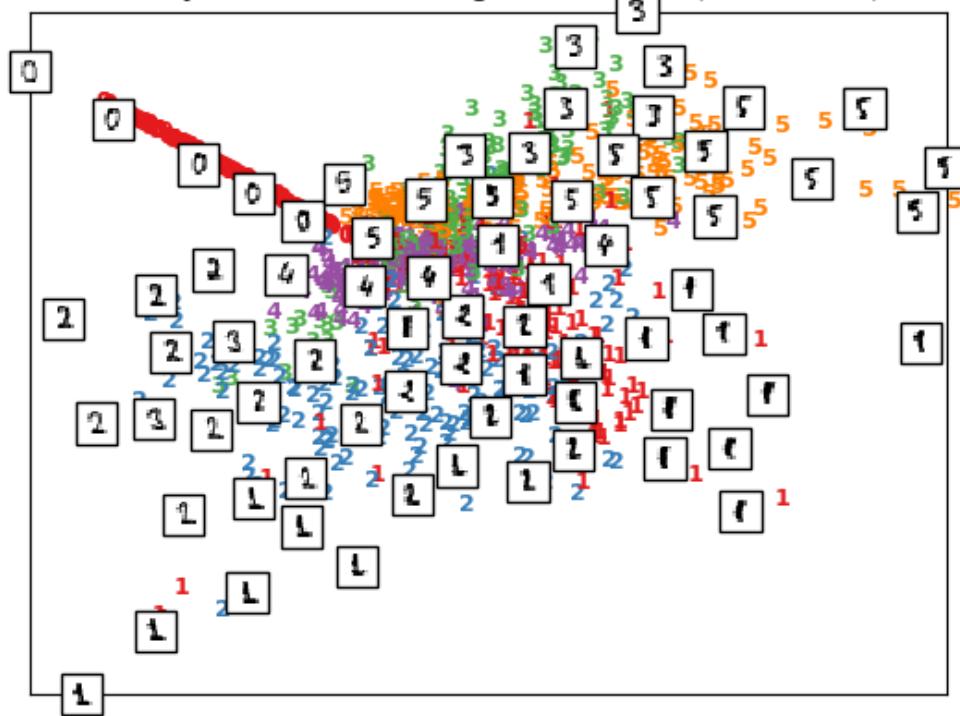
Principal Components projection of the digits (time 0.00s)



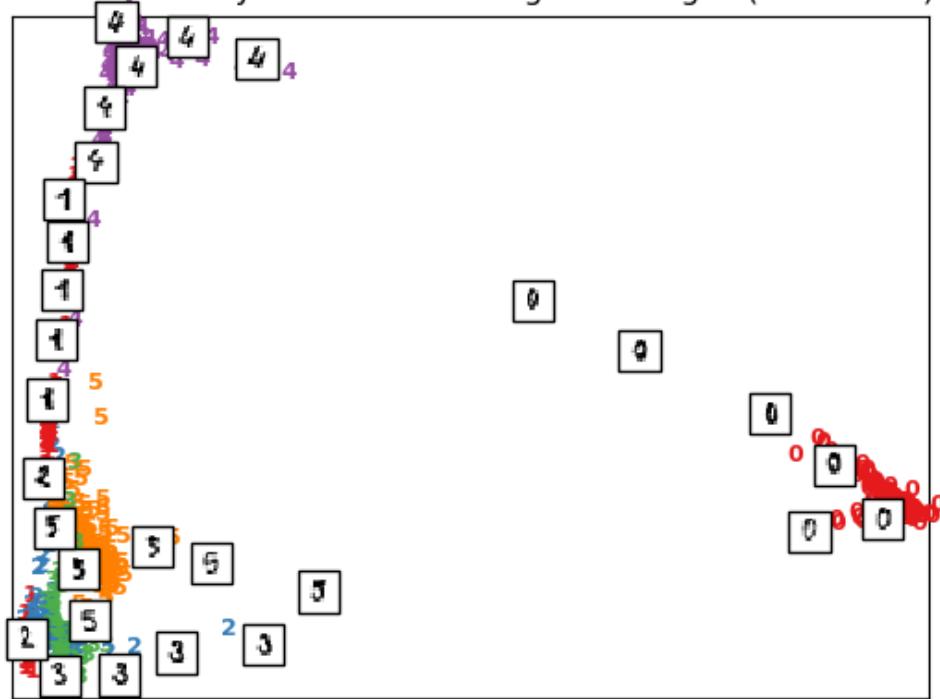




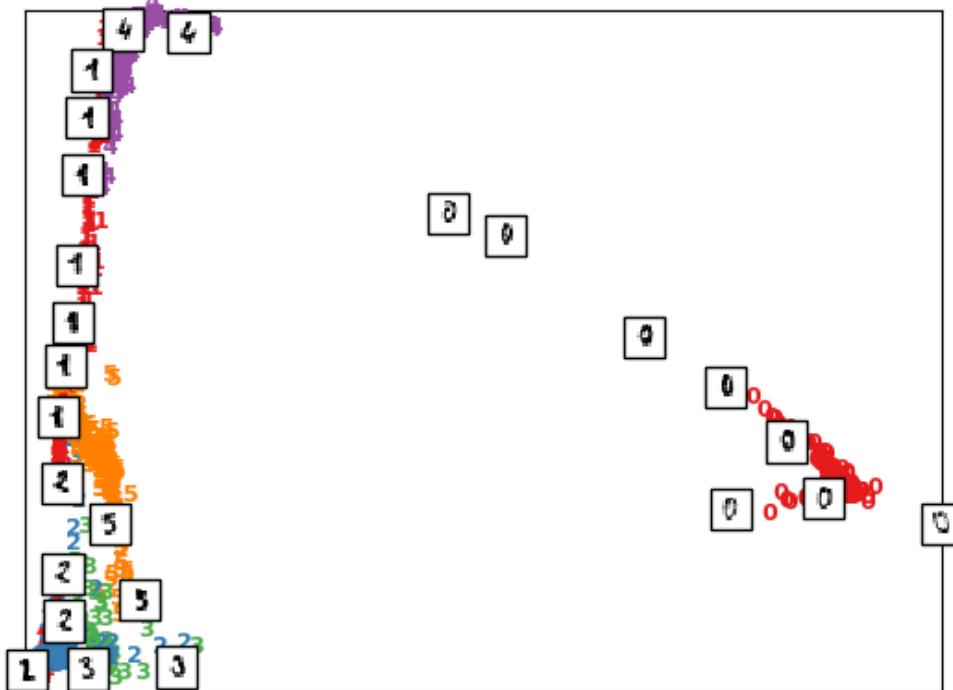
Locally Linear Embedding of the digits (time 0.30s)



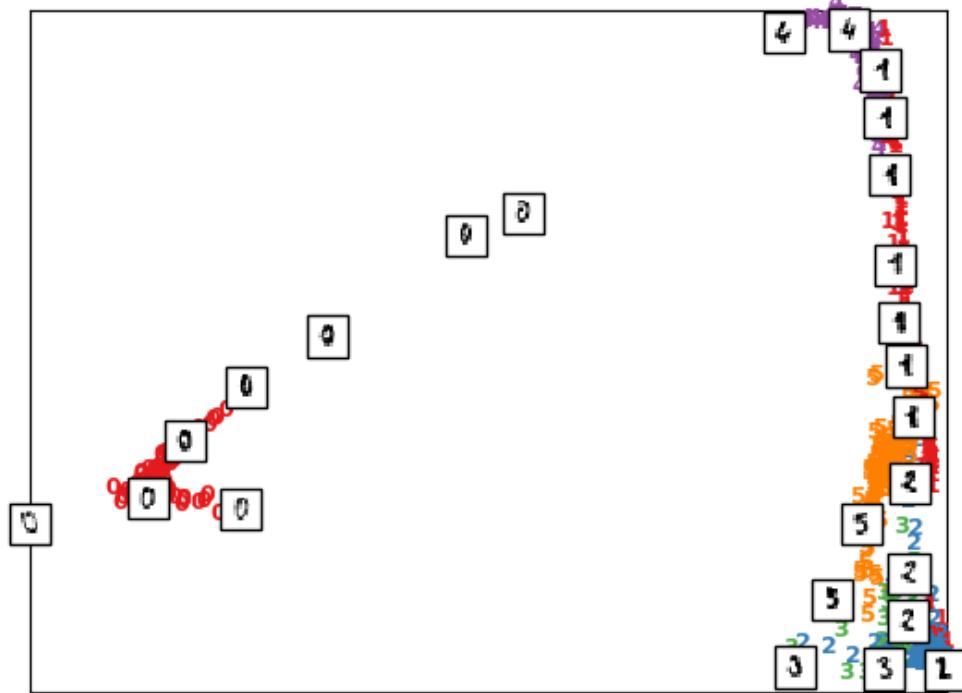
Modified Locally Linear Embedding of the digits (time 0.50s)

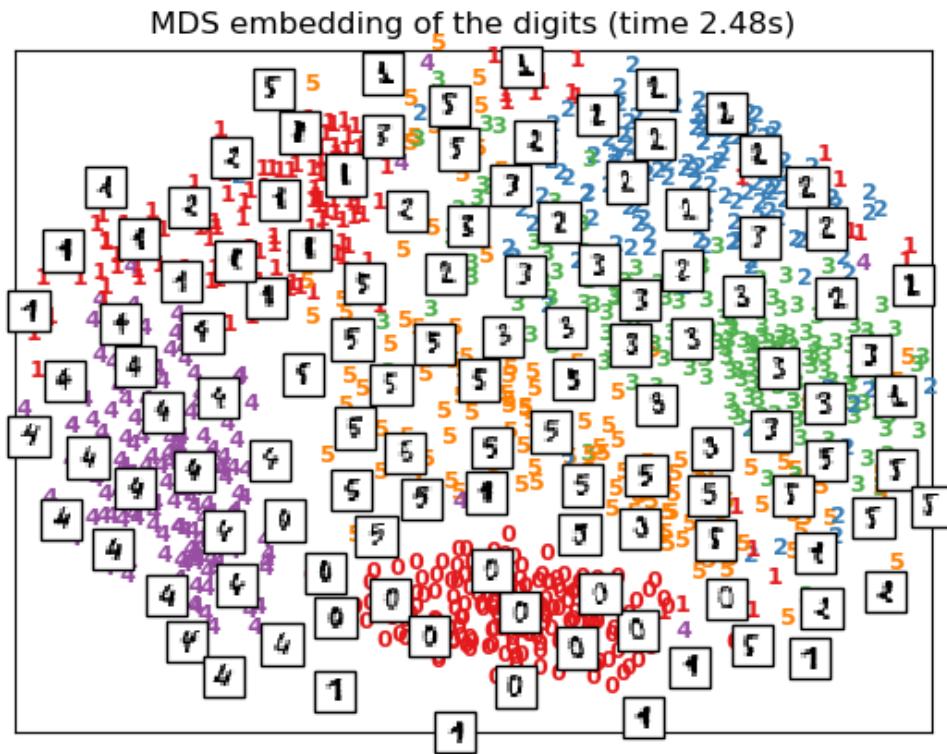


Hessian Locally Linear Embedding of the digits (time 0.62s)

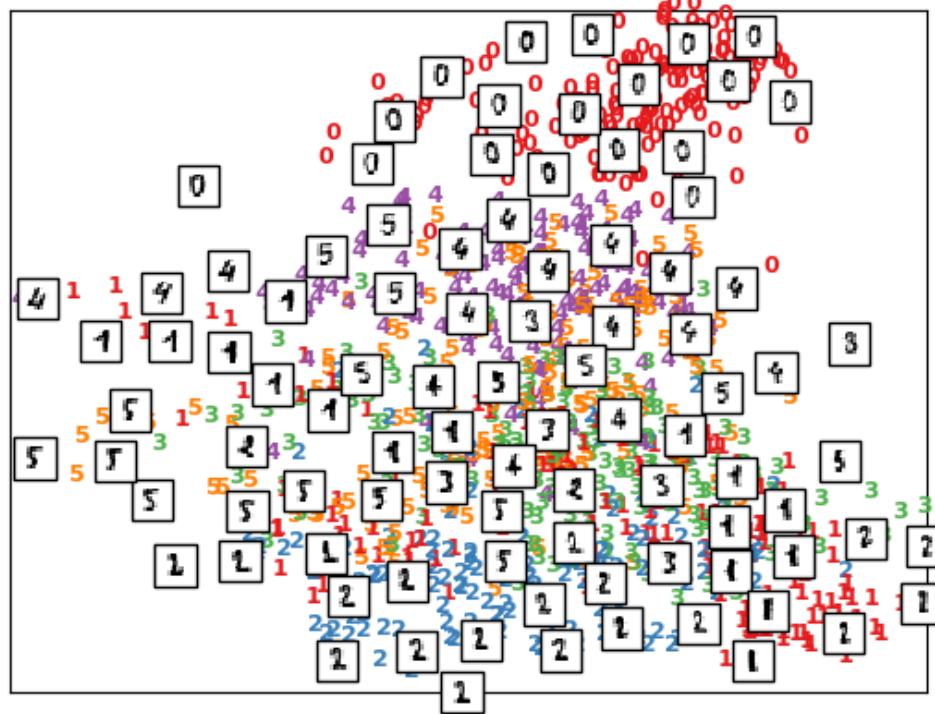


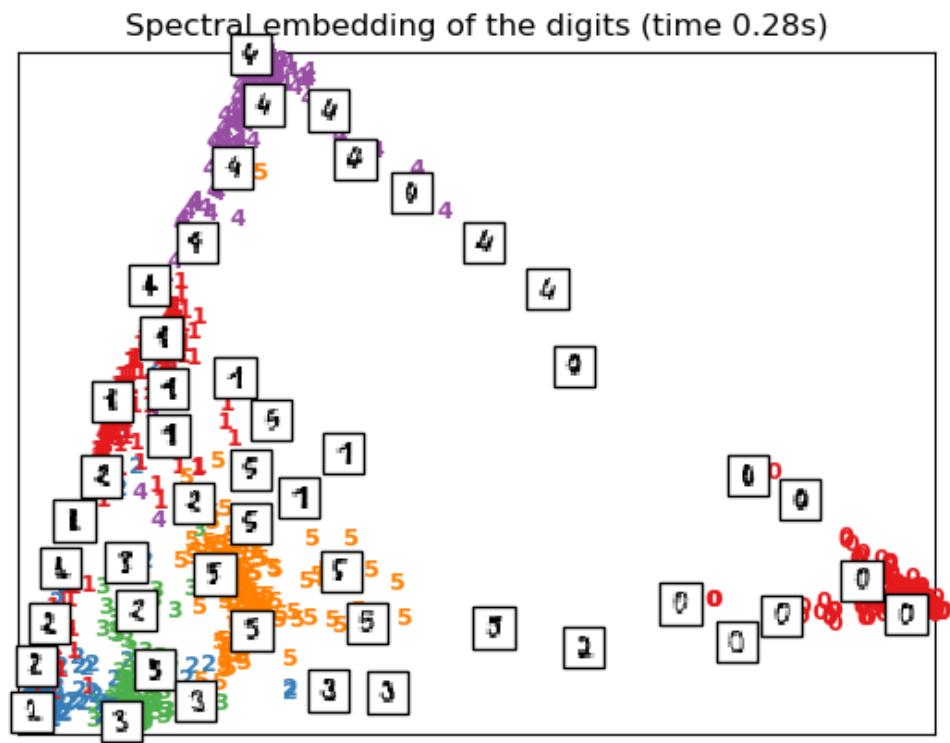
Local Tangent Space Alignment of the digits (time 0.45s)

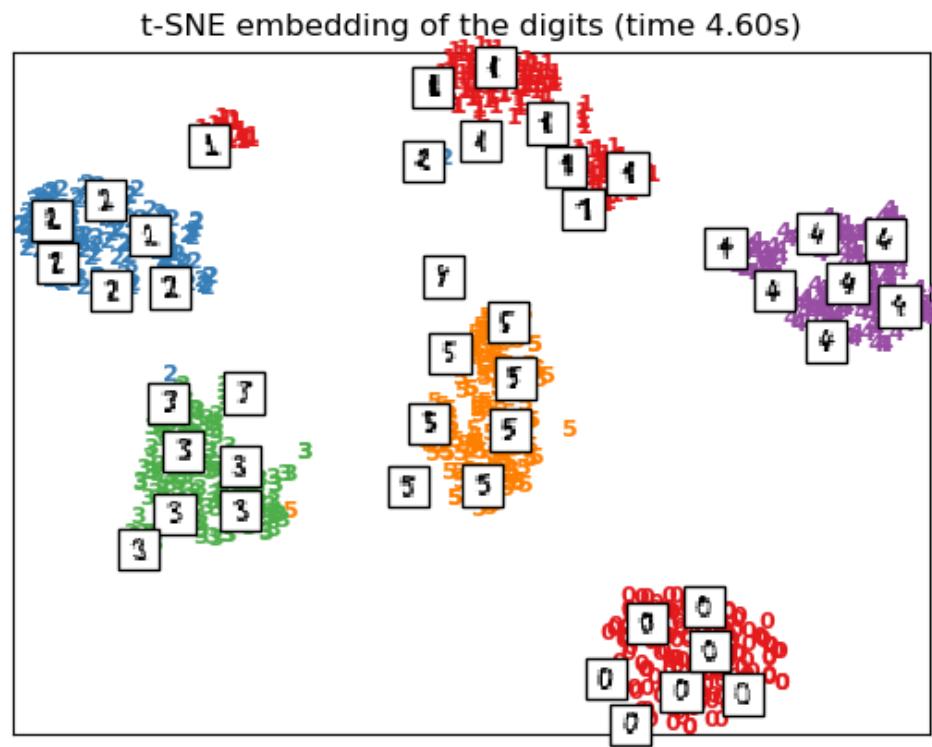


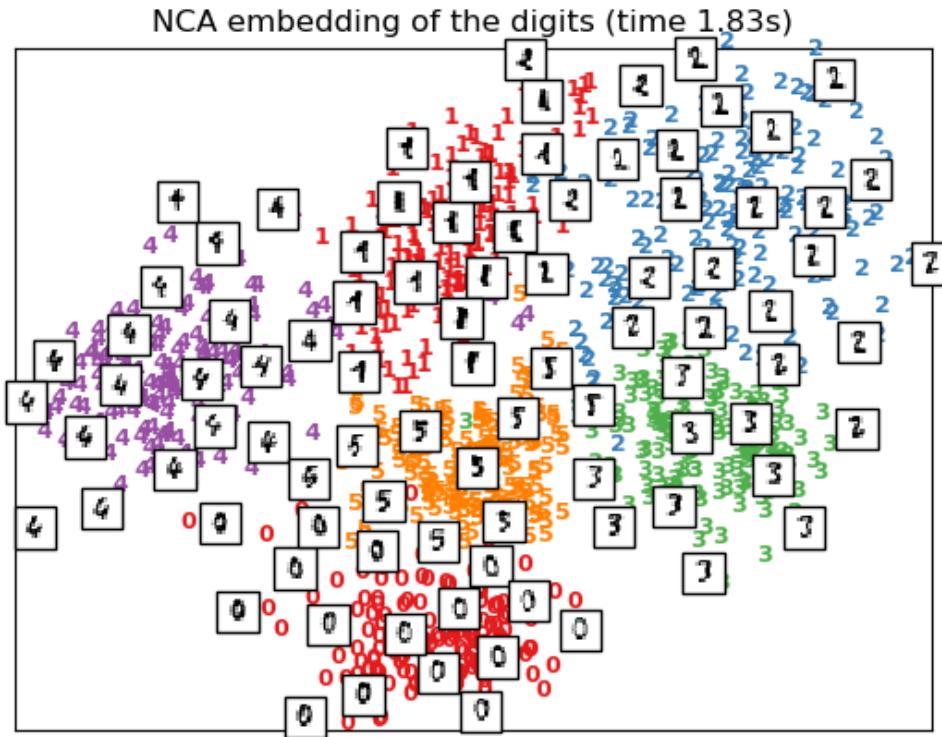


Random forest embedding of the digits (time 0.16s)









Out:

```
Computing random projection
Computing PCA projection
Computing Linear Discriminant Analysis projection
Computing Isomap projection
Done.
Computing LLE embedding
Done. Reconstruction error: 1.63544e-06
Computing modified LLE embedding
Done. Reconstruction error: 0.360652
Computing Hessian LLE embedding
Done. Reconstruction error: 0.212801
Computing LTSA embedding
Done. Reconstruction error: 0.212808
Computing MDS embedding
Done. Stress: 148085982.692961
Computing Totally Random Trees embedding
Computing Spectral embedding
Computing t-SNE embedding
Computing NCA projection
```

```

# Authors: Fabian Pedregosa <fabian.pedregosa@inria.fr>
#          Olivier Grisel <olivier.grisel@ensta.org>
#          Mathieu Blondel <mathieu@mblondel.org>
#          Gael Varoquaux
# License: BSD 3 clause (C) INRIA 2011

print(__doc__)
from time import time

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import offsetbox
from sklearn import (manifold, datasets, decomposition, ensemble,
                     discriminant_analysis, random_projection, neighbors)

digits = datasets.load_digits(n_class=6)
X = digits.data
y = digits.target
n_samples, n_features = X.shape
n_neighbors = 30

# -----
# Scale and visualize the embedding vectors
def plot_embedding(X, title=None):
    x_min, x_max = np.min(X, 0), np.max(X, 0)
    X = (X - x_min) / (x_max - x_min)

    plt.figure()
    ax = plt.subplot(111)
    for i in range(X.shape[0]):
        plt.text(X[i, 0], X[i, 1], str(y[i]),
                 color=plt.cm.Set1(y[i] / 10.),
                 fontdict={'weight': 'bold', 'size': 9})

    if hasattr(offsetbox, 'AnnotationBbox'):
        # only print thumbnails with matplotlib > 1.0
        shown_images = np.array([[1., 1.]]) # just something big
        for i in range(X.shape[0]):
            dist = np.sum((X[i] - shown_images) ** 2, 1)
            if np.min(dist) < 4e-3:
                # don't show points that are too close
                continue
            shown_images = np.r_[shown_images, [X[i]]]
            imagebox = offsetbox.AnnotationBbox(
                offsetbox.OffsetImage(digits.images[i], cmap=plt.cm.gray_r),
                X[i])
            ax.add_artist(imagebox)
    plt.xticks([]), plt.yticks([])
    if title is not None:
        plt.title(title)

# -----
# Plot images of the digits
n_img_per_row = 20
img = np.zeros((10 * n_img_per_row, 10 * n_img_per_row))
for i in range(n_img_per_row):

```

```
ix = 10 * i + 1
for j in range(n_img_per_row):
    iy = 10 * j + 1
    img[ix:ix + 8, iy:iy + 8] = X[i * n_img_per_row + j].reshape((8, 8))

plt.imshow(img, cmap=plt.cm.binary)
plt.xticks([])
plt.yticks([])
plt.title('A selection from the 64-dimensional digits dataset')

# -----
# Random 2D projection using a random unitary matrix
print("Computing random projection")
rp = random_projection.SparseRandomProjection(n_components=2, random_state=42)
X_projected = rp.fit_transform(X)
plot_embedding(X_projected, "Random Projection of the digits")

# -----
# Projection on to the first 2 principal components

print("Computing PCA projection")
t0 = time()
X_pca = decomposition.TruncatedSVD(n_components=2).fit_transform(X)
plot_embedding(X_pca,
               "Principal Components projection of the digits (time %.2fs)" %
               (time() - t0))

# -----
# Projection on to the first 2 linear discriminant components

print("Computing Linear Discriminant Analysis projection")
X2 = X.copy()
X2.flat[::X.shape[1] + 1] += 0.01 # Make X invertible
t0 = time()
X_lda = discriminant_analysis.LinearDiscriminantAnalysis(n_components=2).fit_
       .transform(X2, y)
plot_embedding(X_lda,
               "Linear Discriminant projection of the digits (time %.2fs)" %
               (time() - t0))

# -----
# Isomap projection of the digits dataset
print("Computing Isomap projection")
t0 = time()
X_iso = manifold.Isomap(n_neighbors, n_components=2).fit_transform(X)
print("Done.")
plot_embedding(X_iso,
               "Isomap projection of the digits (time %.2fs)" %
               (time() - t0))

# -----
# Locally linear embedding of the digits dataset
print("Computing LLE embedding")
clf = manifold.LocallyLinearEmbedding(n_neighbors, n_components=2,
```

```

method='standard')

t0 = time()
X_lle = clf.fit_transform(X)
print("Done. Reconstruction error: %g" % clf.reconstruction_error_)
plot_embedding(X_lle,
               "Locally Linear Embedding of the digits (time %.2fs)" %
               (time() - t0))

# -----
# Modified Locally linear embedding of the digits dataset
print("Computing modified LLE embedding")
clf = manifold.LocallyLinearEmbedding(n_neighbors, n_components=2,
                                      method='modified')
t0 = time()
X_mlle = clf.fit_transform(X)
print("Done. Reconstruction error: %g" % clf.reconstruction_error_)
plot_embedding(X_mlle,
               "Modified Locally Linear Embedding of the digits (time %.2fs)" %
               (time() - t0))

# -----
# HLLE embedding of the digits dataset
print("Computing Hessian LLE embedding")
clf = manifold.LocallyLinearEmbedding(n_neighbors, n_components=2,
                                      method='hessian')
t0 = time()
X_hlle = clf.fit_transform(X)
print("Done. Reconstruction error: %g" % clf.reconstruction_error_)
plot_embedding(X_hlle,
               "Hessian Locally Linear Embedding of the digits (time %.2fs)" %
               (time() - t0))

# -----
# LTSA embedding of the digits dataset
print("Computing LTSA embedding")
clf = manifold.LocallyLinearEmbedding(n_neighbors, n_components=2,
                                      method='ltsa')
t0 = time()
X_ltsa = clf.fit_transform(X)
print("Done. Reconstruction error: %g" % clf.reconstruction_error_)
plot_embedding(X_ltsa,
               "Local Tangent Space Alignment of the digits (time %.2fs)" %
               (time() - t0))

# -----
# MDS embedding of the digits dataset
print("Computing MDS embedding")
clf = manifold.MDS(n_components=2, n_init=1, max_iter=100)
t0 = time()
X_mds = clf.fit_transform(X)
print("Done. Stress: %f" % clf.stress_)
plot_embedding(X_mds,
               "MDS embedding of the digits (time %.2fs)" %
               (time() - t0))

```

```

# -----
# Random Trees embedding of the digits dataset
print("Computing Totally Random Trees embedding")
hasher = ensemble.RandomTreesEmbedding(n_estimators=200, random_state=0,
                                         max_depth=5)
t0 = time()
X_transformed = hasher.fit_transform(X)
pca = decomposition.TruncatedSVD(n_components=2)
X_reduced = pca.fit_transform(X_transformed)

plot_embedding(X_reduced,
               "Random forest embedding of the digits (time %.2fs)" % (time() - t0))

# -----
# Spectral embedding of the digits dataset
print("Computing Spectral embedding")
embedder = manifold.SpectralEmbedding(n_components=2, random_state=0,
                                         eigen_solver="arpack")
t0 = time()
X_se = embedder.fit_transform(X)

plot_embedding(X_se,
               "Spectral embedding of the digits (time %.2fs)" % (time() - t0))

# -----
# t-SNE embedding of the digits dataset
print("Computing t-SNE embedding")
tsne = manifold.TSNE(n_components=2, init='pca', random_state=0)
t0 = time()
X_tsne = tsne.fit_transform(X)

plot_embedding(X_tsne,
               "t-SNE embedding of the digits (time %.2fs)" % (time() - t0))

# -----
# NCA projection of the digits dataset
print("Computing NCA projection")
nca = neighbors.NeighborhoodComponentsAnalysis(n_components=2, random_state=0)
t0 = time()
X_nca = nca.fit_transform(X, y)

plot_embedding(X_nca,
               "NCA embedding of the digits (time %.2fs)" % (time() - t0))

plt.show()

```

Total running time of the script: (0 minutes 17.641 seconds)

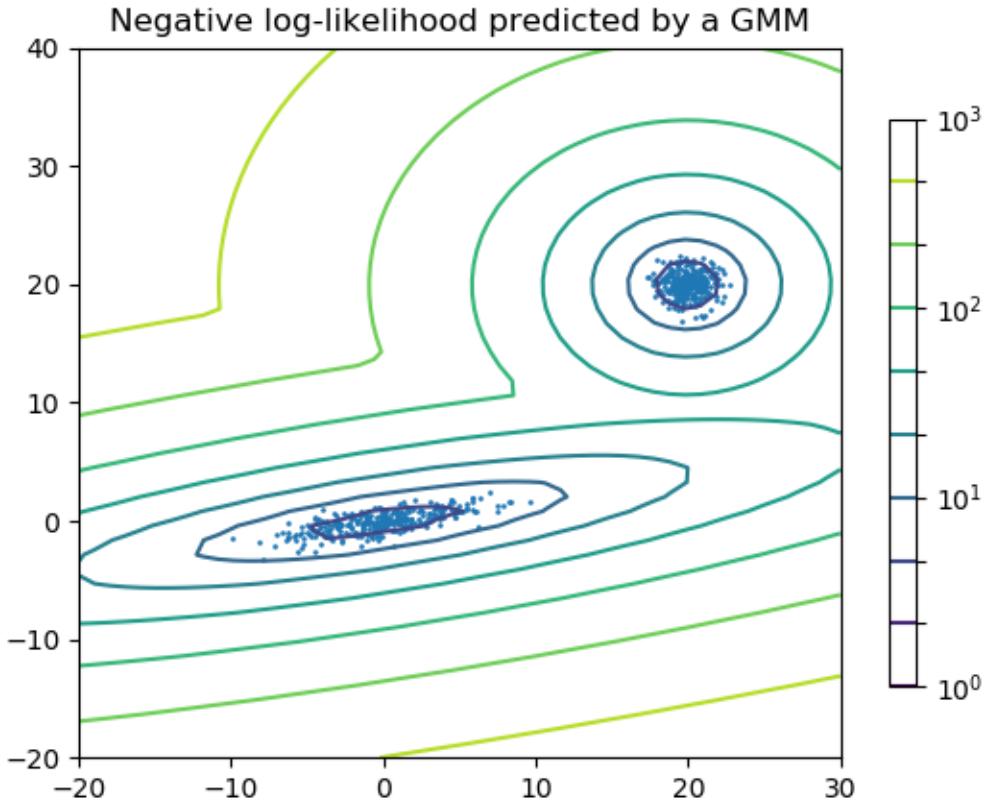
5.20 Gaussian Mixture Models

Examples concerning the `sklearn.mixture` module.

Note: Click [here](#) to download the full example code

5.20.1 Density Estimation for a Gaussian mixture

Plot the density estimation of a mixture of two Gaussians. Data is generated from two Gaussians with different centers and covariance matrices.



```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
from sklearn import mixture

n_samples = 300

# generate random sample, two components
np.random.seed(0)

# generate spherical data centered on (20, 20)
shifted_gaussian = np.random.randn(n_samples, 2) + np.array([20, 20])

# generate zero centered stretched Gaussian data
C = np.array([[0., -0.7], [3.5, .7]])
stretched_gaussian = np.dot(np.random.randn(n_samples, 2), C)
```

```
# concatenate the two datasets into the final training set
X_train = np.vstack([shifted_gaussian, stretched_gaussian])

# fit a Gaussian Mixture Model with two components
clf = mixture.GaussianMixture(n_components=2, covariance_type='full')
clf.fit(X_train)

# display predicted scores by the model as a contour plot
x = np.linspace(-20., 30.)
y = np.linspace(-20., 40.)
X, Y = np.meshgrid(x, y)
XX = np.array([X.ravel(), Y.ravel()]).T
Z = -clf.score_samples(XX)
Z = Z.reshape(X.shape)

CS = plt.contour(X, Y, Z, norm=LogNorm(vmin=1.0, vmax=1000.0),
                 levels=np.logspace(0, 3, 10))
CB = plt.colorbar(CS, shrink=0.8, extend='both')
plt.scatter(X_train[:, 0], X_train[:, 1], .8)

plt.title('Negative log-likelihood predicted by a GMM')
plt.axis('tight')
plt.show()
```

Total running time of the script: (0 minutes 0.040 seconds)

Note: Click [here](#) to download the full example code

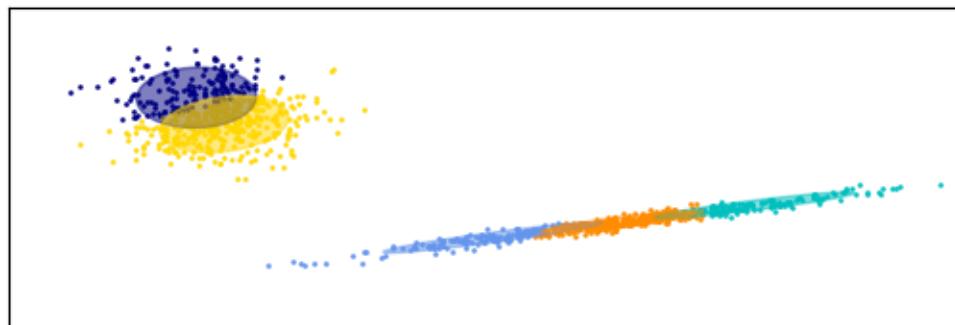
5.20.2 Gaussian Mixture Model Ellipsoids

Plot the confidence ellipsoids of a mixture of two Gaussians obtained with Expectation Maximisation (GaussianMixture class) and Variational Inference (BayesianGaussianMixture class models with a Dirichlet process prior).

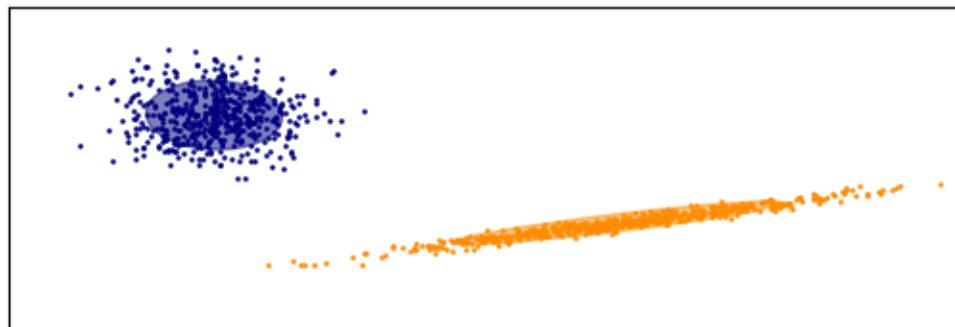
Both models have access to five components with which to fit the data. Note that the Expectation Maximisation model will necessarily use all five components while the Variational Inference model will effectively only use as many as are needed for a good fit. Here we can see that the Expectation Maximisation model splits some components arbitrarily, because it is trying to fit too many components, while the Dirichlet Process model adapts its number of states automatically.

This example doesn't show it, as we're in a low-dimensional space, but another advantage of the Dirichlet process model is that it can fit full covariance matrices effectively even when there are less examples per cluster than there are dimensions in the data, due to regularization properties of the inference algorithm.

Gaussian Mixture



Bayesian Gaussian Mixture with a Dirichlet process prior



```

import itertools

import numpy as np
from scipy import linalg
import matplotlib.pyplot as plt
import matplotlib as mpl

from sklearn import mixture

color_iter = itertools.cycle(['navy', 'c', 'cornflowerblue', 'gold',
                             'darkorange'])

def plot_results(X, Y_, means, covariances, index, title):
    splot = plt.subplot(2, 1, 1 + index)
    for i, (mean, covar, color) in enumerate(zip(
            means, covariances, color_iter)):
        v, w = linalg.eigh(covar)
        v = 2. * np.sqrt(2.) * np.sqrt(v)
        u = w[0] / linalg.norm(w[0])
        # as the DP will not use every component it has access to
        # unless it needs it, we shouldn't plot the redundant
        # components.
        if not np.any(Y_ == i):
            continue
        plt.scatter(X[Y_ == i, 0], X[Y_ == i, 1], .8, color=color)

    plt.title(title)
    plt.xticks(())
    plt.yticks(())

plt.show()

```

```
# Plot an ellipse to show the Gaussian component
angle = np.arctan(u[1] / u[0])
angle = 180. * angle / np.pi # convert to degrees
ell = mpl.patches.Ellipse(mean, v[0], v[1], 180. + angle, color=color)
ell.set_clip_box(splot.bbox)
ell.set_alpha(0.5)
splot.add_artist(ell)

plt.xlim(-9., 5.)
plt.ylim(-3., 6.)
plt.xticks(())
plt.yticks(())
plt.title(title)

# Number of samples per component
n_samples = 500

# Generate random sample, two components
np.random.seed(0)
C = np.array([[0., -0.1], [1.7, .4]])
X = np.r_[np.dot(np.random.randn(n_samples, 2), C),
          .7 * np.random.randn(n_samples, 2) + np.array([-6, 3])]

# Fit a Gaussian mixture with EM using five components
gmm = mixture.GaussianMixture(n_components=5, covariance_type='full').fit(X)
plot_results(X, gmm.predict(X), gmm.means_, gmm.covariances_, 0,
             'Gaussian Mixture')

# Fit a Dirichlet process Gaussian mixture using five components
dpgmm = mixture.BayesianGaussianMixture(n_components=5,
                                         covariance_type='full').fit(X)
plot_results(X, dpgmm.predict(X), dpgmm.means_, dpgmm.covariances_, 1,
             'Bayesian Gaussian Mixture with a Dirichlet process prior')

plt.show()
```

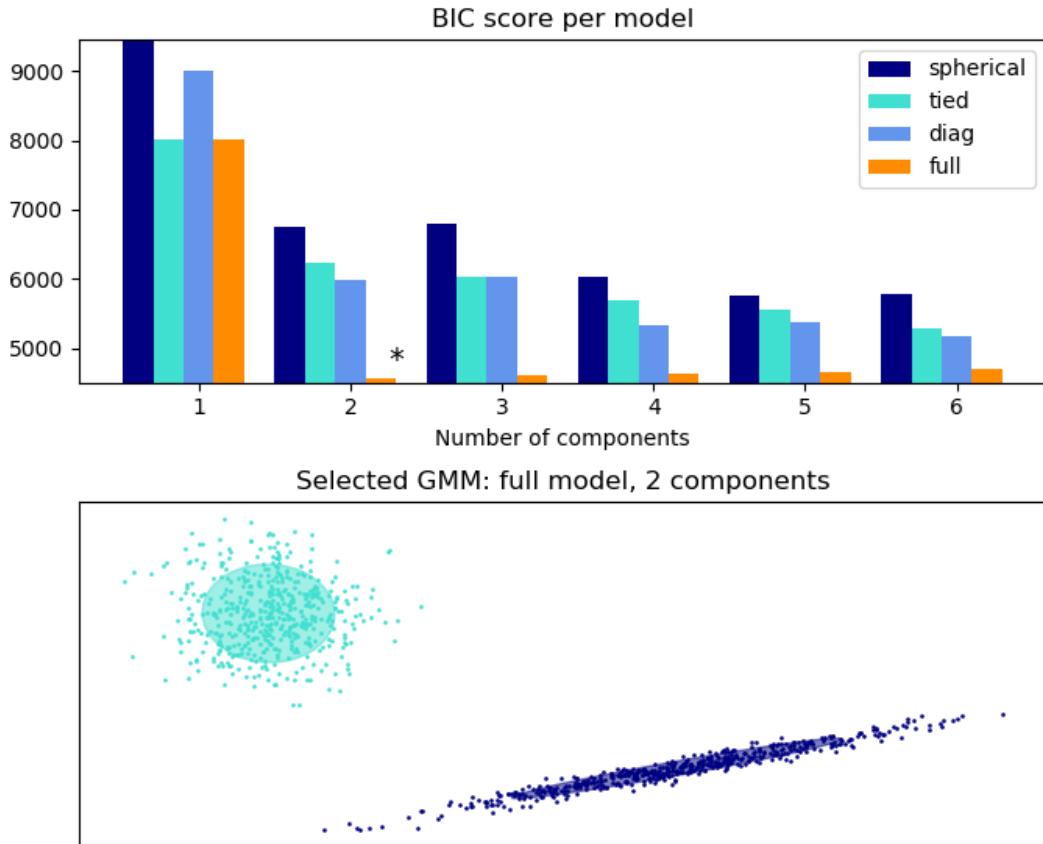
Total running time of the script: (0 minutes 0.138 seconds)

Note: Click [here](#) to download the full example code

5.20.3 Gaussian Mixture Model Selection

This example shows that model selection can be performed with Gaussian Mixture Models using information-theoretic criteria (BIC). Model selection concerns both the covariance type and the number of components in the model. In that case, AIC also provides the right result (not shown to save time), but BIC is better suited if the problem is to identify the right model. Unlike Bayesian procedures, such inferences are prior-free.

In that case, the model with 2 components and full covariance (which corresponds to the true generative model) is selected.



```

import numpy as np
import itertools

from scipy import linalg
import matplotlib.pyplot as plt
import matplotlib as mpl

from sklearn import mixture

print(__doc__)

# Number of samples per component
n_samples = 500

# Generate random sample, two components
np.random.seed(0)
C = np.array([[0., -0.1], [1.7, .4]])
X = np.r_[np.dot(np.random.randn(n_samples, 2), C),
          .7 * np.random.randn(n_samples, 2) + np.array([-6, 3])]

lowest_bic = np.infty
bic = []
n_components_range = range(1, 7)
cv_types = ['spherical', 'tied', 'diag', 'full']
for cv_type in cv_types:
    for n_components in n_components_range:
        # Fit a Gaussian Mixture Model
        gmm = mixture.GaussianMixture(n_components=n_components,
                                      covariance_type=cv_type)
        gmm.fit(X)
        bic.append(gmm.bic(X))

lowest_bic = min(bic)
bic.index(lowest_bic)
cv_type
n_components

```

```

# Fit a Gaussian mixture with EM
gmm = mixture.GaussianMixture(n_components=n_components,
                               covariance_type=cv_type)
gmm.fit(X)
bic.append(gmm.bic(X))
if bic[-1] < lowest_bic:
    lowest_bic = bic[-1]
    best_gmm = gmm

bic = np.array(bic)
color_iter = itertools.cycle(['navy', 'turquoise', 'cornflowerblue',
                             'darkorange'])
clf = best_gmm
bars = []

# Plot the BIC scores
plt.figure(figsize=(8, 6))
spl = plt.subplot(2, 1, 1)
for i, (cv_type, color) in enumerate(zip(cv_types, color_iter)):
    xpos = np.array(n_components_range) + .2 * (i - 2)
    bars.append(plt.bar(xpos, bic[i * len(n_components_range):
                                    (i + 1) * len(n_components_range)],
                         width=.2, color=color))
plt.xticks(n_components_range)
plt.ylim([bic.min() * 1.01 - .01 * bic.max(), bic.max()])
plt.title('BIC score per model')
xpos = np.mod(bic.argmax(), len(n_components_range)) + .65 + \
    .2 * np.floor(bic.argmax() / len(n_components_range))
plt.text(xpos, bic.min() * 0.97 + .03 * bic.max(), '*', fontsize=14)
spl.set_xlabel('Number of components')
spl.legend([b[0] for b in bars], cv_types)

# Plot the winner
splot = plt.subplot(2, 1, 2)
Y_ = clf.predict(X)
for i, (mean, cov, color) in enumerate(zip(clf.means_, clf.covariances_,
                                            color_iter)):
    v, w = linalg.eigh(cov)
    if not np.any(Y_ == i):
        continue
    plt.scatter(X[Y_ == i, 0], X[Y_ == i, 1], .8, color=color)

    # Plot an ellipse to show the Gaussian component
    angle = np.arctan2(w[0][1], w[0][0])
    angle = 180. * angle / np.pi # convert to degrees
    v = 2. * np.sqrt(2.) * np.sqrt(v)
    ell = mpl.patches.Ellipse(mean, v[0], v[1], 180. + angle, color=color)
    ell.set_clip_box(splot.bbox)
    ell.set_alpha(.5)
    splot.add_artist(ell)

plt.xticks(())
plt.yticks(())
plt.title('Selected GMM: full model, 2 components')
plt.subplots_adjust(hspace=.35, bottom=.02)
plt.show()

```

Total running time of the script: (0 minutes 0.207 seconds)

Note: Click [here](#) to download the full example code

5.20.4 GMM covariances

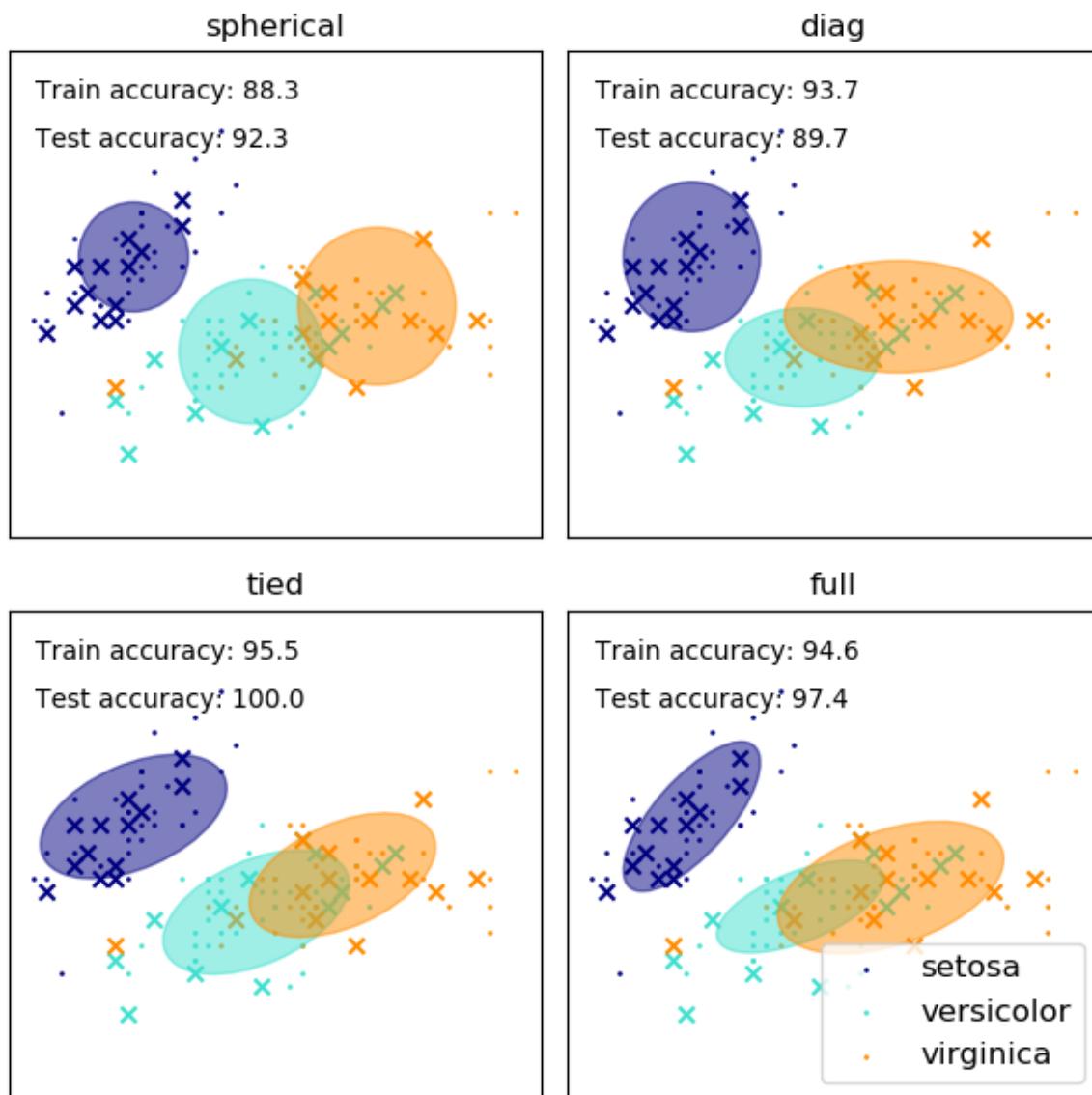
Demonstration of several covariances types for Gaussian mixture models.

See [Gaussian mixture models](#) for more information on the estimator.

Although GMM are often used for clustering, we can compare the obtained clusters with the actual classes from the dataset. We initialize the means of the Gaussians with the means of the classes from the training set to make this comparison valid.

We plot predicted labels on both training and held out test data using a variety of GMM covariance types on the iris dataset. We compare GMMs with spherical, diagonal, full, and tied covariance matrices in increasing order of performance. Although one would expect full covariance to perform best in general, it is prone to overfitting on small datasets and does not generalize well to held out test data.

On the plots, train data is shown as dots, while test data is shown as crosses. The iris dataset is four-dimensional. Only the first two dimensions are shown here, and thus some points are separated in other dimensions.



```
# Author: Ron Weiss <ronweiss@gmail.com>, Gael Varoquaux
# Modified by Thierry Guillemot <thierry.guillemot.work@gmail.com>
# License: BSD 3 clause

import matplotlib as mpl
import matplotlib.pyplot as plt

import numpy as np

from sklearn import datasets
from sklearn.mixture import GaussianMixture
from sklearn.model_selection import StratifiedKFold

print(__doc__)

colors = ['navy', 'turquoise', 'darkorange']
```

```

def make_ellipses(gmm, ax):
    for n, color in enumerate(colors):
        if gmm.covariance_type == 'full':
            covariances = gmm.covariances_[n][:2, :2]
        elif gmm.covariance_type == 'tied':
            covariances = gmm.covariances_[:, :, 2]
        elif gmm.covariance_type == 'diag':
            covariances = np.diag(gmm.covariances_[n][:2])
        elif gmm.covariance_type == 'spherical':
            covariances = np.eye(gmm.means_.shape[1]) * gmm.covariances_[n]
    v, w = np.linalg.eigh(covariances)
    u = w[0] / np.linalg.norm(w[0])
    angle = np.arctan2(u[1], u[0])
    angle = 180 * angle / np.pi # convert to degrees
    v = 2. * np.sqrt(2.) * np.sqrt(v)
    ell = mpl.patches.Ellipse(gmm.means_[n, :2], v[0], v[1],
                               180 + angle, color=color)
    ell.set_clip_box(ax.bbox)
    ell.set_alpha(0.5)
    ax.add_artist(ell)
    ax.set_aspect('equal', 'datalim')

iris = datasets.load_iris()

# Break up the dataset into non-overlapping training (75%) and testing
# (25%) sets.
skf = StratifiedKFold(n_splits=4)
# Only take the first fold.
train_index, test_index = next(iter(skf.split(iris.data, iris.target)))

X_train = iris.data[train_index]
y_train = iris.target[train_index]
X_test = iris.data[test_index]
y_test = iris.target[test_index]

n_classes = len(np.unique(y_train))

# Try GMMs using different types of covariances.
estimators = {cov_type: GaussianMixture(n_components=n_classes,
                                         covariance_type=cov_type, max_iter=20, random_state=0)
              for cov_type in ['spherical', 'diag', 'tied', 'full']}
n_estimators = len(estimators)

plt.figure(figsize=(3 * n_estimators // 2, 6))
plt.subplots_adjust(bottom=.01, top=0.95, hspace=.15, wspace=.05,
                    left=.01, right=.99)

for index, (name, estimator) in enumerate(estimators.items()):
    # Since we have class labels for the training data, we can
    # initialize the GMM parameters in a supervised manner.
    estimator.means_init = np.array([X_train[y_train == i].mean(axis=0)
                                     for i in range(n_classes)])

    # Train the other parameters using the EM algorithm.
    estimator.fit(X_train)

```

```
h = plt.subplot(2, n_estimators // 2, index + 1)
make_ellipses(estimator, h)

for n, color in enumerate(colors):
    data = iris.data[iris.target == n]
    plt.scatter(data[:, 0], data[:, 1], s=0.8, color=color,
                label=iris.target_names[n])
# Plot the test data with crosses
for n, color in enumerate(colors):
    data = X_test[y_test == n]
    plt.scatter(data[:, 0], data[:, 1], marker='x', color=color)

y_train_pred = estimator.predict(X_train)
train_accuracy = np.mean(y_train_pred.ravel() == y_train.ravel()) * 100
plt.text(0.05, 0.9, 'Train accuracy: %.1f' % train_accuracy,
         transform=h.transAxes)

y_test_pred = estimator.predict(X_test)
test_accuracy = np.mean(y_test_pred.ravel() == y_test.ravel()) * 100
plt.text(0.05, 0.8, 'Test accuracy: %.1f' % test_accuracy,
         transform=h.transAxes)

plt.xticks(())
plt.yticks(())
plt.title(name)

plt.legend(scatterpoints=1, loc='lower right', prop=dict(size=12))

plt.show()
```

Total running time of the script: (0 minutes 0.097 seconds)

Note: Click [here](#) to download the full example code

5.20.5 Gaussian Mixture Model Sine Curve

This example demonstrates the behavior of Gaussian mixture models fit on data that was not sampled from a mixture of Gaussian random variables. The dataset is formed by 100 points loosely spaced following a noisy sine curve. There is therefore no ground truth value for the number of Gaussian components.

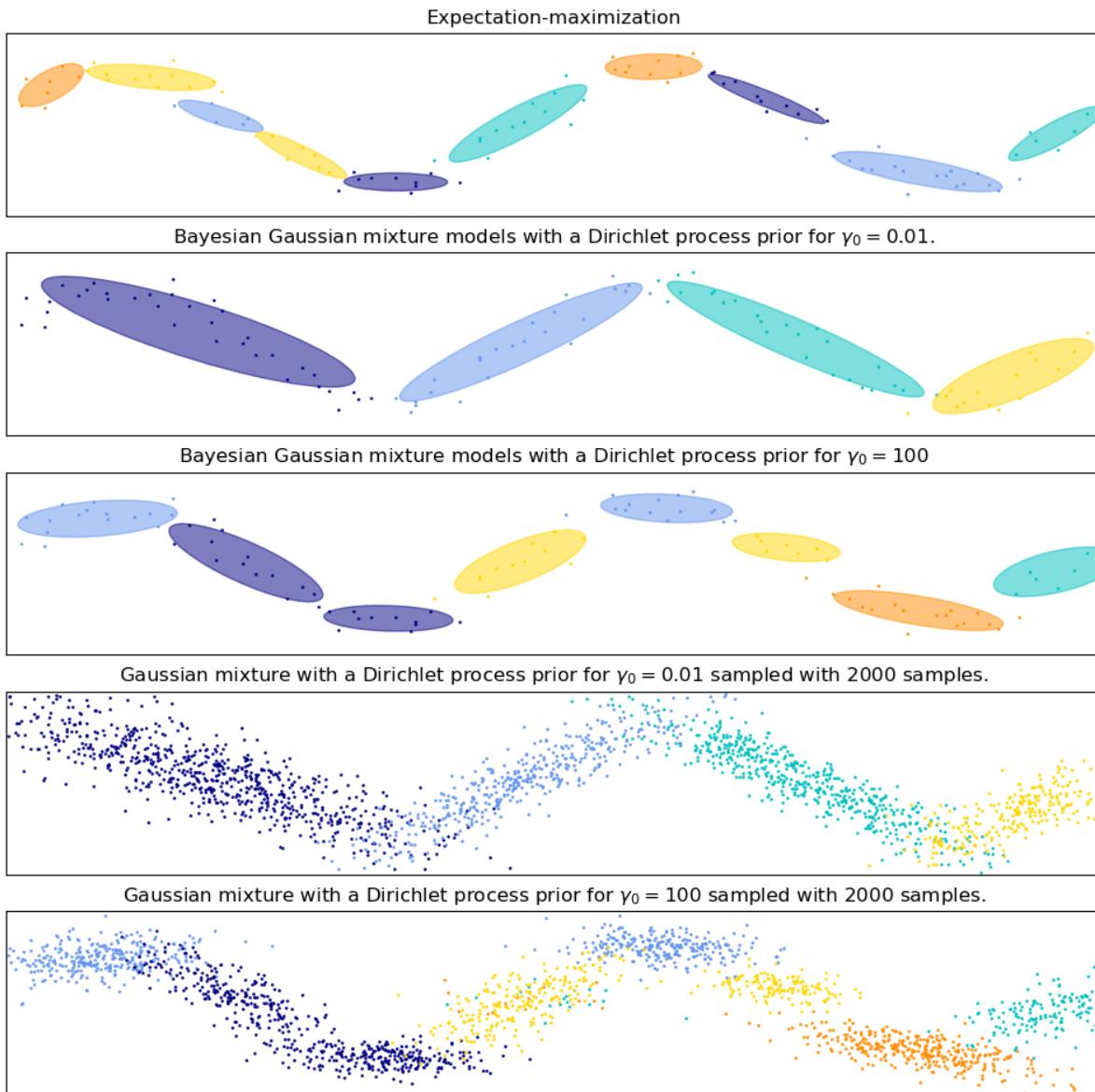
The first model is a classical Gaussian Mixture Model with 10 components fit with the Expectation-Maximization algorithm.

The second model is a Bayesian Gaussian Mixture Model with a Dirichlet process prior fit with variational inference. The low value of the concentration prior makes the model favor a lower number of active components. This models “decides” to focus its modeling power on the big picture of the structure of the dataset: groups of points with alternating directions modeled by non-diagonal covariance matrices. Those alternating directions roughly capture the alternating nature of the original sine signal.

The third model is also a Bayesian Gaussian mixture model with a Dirichlet process prior but this time the value of the concentration prior is higher giving the model more liberty to model the fine-grained structure of the data. The result is a mixture with a larger number of active components that is similar to the first model where we arbitrarily decided to fix the number of components to 10.

Which model is the best is a matter of subjective judgement: do we want to favor models that only capture the big picture to summarize and explain most of the structure of the data while ignoring the details or do we prefer models that closely follow the high density regions of the signal?

The last two panels show how we can sample from the last two models. The resulting samples distributions do not look exactly like the original data distribution. The difference primarily stems from the approximation error we made by using a model that assumes that the data was generated by a finite number of Gaussian components instead of a continuous noisy sine curve.



```
import itertools
import numpy as np
from scipy import linalg
import matplotlib.pyplot as plt
import matplotlib as mpl
```

```

from sklearn import mixture

print(__doc__)

color_iter = itertools.cycle(['navy', 'c', 'cornflowerblue', 'gold',
                             'darkorange'])

def plot_results(X, Y, means, covariances, index, title):
    splot = plt.subplot(5, 1, 1 + index)
    for i, (mean, covar, color) in enumerate(zip(
        means, covariances, color_iter)):
        v, w = linalg.eigh(covar)
        v = 2. * np.sqrt(2.) * np.sqrt(v)
        u = w[0] / linalg.norm(w[0])
        # as the DP will not use every component it has access to
        # unless it needs it, we shouldn't plot the redundant
        # components.
        if not np.any(Y == i):
            continue
        plt.scatter(X[Y == i, 0], X[Y == i, 1], .8, color=color)

        # Plot an ellipse to show the Gaussian component
        angle = np.arctan(u[1] / u[0])
        angle = 180. * angle / np.pi # convert to degrees
        ell = mpl.patches.Ellipse(mean[0], v[1], 180. + angle, color=color)
        ell.set_clip_box(splot.bbox)
        ell.set_alpha(0.5)
        splot.add_artist(ell)

    plt.xlim(-6., 4. * np.pi - 6.)
    plt.ylim(-5., 5.)
    plt.title(title)
    plt.xticks(())
    plt.yticks(())

def plot_samples(X, Y, n_components, index, title):
    plt.subplot(5, 1, 4 + index)
    for i, color in zip(range(n_components), color_iter):
        # as the DP will not use every component it has access to
        # unless it needs it, we shouldn't plot the redundant
        # components.
        if not np.any(Y == i):
            continue
        plt.scatter(X[Y == i, 0], X[Y == i, 1], .8, color=color)

    plt.xlim(-6., 4. * np.pi - 6.)
    plt.ylim(-5., 5.)
    plt.title(title)
    plt.xticks(())
    plt.yticks(())

# Parameters
n_samples = 100

```

```

# Generate random sample following a sine curve
np.random.seed(0)
X = np.zeros((n_samples, 2))
step = 4. * np.pi / n_samples

for i in range(X.shape[0]):
    x = i * step - 6.
    X[i, 0] = x + np.random.normal(0, 0.1)
    X[i, 1] = 3. * (np.sin(x) + np.random.normal(0, .2))

plt.figure(figsize=(10, 10))
plt.subplots_adjust(bottom=.04, top=0.95, hspace=.2, wspace=.05,
                    left=.03, right=.97)

# Fit a Gaussian mixture with EM using ten components
gmm = mixture.GaussianMixture(n_components=10, covariance_type='full',
                               max_iter=100).fit(X)
plot_results(X, gmm.predict(X), gmm.means_, gmm.covariances_, 0,
             'Expectation-maximization')

dpgmm = mixture.BayesianGaussianMixture(
    n_components=10, covariance_type='full', weight_concentration_prior=1e-2,
    weight_concentration_prior_type='dirichlet_process',
    mean_precision_prior=1e-2, covariance_prior=1e0 * np.eye(2),
    init_params="random", max_iter=100, random_state=2).fit(X)
plot_results(X, dpgmm.predict(X), dpgmm.means_, dpgmm.covariances_, 1,
             "Bayesian Gaussian mixture models with a Dirichlet process prior "
             r"for $\gamma_0=0.01$")

X_s, y_s = dpgmm.sample(n_samples=2000)
plot_samples(X_s, y_s, dpgmm.n_components, 0,
             "Gaussian mixture with a Dirichlet process prior "
             r"for $\gamma_0=0.01$ sampled with $2000$ samples.")

dpgmm = mixture.BayesianGaussianMixture(
    n_components=10, covariance_type='full', weight_concentration_prior=1e+2,
    weight_concentration_prior_type='dirichlet_process',
    mean_precision_prior=1e-2, covariance_prior=1e0 * np.eye(2),
    init_params="kmeans", max_iter=100, random_state=2).fit(X)
plot_results(X, dpgmm.predict(X), dpgmm.means_, dpgmm.covariances_, 2,
             "Bayesian Gaussian mixture models with a Dirichlet process prior "
             r"for $\gamma_0=100$")

X_s, y_s = dpgmm.sample(n_samples=2000)
plot_samples(X_s, y_s, dpgmm.n_components, 1,
             "Gaussian mixture with a Dirichlet process prior "
             r"for $\gamma_0=100$ sampled with $2000$ samples.")

plt.show()

```

Total running time of the script: (0 minutes 0.301 seconds)

Note: Click [here](#) to download the full example code

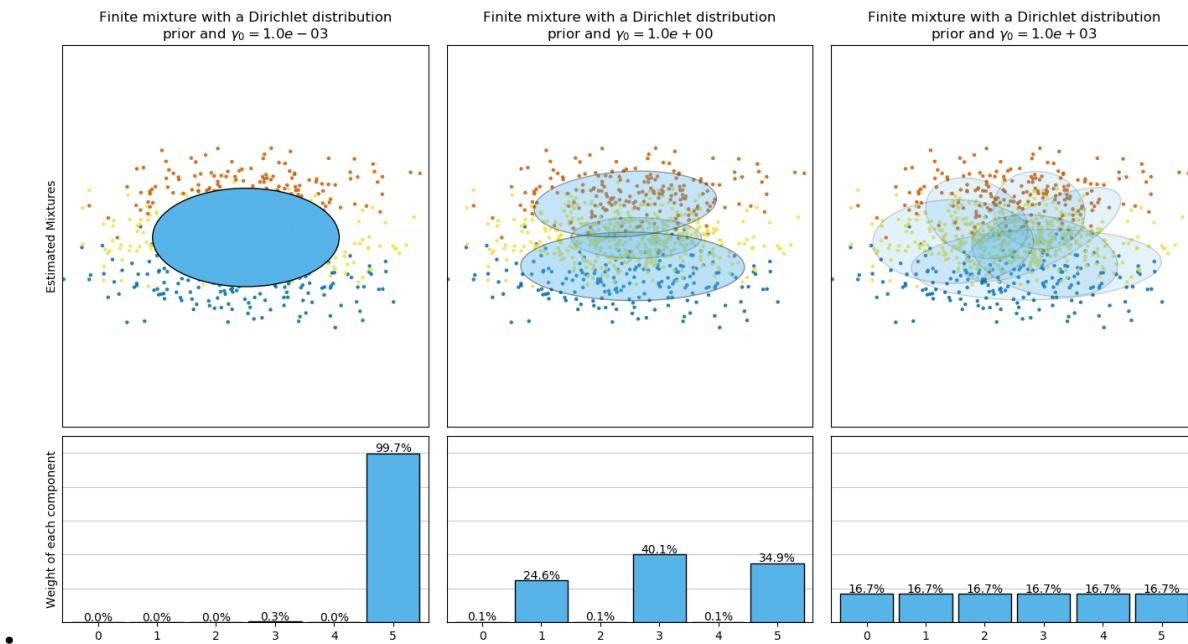
5.20.6 Concentration Prior Type Analysis of Variation Bayesian Gaussian Mixture

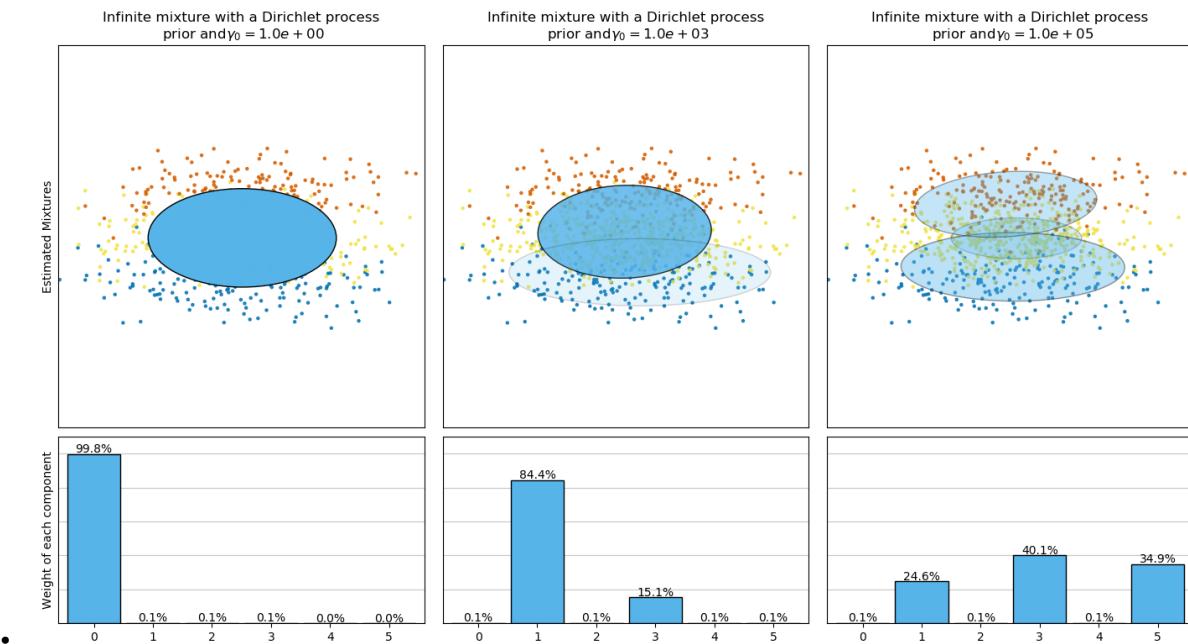
This example plots the ellipsoids obtained from a toy dataset (mixture of three Gaussians) fitted by the `BayesianGaussianMixture` class models with a Dirichlet distribution prior (`weight_concentration_prior_type='dirichlet_distribution'`) and a Dirichlet process prior (`weight_concentration_prior_type='dirichlet_process'`). On each figure, we plot the results for three different values of the weight concentration prior.

The `BayesianGaussianMixture` class can adapt its number of mixture components automatically. The parameter `weight_concentration_prior` has a direct link with the resulting number of components with non-zero weights. Specifying a low value for the concentration prior will make the model put most of the weight on few components set the remaining components weights very close to zero. High values of the concentration prior will allow a larger number of components to be active in the mixture.

The Dirichlet process prior allows to define an infinite number of components and automatically selects the correct number of components: it activates a component only if it is necessary.

On the contrary the classical finite mixture model with a Dirichlet distribution prior will favor more uniformly weighted components and therefore tends to divide natural clusters into unnecessary sub-components.





```
# Author: Thierry Guillemot <thierry.guillemot.work@gmail.com>
# License: BSD 3 clause

import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from sklearn.mixture import BayesianGaussianMixture

print(__doc__)

def plot_ellipses(ax, weights, means, covars):
    for n in range(means.shape[0]):
        eig_vals, eig_vecs = np.linalg.eigh(covars[n])
        unit_eig_vec = eig_vecs[0] / np.linalg.norm(eig_vecs[0])
        angle = np.arctan2(unit_eig_vec[1], unit_eig_vec[0])
        # Ellipse needs degrees
        angle = 180 * angle / np.pi
        # eigenvector normalization
        eig_vals = 2 * np.sqrt(2) * np.sqrt(eig_vals)
        ell = mpl.patches.Ellipse(means[n], eig_vals[0], eig_vals[1],
                                  180 + angle, edgecolor='black')
        ell.set_clip_box(ax.bbox)
        ell.set_alpha(weights[n])
        ell.set_facecolor('#56B4E9')
        ax.add_artist(ell)

def plot_results(ax1, ax2, estimator, X, y, title, plot_title=False):
    ax1.set_title(title)
    ax1.scatter(X[:, 0], X[:, 1], s=5, marker='o', color=colors[y], alpha=0.8)
    ax1.set_xlim(-2., 2.)
    ax1.set_ylim(-3., 3.)
```

```

ax1.set_xticks(())
ax1.set_yticks(())
plot_ellipses(ax1, estimator.weights_, estimator.means_,
              estimator.covariances_)

ax2.get_xaxis().set_tick_params(direction='out')
ax2.yaxis.grid(True, alpha=0.7)
for k, w in enumerate(estimator.weights_):
    ax2.bar(k, w, width=0.9, color="#5B4E9F", zorder=3,
            align='center', edgecolor='black')
    ax2.text(k, w + 0.007, "%." + str(1 - int(np.log10(w))) + "f" % (w * 100.),
              horizontalalignment='center')
ax2.set_xlim(-.6, 2 * n_components - .4)
ax2.set_ylim(0., 1.1)
ax2.tick_params(axis='y', which='both', left=False,
                right=False, labelleft=False)
ax2.tick_params(axis='x', which='both', top=False)

if plot_title:
    ax1.set_ylabel('Estimated Mixtures')
    ax2.set_ylabel('Weight of each component')

# Parameters of the dataset
random_state, n_components, n_features = 2, 3, 2
colors = np.array(['#0072B2', '#F0E442', '#D55E00'])

covars = np.array([[ [.7, .0], [.0, .1]],
                  [[.5, .0], [.0, .1]],
                  [[.5, .0], [.0, .1]]])
samples = np.array([200, 500, 200])
means = np.array([[.0, -.70],
                 [.0, .0],
                 [.0, .70]])

# mean_precision_prior= 0.8 to minimize the influence of the prior
estimators = [
    ("Finite mixture with a Dirichlet distribution\nprior and "
     r"$\gamma_0=$", BayesianGaussianMixture(
        weight_concentration_prior_type="dirichlet_distribution",
        n_components=2 * n_components, reg_covar=0, init_params='random',
        max_iter=1500, mean_precision_prior=.8,
        random_state=random_state), [0.001, 1, 1000]),
    ("Infinite mixture with a Dirichlet process\nprior and" r"$\gamma_0=$",
     BayesianGaussianMixture(
        weight_concentration_prior_type="dirichlet_process",
        n_components=2 * n_components, reg_covar=0, init_params='random',
        max_iter=1500, mean_precision_prior=.8,
        random_state=random_state), [1, 1000, 100000])]

# Generate data
rng = np.random.RandomState(random_state)
X = np.vstack([
    rng.multivariate_normal(means[j], covars[j], samples[j])
    for j in range(n_components)])
y = np.concatenate([np.full(samples[j], j, dtype=int)
                   for j in range(n_components)])

# Plot results in two different figures

```

```
for (title, estimator, concentrations_prior) in estimators:
    plt.figure(figsize=(4.7 * 3, 8))
    plt.subplots_adjust(bottom=.04, top=0.90, hspace=.05, wspace=.05,
                        left=.03, right=.99)

    gs = gridspec.GridSpec(3, len(concentrations_prior))
    for k, concentration in enumerate(concentrations_prior):
        estimator.weight_concentration_prior = concentration
        estimator.fit(X)
        plot_results(plt.subplot(gs[0:2, k]), plt.subplot(gs[2, k]), estimator,
                     X, y, r"%s%.1e" % (title, concentration),
                     plot_title=k == 0)

plt.show()
```

Total running time of the script: (0 minutes 6.923 seconds)

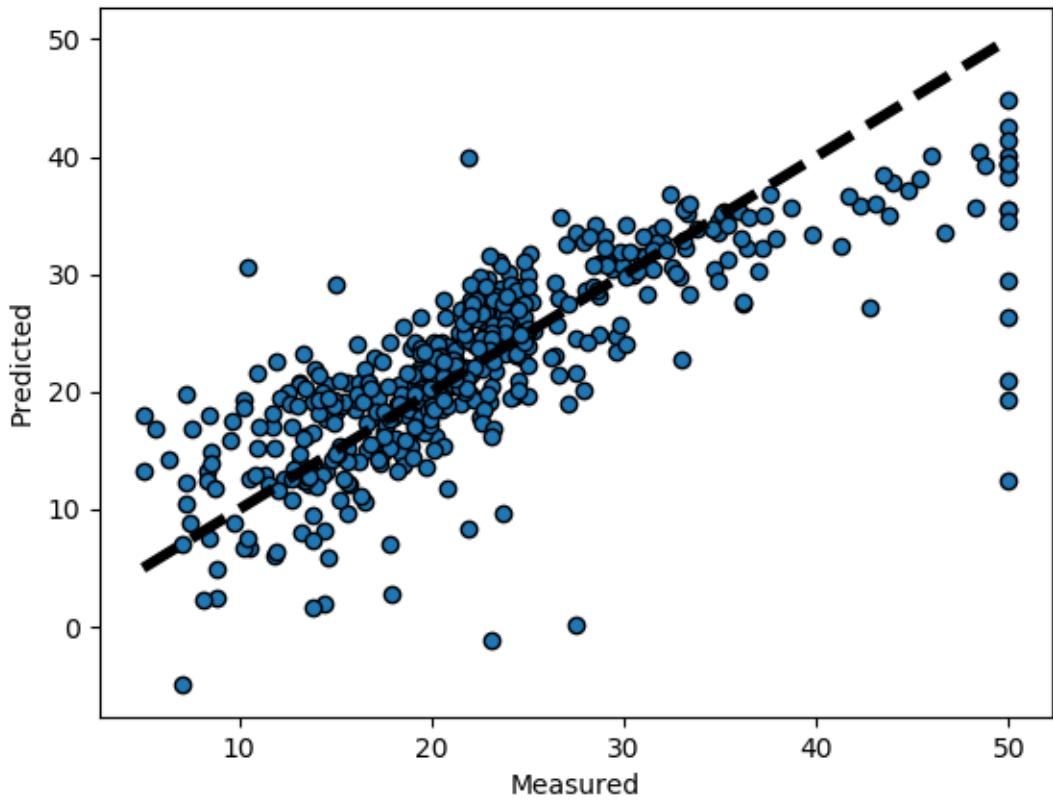
5.21 Model Selection

Examples related to the `sklearn.model_selection` module.

Note: Click [here](#) to download the full example code

5.21.1 Plotting Cross-Validated Predictions

This example shows how to use `cross_val_predict` to visualize prediction errors.



```
from sklearn import datasets
from sklearn.model_selection import cross_val_predict
from sklearn import linear_model
import matplotlib.pyplot as plt

lr = linear_model.LinearRegression()
boston = datasets.load_boston()
y = boston.target

# cross_val_predict returns an array of the same size as `y` where each entry
# is a prediction obtained by cross validation:
predicted = cross_val_predict(lr, boston.data, y, cv=10)

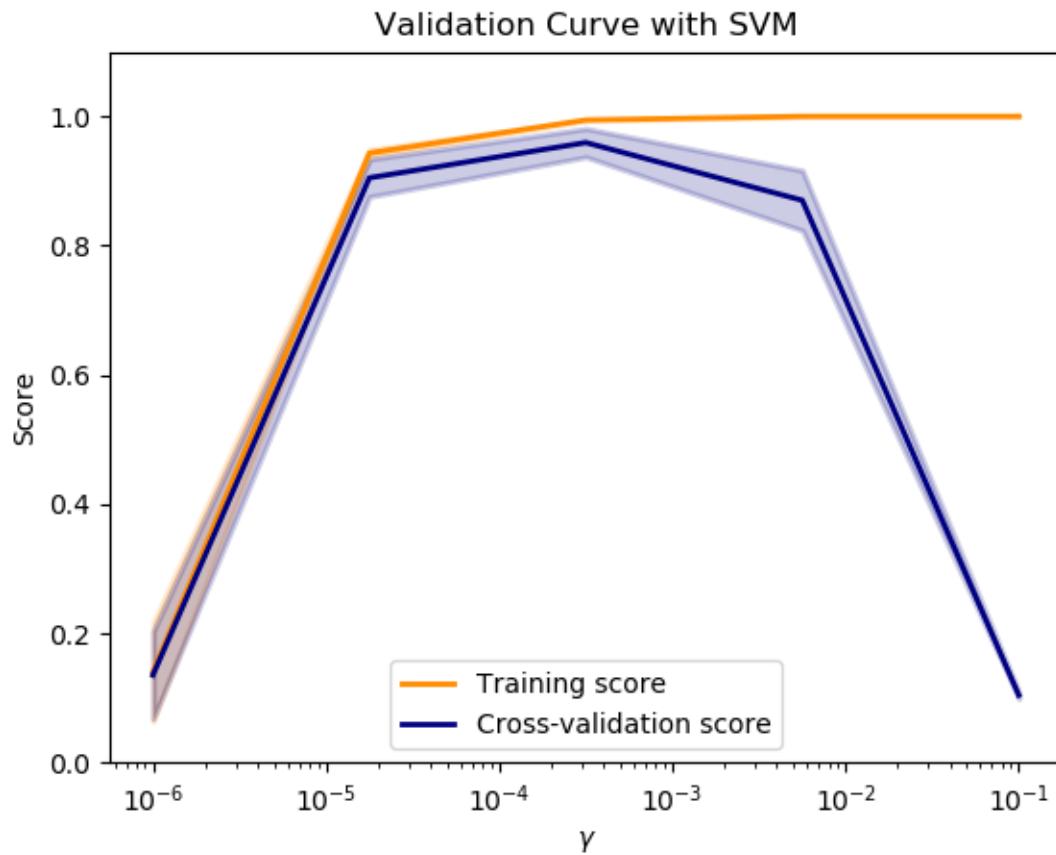
fig, ax = plt.subplots()
ax.scatter(y, predicted, edgecolors=(0, 0, 0))
ax.plot([y.min(), y.max()], [y.min(), y.max()], 'k--', lw=4)
ax.set_xlabel('Measured')
ax.set_ylabel('Predicted')
plt.show()
```

Total running time of the script: (0 minutes 0.024 seconds)

Note: Click [here](#) to download the full example code

5.21.2 Plotting Validation Curves

In this plot you can see the training scores and validation scores of an SVM for different values of the kernel parameter gamma. For very low values of gamma, you can see that both the training score and the validation score are low. This is called underfitting. Medium values of gamma will result in high values for both scores, i.e. the classifier is performing fairly well. If gamma is too high, the classifier will overfit, which means that the training score is good but the validation score is poor.



```
print(__doc__)

import matplotlib.pyplot as plt
import numpy as np

from sklearn.datasets import load_digits
from sklearn.svm import SVC
from sklearn.model_selection import validation_curve

digits = load_digits()
X, y = digits.data, digits.target

param_range = np.logspace(-6, -1, 5)
train_scores, test_scores = validation_curve(
    SVC(), X, y, param_name="gamma", param_range=param_range,
    cv=5, scoring="accuracy", n_jobs=1)
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
```

```

test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

plt.title("Validation Curve with SVM")
plt.xlabel(r"$\gamma$")
plt.ylabel("Score")
plt.ylim(0.0, 1.1)
lw = 2
plt.semilogx(param_range, train_scores_mean, label="Training score",
             color="darkorange", lw=lw)
plt.fill_between(param_range, train_scores_mean - train_scores_std,
                 train_scores_mean + train_scores_std, alpha=0.2,
                 color="darkorange", lw=lw)
plt.semilogx(param_range, test_scores_mean, label="Cross-validation score",
             color="navy", lw=lw)
plt.fill_between(param_range, test_scores_mean - test_scores_std,
                 test_scores_mean + test_scores_std, alpha=0.2,
                 color="navy", lw=lw)
plt.legend(loc="best")
plt.show()

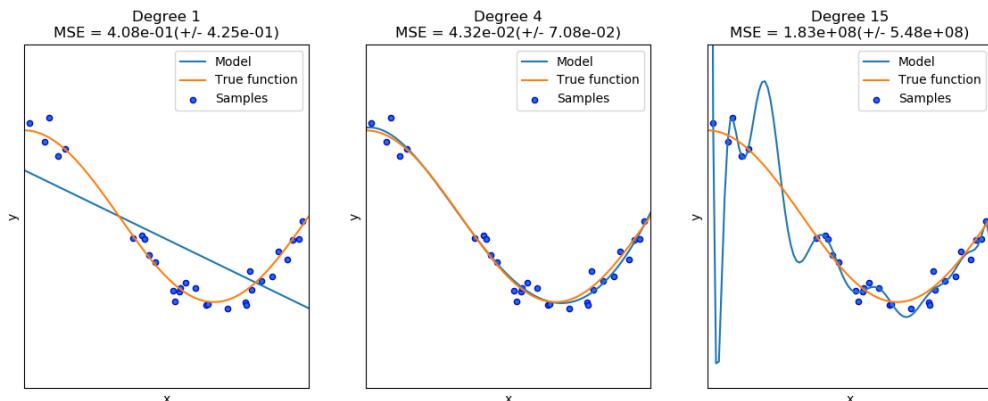
```

Total running time of the script: (0 minutes 13.416 seconds)

Note: Click [here](#) to download the full example code

5.21.3 Underfitting vs. Overfitting

This example demonstrates the problems of underfitting and overfitting and how we can use linear regression with polynomial features to approximate nonlinear functions. The plot shows the function that we want to approximate, which is a part of the cosine function. In addition, the samples from the real function and the approximations of different models are displayed. The models have polynomial features of different degrees. We can see that a linear function (polynomial with degree 1) is not sufficient to fit the training samples. This is called **underfitting**. A polynomial of degree 4 approximates the true function almost perfectly. However, for higher degrees the model will **overfit** the training data, i.e. it learns the noise of the training data. We evaluate quantitatively **overfitting / underfitting** by using cross-validation. We calculate the mean squared error (MSE) on the validation set, the higher, the less likely the model generalizes correctly from the training data.



```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score

def true_fun(X):
    return np.cos(1.5 * np.pi * X)

np.random.seed(0)

n_samples = 30
degrees = [1, 4, 15]

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

plt.figure(figsize=(14, 5))
for i in range(len(degrees)):
    ax = plt.subplot(1, len(degrees), i + 1)
    plt.setp(ax, xticks=(), yticks=())

    polynomial_features = PolynomialFeatures(degree=degrees[i],
                                              include_bias=False)
    linear_regression = LinearRegression()
    pipeline = Pipeline([("polynomial_features", polynomial_features),
                         ("linear_regression", linear_regression)])
    pipeline.fit(X[:, np.newaxis], y)

    # Evaluate the models using crossvalidation
    scores = cross_val_score(pipeline, X[:, np.newaxis], y,
                             scoring="neg_mean_squared_error", cv=10)

    X_test = np.linspace(0, 1, 100)
    plt.plot(X_test, pipeline.predict(X_test[:, np.newaxis]), label="Model")
    plt.plot(X_test, true_fun(X_test), label="True function")
    plt.scatter(X, y, edgecolor='b', s=20, label="Samples")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.xlim((0, 1))
    plt.ylim((-2, 2))
    plt.legend(loc="best")
    plt.title("Degree {} \nMSE = {:.2e} (+/- {:.2e})".format(
        degrees[i], -scores.mean(), scores.std()))
plt.show()

```

Total running time of the script: (0 minutes 0.084 seconds)

Note: Click [here](#) to download the full example code

5.21.4 Parameter estimation using grid search with cross-validation

This examples shows how a classifier is optimized by cross-validation, which is done using the `sklearn.model_selection.GridSearchCV` object on a development set that comprises only half of the available labeled data.

The performance of the selected hyper-parameters and trained model is then measured on a dedicated evaluation set that was not used during the model selection step.

More details on tools available for model selection can be found in the sections on *Cross-validation: evaluating estimator performance* and *Tuning the hyper-parameters of an estimator*.

Out:

```
# Tuning hyper-parameters for precision

Best parameters set found on development set:

{'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}

Grid scores on development set:

0.986 (+/-0.016) for {'C': 1, 'gamma': 0.001, 'kernel': 'rbf'}
0.959 (+/-0.029) for {'C': 1, 'gamma': 0.0001, 'kernel': 'rbf'}
0.988 (+/-0.017) for {'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}
0.982 (+/-0.026) for {'C': 10, 'gamma': 0.0001, 'kernel': 'rbf'}
0.988 (+/-0.017) for {'C': 100, 'gamma': 0.001, 'kernel': 'rbf'}
0.982 (+/-0.025) for {'C': 100, 'gamma': 0.0001, 'kernel': 'rbf'}
0.988 (+/-0.017) for {'C': 1000, 'gamma': 0.001, 'kernel': 'rbf'}
0.982 (+/-0.025) for {'C': 1000, 'gamma': 0.0001, 'kernel': 'rbf'}
0.975 (+/-0.014) for {'C': 1, 'kernel': 'linear'}
0.975 (+/-0.014) for {'C': 10, 'kernel': 'linear'}
0.975 (+/-0.014) for {'C': 100, 'kernel': 'linear'}
0.975 (+/-0.014) for {'C': 1000, 'kernel': 'linear'}
```

Detailed classification report:

The model is trained on the full development set.
The scores are computed on the full evaluation set.

	precision	recall	f1-score	support
0	1.00	1.00	1.00	89
1	0.97	1.00	0.98	90
2	0.99	0.98	0.98	92
3	1.00	0.99	0.99	93
4	1.00	1.00	1.00	76
5	0.99	0.98	0.99	108
6	0.99	1.00	0.99	89
7	0.99	1.00	0.99	78
8	1.00	0.98	0.99	92
9	0.99	0.99	0.99	92
accuracy			0.99	899
macro avg	0.99	0.99	0.99	899
weighted avg	0.99	0.99	0.99	899

```
# Tuning hyper-parameters for recall
```

```
Best parameters set found on development set:

{'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}

Grid scores on development set:

0.986 (+/-0.019) for {'C': 1, 'gamma': 0.001, 'kernel': 'rbf'}
0.957 (+/-0.029) for {'C': 1, 'gamma': 0.0001, 'kernel': 'rbf'}
0.987 (+/-0.019) for {'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}
0.981 (+/-0.028) for {'C': 10, 'gamma': 0.0001, 'kernel': 'rbf'}
0.987 (+/-0.019) for {'C': 100, 'gamma': 0.001, 'kernel': 'rbf'}
0.981 (+/-0.026) for {'C': 100, 'gamma': 0.0001, 'kernel': 'rbf'}
0.987 (+/-0.019) for {'C': 1000, 'gamma': 0.001, 'kernel': 'rbf'}
0.981 (+/-0.026) for {'C': 1000, 'gamma': 0.0001, 'kernel': 'rbf'}
0.972 (+/-0.012) for {'C': 1, 'kernel': 'linear'}
0.972 (+/-0.012) for {'C': 10, 'kernel': 'linear'}
0.972 (+/-0.012) for {'C': 100, 'kernel': 'linear'}
0.972 (+/-0.012) for {'C': 1000, 'kernel': 'linear'}
```

Detailed classification report:

The model is trained on the full development set.
The scores are computed on the full evaluation set.

	precision	recall	f1-score	support
0	1.00	1.00	1.00	89
1	0.97	1.00	0.98	90
2	0.99	0.98	0.98	92
3	1.00	0.99	0.99	93
4	1.00	1.00	1.00	76
5	0.99	0.98	0.99	108
6	0.99	1.00	0.99	89
7	0.99	1.00	0.99	78
8	1.00	0.98	0.99	92
9	0.99	0.99	0.99	92
accuracy			0.99	899
macro avg	0.99	0.99	0.99	899
weighted avg	0.99	0.99	0.99	899

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
from sklearn.svm import SVC

print(__doc__)

# Loading the Digits dataset
digits = datasets.load_digits()
```

```
# To apply an classifier on this data, we need to flatten the image, to
# turn the data in a (samples, feature) matrix:
n_samples = len(digits.images)
X = digits.images.reshape((n_samples, -1))
y = digits.target

# Split the dataset in two equal parts
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.5, random_state=0)

# Set the parameters by cross-validation
tuned_parameters = [{"kernel": ["rbf"], "gamma": [1e-3, 1e-4],
                     "C": [1, 10, 100, 1000]},
                     {"kernel": ["linear"], "C": [1, 10, 100, 1000]}]

scores = ['precision', 'recall']

for score in scores:
    print("# Tuning hyper-parameters for %s" % score)
    print()

    clf = GridSearchCV(SVC(), tuned_parameters, cv=5,
                        scoring='%s_macro' % score)
    clf.fit(X_train, y_train)

    print("Best parameters set found on development set:")
    print()
    print(clf.best_params_)
    print()
    print("Grid scores on development set:")
    print()
    means = clf.cv_results_['mean_test_score']
    stds = clf.cv_results_['std_test_score']
    for mean, std, params in zip(means, stds, clf.cv_results_['params']):
        print("%0.3f (+/-%0.03f) for %r"
              % (mean, std * 2, params))
    print()

    print("Detailed classification report:")
    print()
    print("The model is trained on the full development set.")
    print("The scores are computed on the full evaluation set.")
    print()
    y_true, y_pred = y_test, clf.predict(X_test)
    print(classification_report(y_true, y_pred))
    print()

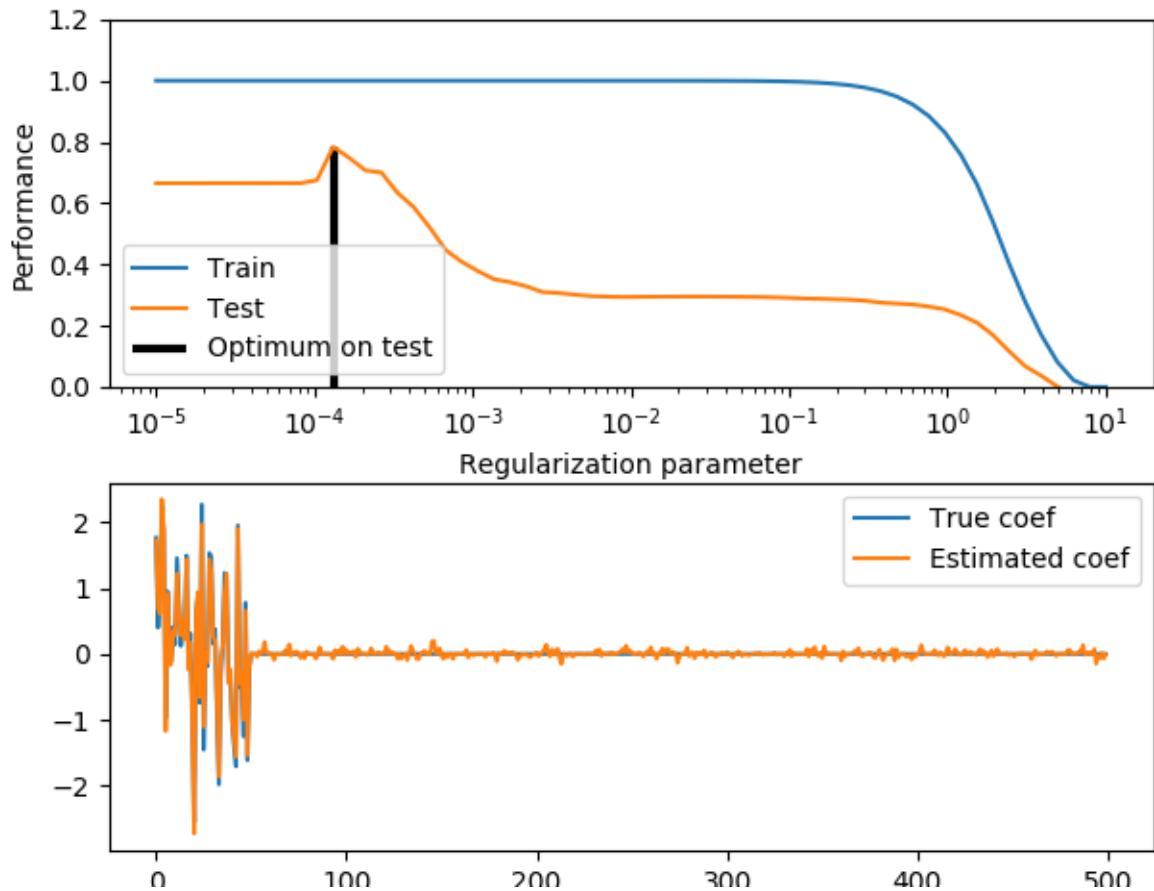
# Note the problem is too easy: the hyperparameter plateau is too flat and the
# output model is the same for precision and recall with ties in quality.
```

Total running time of the script: (0 minutes 4.344 seconds)

Note: Click [here](#) to download the full example code

5.21.5 Train error vs Test error

Illustration of how the performance of an estimator on unseen data (test data) is not the same as the performance on training data. As the regularization increases the performance on train decreases while the performance on test is optimal within a range of values of the regularization parameter. The example with an Elastic-Net regression model and the performance is measured using the explained variance a.k.a. R².



Out:

```
Optimal regularization parameter : 0.00013141473626117567
```

```
print(__doc__)

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause

import numpy as np
from sklearn import linear_model

#####
# Generate sample data
#####
```

```
n_samples_train, n_samples_test, n_features = 75, 150, 500
np.random.seed(0)
coef = np.random.randn(n_features)
coef[50:] = 0.0 # only the top 10 features are impacting the model
X = np.random.randn(n_samples_train + n_samples_test, n_features)
y = np.dot(X, coef)

# Split train and test data
X_train, X_test = X[:n_samples_train], X[n_samples_train:]
y_train, y_test = y[:n_samples_train], y[n_samples_train:]

# ##########
# Compute train and test errors
alphas = np.logspace(-5, 1, 60)
enet = linear_model.ElasticNet(l1_ratio=0.7, max_iter=10000)
train_errors = list()
test_errors = list()
for alpha in alphas:
    enet.set_params(alpha=alpha)
    enet.fit(X_train, y_train)
    train_errors.append(enet.score(X_train, y_train))
    test_errors.append(enet.score(X_test, y_test))

i_alpha_optim = np.argmax(test_errors)
alpha_optim = alphas[i_alpha_optim]
print("Optimal regularization parameter : %s" % alpha_optim)

# Estimate the coef_ on full data with optimal regularization parameter
enet.set_params(alpha=alpha_optim)
coef_ = enet.fit(X, y).coef_

# ##########
# Plot results functions

import matplotlib.pyplot as plt
plt.subplot(2, 1, 1)
plt.semilogx(alphas, train_errors, label='Train')
plt.semilogx(alphas, test_errors, label='Test')
plt.vlines(alpha_optim, plt.ylim()[0], np.max(test_errors), color='k',
           linewidth=3, label='Optimum on test')
plt.legend(loc='lower left')
plt.ylim([0, 1.2])
plt.xlabel('Regularization parameter')
plt.ylabel('Performance')

# Show estimated coef_ vs true coef
plt.subplot(2, 1, 2)
plt.plot(coef, label='True coef')
plt.plot(coef_, label='Estimated coef')
plt.legend()
plt.subplots_adjust(0.09, 0.04, 0.94, 0.94, 0.26, 0.26)
plt.show()
```

Total running time of the script: (0 minutes 3.507 seconds)

Note: Click [here](#) to download the full example code

5.21.6 Receiver Operating Characteristic (ROC) with cross validation

Example of Receiver Operating Characteristic (ROC) metric to evaluate classifier output quality using cross-validation.

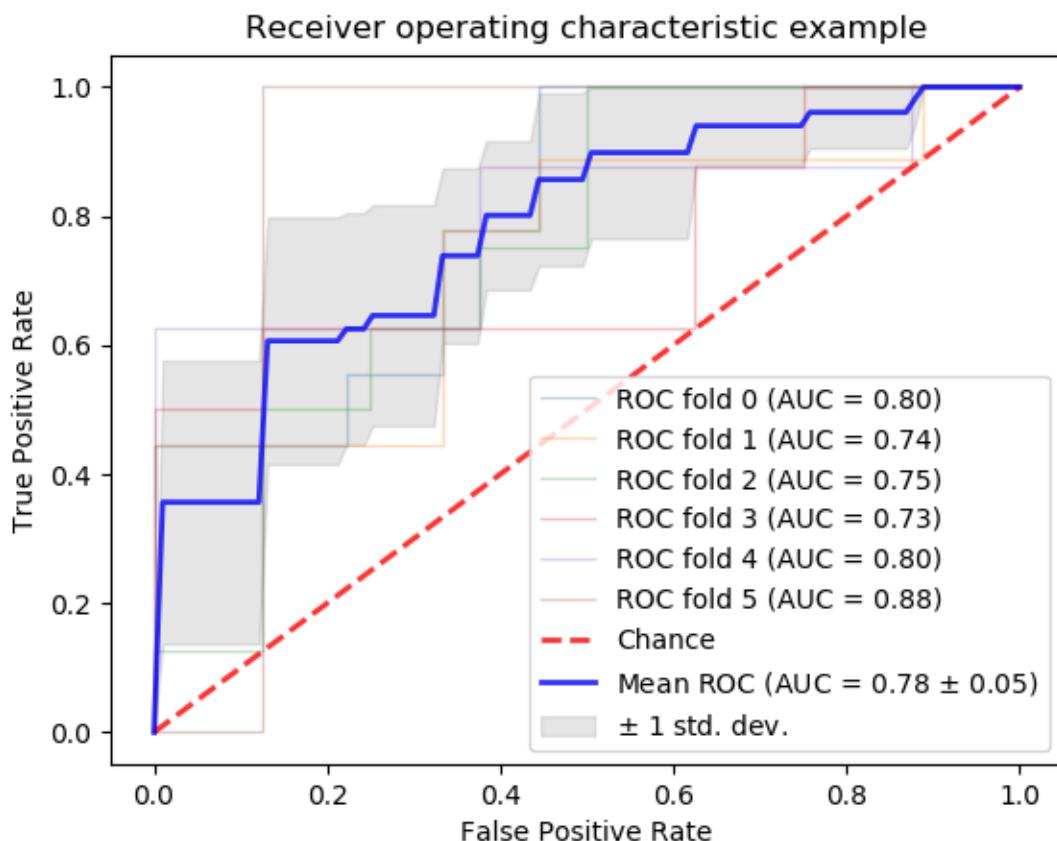
ROC curves typically feature true positive rate on the Y axis, and false positive rate on the X axis. This means that the top left corner of the plot is the “ideal” point - a false positive rate of zero, and a true positive rate of one. This is not very realistic, but it does mean that a larger area under the curve (AUC) is usually better.

The “steepness” of ROC curves is also important, since it is ideal to maximize the true positive rate while minimizing the false positive rate.

This example shows the ROC response of different datasets, created from K-fold cross-validation. Taking all of these curves, it is possible to calculate the mean area under curve, and see the variance of the curve when the training set is split into different subsets. This roughly shows how the classifier output is affected by changes in the training data, and how different the splits generated by K-fold cross-validation are from one another.

Note:

See also [sklearn.metrics.roc_auc_score](#), [sklearn.model_selection.cross_val_score](#),
Receiver Operating Characteristic (ROC),



```
print(__doc__)

import numpy as np
from scipy import interp
```

```

import matplotlib.pyplot as plt

from sklearn import svm, datasets
from sklearn.metrics import roc_curve, auc
from sklearn.model_selection import StratifiedKFold

# ##### Data IO and generation

# Import some data to play with
iris = datasets.load_iris()
X = iris.data
y = iris.target
X, y = X[y != 2], y[y != 2]
n_samples, n_features = X.shape

# Add noisy features
random_state = np.random.RandomState(0)
X = np.c_[X, random_state.randn(n_samples, 200 * n_features)]

# ##### Classification and ROC analysis

# Run classifier with cross-validation and plot ROC curves
cv = StratifiedKFold(n_splits=6)
classifier = svm.SVC(kernel='linear', probability=True,
                     random_state=random_state)

tprs = []
aucs = []
mean_fpr = np.linspace(0, 1, 100)

i = 0
for train, test in cv.split(X, y):
    probas_ = classifier.fit(X[train], y[train]).predict_proba(X[test])
    # Compute ROC curve and area the curve
    fpr, tpr, thresholds = roc_curve(y[test], probas_[:, 1])
    tprs.append(interp(mean_fpr, fpr, tpr))
    tprs[-1][0] = 0.0
    roc_auc = auc(fpr, tpr)
    aucs.append(roc_auc)
    plt.plot(fpr, tpr, lw=1, alpha=0.3,
              label='ROC fold %d (AUC = %.2f)' % (i, roc_auc))

    i += 1
plt.plot([0, 1], [0, 1], linestyle='--', lw=2, color='r',
          label='Chance', alpha=.8)

mean_tpr = np.mean(tprs, axis=0)
mean_tpr[-1] = 1.0
mean_auc = auc(mean_fpr, mean_tpr)
std_auc = np.std(aucs)
plt.plot(mean_fpr, mean_tpr, color='b',
          label=r'Mean ROC (AUC = %.2f $\pm$ %.2f)' % (mean_auc, std_auc),
          lw=2, alpha=.8)

std_tpr = np.std(tprs, axis=0)
tprs_upper = np.minimum(mean_tpr + std_tpr, 1)

```

```

tprs_lower = np.maximum(mean_tpr - std_tpr, 0)
plt.fill_between(mean_fpr, tprs_lower, tprs_upper, color='grey', alpha=.2,
                 label=r'$\pm$ 1 std. dev.')

plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")
plt.show()

```

Total running time of the script: (0 minutes 0.245 seconds)

Note: Click [here](#) to download the full example code

5.21.7 Comparing randomized search and grid search for hyperparameter estimation

Compare randomized search and grid search for optimizing hyperparameters of a random forest. All parameters that influence the learning are searched simultaneously (except for the number of estimators, which poses a time / quality tradeoff).

The randomized search and the grid search explore exactly the same space of parameters. The result in parameter settings is quite similar, while the run time for randomized search is drastically lower.

The performance is slightly worse for the randomized search, though this is most likely a noise effect and would not carry over to a held-out test set.

Note that in practice, one would not search over this many different parameters simultaneously using grid search, but pick only the ones deemed most important.

Out:

```

RandomizedSearchCV took 4.42 seconds for 20 candidates parameter settings.
Model with rank: 1
Mean validation score: 0.939 (std: 0.024)
Parameters: {'bootstrap': False, 'criterion': 'entropy', 'max_depth': None, 'max_
↳features': 7, 'min_samples_split': 3}

Model with rank: 2
Mean validation score: 0.933 (std: 0.022)
Parameters: {'bootstrap': False, 'criterion': 'gini', 'max_depth': None, 'max_features
↳': 6, 'min_samples_split': 6}

Model with rank: 3
Mean validation score: 0.930 (std: 0.031)
Parameters: {'bootstrap': True, 'criterion': 'gini', 'max_depth': None, 'max_features
↳': 6, 'min_samples_split': 6}

GridSearchCV took 13.24 seconds for 72 candidate parameter settings.
Model with rank: 1
Mean validation score: 0.937 (std: 0.019)
Parameters: {'bootstrap': False, 'criterion': 'entropy', 'max_depth': None, 'max_
↳features': 10, 'min_samples_split': 2}

```

```
Model with rank: 2
Mean validation score: 0.936 (std: 0.020)
Parameters: {'bootstrap': False, 'criterion': 'gini', 'max_depth': None, 'max_features':
             10, 'min_samples_split': 2}

Model with rank: 3
Mean validation score: 0.931 (std: 0.029)
Parameters: {'bootstrap': False, 'criterion': 'entropy', 'max_depth': None, 'max_
             features': 10, 'min_samples_split': 3}
```

```
print(__doc__)

import numpy as np

from time import time
from scipy.stats import randint as sp_randint

from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from sklearn.datasets import load_digits
from sklearn.ensemble import RandomForestClassifier

# get some data
digits = load_digits()
X, y = digits.data, digits.target

# build a classifier
clf = RandomForestClassifier(n_estimators=20)

# Utility function to report best scores
def report(results, n_top=3):
    for i in range(1, n_top + 1):
        candidates = np.flatnonzero(results['rank_test_score'] == i)
        for candidate in candidates:
            print("Model with rank: {}".format(i))
            print("Mean validation score: {:.3f} (std: {:.3f})".format(
                results['mean_test_score'][candidate],
                results['std_test_score'][candidate]))
            print("Parameters: {}".format(results['params'][candidate]))
            print("")

# specify parameters and distributions to sample from
param_dist = {"max_depth": [3, None],
              "max_features": sp_randint(1, 11),
              "min_samples_split": sp_randint(2, 11),
              "bootstrap": [True, False],
              "criterion": ["gini", "entropy"]}

# run randomized search
n_iter_search = 20
random_search = RandomizedSearchCV(clf, param_distributions=param_dist,
```

```

n_iter=n_iter_search, cv=5, iid=False)

start = time()
random_search.fit(X, y)
print("RandomizedSearchCV took %.2f seconds for %d candidates"
      " parameter settings." % ((time() - start), n_iter_search))
report(random_search.cv_results_)

# use a full grid over all parameters
param_grid = {"max_depth": [3, None],
              "max_features": [1, 3, 10],
              "min_samples_split": [2, 3, 10],
              "bootstrap": [True, False],
              "criterion": ["gini", "entropy"]}

# run grid search
grid_search = GridSearchCV(clf, param_grid=param_grid, cv=5, iid=False)
start = time()
grid_search.fit(X, y)

print("GridSearchCV took %.2f seconds for %d candidate parameter settings."
      " % (time() - start, len(grid_search.cv_results_['params'])))")
report(grid_search.cv_results_)

```

Total running time of the script: (0 minutes 17.712 seconds)

Note: Click [here](#) to download the full example code

5.21.8 Nested versus non-nested cross-validation

This example compares non-nested and nested cross-validation strategies on a classifier of the iris data set. Nested cross-validation (CV) is often used to train a model in which hyperparameters also need to be optimized. Nested CV estimates the generalization error of the underlying model and its (hyper)parameter search. Choosing the parameters that maximize non-nested CV biases the model to the dataset, yielding an overly-optimistic score.

Model selection without nested CV uses the same data to tune model parameters and evaluate model performance. Information may thus “leak” into the model and overfit the data. The magnitude of this effect is primarily dependent on the size of the dataset and the stability of the model. See Cawley and Talbot¹ for an analysis of these issues.

To avoid this problem, nested CV effectively uses a series of train/validation/test set splits. In the inner loop (here executed by `GridSearchCV`), the score is approximately maximized by fitting a model to each training set, and then directly maximized in selecting (hyper)parameters over the validation set. In the outer loop (here in `cross_val_score`), generalization error is estimated by averaging test set scores over several dataset splits.

The example below uses a support vector classifier with a non-linear kernel to build a model with optimized hyperparameters by grid search. We compare the performance of non-nested and nested CV strategies by taking the difference between their scores.

See Also:

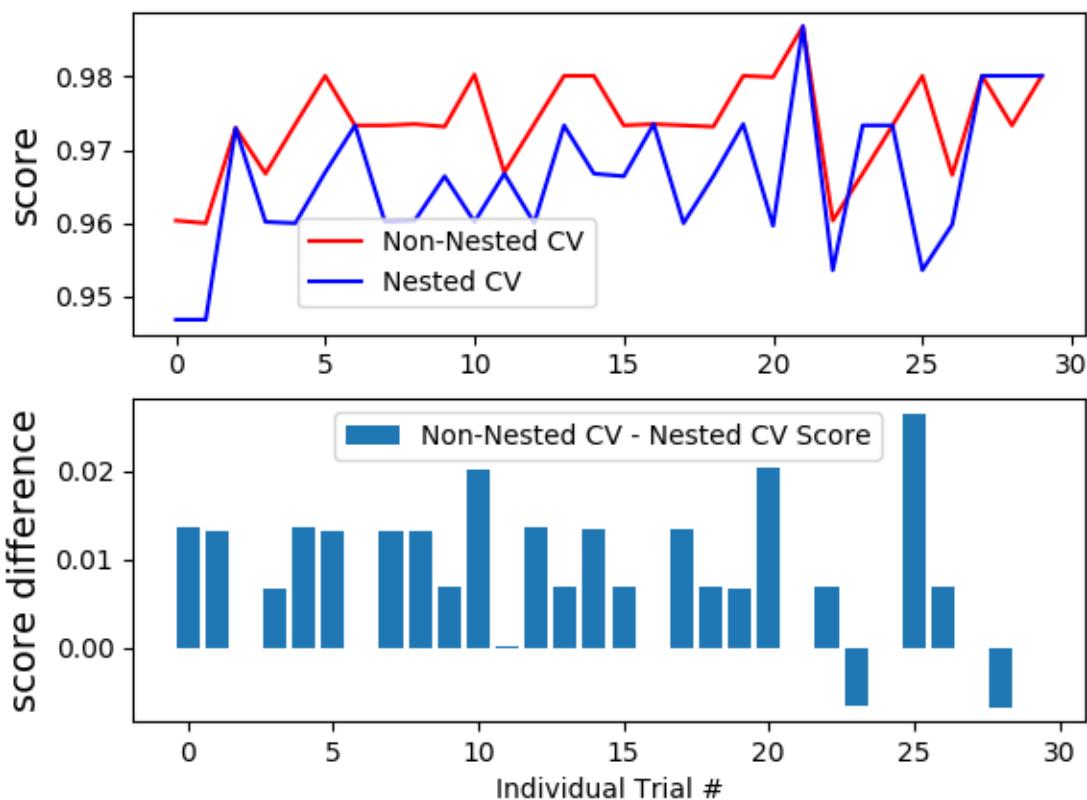
- [Cross-validation: evaluating estimator performance](#)

¹ Cawley, G.C.; Talbot, N.L.C. On over-fitting in model selection and subsequent selection bias in performance evaluation. *J. Mach. Learn. Res.* 2010, 11, 2079-2107.

- Tuning the hyper-parameters of an estimator

References:

Non-Nested and Nested Cross Validation on Iris Dataset



Out:

```
Average difference of 0.007581 with std. dev. of 0.007833.
```

```
from sklearn.datasets import load_iris
from matplotlib import pyplot as plt
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV, cross_val_score, KFold
import numpy as np

print(__doc__)

# Number of random trials
NUM_TRIALS = 30
```

```

# Load the dataset
iris = load_iris()
X_iris = iris.data
y_iris = iris.target

# Set up possible values of parameters to optimize over
p_grid = {"C": [1, 10, 100],
           "gamma": [.01, .1]}

# We will use a Support Vector Classifier with "rbf" kernel
svm = SVC(kernel="rbf")

# Arrays to store scores
non_nested_scores = np.zeros(NUM_TRIALS)
nested_scores = np.zeros(NUM_TRIALS)

# Loop for each trial
for i in range(NUM_TRIALS):

    # Choose cross-validation techniques for the inner and outer loops,
    # independently of the dataset.
    # E.g "GroupKFold", "LeaveOneOut", "LeaveOneGroupOut", etc.
    inner_cv = KFold(n_splits=4, shuffle=True, random_state=i)
    outer_cv = KFold(n_splits=4, shuffle=True, random_state=i)

    # Non_nested parameter search and scoring
    clf = GridSearchCV(estimator=svm, param_grid=p_grid, cv=inner_cv,
                        iid=False)
    clf.fit(X_iris, y_iris)
    non_nested_scores[i] = clf.best_score_

    # Nested CV with parameter optimization
    nested_score = cross_val_score(clf, X=X_iris, y=y_iris, cv=outer_cv)
    nested_scores[i] = nested_score.mean()

score_difference = non_nested_scores - nested_scores

print("Average difference of {:.6f} with std. dev. of {:.6f}."
      .format(score_difference.mean(), score_difference.std()))

# Plot scores on each trial for nested and non-nested CV
plt.figure()
plt.subplot(211)
non_nested_scores_line, = plt.plot(non_nested_scores, color='r')
nested_line, = plt.plot(nested_scores, color='b')
plt.ylabel("score", fontsize="14")
plt.legend([non_nested_scores_line, nested_line],
           ["Non-Nested CV", "Nested CV"],
           bbox_to_anchor=(0, .4, .5, 0))
plt.title("Non-Nested and Nested Cross Validation on Iris Dataset",
          x=.5, y=1.1, fontsize="15")

# Plot bar chart of the difference.
plt.subplot(212)
difference_plot = plt.bar(range(NUM_TRIALS), score_difference)
plt.xlabel("Individual Trial #")
plt.legend([difference_plot],

```

```
    ["Non-Nested CV - Nested CV Score"],  
    bbox_to_anchor=(0, 1, .8, 0))  
plt.ylabel("score difference", fontsize="14")  
  
plt.show()
```

Total running time of the script: (0 minutes 3.447 seconds)

Note: Click [here](#) to download the full example code

5.21.9 Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV

Multiple metric parameter search can be done by setting the `scoring` parameter to a list of metric scorer names or a dict mapping the scorer names to the scorer callables.

The scores of all the scorers are available in the `cv_results_` dict at keys ending in '`_<scorer_name>`' ('`mean_test_precision`', '`rank_test_precision`', etc...)

The `best_estimator_`, `best_index_`, `best_score_` and `best_params_` correspond to the scorer (key) that is set to the `refit` attribute.

```
# Author: Raghav RV <rvraghav93@gmail.com>  
# License: BSD  
  
import numpy as np  
from matplotlib import pyplot as plt  
  
from sklearn.datasets import make_hastie_10_2  
from sklearn.model_selection import GridSearchCV  
from sklearn.metrics import make_scorer  
from sklearn.metrics import accuracy_score  
from sklearn.tree import DecisionTreeClassifier  
  
print(__doc__)
```

Running GridSearchCV using multiple evaluation metrics

```
x, y = make_hastie_10_2(n_samples=8000, random_state=42)  
  
# The scorers can be either be one of the predefined metric strings or a scorer  
# callable, like the one returned by make_scorer  
scoring = {'AUC': 'roc_auc', 'Accuracy': make_scorer(accuracy_score)}  
  
# Setting refit='AUC', refits an estimator on the whole dataset with the  
# parameter setting that has the best cross-validated AUC score.  
# That estimator is made available at ``gs.best_estimator_`` along with  
# parameters like ``gs.best_score_``, ``gs.best_params_`` and  
# ``gs.best_index_``  
gs = GridSearchCV(DecisionTreeClassifier(random_state=42),  
                  param_grid={'min_samples_split': range(2, 403, 10)},  
                  scoring=scoring, cv=5, refit='AUC', return_train_score=True)
```

```
gs.fit(X, y)
results = gs.cv_results_
```

Plotting the result

```
plt.figure(figsize=(13, 13))
plt.title("GridSearchCV evaluating using multiple scorers simultaneously",
          fontsize=16)

plt.xlabel("min_samples_split")
plt.ylabel("Score")

ax = plt.gca()
ax.set_xlim(0, 402)
ax.set_ylim(0.73, 1)

# Get the regular numpy array from the MaskedArray
X_axis = np.array(results['param_min_samples_split'].data, dtype=float)

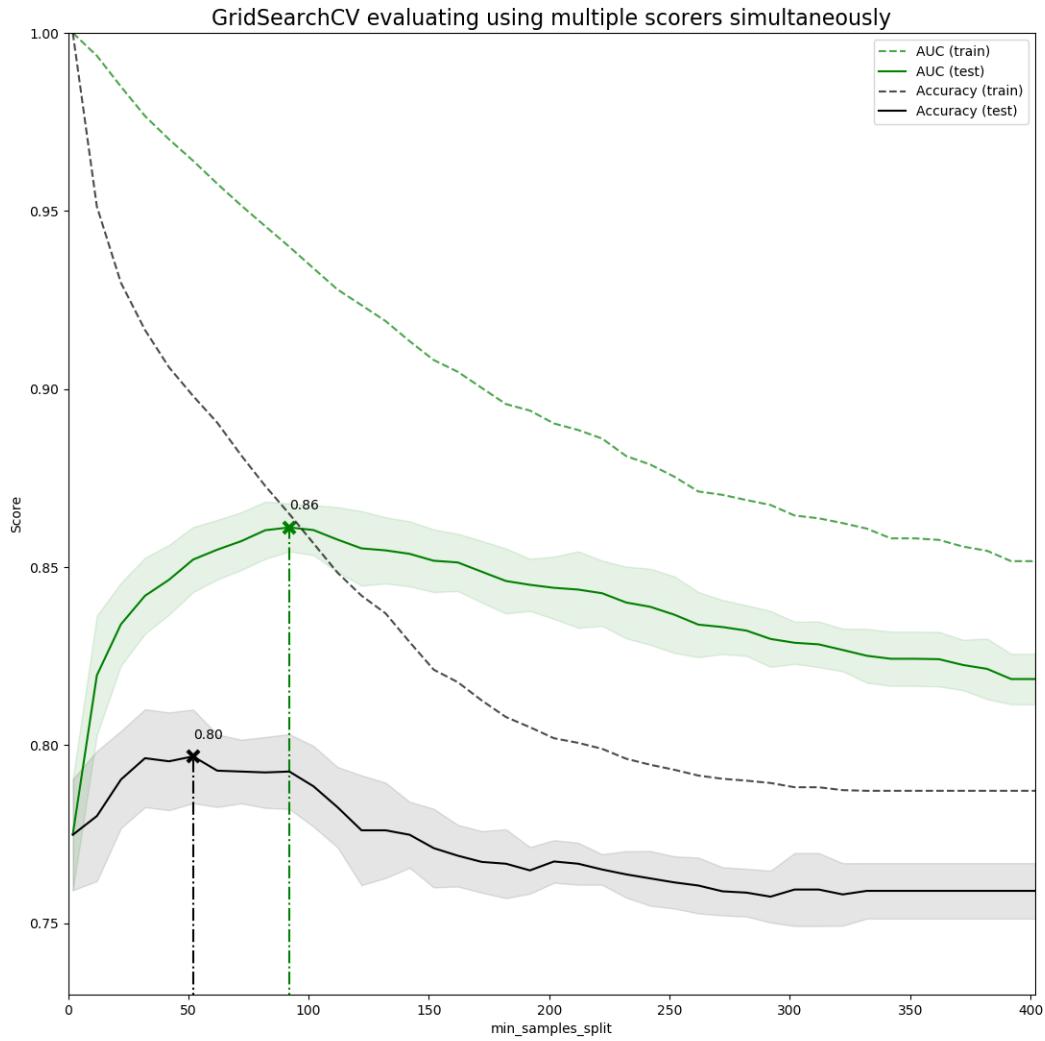
for scorer, color in zip(sorted(scoring), ['g', 'k']):
    for sample, style in (('train', '--'), ('test', '-')):
        sample_score_mean = results['mean_%s_%s' % (sample, scorer)]
        sample_score_std = results['std_%s_%s' % (sample, scorer)]
        ax.fill_between(X_axis, sample_score_mean - sample_score_std,
                        sample_score_mean + sample_score_std,
                        alpha=0.1 if sample == 'test' else 0, color=color)
        ax.plot(X_axis, sample_score_mean, style, color=color,
                alpha=1 if sample == 'test' else 0.7,
                label="%s (%s)" % (scorer, sample))

best_index = np.nonzero(results['rank_test_%s' % scorer] == 1)[0][0]
best_score = results['mean_test_%s' % scorer][best_index]

# Plot a dotted vertical line at the best score for that scorer marked by x
ax.plot([X_axis[best_index], ] * 2, [0, best_score],
        linestyle='-.', color=color, marker='x', markeredgewidth=3, ms=8)

# Annotate the best score for that scorer
ax.annotate("%0.2f" % best_score,
            (X_axis[best_index], best_score + 0.005))

plt.legend(loc="best")
plt.grid(False)
plt.show()
```



Total running time of the script: (0 minutes 20.458 seconds)

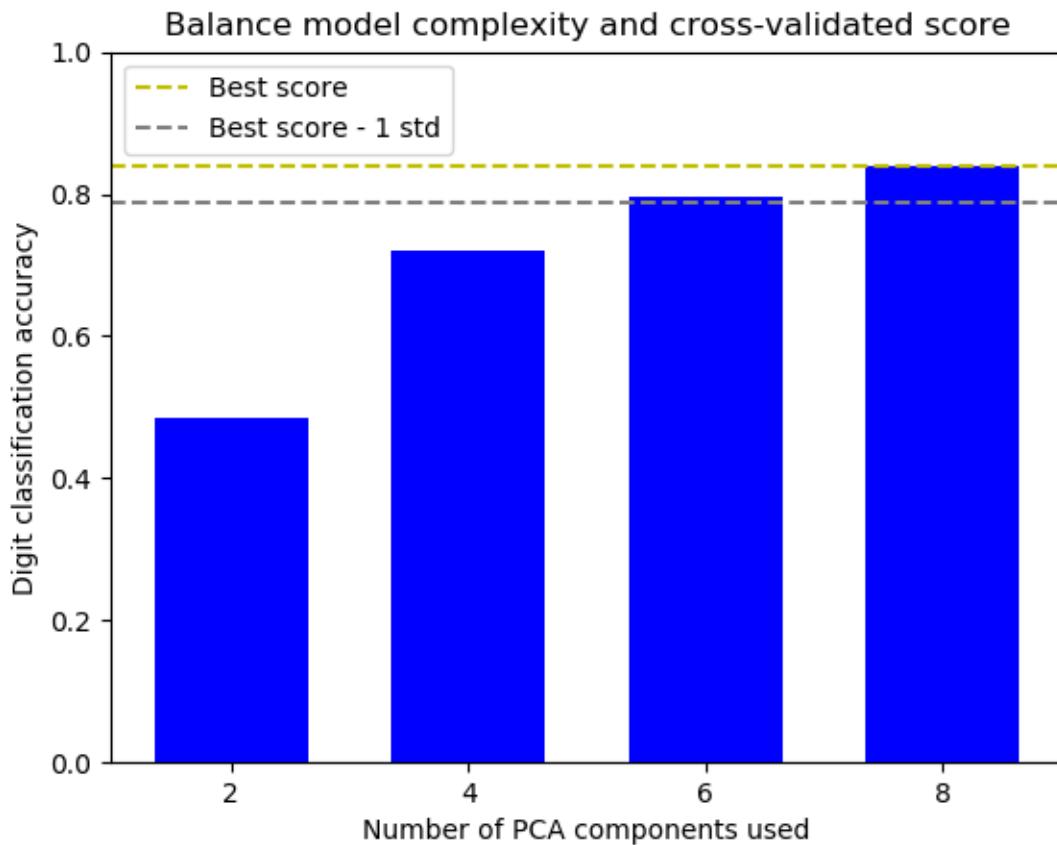
Note: Click [here](#) to download the full example code

5.21.10 Balance model complexity and cross-validated score

This example balances model complexity and cross-validated score by finding a decent accuracy within 1 standard deviation of the best accuracy score while minimising the number of PCA components [1].

The figure shows the trade-off between cross-validated score and the number of PCA components. The balanced case is when $n_{\text{components}}=6$ and $\text{accuracy}=0.80$, which falls into the range within 1 standard deviation of the best accuracy score.

[1] Hastie, T., Tibshirani, R., Friedman, J. (2001). Model Assessment and Selection. The Elements of Statistical Learning (pp. 219-260). New York, NY, USA: Springer New York Inc..



Out:

```
The best_index_ is 2
The n_components selected is 6
The corresponding accuracy score is 0.80
```

```
# Author: Wenhao Zhang <wenhaoz@ucla.edu>

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import load_digits
from sklearn.decomposition import PCA
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.svm import LinearSVC
```

```
def lower_bound(cv_results):
    """
    Calculate the lower bound within 1 standard deviation
    of the best `mean_test_scores`.

    Parameters
    -----
    cv_results : dict of numpy (masked) ndarrays
        See attribute cv_results_ of 'GridSearchCV'

    Returns
    -----
    float
        Lower bound within 1 standard deviation of the
        best `mean_test_score`.
    """
    best_score_idx = np.argmax(cv_results['mean_test_score'])

    return (cv_results['mean_test_score'][best_score_idx]
            - cv_results['std_test_score'][best_score_idx])

def best_low_complexity(cv_results):
    """
    Balance model complexity with cross-validated score.

    Parameters
    -----
    cv_results : dict of numpy (masked) ndarrays
        See attribute cv_results_ of 'GridSearchCV'.

    Returns
    -----
    int
        Index of a model that has the fewest PCA components
        while has its test score within 1 standard deviation of the best
        `mean_test_score`.
    """
    threshold = lower_bound(cv_results)
    candidate_idx = np.flatnonzero(cv_results['mean_test_score'] >= threshold)
    best_idx = candidate_idx[cv_results['param_reduce_dim__n_components']
                            [candidate_idx].argmin()]
    return best_idx

pipe = Pipeline([
    ('reduce_dim', PCA(random_state=42)),
    ('classify', LinearSVC(random_state=42)),
])
param_grid = {
    'reduce_dim__n_components': [2, 4, 6, 8]
}
grid = GridSearchCV(pipe, cv=10, n_jobs=1, param_grid=param_grid,
                     scoring='accuracy', refit=best_low_complexity)
```

```

digits = load_digits()
grid.fit(digits.data, digits.target)

n_components = grid.cv_results_['param_reduce_dim__n_components']
test_scores = grid.cv_results_['mean_test_score']

plt.figure()
plt.bar(n_components, test_scores, width=1.3, color='b')

lower = lower_bound(grid.cv_results_)
plt.axhline(np.max(test_scores), linestyle='--', color='y',
            label='Best score')
plt.axhline(lower, linestyle='--', color='.5', label='Best score - 1 std')

plt.title("Balance model complexity and cross-validated score")
plt.xlabel('Number of PCA components used')
plt.ylabel('Digit classification accuracy')
plt.xticks(n_components.tolist())
plt.ylim((0, 1.0))
plt.legend(loc='upper left')

best_index_ = grid.best_index_

print("The best_index_ is %d" % best_index_)
print("The n_components selected is %d" % n_components[best_index_])
print("The corresponding accuracy score is %.2f"
      % grid.cv_results_['mean_test_score'][best_index_])
plt.show()

```

Total running time of the script: (0 minutes 16.219 seconds)

Note: Click [here](#) to download the full example code

5.21.11 Sample pipeline for text feature extraction and evaluation

The dataset used in this example is the 20 newsgroups dataset which will be automatically downloaded and then cached and reused for the document classification example.

You can adjust the number of categories by giving their names to the dataset loader or setting them to None to get the 20 of them.

Here is a sample output of a run on a quad-core machine:

```

Loading 20 newsgroups dataset for categories:
['alt.atheism', 'talk.religion.misc']
1427 documents
2 categories

Performing grid search...
pipeline: ['vect', 'tfidf', 'clf']
parameters:
{'clf_alpha': (1.000000000000001e-05, 9.99999999999995e-07),
 'clf_max_iter': (10, 50, 80),
 'clf_penalty': ('l2', 'elasticnet'),
 'tfidf_use_idf': (True, False),

```

```
'vect__max_n': (1, 2),
'vect__max_df': (0.5, 0.75, 1.0),
'vect__max_features': (None, 5000, 10000, 50000)}
done in 1737.030s

Best score: 0.940
Best parameters set:
  clf__alpha: 9.99999999999995e-07
  clf__max_iter: 50
  clf__penalty: 'elasticnet'
  tfidf__use_idf: True
  vect__max_n: 2
  vect__max_df: 0.75
  vect__max_features: 50000
```

```
# Author: Olivier Grisel <olivier.grisel@ensta.org>
#         Peter Prettenhofer <peter.prettenhofer@gmail.com>
#         Mathieu Blondel <mathieu@mblondel.org>
# License: BSD 3 clause
from pprint import pprint
from time import time
import logging

from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline

print(__doc__)

# Display progress logs on stdout
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)s %(message)s')

# ##########
# Load some categories from the training set
categories = [
    'alt.atheism',
    'talk.religion.misc',
]
# Uncomment the following to do the analysis on all the categories
#categories = None

print("Loading 20 newsgroups dataset for categories:")
print(categories)

data = fetch_20newsgroups(subset='train', categories=categories)
print("%d documents" % len(data.filenames))
print("%d categories" % len(data.target_names))
print()

# ##########
# Define a pipeline combining a text feature extractor with a simple
# classifier
pipeline = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('clf', SGDClassifier(loss='hinge', penalty='l2', max_iter=50, tol=1e-3)),
])
```

```

('vect', CountVectorizer()),
('tfidf', TfidfTransformer()),
('clf', SGDClassifier(tol=1e-3)),
])

# uncommenting more parameters will give better exploring power but will
# increase processing time in a combinatorial way
parameters = {
    'vect__max_df': (0.5, 0.75, 1.0),
    # 'vect__max_features': (None, 5000, 10000, 50000),
    'vect__ngram_range': ((1, 1), (1, 2)), # unigrams or bigrams
    # 'tfidf__use_idf': (True, False),
    # 'tfidf__norm': ('l1', 'l2'),
    'clf__max_iter': (20,),
    'clf__alpha': (0.00001, 0.000001),
    'clf__penalty': ('l2', 'elasticnet'),
    # 'clf__max_iter': (10, 50, 80),
}

if __name__ == "__main__":
    # multiprocessing requires the fork to happen in a __main__ protected
    # block

    # find the best parameters for both the feature extraction and the
    # classifier
    grid_search = GridSearchCV(pipeline, parameters, cv=5,
                                n_jobs=-1, verbose=1)

    print("Performing grid search...")
    print("pipeline:", [name for name, _ in pipeline.steps])
    print("parameters:")
    pprint(parameters)
    t0 = time()
    grid_search.fit(data.data, data.target)
    print("done in %0.3fs" % (time() - t0))
    print()

    print("Best score: %0.3f" % grid_search.best_score_)
    print("Best parameters set:")
    best_parameters = grid_search.best_estimator_.get_params()
    for param_name in sorted(parameters.keys()):
        print("\t%s: %r" % (param_name, best_parameters[param_name]))

```

Total running time of the script: (0 minutes 0.000 seconds)

Note: Click [here](#) to download the full example code

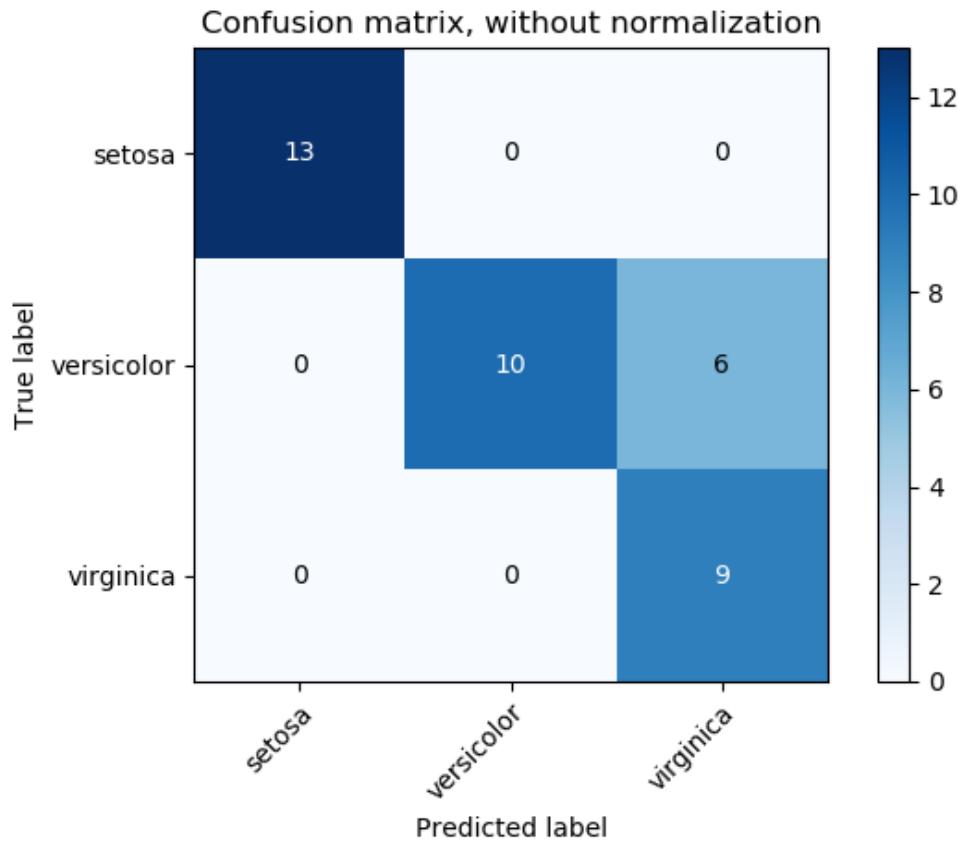
5.21.12 Confusion matrix

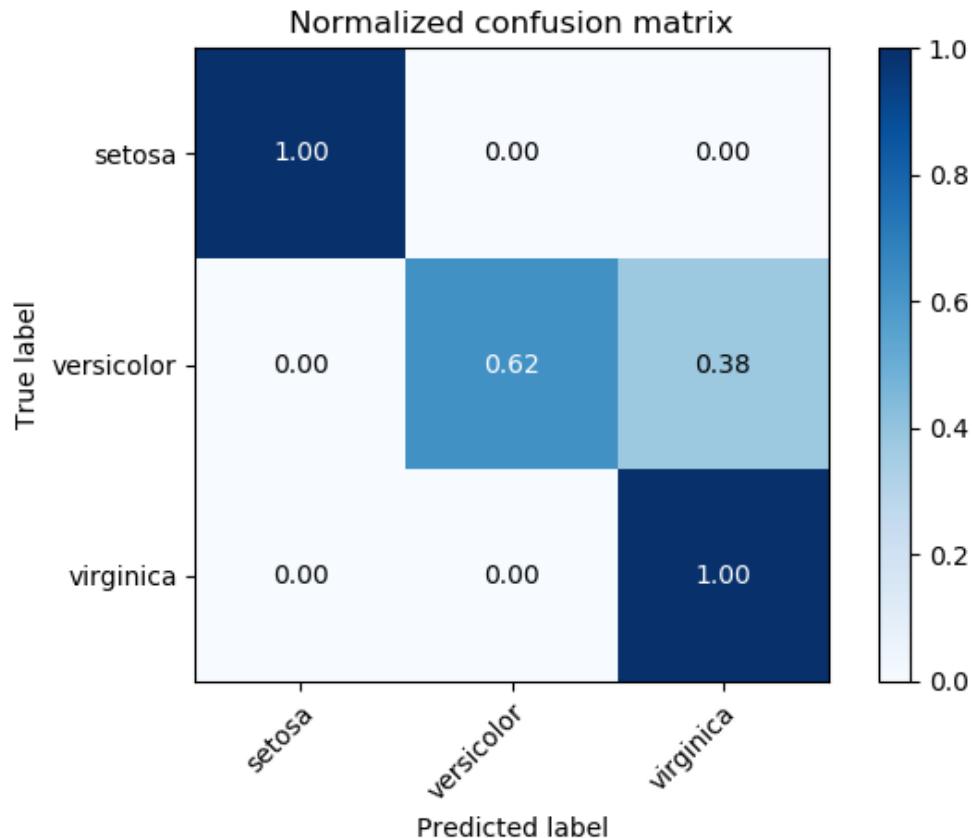
Example of confusion matrix usage to evaluate the quality of the output of a classifier on the iris data set. The diagonal elements represent the number of points for which the predicted label is equal to the true label, while off-diagonal elements are those that are mislabeled by the classifier. The higher the diagonal values of the confusion matrix the better, indicating many correct predictions.

The figures show the confusion matrix with and without normalization by class support size (number of elements in

each class). This kind of normalization can be interesting in case of class imbalance to have a more visual interpretation of which class is being misclassified.

Here the results are not as good as they could be as our choice for the regularization parameter C was not the best. In real life applications this parameter is usually chosen using [Tuning the hyper-parameters of an estimator](#).





Out:

```
Confusion matrix, without normalization
[[13  0  0]
 [ 0 10  6]
 [ 0  0  9]]
Normalized confusion matrix
[[1.   0.   0. ]
 [0.   0.62 0.38]
 [0.   0.   1. ]]
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn import svm, datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.utils.multiclass import unique_labels

# import some data to play with
```

```

iris = datasets.load_iris()
X = iris.data
y = iris.target
class_names = iris.target_names

# Split the data into a training set and a test set
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# Run classifier, using a model that is too regularized (C too low) to see
# the impact on the results
classifier = svm.SVC(kernel='linear', C=0.01)
y_pred = classifier.fit(X_train, y_train).predict(X_test)

def plot_confusion_matrix(y_true, y_pred, classes,
                           normalize=False,
                           title=None,
                           cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if not title:
        if normalize:
            title = 'Normalized confusion matrix'
        else:
            title = 'Confusion matrix, without normalization'

    # Compute confusion matrix
    cm = confusion_matrix(y_true, y_pred)
    # Only use the labels that appear in the data
    classes = classes[unique_labels(y_true, y_pred)]
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    fig, ax = plt.subplots()
    im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
    ax.figure.colorbar(im, ax=ax)
    # We want to show all ticks...
    ax.set(xticks=np.arange(cm.shape[1]),
           yticks=np.arange(cm.shape[0]),
           # ... and label them with the respective list entries
           xticklabels=classes, yticklabels=classes,
           title=title,
           ylabel='True label',
           xlabel='Predicted label')

    # Rotate the tick labels and set their alignment.
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
             rotation_mode="anchor")

    # Loop over data dimensions and create text annotations.
    fmt = '.2f' if normalize else 'd'

```

```

thresh = cm.max() / 2.
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        ax.text(j, i, format(cm[i, j], fmt),
                ha="center", va="center",
                color="white" if cm[i, j] > thresh else "black")
fig.tight_layout()
return ax

np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix
plot_confusion_matrix(y_test, y_pred, classes=class_names,
                      title='Confusion matrix, without normalization')

# Plot normalized confusion matrix
plot_confusion_matrix(y_test, y_pred, classes=class_names, normalize=True,
                      title='Normalized confusion matrix')

plt.show()

```

Total running time of the script: (0 minutes 0.217 seconds)

Note: Click [here](#) to download the full example code

5.21.13 Visualizing cross-validation behavior in scikit-learn

Choosing the right cross-validation object is a crucial part of fitting a model properly. There are many ways to split data into training and test sets in order to avoid model overfitting, to standardize the number of groups in test sets, etc.

This example visualizes the behavior of several common scikit-learn objects for comparison.

```

from sklearn.model_selection import (TimeSeriesSplit, KFold, ShuffleSplit,
                                    StratifiedKFold, GroupShuffleSplit,
                                    GroupKFold, StratifiedShuffleSplit)
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Patch
np.random.seed(1338)
cmap_data = plt.cm.Paired
cmap_cv = plt.cm.coolwarm
n_splits = 4

```

Visualize our data

First, we must understand the structure of our data. It has 100 randomly generated input datapoints, 3 classes split unevenly across datapoints, and 10 “groups” split evenly across datapoints.

As we'll see, some cross-validation objects do specific things with labeled data, others behave differently with grouped data, and others do not use this information.

To begin, we'll visualize our data.

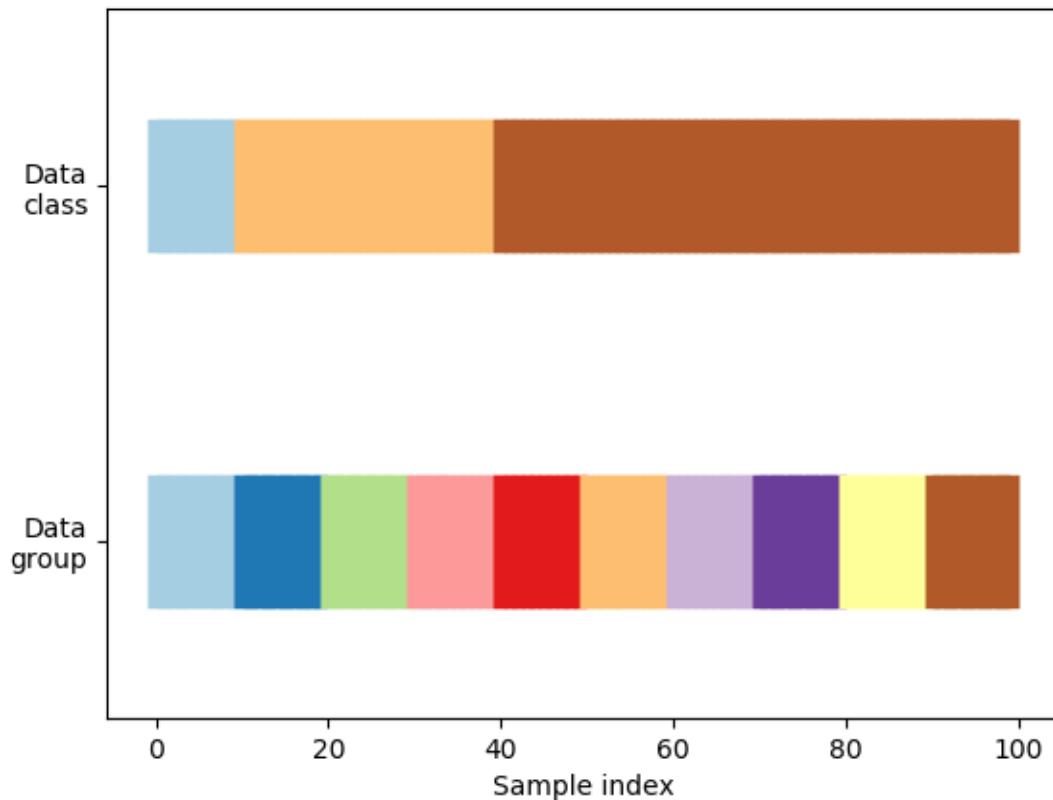
```
# Generate the class/group data
n_points = 100
X = np.random.randn(100, 10)

percentiles_classes = [.1, .3, .6]
y = np.hstack([[ii] * int(100 * perc)
               for ii, perc in enumerate(percentiles_classes)])

# Evenly spaced groups repeated once
groups = np.hstack([[ii] * 10 for ii in range(10)])


def visualize_groups(classes, groups, name):
    # Visualize dataset groups
    fig, ax = plt.subplots()
    ax.scatter(range(len(groups)), [.5] * len(groups), c=groups, marker='_',
               lw=50, cmap=cmap_data)
    ax.scatter(range(len(groups)), [3.5] * len(groups), c=classes, marker='_',
               lw=50, cmap=cmap_data)
    ax.set(ylim=[-1, 5], yticks=[.5, 3.5],
           yticklabels=['Data\ngroup', 'Data\niclass'], xlabel="Sample index")

visualize_groups(y, groups, 'no groups')
```



Define a function to visualize cross-validation behavior

We'll define a function that lets us visualize the behavior of each cross-validation object. We'll perform 4 splits of the data. On each split, we'll visualize the indices chosen for the training set (in blue) and the test set (in red).

```
def plot_cv_indices(cv, X, y, group, ax, n_splits, lw=10):
    """Create a sample plot for indices of a cross-validation object."""

    # Generate the training/testing visualizations for each CV split
    for ii, (tr, tt) in enumerate(cv.split(X=X, y=y, groups=group)):
        # Fill in indices with the training/test groups
        indices = np.array([np.nan] * len(X))
        indices[tt] = 1
        indices[tr] = 0

        # Visualize the results
        ax.scatter(range(len(indices)), [ii + .5] * len(indices),
                   c=indices, marker='_', lw=lw, cmap=cmap_cv,
                   vmin=-.2, vmax=1.2)

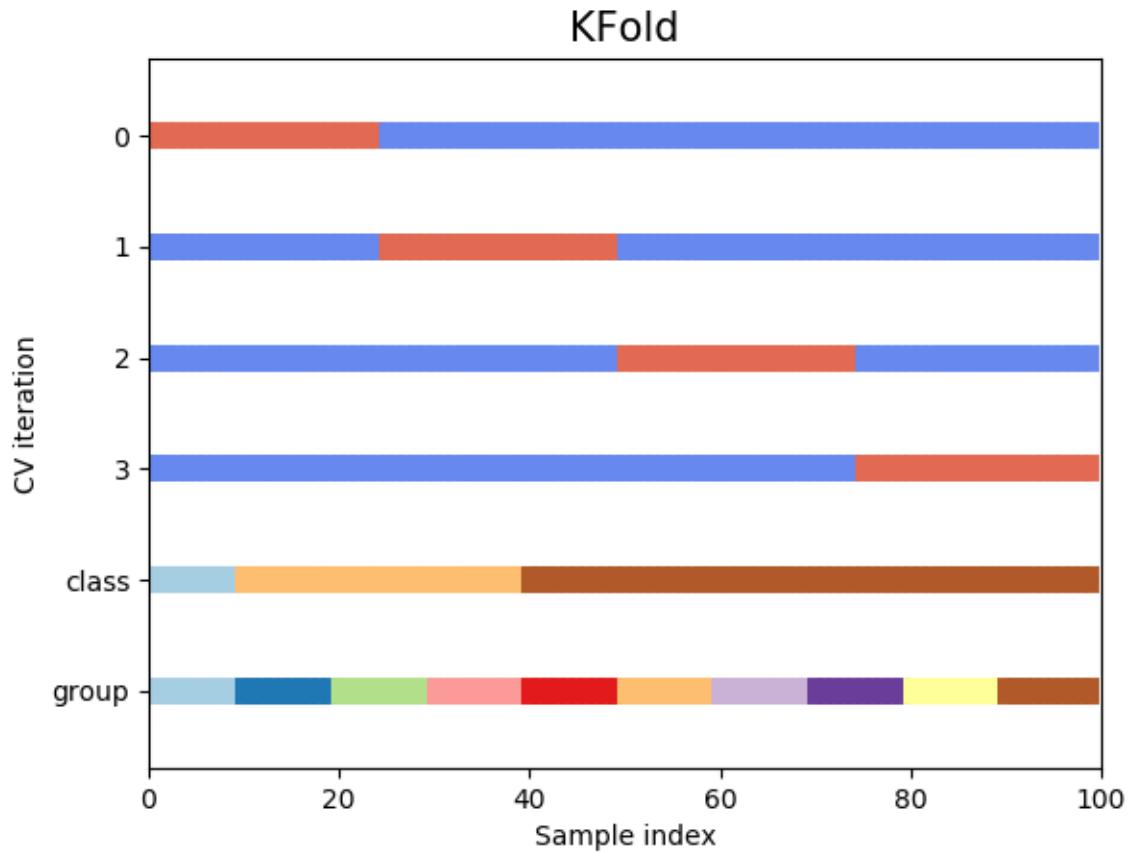
    # Plot the data classes and groups at the end
    ax.scatter(range(len(X)), [ii + 1.5] * len(X),
               c=y, marker='_', lw=lw, cmap=cmap_data)

    ax.scatter(range(len(X)), [ii + 2.5] * len(X),
               c=group, marker='_', lw=lw, cmap=cmap_data)

    # Formatting
    yticklabels = list(range(n_splits)) + ['class', 'group']
    ax.set(yticks=np.arange(n_splits+2) + .5, yticklabels=yticklabels,
           xlabel='Sample index', ylabel="CV iteration",
           ylim=[n_splits+2.2, -.2], xlim=[0, 100])
    ax.set_title('{0}'.format(type(cv).__name__), fontsize=15)
    return ax
```

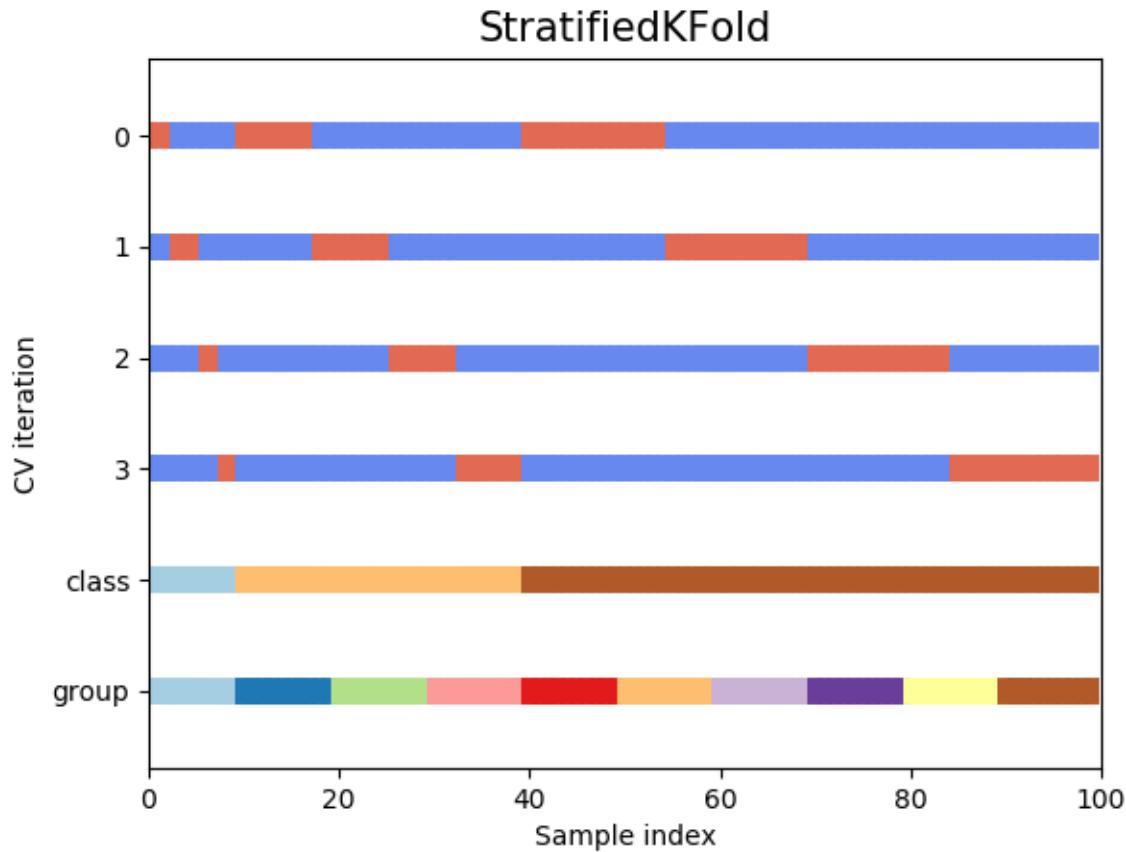
Let's see how it looks for the `KFold` cross-validation object:

```
fig, ax = plt.subplots()
cv = KFold(n_splits)
plot_cv_indices(cv, X, y, groups, ax, n_splits)
```



As you can see, by default the KFold cross-validation iterator does not take either datapoint class or group into consideration. We can change this by using the StratifiedKFold like so.

```
fig, ax = plt.subplots()
cv = StratifiedKFold(n_splits)
plot_cv_indices(cv, X, y, groups, ax, n_splits)
```



In this case, the cross-validation retained the same ratio of classes across each CV split. Next we'll visualize this behavior for a number of CV iterators.

Visualize cross-validation indices for many CV objects

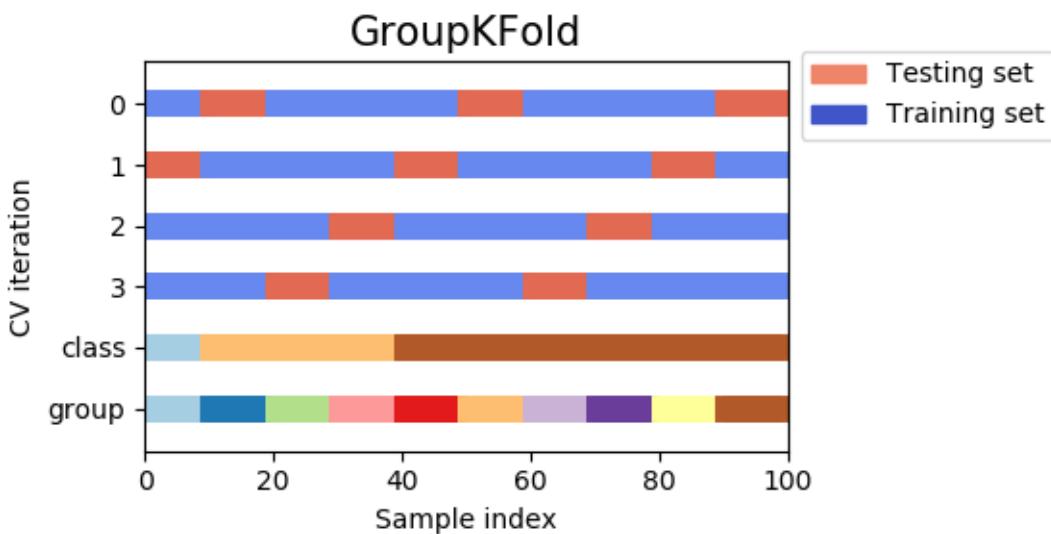
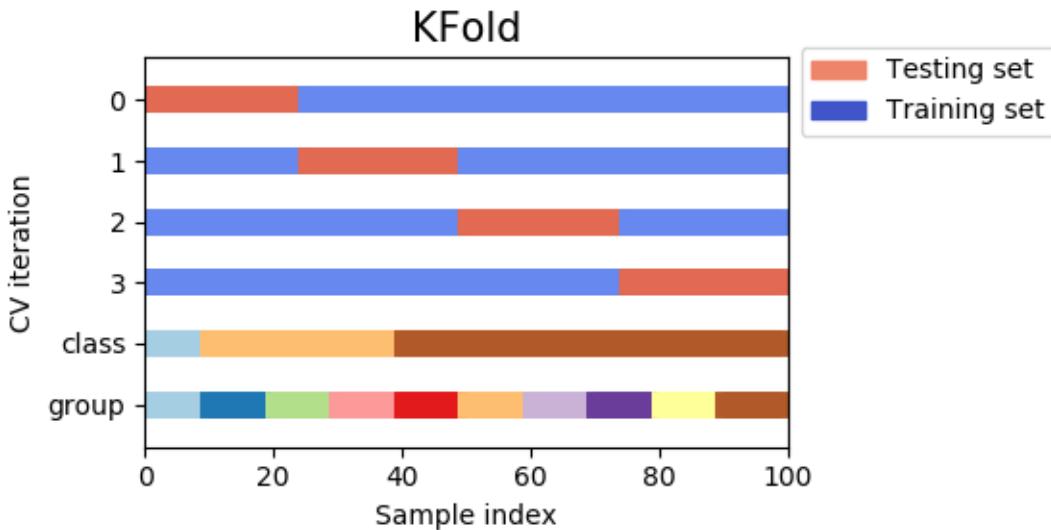
Let's visually compare the cross validation behavior for many scikit-learn cross-validation objects. Below we will loop through several common cross-validation objects, visualizing the behavior of each.

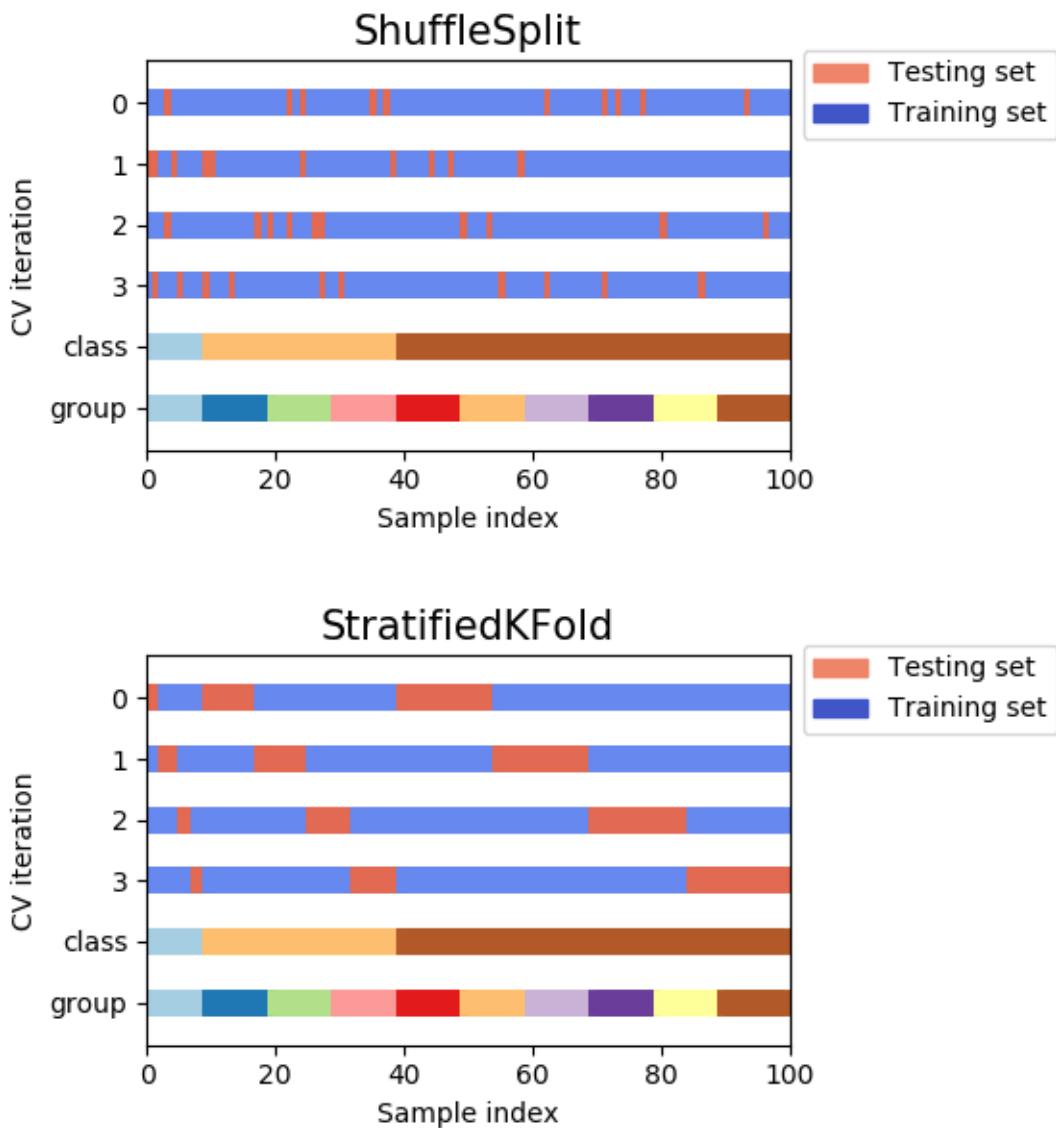
Note how some use the group/class information while others do not.

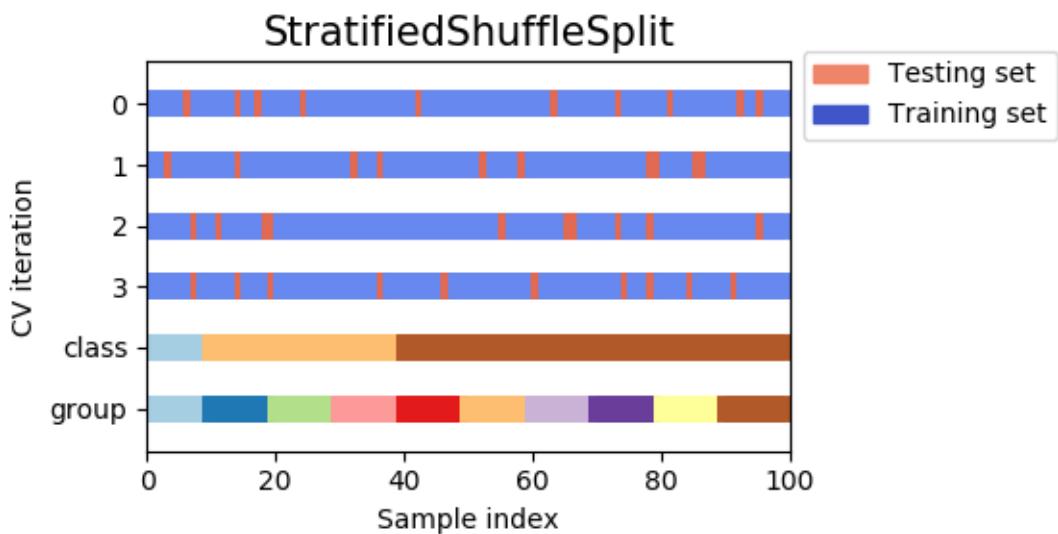
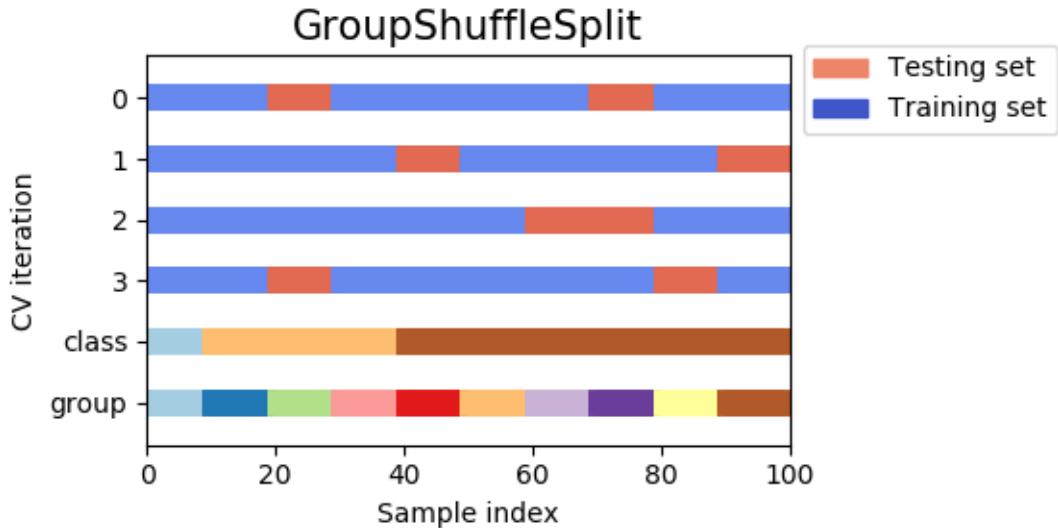
```
cvs = [KFold, GroupKFold, ShuffleSplit, StratifiedKFold,
       GroupShuffleSplit, StratifiedShuffleSplit, TimeSeriesSplit]

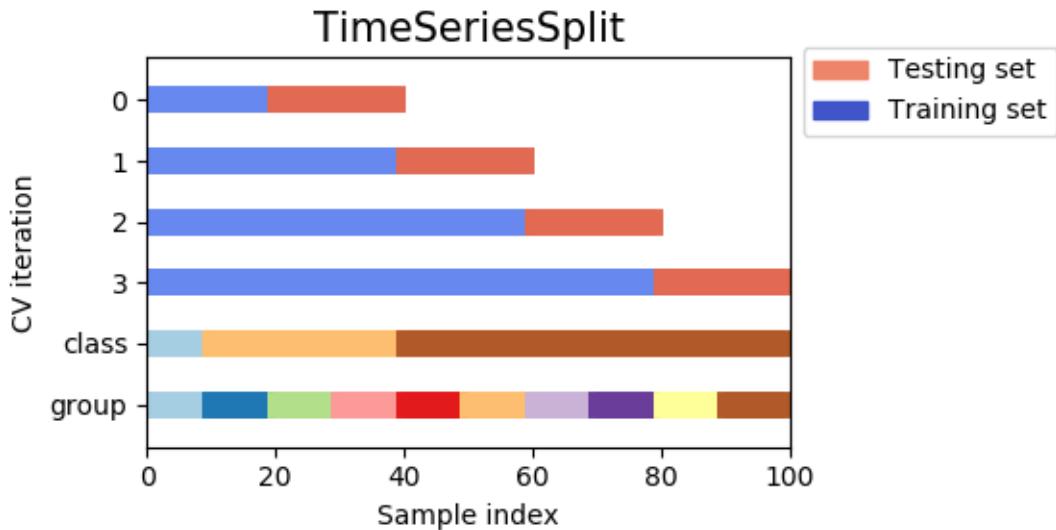
for cv in cvs:
    this_cv = cv(n_splits=n_splits)
    fig, ax = plt.subplots(figsize=(6, 3))
    plot_cv_indices(this_cv, X, y, groups, ax, n_splits)

    ax.legend([Patch(color=cmap_cv(.8)), Patch(color=cmap_cv(.02))],
              ['Testing set', 'Training set'], loc=(1.02, .8))
    # Make the legend fit
    plt.tight_layout()
    fig.subplots_adjust(right=.7)
plt.show()
```









Total running time of the script: (0 minutes 0.401 seconds)

Note: Click [here](#) to download the full example code

5.21.14 Receiver Operating Characteristic (ROC)

Example of Receiver Operating Characteristic (ROC) metric to evaluate classifier output quality.

ROC curves typically feature true positive rate on the Y axis, and false positive rate on the X axis. This means that the top left corner of the plot is the “ideal” point - a false positive rate of zero, and a true positive rate of one. This is not very realistic, but it does mean that a larger area under the curve (AUC) is usually better.

The “steepness” of ROC curves is also important, since it is ideal to maximize the true positive rate while minimizing the false positive rate.

Multiclass settings

ROC curves are typically used in binary classification to study the output of a classifier. In order to extend ROC curve and ROC area to multi-class or multi-label classification, it is necessary to binarize the output. One ROC curve can be drawn per label, but one can also draw a ROC curve by considering each element of the label indicator matrix as a binary prediction (micro-averaging).

Another evaluation measure for multi-class classification is macro-averaging, which gives equal weight to the classification of each label.

Note:

See also `sklearn.metrics.roc_auc_score`, *Receiver Operating Characteristic (ROC) with cross validation*.

```
print(__doc__)
import numpy as np
```

```

import matplotlib.pyplot as plt
from itertools import cycle

from sklearn import svm, datasets
from sklearn.metrics import roc_curve, auc
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import label_binarize
from sklearn.multiclass import OneVsRestClassifier
from scipy import interp

# Import some data to play with
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Binarize the output
y = label_binarize(y, classes=[0, 1, 2])
n_classes = y.shape[1]

# Add noisy features to make the problem harder
random_state = np.random.RandomState(0)
n_samples, n_features = X.shape
X = np.c_[X, random_state.randn(n_samples, 200 * n_features)]

# shuffle and split training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.5,
                                                    random_state=0)

# Learn to predict each class against the other
classifier = OneVsRestClassifier(svm.SVC(kernel='linear', probability=True,
                                           random_state=random_state))
y_score = classifier.fit(X_train, y_train).decision_function(X_test)

# Compute ROC curve and ROC area for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_score[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(y_test.ravel(), y_score.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

```

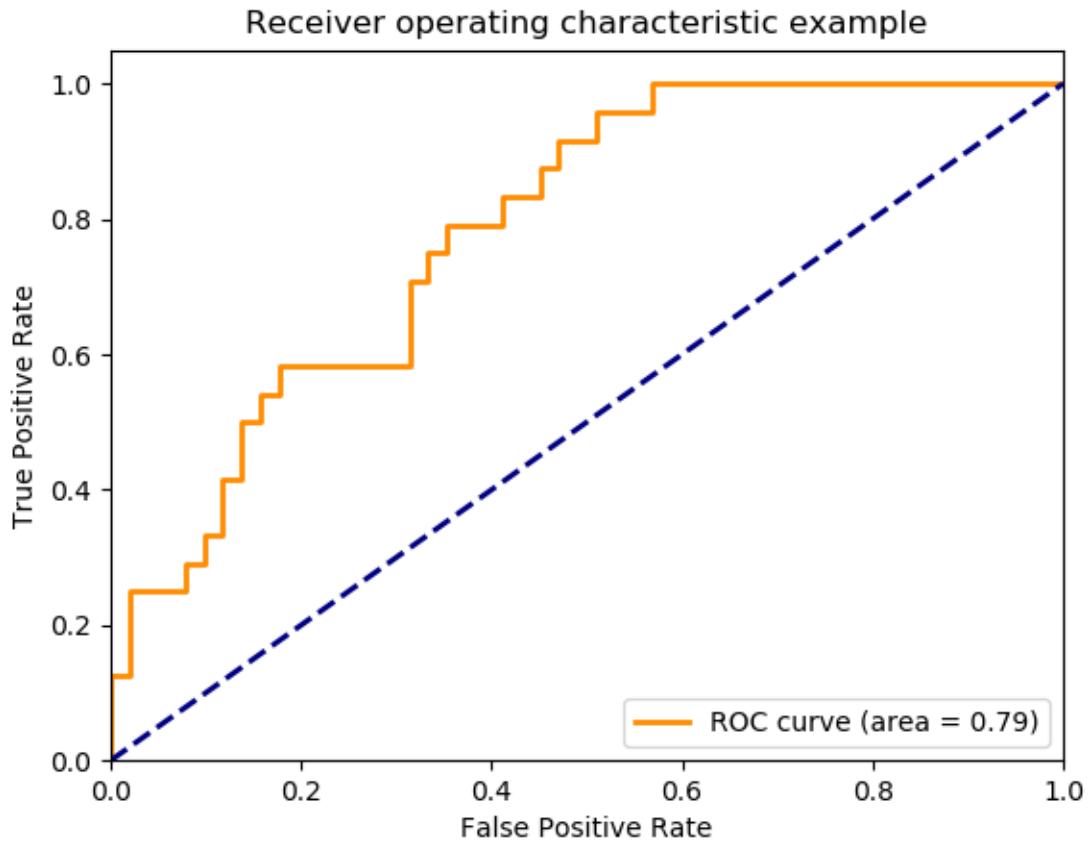
Plot of a ROC curve for a specific class

```

plt.figure()
lw = 2
plt.plot(fpr[2], tpr[2], color='darkorange',
          lw=lw, label='ROC curve (area = %0.2f)' % roc_auc[2])
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")

```

```
plt.show()
```



Plot ROC curves for the multiclass problem

```
# Compute macro-average ROC curve and ROC area

# First aggregate all false positive rates
all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))

# Then interpolate all ROC curves at this points
mean_tpr = np.zeros_like(all_fpr)
for i in range(n_classes):
    mean_tpr += interp(all_fpr, fpr[i], tpr[i])

# Finally average it and compute AUC
mean_tpr /= n_classes

fpr["macro"] = all_fpr
tpr["macro"] = mean_tpr
roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

# Plot all ROC curves
plt.figure()
plt.plot(fpr["micro"], tpr["micro"],
          label='micro-average ROC curve (area = {0:0.2f})'
          ''.format(roc_auc["micro"]),
          color='darkorange', linestyle='solid')
```

```

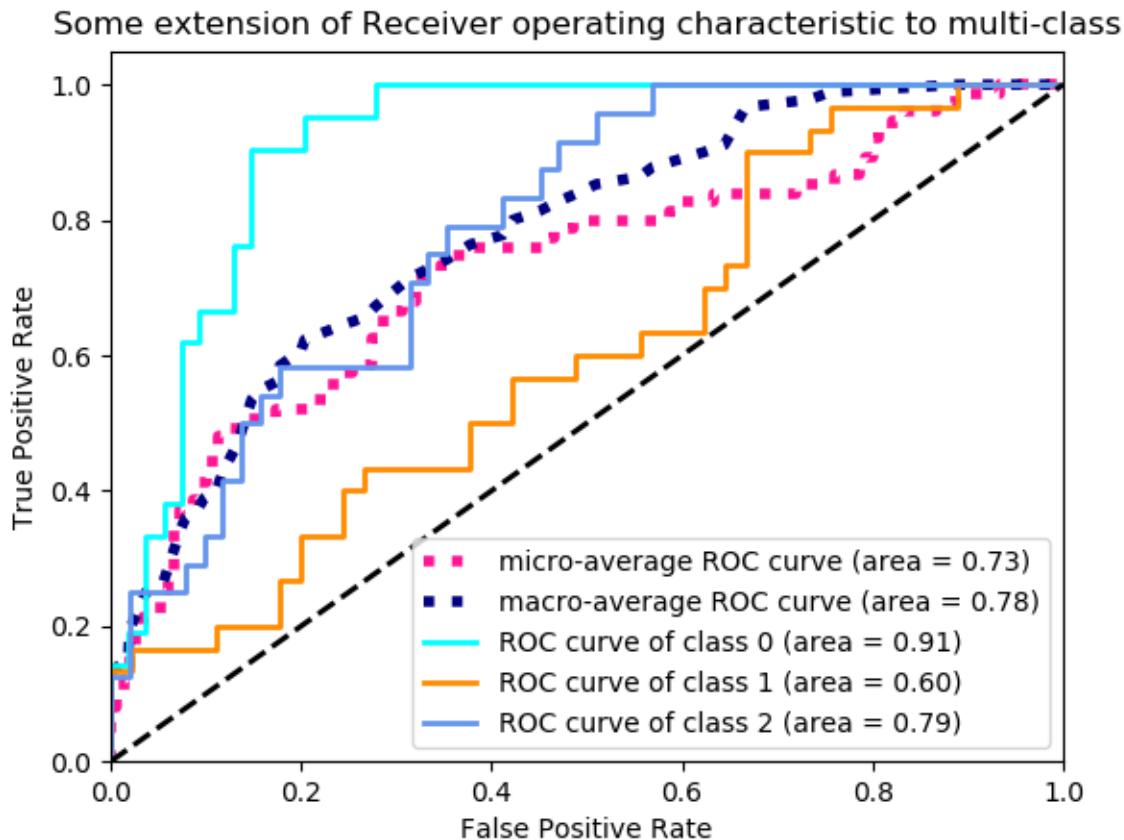
        color='deppink', linestyle=':', linewidth=4)

plt.plot(fpr["macro"], tpr["macro"],
          label='macro-average ROC curve (area = {0:0.2f})'
                  ''.format(roc_auc["macro"]),
          color='navy', linestyle=':', linewidth=4)

colors = cycle(['aqua', 'darkorange', 'cornflowerblue'])
for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=lw,
              label='ROC curve of class {0} (area = {1:0.2f})'
                  ''.format(i, roc_auc[i]))

plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Some extension of Receiver operating characteristic to multi-class')
plt.legend(loc="lower right")
plt.show()

```

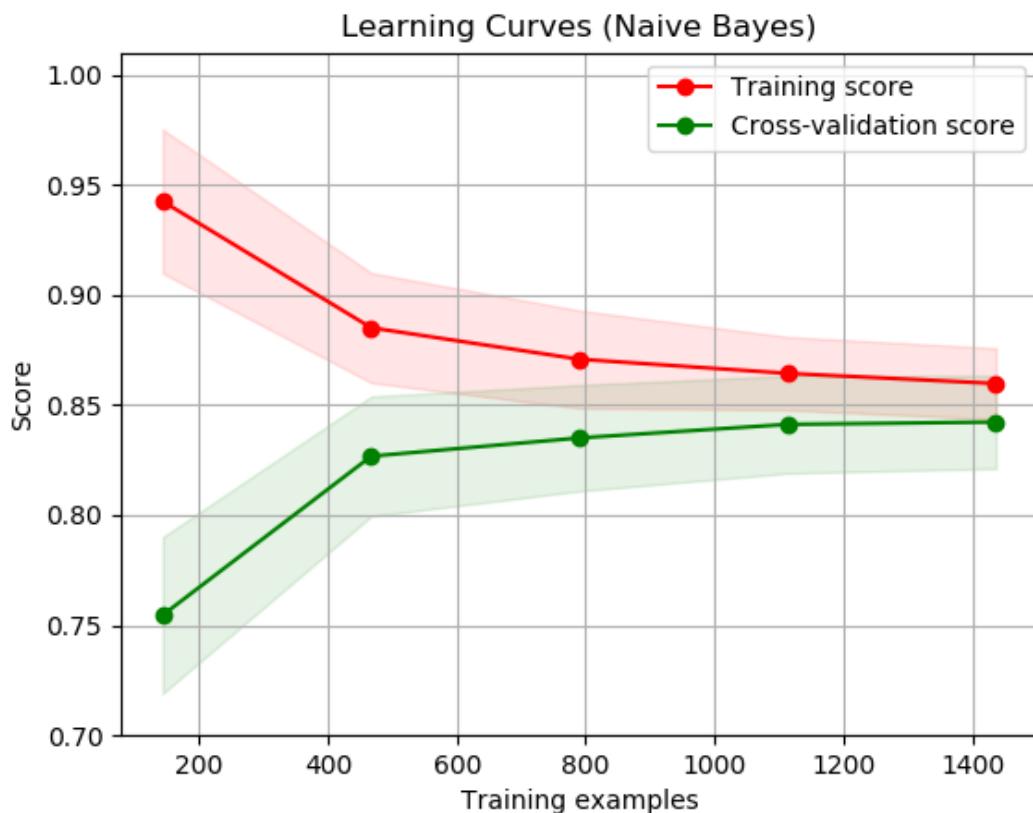


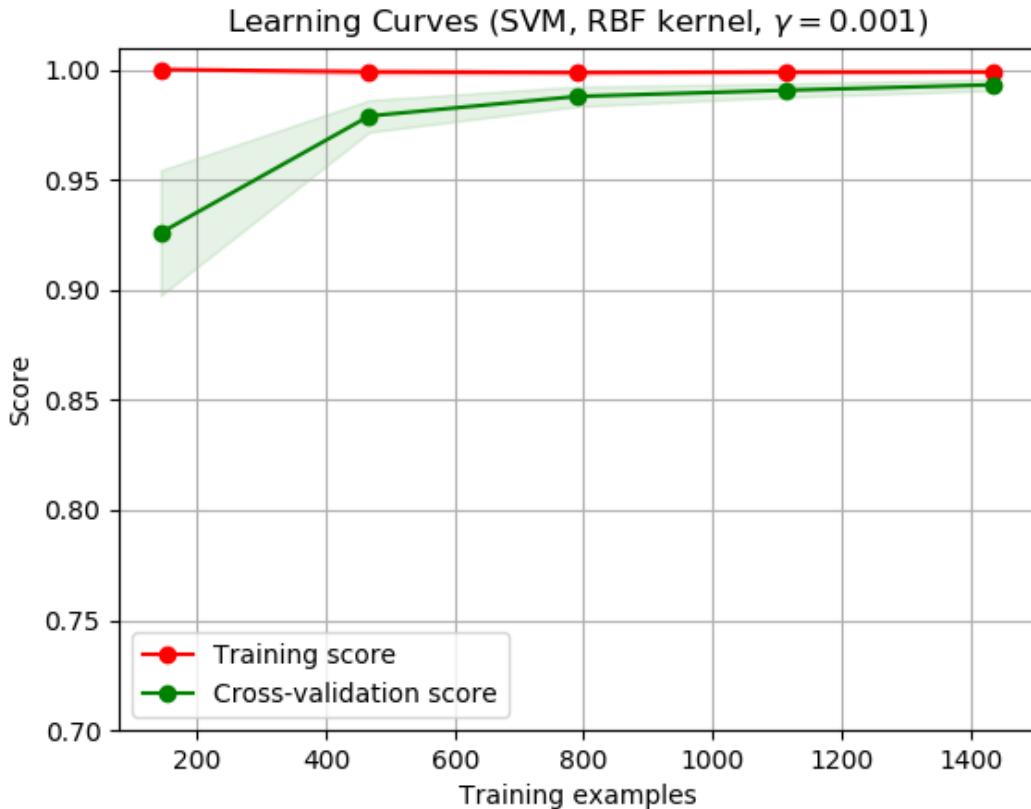
Total running time of the script: (0 minutes 0.145 seconds)

Note: Click [here](#) to download the full example code

5.21.15 Plotting Learning Curves

On the left side the learning curve of a naive Bayes classifier is shown for the digits dataset. Note that the training score and the cross-validation score are both not very good at the end. However, the shape of the curve can be found in more complex datasets very often: the training score is very high at the beginning and decreases and the cross-validation score is very low at the beginning and increases. On the right side we see the learning curve of an SVM with RBF kernel. We can see clearly that the training score is still around the maximum and the validation score could be increased with more training samples.





```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.datasets import load_digits
from sklearn.model_selection import learning_curve
from sklearn.model_selection import ShuffleSplit

def plot_learning_curve(estimator, title, X, y, ylim=None, cv=None,
                       n_jobs=None, train_sizes=np.linspace(.1, 1.0, 5)):
    """
    Generate a simple plot of the test and training learning curve.

    Parameters
    ----------
    estimator : object type that implements the "fit" and "predict" methods
        An object of that type which is cloned for each validation.

    title : string
        Title for the chart.

    X : array-like, shape (n_samples, n_features)
        Training vector, where n_samples is the number of samples and
        n_features is the number of features.

    y : array-like, shape (n_samples)
        Target values.
    ylim : tuple, shape (ymin, ymax), optional
        Limit the range of the vertical axis.
    cv : int, cross-validation generator or an iterable, optional
        If an integer is passed, it is treated as a number of folds.
        Must be None if shuffle is False.
    n_jobs : integer, optional
        Number of jobs to run in parallel.
        -1 means using all processors.
    train_sizes : array-like, shape (n_ticks,), dtype float or int
        Minimum number of training samples per bin.
        Maximum size is set in cv (can be passed as n_folds).
    """
    plt.figure()
    plt.title(title)
    if ylim is not None:
        plt.ylim(*ylim)
    plt.xlabel("Training examples")
    plt.ylabel("Score")
    train_sizes, train_scores, test_scores = learning_curve(
        estimator, X, y, cv=cv, n_jobs=n_jobs, train_sizes=train_sizes)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)

    plt.grid()

    plt.fill_between(train_sizes, train_scores_mean - train_scores_std,
                    train_scores_mean + train_scores_std, alpha=0.1,
                    color="green")
    plt.fill_between(train_sizes, test_scores_mean - test_scores_std,
                    test_scores_mean + test_scores_std, alpha=0.1, color="red")
    plt.plot(train_sizes, train_scores_mean, 'o-', color="green",
             label="Training score")
    plt.plot(train_sizes, test_scores_mean, 'o-', color="red",
             label="Cross-validation score")

    plt.legend(loc="best")
    return plt

```

```

y : array-like, shape (n_samples) or (n_samples, n_features), optional
    Target relative to X for classification or regression;
    None for unsupervised learning.

ylim : tuple, shape (ymin, ymax), optional
    Defines minimum and maximum yvalues plotted.

cv : int, cross-validation generator or an iterable, optional
    Determines the cross-validation splitting strategy.
    Possible inputs for cv are:
        - None, to use the default 3-fold cross-validation,
        - integer, to specify the number of folds.
        - :term:`CV splitter`,
        - An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if ``y`` is binary or multiclass,
:class:`StratifiedKFold` used. If the estimator is not a classifier
or if ``y`` is neither binary nor multiclass, :class:`KFold` is used.

Refer :ref:`User Guide <cross_validation>` for the various
cross-validators that can be used here.

n_jobs : int or None, optional (default=None)
    Number of jobs to run in parallel.
    ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
    ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
    for more details.

train_sizes : array-like, shape (n_ticks,), dtype float or int
    Relative or absolute numbers of training examples that will be used to
    generate the learning curve. If the dtype is float, it is regarded as a
    fraction of the maximum size of the training set (that is determined
    by the selected validation method), i.e. it has to be within (0, 1].
    Otherwise it is interpreted as absolute sizes of the training sets.
    Note that for classification the number of samples usually have to
    be big enough to contain at least one sample from each class.
    (default: np.linspace(0.1, 1.0, 5))

"""
plt.figure()
plt.title(title)
if ylim is not None:
    plt.ylim(*ylim)
plt.xlabel("Training examples")
plt.ylabel("Score")
train_sizes, train_scores, test_scores = learning_curve(
    estimator, X, y, cv=cv, n_jobs=n_jobs, train_sizes=train_sizes)
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)
plt.grid()

plt.fill_between(train_sizes, train_scores_mean - train_scores_std,
                train_scores_mean + train_scores_std, alpha=0.1,
                color="r")
plt.fill_between(train_sizes, test_scores_mean - test_scores_std,
                test_scores_mean + test_scores_std, alpha=0.1, color="g")

```

```
plt.plot(train_sizes, train_scores_mean, 'o-', color="r",
         label="Training score")
plt.plot(train_sizes, test_scores_mean, 'o-', color="g",
         label="Cross-validation score")

plt.legend(loc="best")
return plt

digits = load_digits()
X, y = digits.data, digits.target

title = "Learning Curves (Naive Bayes)"
# Cross validation with 100 iterations to get smoother mean test and train
# score curves, each time with 20% data randomly selected as a validation set.
cv = ShuffleSplit(n_splits=100, test_size=0.2, random_state=0)

estimator = GaussianNB()
plot_learning_curve(estimator, title, X, y, ylim=(0.7, 1.01), cv=cv, n_jobs=4)

title = r"Learning Curves (SVM, RBF kernel, $\gamma=0.001$)"
# SVC is more expensive so we do a lower number of CV iterations:
cv = ShuffleSplit(n_splits=10, test_size=0.2, random_state=0)
estimator = SVC(gamma=0.001)
plot_learning_curve(estimator, title, X, y, (0.7, 1.01), cv=cv, n_jobs=4)

plt.show()
```

Total running time of the script: (0 minutes 2.856 seconds)

Note: Click [here](#) to download the full example code

5.21.16 Precision-Recall

Example of Precision-Recall metric to evaluate classifier output quality.

Precision-Recall is a useful measure of success of prediction when the classes are very imbalanced. In information retrieval, precision is a measure of result relevancy, while recall is a measure of how many truly relevant results are returned.

The precision-recall curve shows the tradeoff between precision and recall for different threshold. A high area under the curve represents both high recall and high precision, where high precision relates to a low false positive rate, and high recall relates to a low false negative rate. High scores for both show that the classifier is returning accurate results (high precision), as well as returning a majority of all positive results (high recall).

A system with high recall but low precision returns many results, but most of its predicted labels are incorrect when compared to the training labels. A system with high precision but low recall is just the opposite, returning very few results, but most of its predicted labels are correct when compared to the training labels. An ideal system with high precision and high recall will return many results, with all results labeled correctly.

Precision (P) is defined as the number of true positives (T_p) over the number of true positives plus the number of false positives (F_p).

$$P = \frac{T_p}{T_p + F_p}$$

Recall (R) is defined as the number of true positives (T_p) over the number of true positives plus the number of false negatives (F_n).

$$R = \frac{T_p}{T_p + F_n}$$

These quantities are also related to the (F_1) score, which is defined as the harmonic mean of precision and recall.

$$F1 = 2 \frac{P \times R}{P + R}$$

Note that the precision may not decrease with recall. The definition of precision ($\frac{T_p}{T_p + F_p}$) shows that lowering the threshold of a classifier may increase the denominator, by increasing the number of results returned. If the threshold was previously set too high, the new results may all be true positives, which will increase precision. If the previous threshold was about right or too low, further lowering the threshold will introduce false positives, decreasing precision.

Recall is defined as $\frac{T_p}{T_p + F_n}$, where $T_p + F_n$ does not depend on the classifier threshold. This means that lowering the classifier threshold may increase recall, by increasing the number of true positive results. It is also possible that lowering the threshold may leave recall unchanged, while the precision fluctuates.

The relationship between recall and precision can be observed in the staircase area of the plot - at the edges of these steps a small change in the threshold considerably reduces precision, with only a minor gain in recall.

Average precision (AP) summarizes such a plot as the weighted mean of precisions achieved at each threshold, with the increase in recall from the previous threshold used as the weight:

$$AP = \sum_n (R_n - R_{n-1}) P_n$$

where P_n and R_n are the precision and recall at the nth threshold. A pair (R_k, P_k) is referred to as an *operating point*.

AP and the trapezoidal area under the operating points (`sklearn.metrics.auc`) are common ways to summarize a precision-recall curve that lead to different results. Read more in the [User Guide](#).

Precision-recall curves are typically used in binary classification to study the output of a classifier. In order to extend the precision-recall curve and average precision to multi-class or multi-label classification, it is necessary to binarize the output. One curve can be drawn per label, but one can also draw a precision-recall curve by considering each element of the label indicator matrix as a binary prediction (micro-averaging).

Note:

See also `sklearn.metrics.average_precision_score`, `sklearn.metrics.recall_score`, `sklearn.metrics.precision_score`, `sklearn.metrics.f1_score`

In binary classification settings

Create simple data

Try to differentiate the two first classes of the iris data

```
from sklearn import svm, datasets
from sklearn.model_selection import train_test_split
import numpy as np

iris = datasets.load_iris()
X = iris.data
y = iris.target

# Add noisy features
random_state = np.random.RandomState(0)
n_samples, n_features = X.shape
```

```
X = np.c_[X, random_state.randn(n_samples, 200 * n_features)]  
  
# Limit to the two first classes, and split into training and test  
X_train, X_test, y_train, y_test = train_test_split(X[y < 2], y[y < 2],  
                                                test_size=.5,  
                                                random_state=random_state)  
  
# Create a simple classifier  
classifier = svm.LinearSVC(random_state=random_state)  
classifier.fit(X_train, y_train)  
y_score = classifier.decision_function(X_test)
```

Compute the average precision score

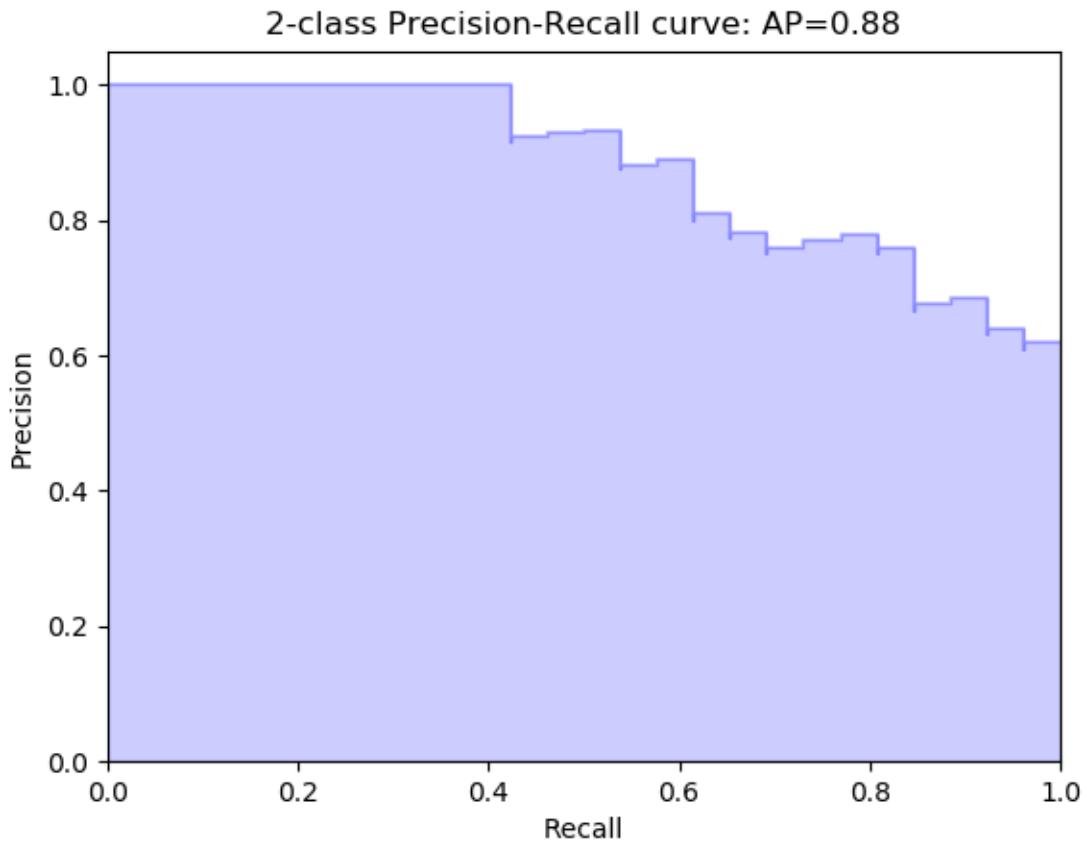
```
from sklearn.metrics import average_precision_score  
average_precision = average_precision_score(y_test, y_score)  
  
print('Average precision-recall score: {:.2f}'.format(  
    average_precision))
```

Out:

```
Average precision-recall score: 0.88
```

Plot the Precision-Recall curve

```
from sklearn.metrics import precision_recall_curve  
import matplotlib.pyplot as plt  
from inspect import signature  
  
precision, recall, _ = precision_recall_curve(y_test, y_score)  
  
# In matplotlib < 1.5, plt.fill_between does not have a 'step' argument  
step_kwargs = ({'step': 'post'}  
              if 'step' in signature(plt.fill_between).parameters  
              else {})  
plt.step(recall, precision, color='b', alpha=0.2,  
         where='post')  
plt.fill_between(recall, precision, alpha=0.2, color='b', **step_kwargs)  
  
plt.xlabel('Recall')  
plt.ylabel('Precision')  
plt.ylim([0.0, 1.05])  
plt.xlim([0.0, 1.0])  
plt.title('2-class Precision-Recall curve: AP={:.2f}'.format(  
    average_precision))
```



In multi-label settings

Create multi-label data, fit, and predict

We create a multi-label dataset, to illustrate the precision-recall in multi-label settings

```
from sklearn.preprocessing import label_binarize

# Use label_binarize to be multi-label like settings
Y = label_binarize(y, classes=[0, 1, 2])
n_classes = Y.shape[1]

# Split into training and test
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=.5,
                                                    random_state=random_state)

# We use OneVsRestClassifier for multi-label prediction
from sklearn.multiclass import OneVsRestClassifier

# Run classifier
classifier = OneVsRestClassifier(svm.LinearSVC(random_state=random_state))
classifier.fit(X_train, Y_train)
y_score = classifier.decision_function(X_test)
```

The average precision score in multi-label settings

```
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import average_precision_score

# For each class
precision = dict()
recall = dict()
average_precision = dict()
for i in range(n_classes):
    precision[i], recall[i], _ = precision_recall_curve(Y_test[:, i],
                                                       y_score[:, i])
    average_precision[i] = average_precision_score(Y_test[:, i], y_score[:, i])

# A "micro-average": quantifying score on all classes jointly
precision["micro"], recall["micro"], _ = precision_recall_curve(Y_test.ravel(),
    y_score.ravel())
average_precision["micro"] = average_precision_score(Y_test, y_score,
                                                      average="micro")
print('Average precision score, micro-averaged over all classes: {0:0.2f}'
      .format(average_precision["micro"]))
```

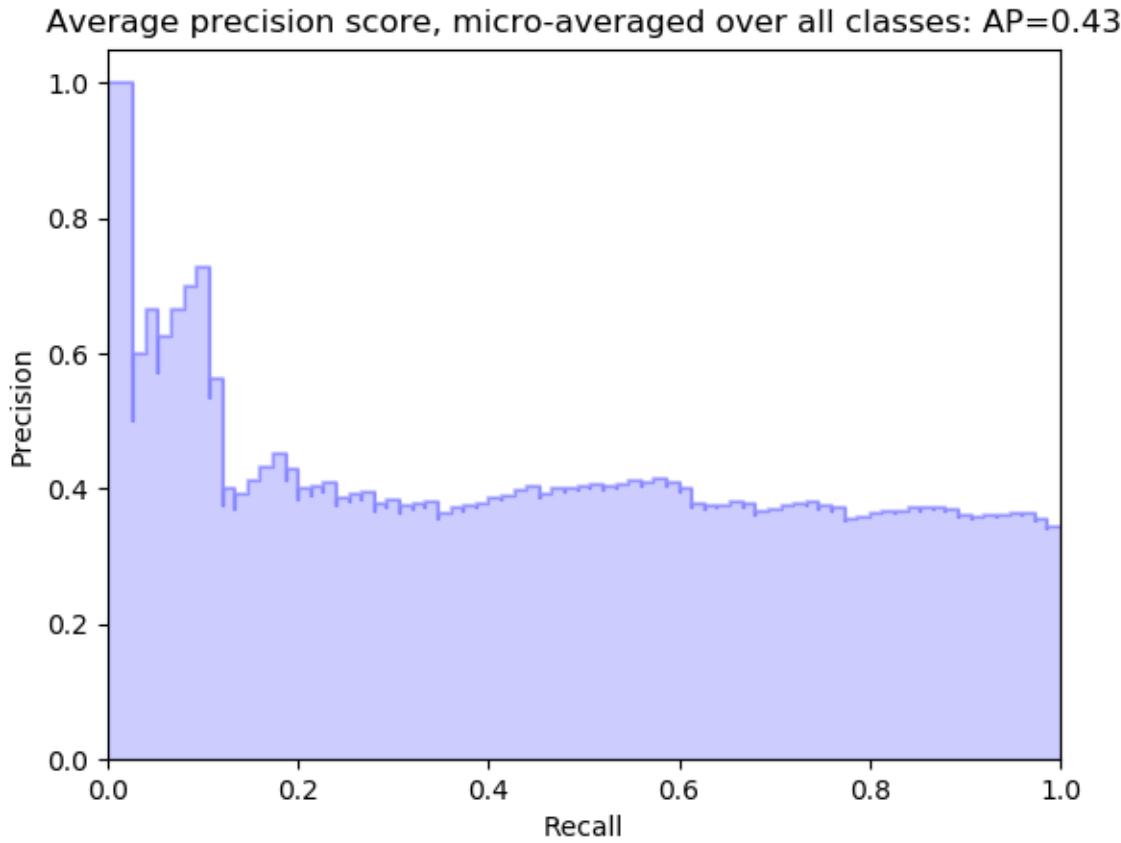
Out:

```
Average precision score, micro-averaged over all classes: 0.43
```

Plot the micro-averaged Precision-Recall curve

```
plt.figure()
plt.step(recall['micro'], precision['micro'], color='b', alpha=0.2,
          where='post')
plt.fill_between(recall["micro"], precision["micro"], alpha=0.2, color='b',
                 **step_kwargs)

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title(
    'Average precision score, micro-averaged over all classes: AP={0:0.2f}'.
    format(average_precision["micro"]))
```



Plot Precision-Recall curve for each class and iso-f1 curves

```
from itertools import cycle
# setup plot details
colors = cycle(['navy', 'turquoise', 'darkorange', 'cornflowerblue', 'teal'])

plt.figure(figsize=(7, 8))
f_scores = np.linspace(0.2, 0.8, num=4)
lines = []
labels = []
for f_score in f_scores:
    x = np.linspace(0.01, 1)
    y = f_score * x / (2 * x - f_score)
    l, = plt.plot(x[y >= 0], y[y >= 0], color='gray', alpha=0.2)
    plt.annotate('f1={0:0.1f}'.format(f_score), xy=(0.9, y[45] + 0.02))

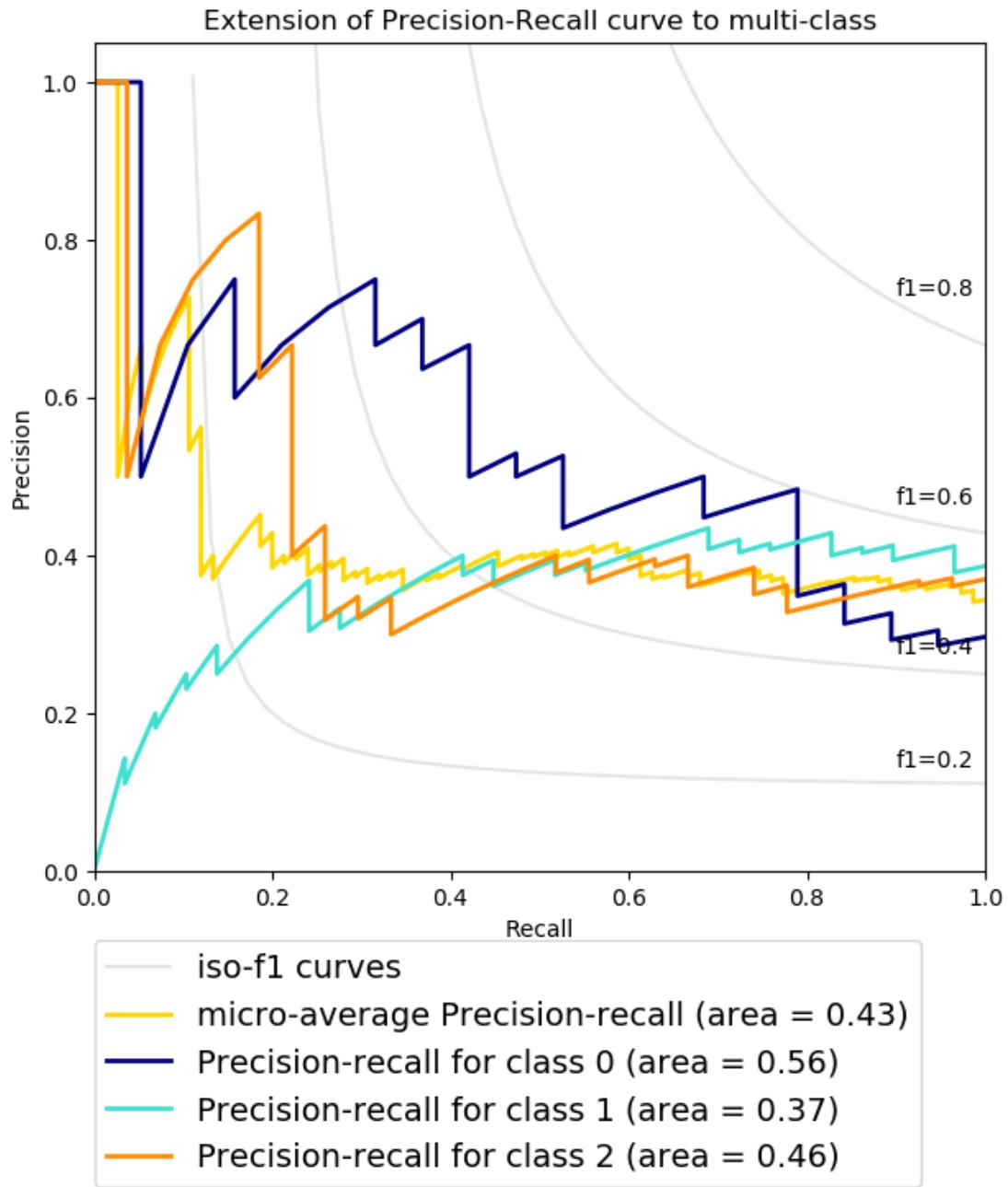
lines.append(l)
labels.append('iso-f1 curves')
l, = plt.plot(recall["micro"], precision["micro"], color='gold', lw=2)
lines.append(l)
labels.append('micro-average Precision-recall (area = {0:0.2f})'
              ''.format(average_precision["micro"]))

for i, color in zip(range(n_classes), colors):
```

```
l, = plt.plot(recall[i], precision[i], color=color, lw=2)
lines.append(l)
labels.append('Precision-recall for class {0} (area = {1:0.2f})' +
              ''.format(i, average_precision[i]))
```

```
fig = plt.gcf()
fig.subplots_adjust(bottom=0.25)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Extension of Precision-Recall curve to multi-class')
plt.legend(lines, labels, loc=(0, -.38), prop=dict(size=14))
```

```
plt.show()
```



Total running time of the script: (0 minutes 0.064 seconds)

5.22 Multioutput methods

Examples concerning the `sklearn.multioutput` module.

Note: Click [here](#) to download the full example code

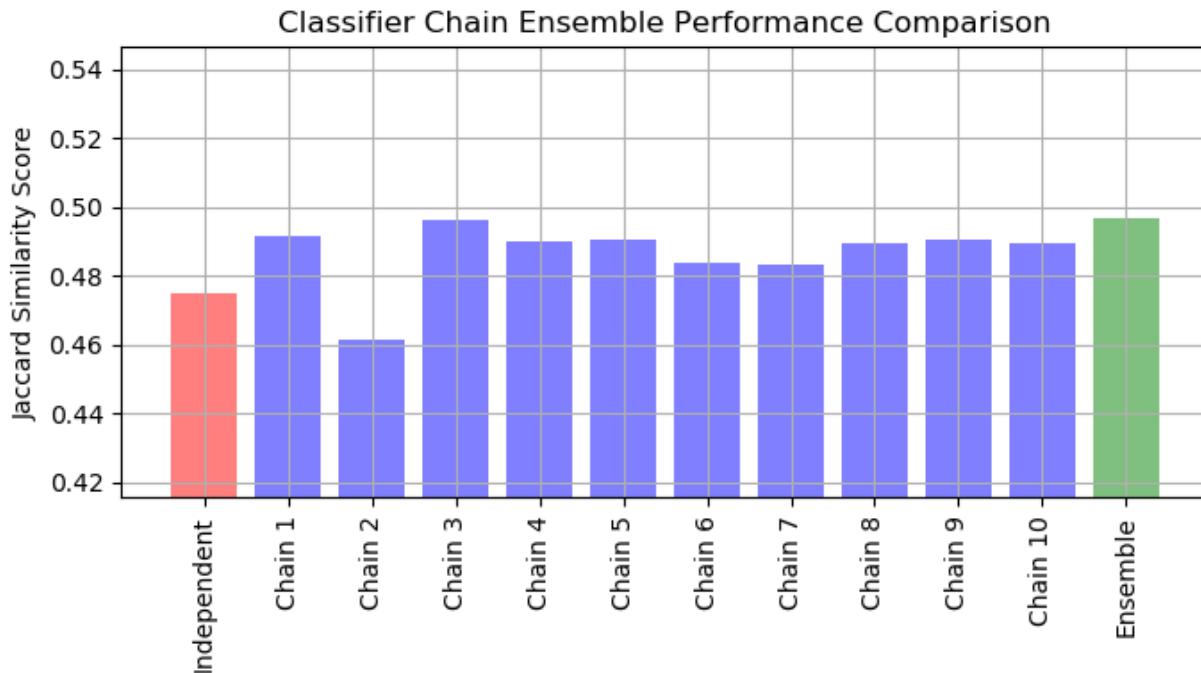
5.22.1 Classifier Chain

Example of using classifier chain on a multilabel dataset.

For this example we will use the `yeast` dataset which contains 2417 datapoints each with 103 features and 14 possible labels. Each data point has at least one label. As a baseline we first train a logistic regression classifier for each of the 14 labels. To evaluate the performance of these classifiers we predict on a held-out test set and calculate the jaccard score for each sample.

Next we create 10 classifier chains. Each classifier chain contains a logistic regression model for each of the 14 labels. The models in each chain are ordered randomly. In addition to the 103 features in the dataset, each model gets the predictions of the preceding models in the chain as features (note that by default at training time each model gets the true labels as features). These additional features allow each chain to exploit correlations among the classes. The Jaccard similarity score for each chain tends to be greater than that of the set independent logistic models.

Because the models in each chain are arranged randomly there is significant variation in performance among the chains. Presumably there is an optimal ordering of the classes in a chain that will yield the best performance. However we do not know that ordering a priori. Instead we can construct an voting ensemble of classifier chains by averaging the binary predictions of the chains and apply a threshold of 0.5. The Jaccard similarity score of the ensemble is greater than that of the independent models and tends to exceed the score of each chain in the ensemble (although this is not guaranteed with randomly ordered chains).



```
# Author: Adam Kleczewski
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
```

```

from sklearn.multioutput import ClassifierChain
from sklearn.model_selection import train_test_split
from sklearn.multiclass import OneVsRestClassifier
from sklearn.metrics import jaccard_score
from sklearn.linear_model import LogisticRegression

print(__doc__)

# Load a multi-label dataset from https://www.openml.org/d/40597
X, Y = fetch_openml('yeast', version=4, return_X_y=True)
Y = Y == 'TRUE'
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=.2,
                                                    random_state=0)

# Fit an independent logistic regression model for each class using the
# OneVsRestClassifier wrapper.
base_lr = LogisticRegression(solver='lbfgs')
ovr = OneVsRestClassifier(base_lr)
ovr.fit(X_train, Y_train)
Y_pred_ovr = ovr.predict(X_test)
ovr_jaccard_score = jaccard_score(Y_test, Y_pred_ovr, average='samples')

# Fit an ensemble of logistic regression classifier chains and take the
# take the average prediction of all the chains.
chains = [ClassifierChain(base_lr, order='random', random_state=i)
          for i in range(10)]
for chain in chains:
    chain.fit(X_train, Y_train)

Y_pred_chains = np.array([chain.predict(X_test) for chain in
                           chains])
chain_jaccard_scores = [jaccard_score(Y_test, Y_pred_chain >= .5,
                                       average='samples')
                        for Y_pred_chain in Y_pred_chains]

Y_pred_ensemble = Y_pred_chains.mean(axis=0)
ensemble_jaccard_score = jaccard_score(Y_test,
                                         Y_pred_ensemble >= .5,
                                         average='samples')

model_scores = [ovr_jaccard_score] + chain_jaccard_scores
model_scores.append(ensemble_jaccard_score)

model_names = ('Independent',
               'Chain 1',
               'Chain 2',
               'Chain 3',
               'Chain 4',
               'Chain 5',
               'Chain 6',
               'Chain 7',
               'Chain 8',
               'Chain 9',
               'Chain 10',
               'Ensemble')

x_pos = np.arange(len(model_names))

```

```
# Plot the Jaccard similarity scores for the independent model, each of the
# chains, and the ensemble (note that the vertical axis on this plot does
# not begin at 0).

fig, ax = plt.subplots(figsize=(7, 4))
ax.grid(True)
ax.set_title('Classifier Chain Ensemble Performance Comparison')
ax.set_xticks(x_pos)
ax.set_xticklabels(model_names, rotation='vertical')
ax.set_ylabel('Jaccard Similarity Score')
ax.set_ylim([min(model_scores) * .9, max(model_scores) * 1.1])
colors = ['r'] + ['b'] * len(chain_jaccard_scores) + ['g']
ax.bar(x_pos, model_scores, alpha=0.5, color=colors)
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 7.947 seconds)

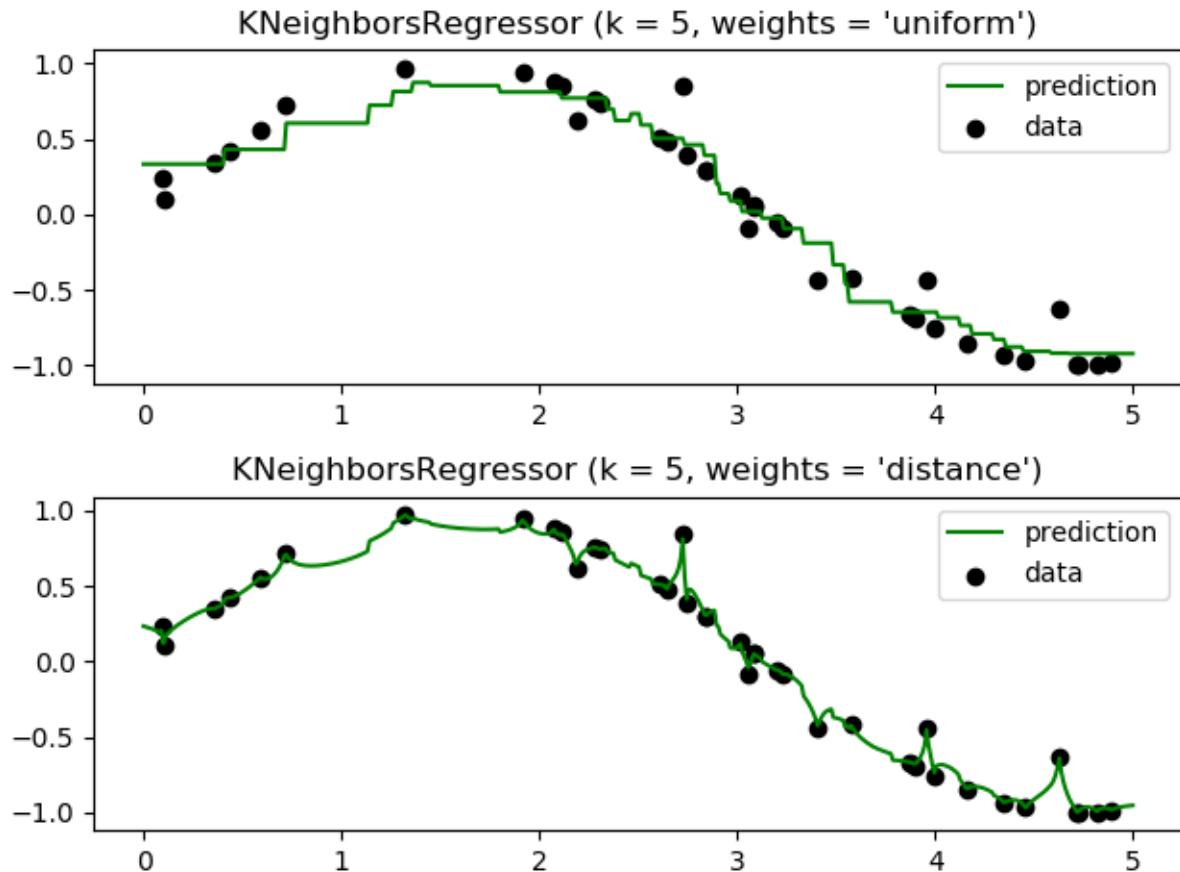
5.23 Nearest Neighbors

Examples concerning the `sklearn.neighbors` module.

Note: Click [here](#) to download the full example code

5.23.1 Nearest Neighbors regression

Demonstrate the resolution of a regression problem using a k-Nearest Neighbor and the interpolation of the target using both barycenter and constant weights.



```

print(__doc__)

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#         Fabian Pedregosa <fabian.pedregosa@inria.fr>
#
# License: BSD 3 clause (C) INRIA

#####
# Generate sample data
import numpy as np
import matplotlib.pyplot as plt
from sklearn import neighbors

np.random.seed(0)
X = np.sort(5 * np.random.rand(40, 1), axis=0)
T = np.linspace(0, 5, 500)[:, np.newaxis]
y = np.sin(X).ravel()

# Add noise to targets
y[::5] += 1 * (0.5 - np.random.rand(8))

#####
# Fit regression model
n_neighbors = 5

```

```
for i, weights in enumerate(['uniform', 'distance']):
    knn = neighbors.KNeighborsRegressor(n_neighbors, weights=weights)
    y_ = knn.fit(X, y).predict(T)

    plt.subplot(2, 1, i + 1)
    plt.scatter(X, y, c='k', label='data')
    plt.plot(T, y_, c='g', label='prediction')
    plt.axis('tight')
    plt.legend()
    plt.title("KNeighborsRegressor (k = %i, weights = '%s') " % (n_neighbors,
                                                               weights))

plt.tight_layout()
plt.show()
```

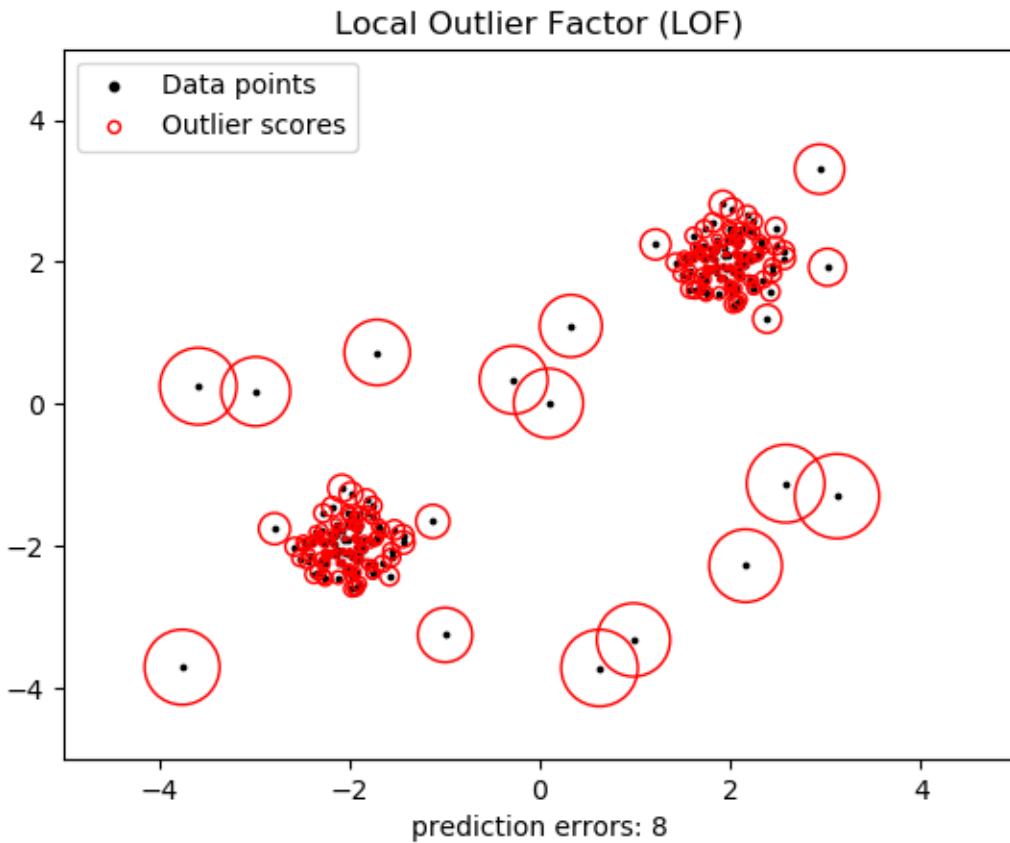
Total running time of the script: (0 minutes 0.089 seconds)

Note: Click [here](#) to download the full example code

5.23.2 Outlier detection with Local Outlier Factor (LOF)

The Local Outlier Factor (LOF) algorithm is an unsupervised anomaly detection method which computes the local density deviation of a given data point with respect to its neighbors. It considers as outliers the samples that have a substantially lower density than their neighbors. This example shows how to use LOF for outlier detection which is the default use case of this estimator in scikit-learn. Note that when LOF is used for outlier detection it has no predict, decision_function and score_samples methods. See [User Guide](#): for details on the difference between outlier detection and novelty detection and how to use LOF for novelty detection.

The number of neighbors considered (parameter n_neighbors) is typically set 1) greater than the minimum number of samples a cluster has to contain, so that other samples can be local outliers relative to this cluster, and 2) smaller than the maximum number of close by samples that can potentially be local outliers. In practice, such informations are generally not available, and taking n_neighbors=20 appears to work well in general.



```

import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import LocalOutlierFactor

print(__doc__)

np.random.seed(42)

# Generate train data
X_inliers = 0.3 * np.random.randn(100, 2)
X_inliers = np.r_[X_inliers + 2, X_inliers - 2]

# Generate some outliers
X_outliers = np.random.uniform(low=-4, high=4, size=(20, 2))
X = np.r_[X_inliers, X_outliers]

n_outliers = len(X_outliers)
ground_truth = np.ones(len(X), dtype=int)
ground_truth[-n_outliers:] = -1

# fit the model for outlier detection (default)
clf = LocalOutlierFactor(n_neighbors=20, contamination=0.1)
# use fit_predict to compute the predicted labels of the training samples
# (when LOF is used for outlier detection, the estimator has no predict,
# decision_function and score_samples methods).
y_pred = clf.fit_predict(X)

```

```
n_errors = (y_pred != ground_truth).sum()
X_scores = clf.negative_outlier_factor_

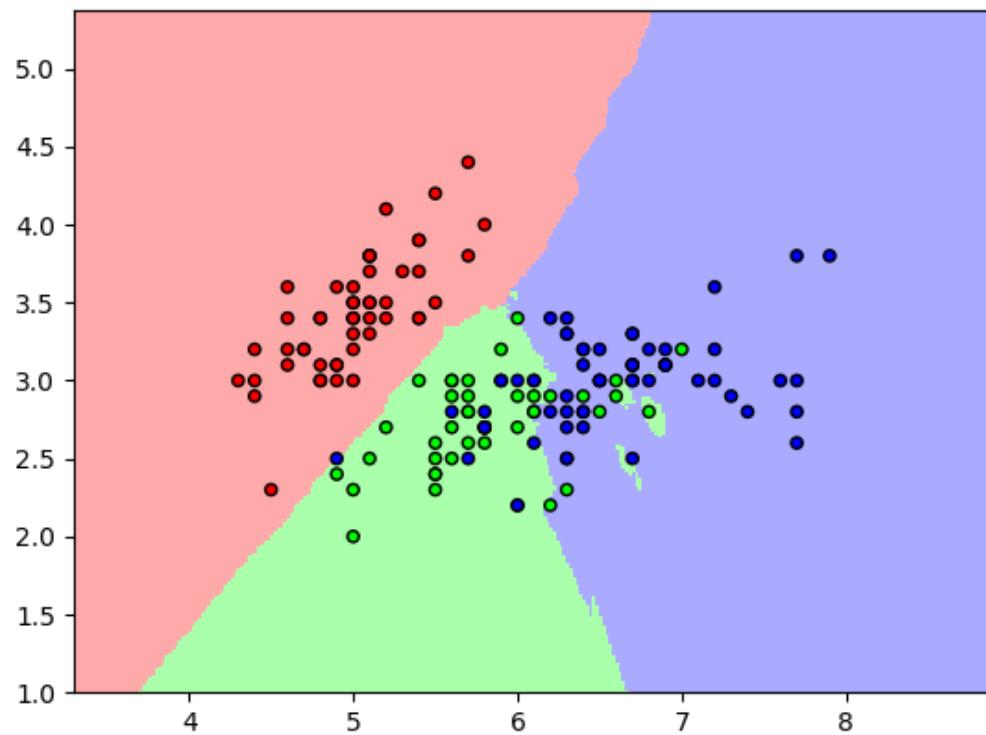
plt.title("Local Outlier Factor (LOF)")
plt.scatter(X[:, 0], X[:, 1], color='k', s=3., label='Data points')
# plot circles with radius proportional to the outlier scores
radius = (X_scores.max() - X_scores) / (X_scores.max() - X_scores.min())
plt.scatter(X[:, 0], X[:, 1], s=1000 * radius, edgecolors='r',
            facecolors='none', label='Outlier scores')
plt.axis('tight')
plt.xlim((-5, 5))
plt.ylim((-5, 5))
plt.xlabel("prediction errors: %d" % (n_errors))
legend = plt.legend(loc='upper left')
legend.legendHandles[0]._sizes = [10]
legend.legendHandles[1]._sizes = [20]
plt.show()
```

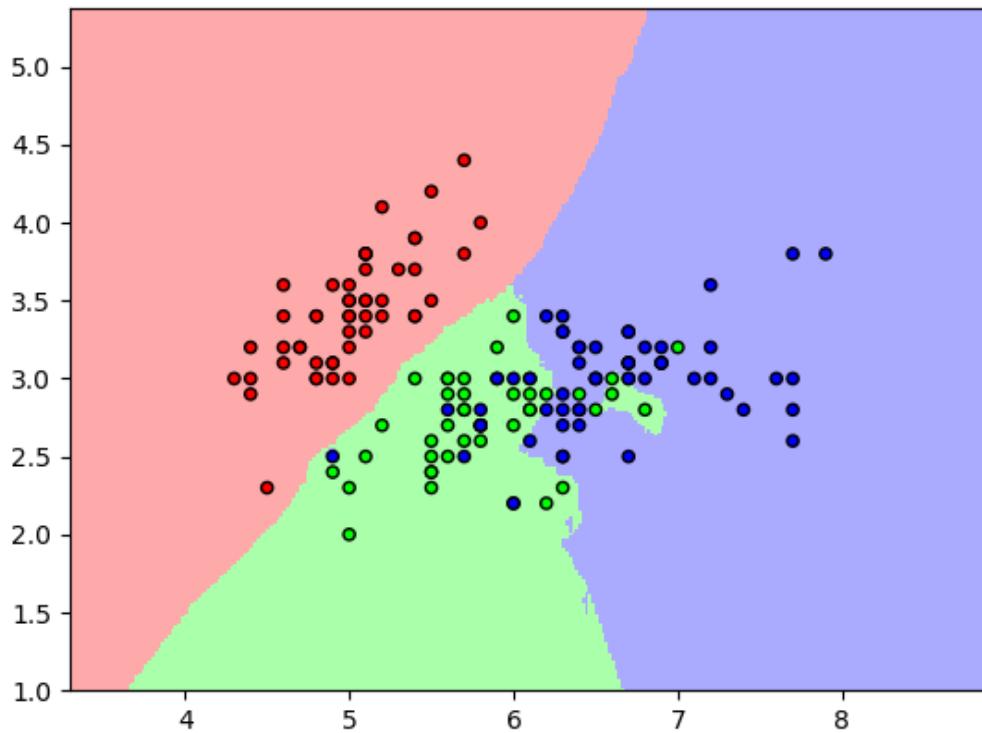
Total running time of the script: (0 minutes 0.017 seconds)

Note: Click [here](#) to download the full example code

5.23.3 Nearest Neighbors Classification

Sample usage of Nearest Neighbors classification. It will plot the decision boundaries for each class.

3-Class classification ($k = 15$, weights = 'uniform')

3-Class classification ($k = 15$, weights = 'distance')

```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import neighbors, datasets

n_neighbors = 15

# import some data to play with
iris = datasets.load_iris()

# we only take the first two features. We could avoid this ugly
# slicing by using a two-dim dataset
X = iris.data[:, :2]
y = iris.target

h = .02 # step size in the mesh

# Create color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

for weights in ['uniform', 'distance']:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = neighbors.KNeighborsClassifier(n_neighbors, weights=weights)
    clf.fit(X, y)

```

```

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                      np.arange(y_min, y_max, h))
z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure()
plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold,
            edgecolor='k', s=20)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.title("3-Class classification (k = %i, weights = '%s')"
          % (n_neighbors, weights))

plt.show()

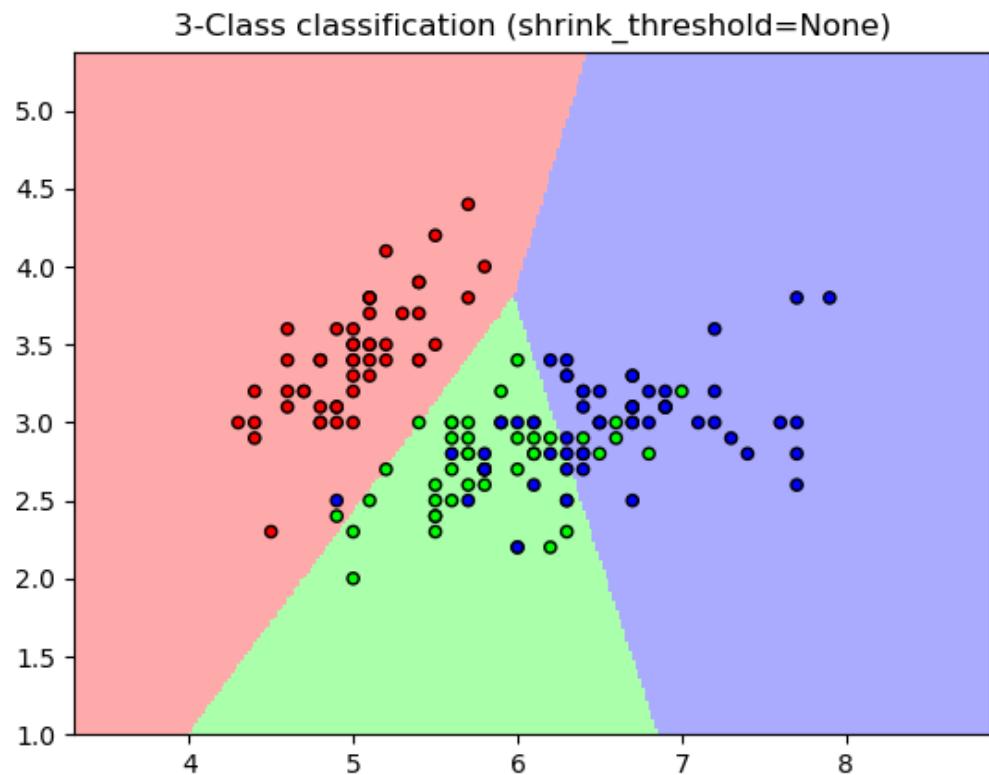
```

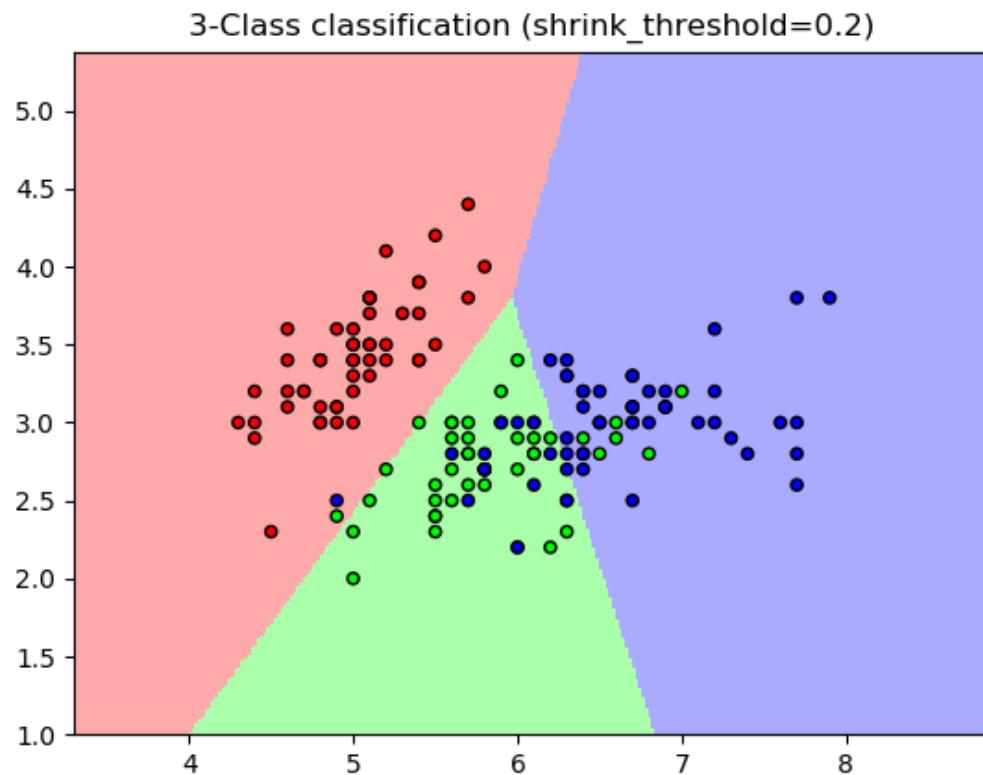
Total running time of the script: (0 minutes 1.416 seconds)

Note: Click [here](#) to download the full example code

5.23.4 Nearest Centroid Classification

Sample usage of Nearest Centroid classification. It will plot the decision boundaries for each class.





Out:

```
None 0.8133333333333334
0.2 0.82
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets
from sklearn.neighbors import NearestCentroid

n_neighbors = 15

# import some data to play with
iris = datasets.load_iris()
# we only take the first two features. We could avoid this ugly
# slicing by using a two-dim dataset
X = iris.data[:, :2]
y = iris.target
```

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

# Create color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

for shrinkage in [None, .2]:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = NearestCentroid(shrink_threshold=shrinkage)
    clf.fit(X, y)
    y_pred = clf.predict(X)
    print(shrinkage, np.mean(y == y_pred))
    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                          np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold,
                edgecolor='k', s=20)
    plt.title("3-Class classification (shrink_threshold=%r)" % shrinkage)
    plt.axis('tight')

plt.show()
```

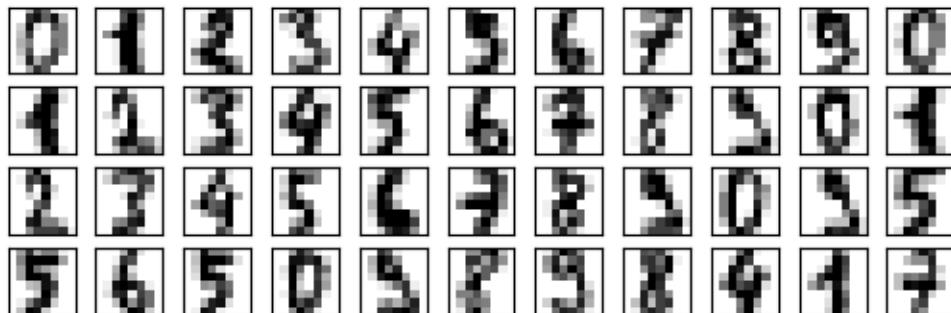
Total running time of the script: (0 minutes 0.051 seconds)

Note: Click [here](#) to download the full example code

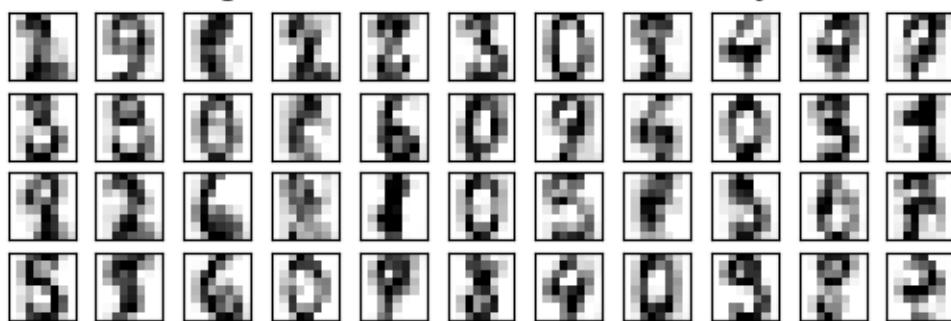
5.23.5 Kernel Density Estimation

This example shows how kernel density estimation (KDE), a powerful non-parametric density estimation technique, can be used to learn a generative model for a dataset. With this generative model in place, new samples can be drawn. These new samples reflect the underlying model of the data.

Selection from the input data



'New' digits drawn from the kernel density model



Out:

```
best bandwidth: 3.79269019073225
```

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import load_digits
from sklearn.neighbors import KernelDensity
from sklearn.decomposition import PCA
from sklearn.model_selection import GridSearchCV

# load the data
digits = load_digits()

# project the 64-dimensional data to a lower dimension
pca = PCA(n_components=15, whiten=False)
data = pca.fit_transform(digits.data)

# use grid search cross-validation to optimize the bandwidth
params = {'bandwidth': np.logspace(-1, 1, 20)}
```

```
grid = GridSearchCV(KernelDensity(), params, cv=5, iid=False)
grid.fit(data)

print("best bandwidth: {}".format(grid.best_estimator_.bandwidth))

# use the best estimator to compute the kernel density estimate
kde = grid.best_estimator_

# sample 44 new points from the data
new_data = kde.sample(44, random_state=0)
new_data = pca.inverse_transform(new_data)

# turn data into a 4x11 grid
new_data = new_data.reshape((4, 11, -1))
real_data = digits.data[:44].reshape((4, 11, -1))

# plot real digits and resampled digits
fig, ax = plt.subplots(9, 11, subplot_kw=dict(xticks=[], yticks=[]))
for j in range(11):
    ax[4, j].set_visible(False)
    for i in range(4):
        im = ax[i, j].imshow(real_data[i, j].reshape((8, 8)),
                             cmap=plt.cm.binary, interpolation='nearest')
        im.set_clim(0, 16)
        im = ax[i + 5, j].imshow(new_data[i, j].reshape((8, 8)),
                               cmap=plt.cm.binary, interpolation='nearest')
        im.set_clim(0, 16)

ax[0, 5].set_title('Selection from the input data')
ax[5, 5].set_title('"New" digits drawn from the kernel density model')

plt.show()
```

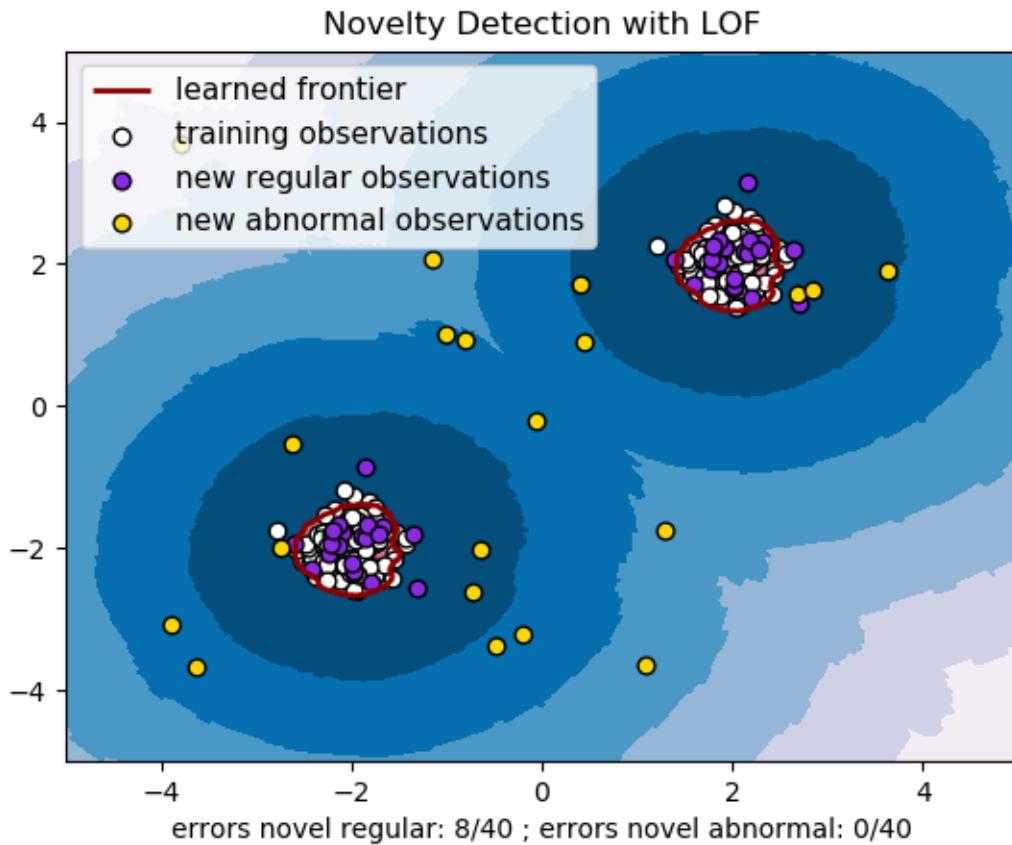
Total running time of the script: (0 minutes 4.482 seconds)

Note: Click [here](#) to download the full example code

5.23.6 Novelty detection with Local Outlier Factor (LOF)

The Local Outlier Factor (LOF) algorithm is an unsupervised anomaly detection method which computes the local density deviation of a given data point with respect to its neighbors. It considers as outliers the samples that have a substantially lower density than their neighbors. This example shows how to use LOF for novelty detection. Note that when LOF is used for novelty detection you MUST not use predict, decision_function and score_samples on the training set as this would lead to wrong results. You must only use these methods on new unseen data (which are not in the training set). See [User Guide](#): for details on the difference between outlier detection and novelty detection and how to use LOF for outlier detection.

The number of neighbors considered, (parameter n_neighbors) is typically set 1) greater than the minimum number of samples a cluster has to contain, so that other samples can be local outliers relative to this cluster, and 2) smaller than the maximum number of close by samples that can potentially be local outliers. In practice, such informations are generally not available, and taking n_neighbors=20 appears to work well in general.



```

import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from sklearn.neighbors import LocalOutlierFactor

print(__doc__)

np.random.seed(42)

xx, yy = np.meshgrid(np.linspace(-5, 5, 500), np.linspace(-5, 5, 500))
# Generate normal (not abnormal) training observations
X = 0.3 * np.random.randn(100, 2)
X_train = np.r_[X + 2, X - 2]
# Generate new normal (not abnormal) observations
X = 0.3 * np.random.randn(20, 2)
X_test = np.r_[X + 2, X - 2]
# Generate some abnormal novel observations
X_outliers = np.random.uniform(low=-4, high=4, size=(20, 2))

# fit the model for novelty detection (novelty=True)
clf = LocalOutlierFactor(n_neighbors=20, novelty=True, contamination=0.1)
clf.fit(X_train)
# DO NOT use predict, decision_function and score_samples on X_train as this
# would give wrong results but only on new unseen data (not used in X_train),
# e.g. X_test, X_outliers or the meshgrid
y_pred_test = clf.predict(X_test)

```

```
y_pred_outliers = clf.predict(X_outliers)
n_error_test = y_pred_test[y_pred_test == -1].size
n_error_outliers = y_pred_outliers[y_pred_outliers == 1].size

# plot the learned frontier, the points, and the nearest vectors to the plane
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.title("Novelty Detection with LOF")
plt.contourf(xx, yy, Z, levels=np.linspace(Z.min(), 0, 7), cmap=plt.cm.PuBu)
a = plt.contour(xx, yy, Z, levels=[0], linewidths=2, colors='darkred')
plt.contourf(xx, yy, Z, levels=[0, Z.max()], colors='palevioletred')

s = 40
b1 = plt.scatter(X_train[:, 0], X_train[:, 1], c='white', s=s, edgecolors='k')
b2 = plt.scatter(X_test[:, 0], X_test[:, 1], c='blueviolet', s=s,
                 edgecolors='k')
c = plt.scatter(X_outliers[:, 0], X_outliers[:, 1], c='gold', s=s,
                 edgecolors='k')
plt.axis('tight')
plt.xlim((-5, 5))
plt.ylim((-5, 5))
plt.legend([a.collections[0], b1, b2, c],
           ["learned frontier", "training observations",
            "new regular observations", "new abnormal observations"],
           loc="upper left",
           prop=matplotlib.font_manager.FontProperties(size=11))
plt.xlabel(
    "errors novel regular: %d/40 ; errors novel abnormal: %d/40"
    % (n_error_test, n_error_outliers))
plt.show()
```

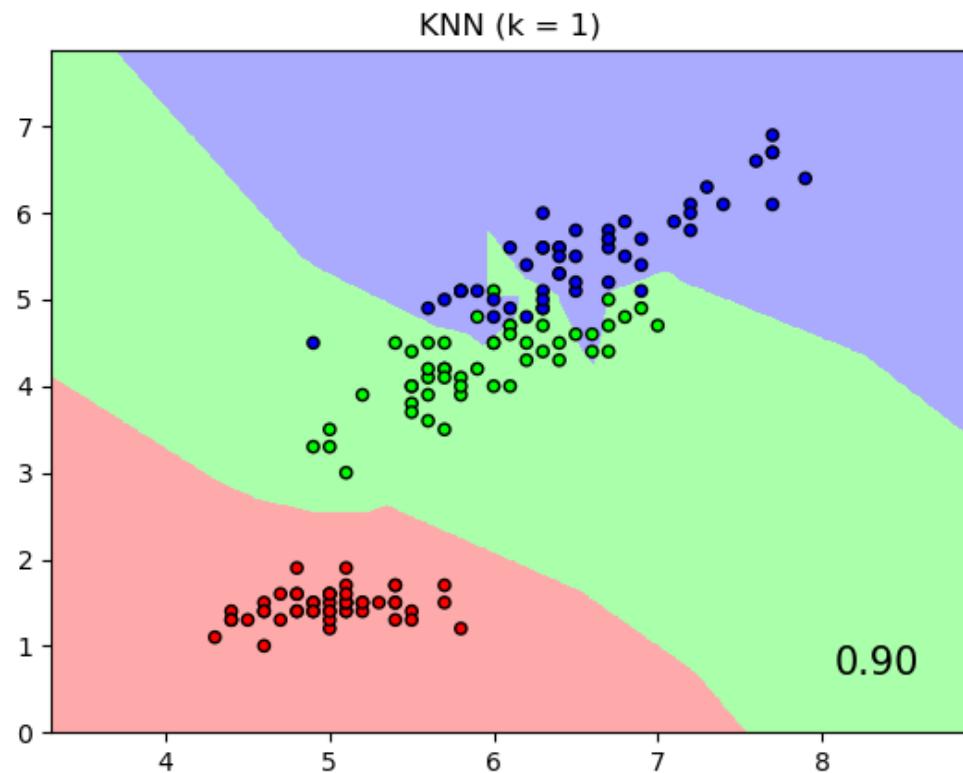
Total running time of the script: (0 minutes 0.634 seconds)

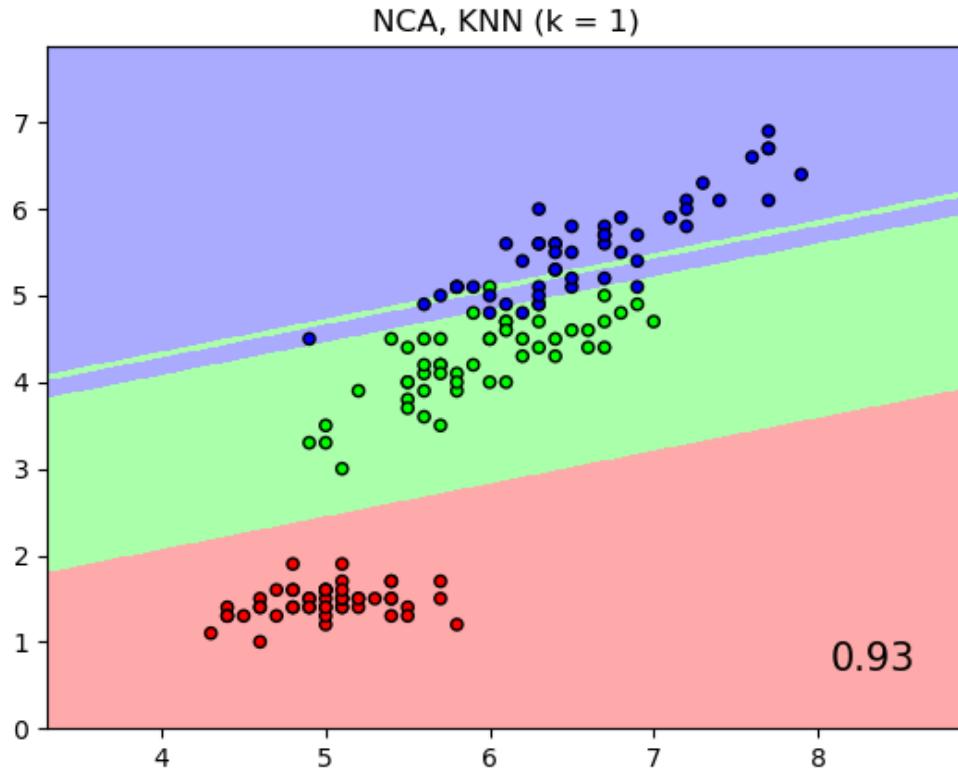
Note: Click [here](#) to download the full example code

5.23.7 Comparing Nearest Neighbors with and without Neighborhood Components Analysis

An example comparing nearest neighbors classification with and without Neighborhood Components Analysis.

It will plot the class decision boundaries given by a Nearest Neighbors classifier when using the Euclidean distance on the original features, versus using the Euclidean distance after the transformation learned by Neighborhood Components Analysis. The latter aims to find a linear transformation that maximises the (stochastic) nearest neighbor classification accuracy on the training set.





```
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import (KNeighborsClassifier,
                               NeighborhoodComponentsAnalysis)
from sklearn.pipeline import Pipeline

print(__doc__)

n_neighbors = 1

dataset = datasets.load_iris()
X, y = dataset.data, dataset.target

# we only take two features. We could avoid this ugly
# slicing by using a two-dim dataset
X = X[:, [0, 2]]

X_train, X_test, y_train, y_test = \
    train_test_split(X, y, stratify=y, test_size=0.7, random_state=42)
```

```

h = .01 # step size in the mesh

# Create color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

names = ['KNN', 'NCA', 'KNN']

classifiers = [Pipeline([('scaler', StandardScaler()),
                         ('knn', KNeighborsClassifier(n_neighbors=n_neighbors))
                         ]),
                Pipeline([('scaler', StandardScaler()),
                          ('nca', NeighborhoodComponentsAnalysis()),
                          ('knn', KNeighborsClassifier(n_neighbors=n_neighbors))
                          ])
               ]

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

for name, clf in zip(names, classifiers):

    clf.fit(X_train, y_train)
    score = clf.score(X_test, y_test)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light, alpha=.8)

    # Plot also the training and testing points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold, edgecolor='k', s=20)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title("{} (k = {})".format(name, n_neighbors))
    plt.text(0.9, 0.1, '{:.2f}'.format(score), size=15,
             ha='center', va='center', transform=plt.gca().transAxes)

plt.show()

```

Total running time of the script: (0 minutes 16.006 seconds)

Note: Click [here](#) to download the full example code

5.23.8 Dimensionality Reduction with Neighborhood Components Analysis

Sample usage of Neighborhood Components Analysis for dimensionality reduction.

This example compares different (linear) dimensionality reduction methods applied on the Digits data set. The data

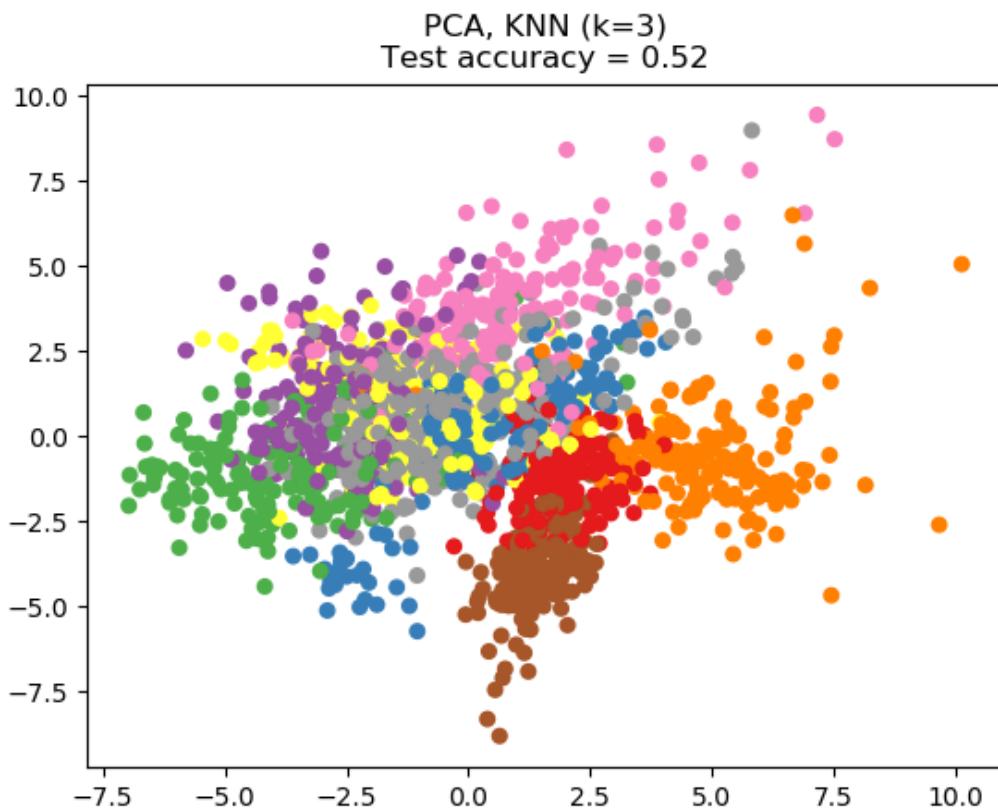
set contains images of digits from 0 to 9 with approximately 180 samples of each class. Each image is of dimension $8 \times 8 = 64$, and is reduced to a two-dimensional data point.

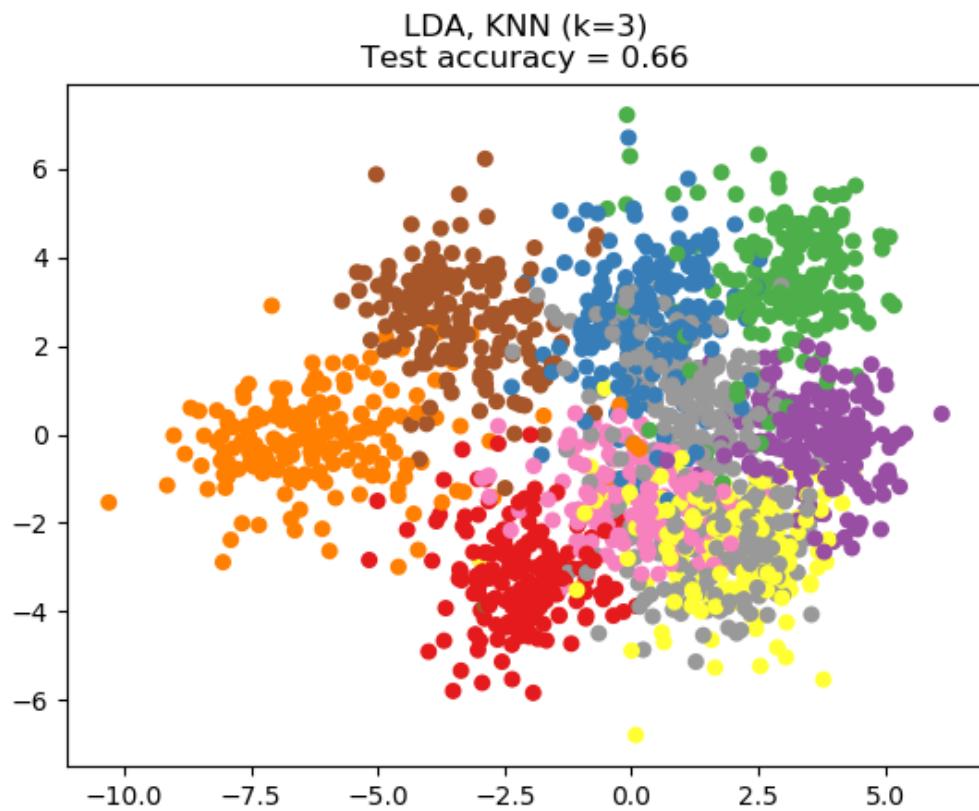
Principal Component Analysis (PCA) applied to this data identifies the combination of attributes (principal components, or directions in the feature space) that account for the most variance in the data. Here we plot the different samples on the 2 first principal components.

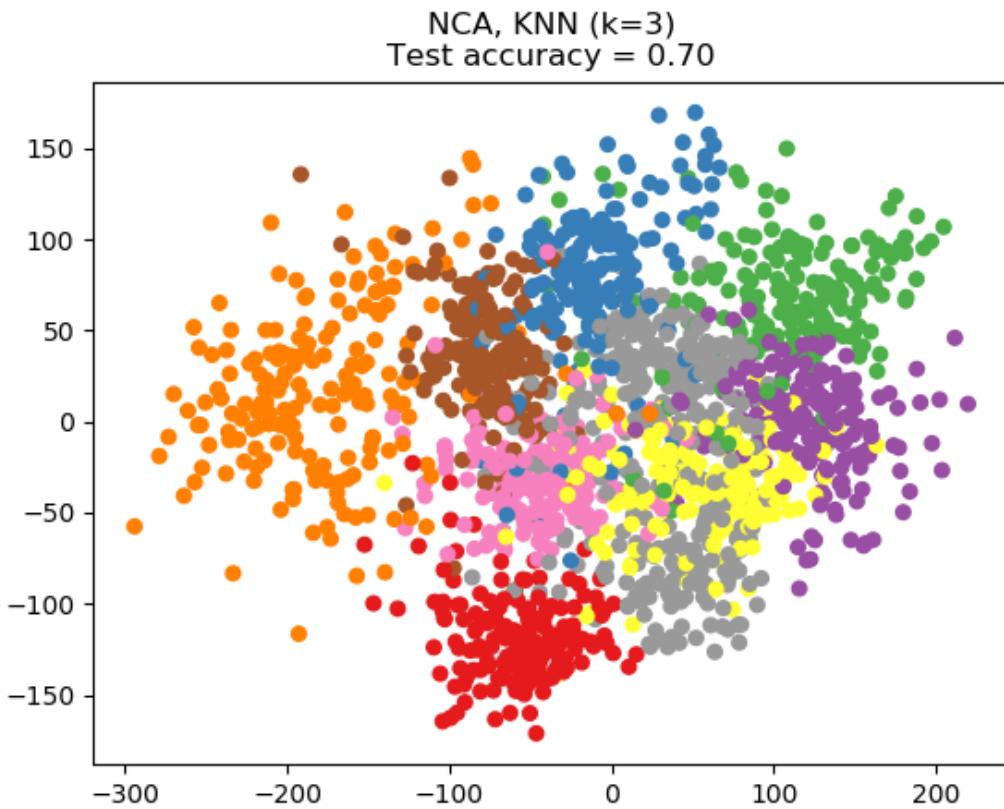
Linear Discriminant Analysis (LDA) tries to identify attributes that account for the most variance *between classes*. In particular, LDA, in contrast to PCA, is a supervised method, using known class labels.

Neighborhood Components Analysis (NCA) tries to find a feature space such that a stochastic nearest neighbor algorithm will give the best accuracy. Like LDA, it is a supervised method.

One can see that NCA enforces a clustering of the data that is visually meaningful despite the large reduction in dimension.







```
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neighbors import (KNeighborsClassifier,
                               NeighborhoodComponentsAnalysis)
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

print(__doc__)

n_neighbors = 3
random_state = 0

# Load Digits dataset
digits = datasets.load_digits()
X, y = digits.data, digits.target

# Split into train/test
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.5, stratify=y,
                     random_state=random_state)
```

```

dim = len(X[0])
n_classes = len(np.unique(y))

# Reduce dimension to 2 with PCA
pca = make_pipeline(StandardScaler(),
                     PCA(n_components=2, random_state=random_state))

# Reduce dimension to 2 with LinearDiscriminantAnalysis
lda = make_pipeline(StandardScaler(),
                     LinearDiscriminantAnalysis(n_components=2))

# Reduce dimension to 2 with NeighborhoodComponentAnalysis
nca = make_pipeline(StandardScaler(),
                     NeighborhoodComponentsAnalysis(n_components=2,
                                                    random_state=random_state))

# Use a nearest neighbor classifier to evaluate the methods
knn = KNeighborsClassifier(n_neighbors=n_neighbors)

# Make a list of the methods to be compared
dim_reduction_methods = [('PCA', pca), ('LDA', lda), ('NCA', nca)]

# plt.figure()
for i, (name, model) in enumerate(dim_reduction_methods):
    plt.figure()
    # plt.subplot(1, 3, i + 1, aspect=1)

    # Fit the method's model
    model.fit(X_train, y_train)

    # Fit a nearest neighbor classifier on the embedded training set
    knn.fit(model.transform(X_train), y_train)

    # Compute the nearest neighbor accuracy on the embedded test set
    acc_knn = knn.score(model.transform(X_test), y_test)

    # Embed the data set in 2 dimensions using the fitted model
    X_embedded = model.transform(X)

    # Plot the projected points and show the evaluation score
    plt.scatter(X_embedded[:, 0], X_embedded[:, 1], c=y, s=30, cmap='Set1')
    plt.title("{}\nKNN (k={})\nTest accuracy = {:.2f}\n".format(name,
                                                               n_neighbors,
                                                               acc_knn))
plt.show()

```

Total running time of the script: (0 minutes 2.879 seconds)

Note: Click [here](#) to download the full example code

5.23.9 Kernel Density Estimate of Species Distributions

This shows an example of a neighbors-based query (in particular a kernel density estimate) on geospatial data, using a Ball Tree built upon the Haversine distance metric – i.e. distances over points in latitude/longitude. The dataset

is provided by Phillips et. al. (2006). If available, the example uses `basemap` to plot the coast lines and national boundaries of South America.

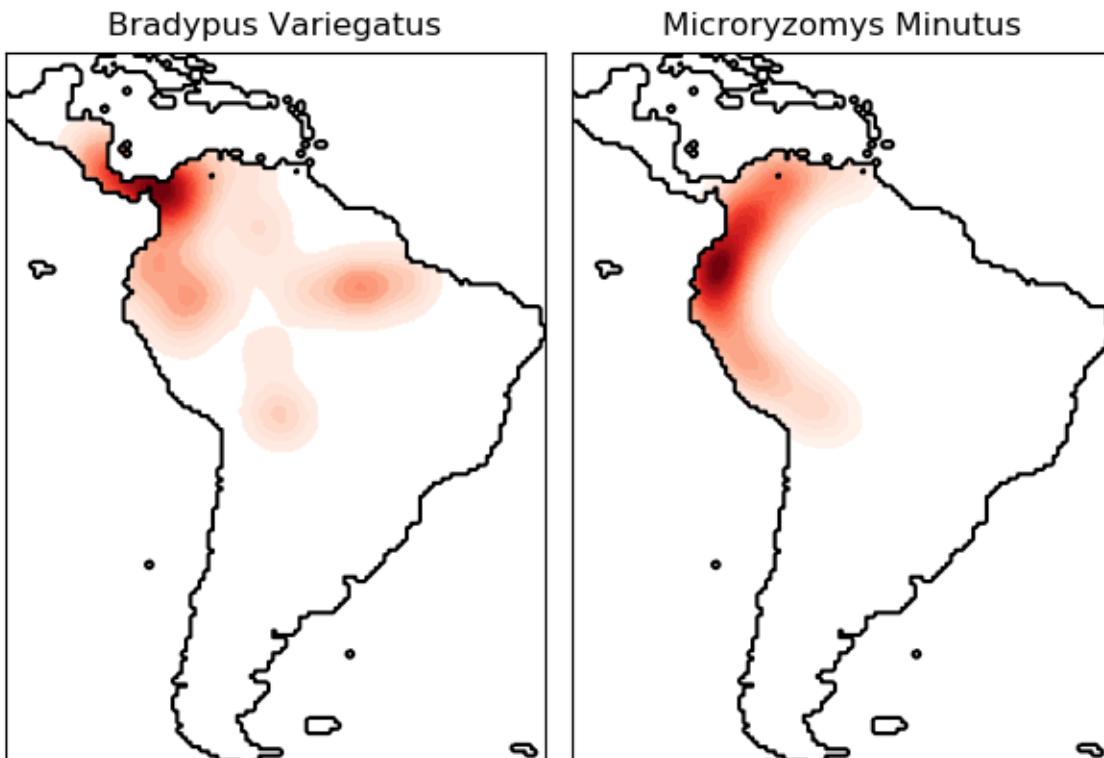
This example does not perform any learning over the data (see [Species distribution modeling](#) for an example of classification based on the attributes in this dataset). It simply shows the kernel density estimate of observed data points in geospatial coordinates.

The two species are:

- “*Bradypus variegatus*”, the Brown-throated Sloth.
- “*Microryzomys minutus*”, also known as the Forest Small Rice Rat, a rodent that lives in Peru, Colombia, Ecuador, Peru, and Venezuela.

References

- “Maximum entropy modeling of species geographic distributions” S. J. Phillips, R. P. Anderson, R. E. Schapire
- Ecological Modelling, 190:231-259, 2006.



Out:

- ```
- computing KDE in spherical coordinates
- plot coastlines from coverage
- computing KDE in spherical coordinates
- plot coastlines from coverage
```

```

Author: Jake Vanderplas <jakevdp@cs.washington.edu>
#
License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_species_distributions
from sklearn.datasets.species_distributions import construct_grids
from sklearn.neighbors import KernelDensity

if basemap is available, we'll use it.
otherwise, we'll improvise later...
try:
 from mpl_toolkits.basemap import Basemap
 basemap = True
except ImportError:
 basemap = False

Get matrices/arrays of species IDs and locations
data = fetch_species_distributions()
species_names = ['Bradypus Variegatus', 'Microryzomys Minutus']

Xtrain = np.vstack([data['train']['dd lat'],
 data['train']['dd long']]).T
ytrain = np.array([d.decode('ascii').startswith('micro')
 for d in data['train']['species']], dtype='int')
Xtrain *= np.pi / 180. # Convert lat/long to radians

Set up the data grid for the contour plot
xgrid, ygrid = construct_grids(data)
X, Y = np.meshgrid(xgrid[::5], ygrid[::5][::-1])
land_reference = data.coverages[6][::5, ::5]
land_mask = (land_reference > -9999).ravel()

xy = np.vstack([Y.ravel(), X.ravel()]).T
xy = xy[land_mask]
xy *= np.pi / 180.

Plot map of South America with distributions of each species
fig = plt.figure()
fig.subplots_adjust(left=0.05, right=0.95, wspace=0.05)

for i in range(2):
 plt.subplot(1, 2, i + 1)

 # construct a kernel density estimate of the distribution
 print(" - computing KDE in spherical coordinates")
 kde = KernelDensity(bandwidth=0.04, metric='haversine',
 kernel='gaussian', algorithm='ball_tree')
 kde.fit(Xtrain[ytrain == i])

 # evaluate only on the land: -9999 indicates ocean
 Z = np.full(land_mask.shape[0], -9999, dtype='int')
 Z[land_mask] = np.exp(kde.score_samples(xy))
 Z = Z.reshape(X.shape)

```

```
plot contours of the density
levels = np.linspace(0, Z.max(), 25)
plt.contourf(X, Y, Z, levels=levels, cmap=plt.cm.Reds)

if basemap:
 print(" - plot coastlines using basemap")
 m = Basemap(projection='cyl', llcrnrlat=Y.min(),
 urcrnrlat=Y.max(), llcrnrlon=X.min(),
 urcrnrlon=X.max(), resolution='c')
 m.drawcoastlines()
 m.drawcountries()
else:
 print(" - plot coastlines from coverage")
 plt.contour(X, Y, land_reference,
 levels=[-9998], colors="k",
 linestyles="solid")
 plt.xticks([])
 plt.yticks([])

plt.title(species_names[i])

plt.show()
```

**Total running time of the script:** ( 0 minutes 5.470 seconds)

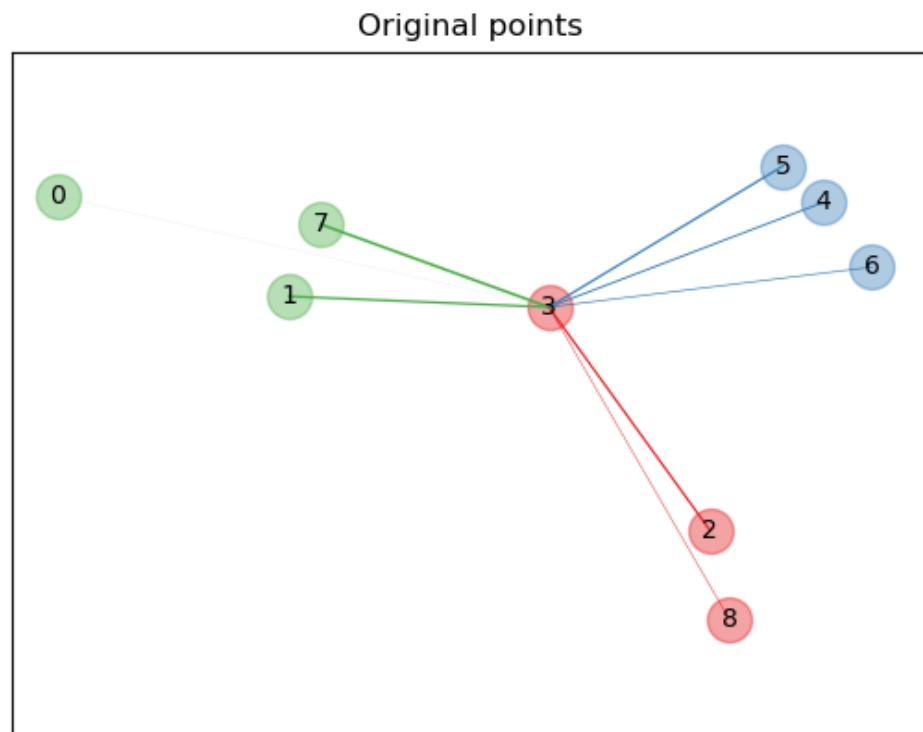
---

**Note:** Click [here](#) to download the full example code

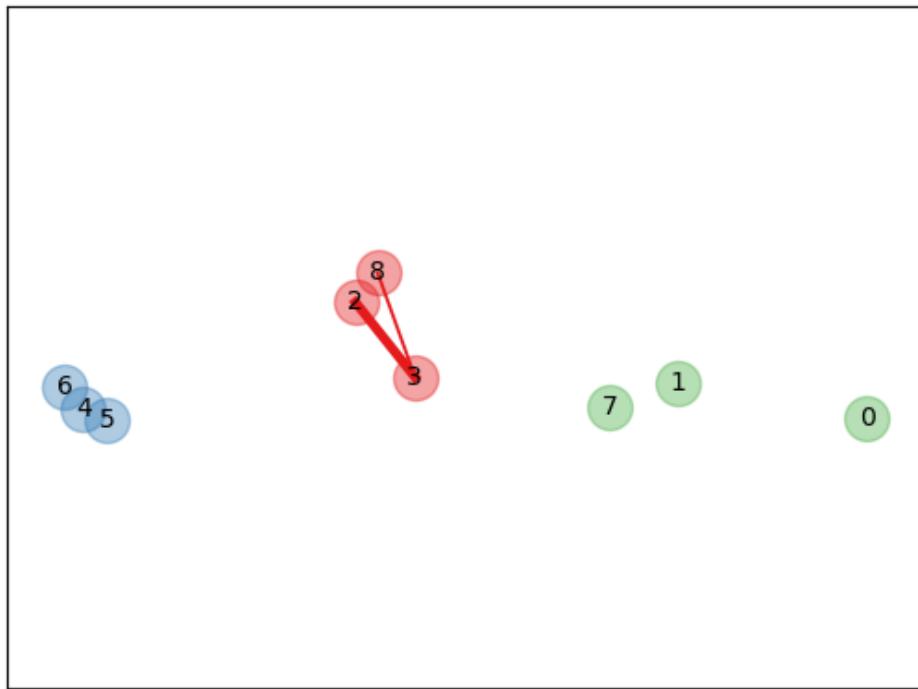
---

## 5.23.10 Neighborhood Components Analysis Illustration

An example illustrating the goal of learning a distance metric that maximizes the nearest neighbors classification accuracy. The example is solely for illustration purposes. Please refer to the *User Guide* for more information.



NCA embedding



```
License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.neighbors import NeighborhoodComponentsAnalysis
from matplotlib import cm
from sklearn.utils.fixes import logsumexp

print(__doc__)

n_neighbors = 1
random_state = 0

Create a tiny data set of 9 samples from 3 classes
X, y = make_classification(n_samples=9, n_features=2, n_informative=2,
 n_redundant=0, n_classes=3, n_clusters_per_class=1,
 class_sep=1.0, random_state=random_state)

Plot the points in the original space
plt.figure()
ax = plt.gca()

Draw the graph nodes
for i in range(X.shape[0]):
 ax.text(X[i, 0], X[i, 1], str(i), va='center', ha='center')
 ax.scatter(X[i, 0], X[i, 1], s=300, c=cm.Set1(y[i]), alpha=0.4)
```

```

def p_i(X, i):
 diff_embedded = X[i] - X
 dist_embedded = np.einsum('ij,ij->i', diff_embedded,
 diff_embedded)
 dist_embedded[i] = np.inf

 # compute exponentiated distances (use the log-sum-exp trick to
 # avoid numerical instabilities
 exp_dist_embedded = np.exp(-dist_embedded -
 logsumexp(-dist_embedded))
 return exp_dist_embedded

def relate_point(X, i, ax):
 pt_i = X[i]
 for j, pt_j in enumerate(X):
 thickness = p_i(X, i)
 if i != j:
 line = ([pt_i[0], pt_j[0]], [pt_i[1], pt_j[1]])
 ax.plot(*line, c=cm.Set1(y[j]),
 linewidth=5*thickness[j])

 # we consider only point 3
 i = 3

 # Plot bonds linked to sample i in the original space
 relate_point(X, i, ax)
 ax.set_title("Original points")
 ax.axes.get_xaxis().set_visible(False)
 ax.axes.get_yaxis().set_visible(False)
 ax.axis('equal')

 # Learn an embedding with NeighborhoodComponentsAnalysis
 nca = NeighborhoodComponentsAnalysis(max_iter=30, random_state=random_state)
 nca = nca.fit(X, y)

 # Plot the points after transformation with NeighborhoodComponentsAnalysis
 plt.figure()
 ax2 = plt.gca()

 # Get the embedding and find the new nearest neighbors
 X_embedded = nca.transform(X)

 relate_point(X_embedded, i, ax2)

 for i in range(len(X)):
 ax2.text(X_embedded[i, 0], X_embedded[i, 1], str(i),
 va='center', ha='center')
 ax2.scatter(X_embedded[i, 0], X_embedded[i, 1], s=300, c=cm.Set1(y[i]),
 alpha=0.4)

 # Make axes equal so that boundaries are displayed correctly as circles
 ax2.set_title("NCA embedding")
 ax2.axes.get_xaxis().set_visible(False)
 ax2.axes.get_yaxis().set_visible(False)

```

```
ax2.axis('equal')
plt.show()
```

**Total running time of the script:** ( 0 minutes 0.069 seconds)

---

**Note:** Click [here](#) to download the full example code

---

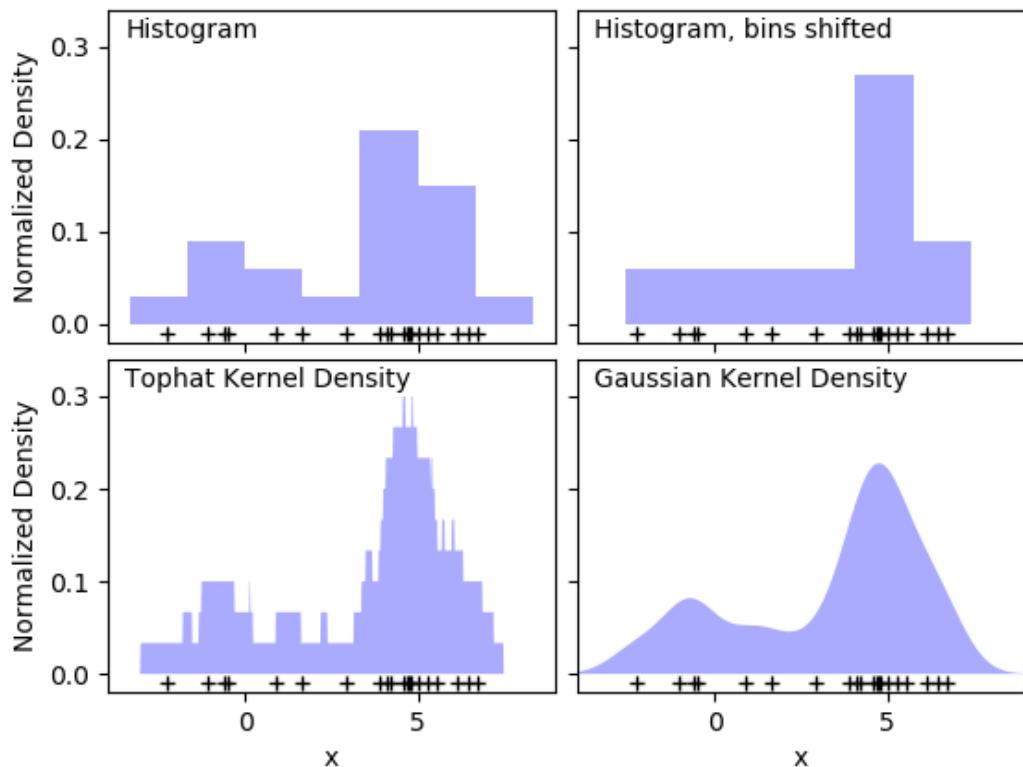
### 5.23.11 Simple 1D Kernel Density Estimation

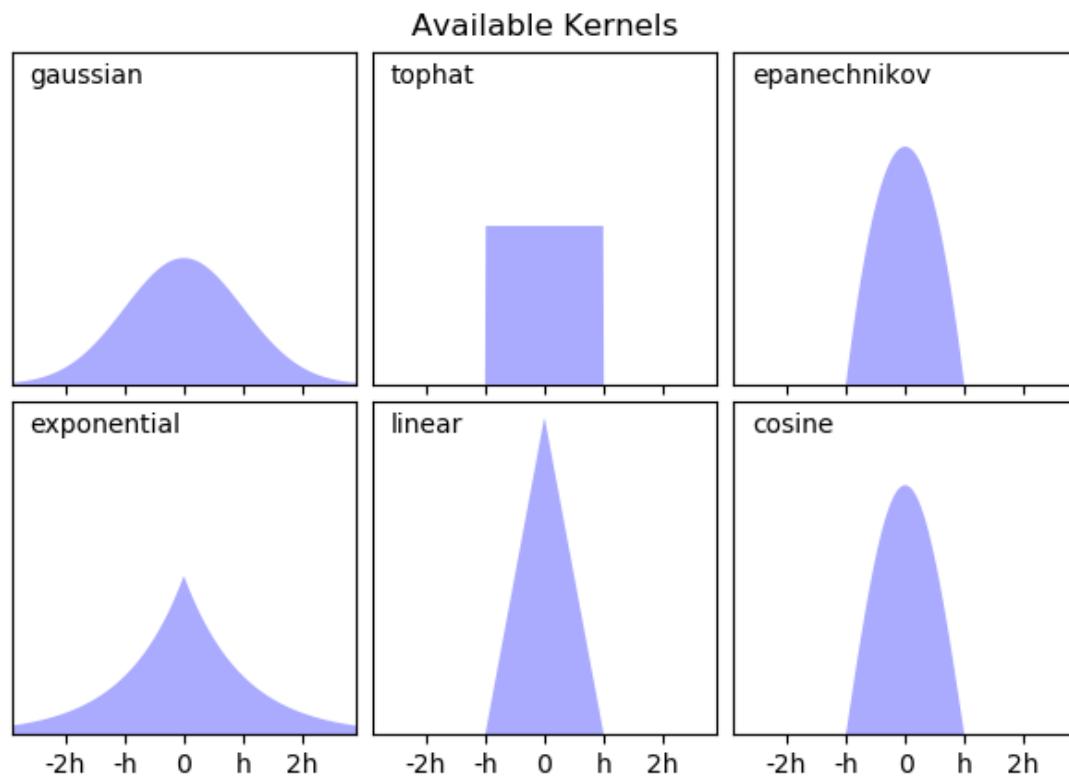
This example uses the `sklearn.neighbors.KernelDensity` class to demonstrate the principles of Kernel Density Estimation in one dimension.

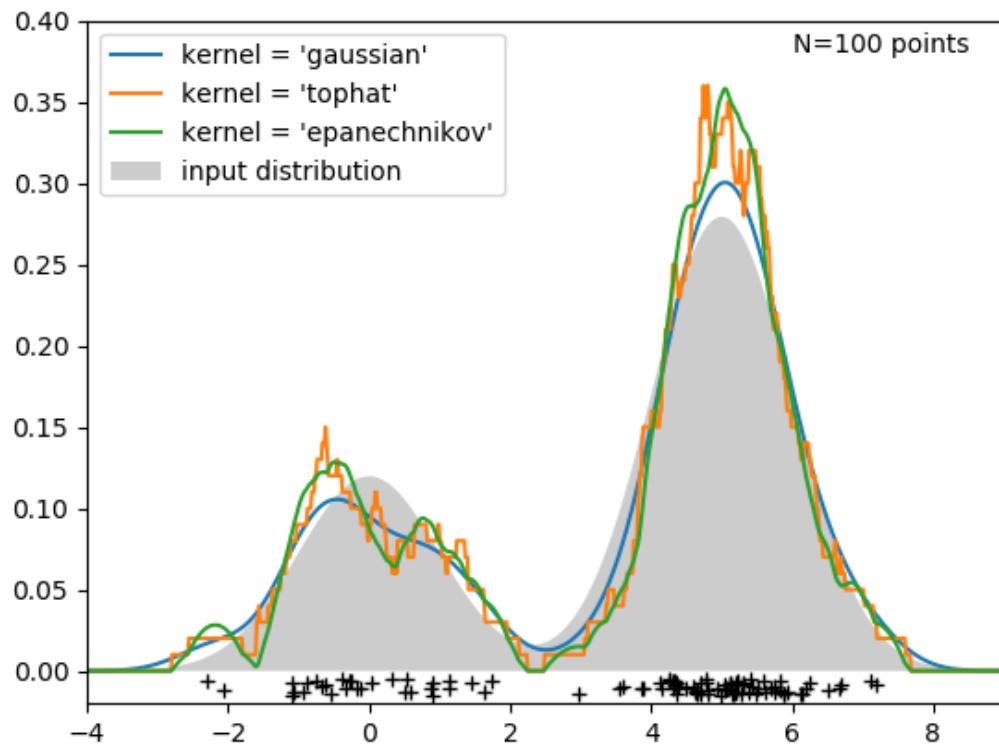
The first plot shows one of the problems with using histograms to visualize the density of points in 1D. Intuitively, a histogram can be thought of as a scheme in which a unit “block” is stacked above each point on a regular grid. As the top two panels show, however, the choice of gridding for these blocks can lead to wildly divergent ideas about the underlying shape of the density distribution. If we instead center each block on the point it represents, we get the estimate shown in the bottom left panel. This is a kernel density estimation with a “top hat” kernel. This idea can be generalized to other kernel shapes: the bottom-right panel of the first figure shows a Gaussian kernel density estimate over the same distribution.

Scikit-learn implements efficient kernel density estimation using either a Ball Tree or KD Tree structure, through the `sklearn.neighbors.KernelDensity` estimator. The available kernels are shown in the second figure of this example.

The third figure compares kernel density estimates for a distribution of 100 samples in 1 dimension. Though this example uses 1D distributions, kernel density estimation is easily and efficiently extensible to higher dimensions as well.







```
Author: Jake Vanderplas <jakevdp@cs.washington.edu>
#
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from distutils.version import LooseVersion
from scipy.stats import norm
from sklearn.neighbors import KernelDensity

`normed` is being deprecated in favor of `density` in histograms
if LooseVersion(matplotlib.__version__) >= '2.1':
 density_param = {'density': True}
else:
 density_param = {'normed': True}

#-----
Plot the progression of histograms to kernels
np.random.seed(1)
N = 20
X = np.concatenate((np.random.normal(0, 1, int(0.3 * N)),
 np.random.normal(5, 1, int(0.7 * N))))[:, np.newaxis]
X_plot = np.linspace(-5, 10, 1000)[:, np.newaxis]
bins = np.linspace(-5, 10, 10)

fig, ax = plt.subplots(2, 2, sharex=True, sharey=True)
fig.subplots_adjust(hspace=0.05, wspace=0.05)
```

```

histogram 1
ax[0, 0].hist(X[:, 0], bins=bins, fc='#AAAAFF', **density_param)
ax[0, 0].text(-3.5, 0.31, "Histogram")

histogram 2
ax[0, 1].hist(X[:, 0], bins=bins + 0.75, fc='#AAAAFF', **density_param)
ax[0, 1].text(-3.5, 0.31, "Histogram, bins shifted")

tophat KDE
kde = KernelDensity(kernel='tophat', bandwidth=0.75).fit(X)
log_dens = kde.score_samples(X_plot)
ax[1, 0].fill(X_plot[:, 0], np.exp(log_dens), fc='#AAAAFF')
ax[1, 0].text(-3.5, 0.31, "Tophat Kernel Density")

Gaussian KDE
kde = KernelDensity(kernel='gaussian', bandwidth=0.75).fit(X)
log_dens = kde.score_samples(X_plot)
ax[1, 1].fill(X_plot[:, 0], np.exp(log_dens), fc='#AAAAFF')
ax[1, 1].text(-3.5, 0.31, "Gaussian Kernel Density")

for axi in ax.ravel():
 axi.plot(X[:, 0], np.full(X.shape[0], -0.01), '+k')
 axi.set_xlim(-4, 9)
 axi.set_ylim(-0.02, 0.34)

for axi in ax[:, 0]:
 axi.set_ylabel('Normalized Density')

for axi in ax[1, :]:
 axi.set_xlabel('x')

#-----
Plot all available kernels
X_plot = np.linspace(-6, 6, 1000)[:, None]
X_src = np.zeros((1, 1))

fig, ax = plt.subplots(2, 3, sharex=True, sharey=True)
fig.subplots_adjust(left=0.05, right=0.95, hspace=0.05, wspace=0.05)

def format_func(x, loc):
 if x == 0:
 return '0'
 elif x == 1:
 return 'h'
 elif x == -1:
 return '-h'
 else:
 return '%ih' % x

for i, kernel in enumerate(['gaussian', 'tophat', 'epanechnikov',
 'exponential', 'linear', 'cosine']):
 axi = ax.ravel()[i]
 log_dens = KernelDensity(kernel=kernel).fit(X_src).score_samples(X_plot)
 axi.fill(X_plot[:, 0], np.exp(log_dens), '-k', fc='#AAAAFF')
 axi.text(-2.6, 0.95, kernel)

 axi.xaxis.set_major_formatter(plt.FuncFormatter(format_func))

```

```

axi.xaxis.set_major_locator(plt.MultipleLocator(1))
axi.yaxis.set_major_locator(plt.NullLocator())

axi.set_ylim(0, 1.05)
axi.set_xlim(-2.9, 2.9)

ax[0, 1].set_title('Available Kernels')

#-----
Plot a 1D density example
N = 100
np.random.seed(1)
X = np.concatenate((np.random.normal(0, 1, int(0.3 * N)),
 np.random.normal(5, 1, int(0.7 * N))))[:, np.newaxis]

X_plot = np.linspace(-5, 10, 1000)[:, np.newaxis]

true_dens = (0.3 * norm(0, 1).pdf(X_plot[:, 0])
 + 0.7 * norm(5, 1).pdf(X_plot[:, 0]))

fig, ax = plt.subplots()
ax.fill(X_plot[:, 0], true_dens, fc='black', alpha=0.2,
 label='input distribution')

for kernel in ['gaussian', 'tophat', 'epanechnikov']:
 kde = KernelDensity(kernel=kernel, bandwidth=0.5).fit(X)
 log_dens = kde.score_samples(X_plot)
 ax.plot(X_plot[:, 0], np.exp(log_dens), '-',
 label="kernel = '{0}'".format(kernel))

ax.text(6, 0.38, "N={0} points".format(N))

ax.legend(loc='upper left')
ax.plot(X[:, 0], -0.005 - 0.01 * np.random.random(X.shape[0]), '+k')

ax.set_xlim(-4, 9)
ax.set_ylim(-0.02, 0.4)
plt.show()

```

**Total running time of the script:** ( 0 minutes 0.145 seconds)

## 5.24 Neural Networks

Examples concerning the `sklearn.neural_network` module.

---

**Note:** Click [here](#) to download the full example code

---

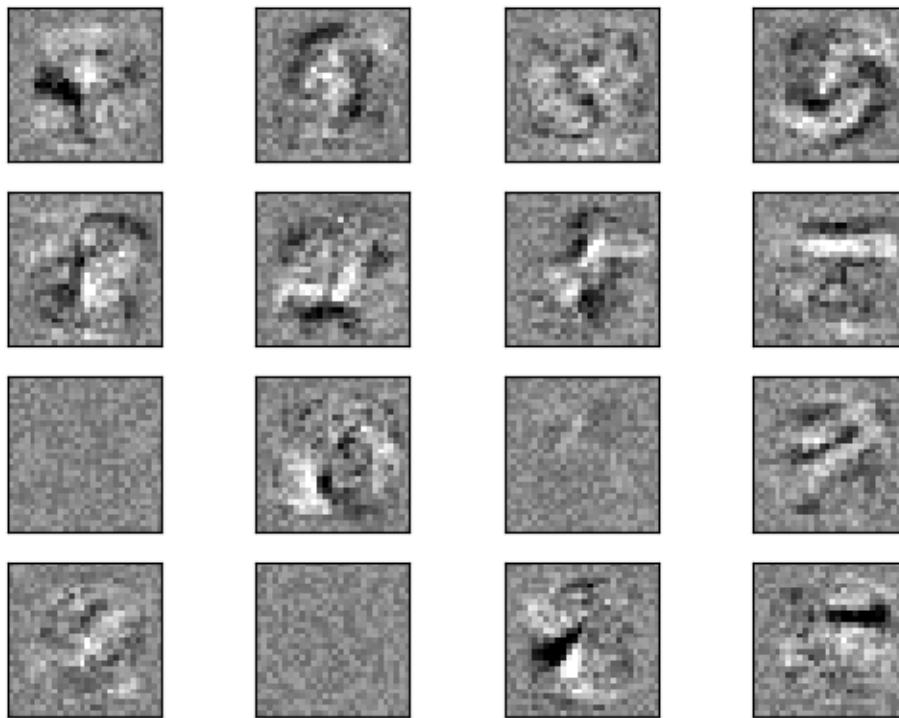
### 5.24.1 Visualization of MLP weights on MNIST

Sometimes looking at the learned coefficients of a neural network can provide insight into the learning behavior. For example if weights look unstructured, maybe some were not used at all, or if very large coefficients exist, maybe regularization was too low or the learning rate too high.

This example shows how to plot some of the first layer weights in a MLPClassifier trained on the MNIST dataset.

The input data consists of 28x28 pixel handwritten digits, leading to 784 features in the dataset. Therefore the first layer weight matrix have the shape (784, hidden\_layer\_sizes[0]). We can therefore visualize a single column of the weight matrix as a 28x28 pixel image.

To make the example run faster, we use very few hidden units, and train only for a very short time. Training longer would result in weights with a much smoother spatial appearance.



Out:

```
Iteration 1, loss = 0.32009978
Iteration 2, loss = 0.15347534
Iteration 3, loss = 0.11544755
Iteration 4, loss = 0.09279764
Iteration 5, loss = 0.07889367
Iteration 6, loss = 0.07170497
Iteration 7, loss = 0.06282111
Iteration 8, loss = 0.05530788
Iteration 9, loss = 0.04960484
Iteration 10, loss = 0.04645355
Training set score: 0.986800
Test set score: 0.970000
```

```

import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.neural_network import MLPClassifier

print(__doc__)

Load data from https://www.openml.org/d/554
X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
X = X / 255.

rescale the data, use the traditional train/test split
X_train, X_test = X[:60000], X[60000:]
y_train, y_test = y[:60000], y[60000:]

mlp = MLPClassifier(hidden_layer_sizes=(100, 100), max_iter=400, alpha=1e-4,
solver='sgd', verbose=10, tol=1e-4, random_state=1)
mlp = MLPClassifier(hidden_layer_sizes=(50,), max_iter=10, alpha=1e-4,
 solver='sgd', verbose=10, tol=1e-4, random_state=1,
 learning_rate_init=.1)

mlp.fit(X_train, y_train)
print("Training set score: %f" % mlp.score(X_train, y_train))
print("Test set score: %f" % mlp.score(X_test, y_test))

fig, axes = plt.subplots(4, 4)
use global min / max to ensure all weights are shown on the same scale
vmin, vmax = mlp.coefs_[0].min(), mlp.coefs_[0].max()
for coef, ax in zip(mlp.coefs_[0].T, axes.ravel()):
 ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray, vmin=.5 * vmin,
 vmax=.5 * vmax)
 ax.set_xticks(())
 ax.set_yticks(())

plt.show()

```

**Total running time of the script:** ( 0 minutes 27.631 seconds)

---

**Note:** Click [here](#) to download the full example code

---

## 5.24.2 Restricted Boltzmann Machine features for digit classification

For greyscale image data where pixel values can be interpreted as degrees of blackness on a white background, like handwritten digit recognition, the Bernoulli Restricted Boltzmann machine model ([BernoulliRBM](#)) can perform effective non-linear feature extraction.

In order to learn good latent representations from a small dataset, we artificially generate more labeled data by perturbing the training data with linear shifts of 1 pixel in each direction.

This example shows how to build a classification pipeline with a BernoulliRBM feature extractor and a [LogisticRegression](#) classifier. The hyperparameters of the entire model (learning rate, hidden layer size, regularization) were optimized by grid search, but the search is not reproduced here because of runtime constraints.

Logistic regression on raw pixel values is presented for comparison. The example shows that the features extracted by the BernoulliRBM help improve the classification accuracy.

## 100 components extracted by RBM



Out:

```
[BernoulliRBM] Iteration 1, pseudo-likelihood = -25.39, time = 0.15s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -23.72, time = 0.34s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -22.72, time = 0.33s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -21.86, time = 0.32s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -21.66, time = 0.35s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -21.00, time = 0.37s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -20.75, time = 0.35s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -20.52, time = 0.34s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -20.38, time = 0.31s
[BernoulliRBM] Iteration 10, pseudo-likelihood = -20.23, time = 0.35s
[BernoulliRBM] Iteration 11, pseudo-likelihood = -20.02, time = 0.34s
[BernoulliRBM] Iteration 12, pseudo-likelihood = -19.93, time = 0.35s
[BernoulliRBM] Iteration 13, pseudo-likelihood = -19.71, time = 0.33s
[BernoulliRBM] Iteration 14, pseudo-likelihood = -19.69, time = 0.33s
[BernoulliRBM] Iteration 15, pseudo-likelihood = -19.61, time = 0.33s
[BernoulliRBM] Iteration 16, pseudo-likelihood = -19.57, time = 0.32s
[BernoulliRBM] Iteration 17, pseudo-likelihood = -19.36, time = 0.33s
[BernoulliRBM] Iteration 18, pseudo-likelihood = -19.22, time = 0.36s
[BernoulliRBM] Iteration 19, pseudo-likelihood = -19.31, time = 0.35s
[BernoulliRBM] Iteration 20, pseudo-likelihood = -19.21, time = 0.34s
Logistic regression using RBM features:
```

|   | precision | recall | f1-score | support |
|---|-----------|--------|----------|---------|
| 0 | 0.99      | 0.98   | 0.99     | 174     |
| 1 | 0.94      | 0.95   | 0.94     | 184     |
| 2 | 0.89      | 0.96   | 0.92     | 166     |
| 3 | 0.93      | 0.88   | 0.90     | 194     |
| 4 | 0.96      | 0.94   | 0.95     | 186     |
| 5 | 0.92      | 0.90   | 0.91     | 181     |
| 6 | 0.98      | 0.98   | 0.98     | 207     |

|                                                   |           |        |          |         |
|---------------------------------------------------|-----------|--------|----------|---------|
| 7                                                 | 0.92      | 0.99   | 0.96     | 154     |
| 8                                                 | 0.87      | 0.84   | 0.86     | 182     |
| 9                                                 | 0.91      | 0.91   | 0.91     | 169     |
| accuracy                                          |           |        | 0.93     | 1797    |
| macro avg                                         | 0.93      | 0.93   | 0.93     | 1797    |
| weighted avg                                      | 0.93      | 0.93   | 0.93     | 1797    |
| <br>Logistic regression using raw pixel features: |           |        |          |         |
|                                                   | precision | recall | f1-score | support |
| 0                                                 | 0.90      | 0.91   | 0.91     | 174     |
| 1                                                 | 0.60      | 0.58   | 0.59     | 184     |
| 2                                                 | 0.75      | 0.85   | 0.80     | 166     |
| 3                                                 | 0.78      | 0.78   | 0.78     | 194     |
| 4                                                 | 0.81      | 0.84   | 0.83     | 186     |
| 5                                                 | 0.76      | 0.77   | 0.77     | 181     |
| 6                                                 | 0.91      | 0.87   | 0.89     | 207     |
| 7                                                 | 0.85      | 0.88   | 0.87     | 154     |
| 8                                                 | 0.67      | 0.57   | 0.62     | 182     |
| 9                                                 | 0.75      | 0.77   | 0.76     | 169     |
| accuracy                                          |           |        | 0.78     | 1797    |
| macro avg                                         | 0.78      | 0.78   | 0.78     | 1797    |
| weighted avg                                      | 0.78      | 0.78   | 0.78     | 1797    |

```
print(__doc__)

Authors: Yann N. Dauphin, Vlad Niculae, Gabriel Synnaeve
License: BSD

import numpy as np
import matplotlib.pyplot as plt

from scipy.ndimage import convolve
from sklearn import linear_model, datasets, metrics
from sklearn.model_selection import train_test_split
from sklearn.neural_network import BernoulliRBM
from sklearn.pipeline import Pipeline
from sklearn.base import clone

Setting up
Setting up

def nudge_dataset(X, Y):
 """
 This produces a dataset 5 times bigger than the original one,
 by moving the 8x8 images in X around by 1px to left, right, down, up
 """
 direction_vectors = [
 [0, 1, 0],
 [-1, 0, 0],
 [0, -1, 0],
 [1, 0, 0]
]
```

```

[0, 0, 0],
[0, 0, 0]],

[[0, 0, 0],
 [1, 0, 0],
 [0, 0, 0]],

[[0, 0, 0],
 [0, 0, 1],
 [0, 0, 0]],

[[0, 0, 0],
 [0, 0, 0],
 [0, 1, 0]]]

def shift(x, w):
 return convolve(x.reshape((8, 8)), mode='constant', weights=w).ravel()

X = np.concatenate([X] +
 [np.apply_along_axis(shift, 1, X, vector)
 for vector in direction_vectors])
Y = np.concatenate([Y for _ in range(5)], axis=0)
return X, Y

Load Data
digits = datasets.load_digits()
X = np.asarray(digits.data, 'float32')
X, Y = nudge_dataset(X, digits.target)
X = (X - np.min(X, 0)) / (np.max(X, 0) + 0.0001) # 0-1 scaling

X_train, X_test, Y_train, Y_test = train_test_split(
 X, Y, test_size=0.2, random_state=0)

Models we will use
logistic = linear_model.LogisticRegression(solver='newton-cg', tol=1,
 multi_class='multinomial')
rbm = BernoulliRBM(random_state=0, verbose=True)

rbm_features_classifier = Pipeline(
 steps=[('rbm', rbm), ('logistic', logistic)])

#####
Training

Hyper-parameters. These were set by cross-validation,
using a GridSearchCV. Here we are not performing cross-validation to
save time.
rbm.learning_rate = 0.06
rbm.n_iter = 20
More components tend to give better prediction performance, but larger
fitting time
rbm.n_components = 100
logistic.C = 6000

Training RBM-Logistic Pipeline
rbm_features_classifier.fit(X_train, Y_train)

```

```

Training the Logistic regression classifier directly on the pixel
raw_pixel_classifier = clone(logistic)
raw_pixel_classifier.C = 100.
raw_pixel_classifier.fit(X_train, Y_train)

##########
Evaluation

Y_pred = rbm_features_classifier.predict(X_test)
print("Logistic regression using RBM features:\n%s\n" % (
 metrics.classification_report(Y_test, Y_pred)))

Y_pred = raw_pixel_classifier.predict(X_test)
print("Logistic regression using raw pixel features:\n%s\n" % (
 metrics.classification_report(Y_test, Y_pred)))

#####
Plotting

plt.figure(figsize=(4.2, 4))
for i, comp in enumerate(rbm.components_):
 plt.subplot(10, 10, i + 1)
 plt.imshow(comp.reshape((8, 8)), cmap=plt.cm.gray_r,
 interpolation='nearest')
 plt.xticks(())
 plt.yticks(())
plt.suptitle('100 components extracted by RBM', fontsize=16)
plt.subplots_adjust(0.08, 0.02, 0.92, 0.85, 0.08, 0.23)

plt.show()

```

**Total running time of the script:** ( 0 minutes 11.939 seconds)

---

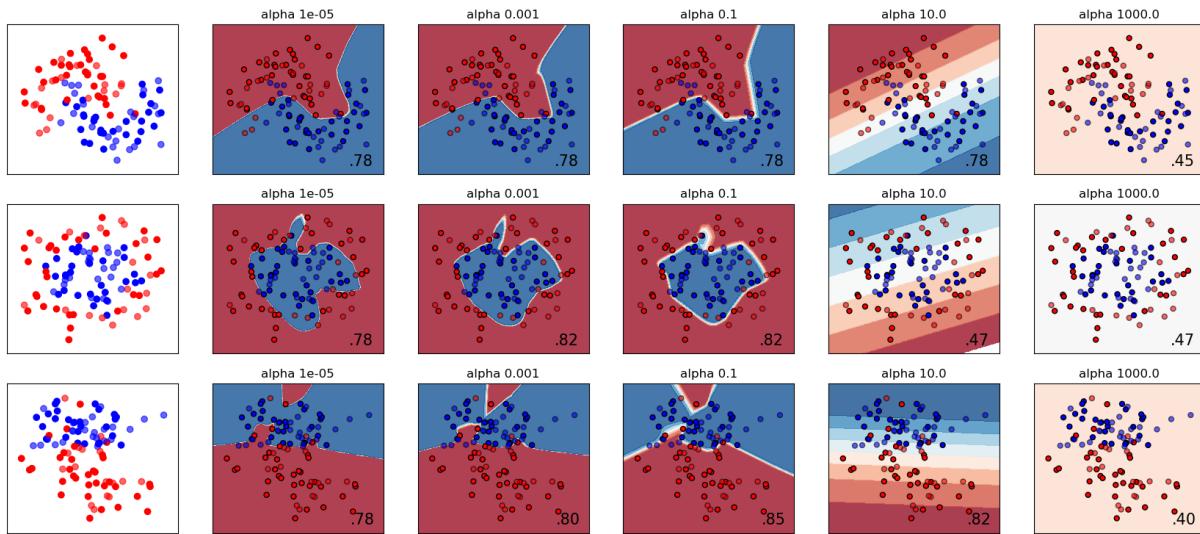
**Note:** Click [here](#) to download the full example code

---

### 5.24.3 Varying regularization in Multi-layer Perceptron

A comparison of different values for regularization parameter ‘alpha’ on synthetic datasets. The plot shows that different alphas yield different decision functions.

Alpha is a parameter for regularization term, aka penalty term, that combats overfitting by constraining the size of the weights. Increasing alpha may fix high variance (a sign of overfitting) by encouraging smaller weights, resulting in a decision boundary plot that appears with lesser curvatures. Similarly, decreasing alpha may fix high bias (a sign of underfitting) by encouraging larger weights, potentially resulting in a more complicated decision boundary.



```

print(__doc__)

Author: Issam H. Laradji
License: BSD 3 clause

import numpy as np
from matplotlib import pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.neural_network import MLPClassifier

h = .02 # step size in the mesh

alphas = np.logspace(-5, 3, 5)
names = ['alpha ' + str(i) for i in alphas]

classifiers = []
for i in alphas:
 classifiers.append(MLPClassifier(solver='lbfgs', alpha=i, random_state=1,
 hidden_layer_sizes=[100, 100]))

X, y = make_classification(n_features=2, n_redundant=0, n_informative=2,
 random_state=0, n_clusters_per_class=1)
rng = np.random.RandomState(2)
X += 2 * rng.uniform(size=X.shape)
linearly_separable = (X, y)

datasets = [make_moons(noise=0.3, random_state=0),
 make_circles(noise=0.2, factor=0.5, random_state=1),
 linearly_separable]

figure = plt.figure(figsize=(17, 9))

```

```

i = 1
iterate over datasets
for X, y in datasets:
 # preprocess dataset, split into training and test part
 X = StandardScaler().fit_transform(X)
 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.4)

 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))

 # just plot the dataset first
 cm = plt.cm.RdBu
 cm_bright = ListedColormap(['#FF0000', '#0000FF'])
 ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
 # Plot the training points
 ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright)
 # and testing points
 ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, alpha=0.6)
 ax.set_xlim(xx.min(), xx.max())
 ax.set_ylim(yy.min(), yy.max())
 ax.set_xticks(())
 ax.set_yticks(())
 i += 1

 # iterate over classifiers
 for name, clf in zip(names, classifiers):
 ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
 clf.fit(X_train, y_train)
 score = clf.score(X_test, y_test)

 # Plot the decision boundary. For that, we will assign a color to each
 # point in the mesh [x_min, x_max]x[y_min, y_max].
 if hasattr(clf, "decision_function"):
 Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
 else:
 Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

 # Put the result into a color plot
 Z = Z.reshape(xx.shape)
 ax.contourf(xx, yy, Z, cmap=cm, alpha=.8)

 # Plot also the training points
 ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
 edgecolors='black', s=25)
 # and testing points
 ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright,
 alpha=0.6, edgecolors='black', s=25)

 ax.set_xlim(xx.min(), xx.max())
 ax.set_ylim(yy.min(), yy.max())
 ax.set_xticks(())
 ax.set_yticks(())
 ax.set_title(name)
 ax.text(xx.max() - .3, yy.min() + .3, ('%.2f' % score).lstrip('0'),
 size=15, horizontalalignment='right')
 i += 1

```

```
figure.subplots_adjust(left=.02, right=.98)
plt.show()
```

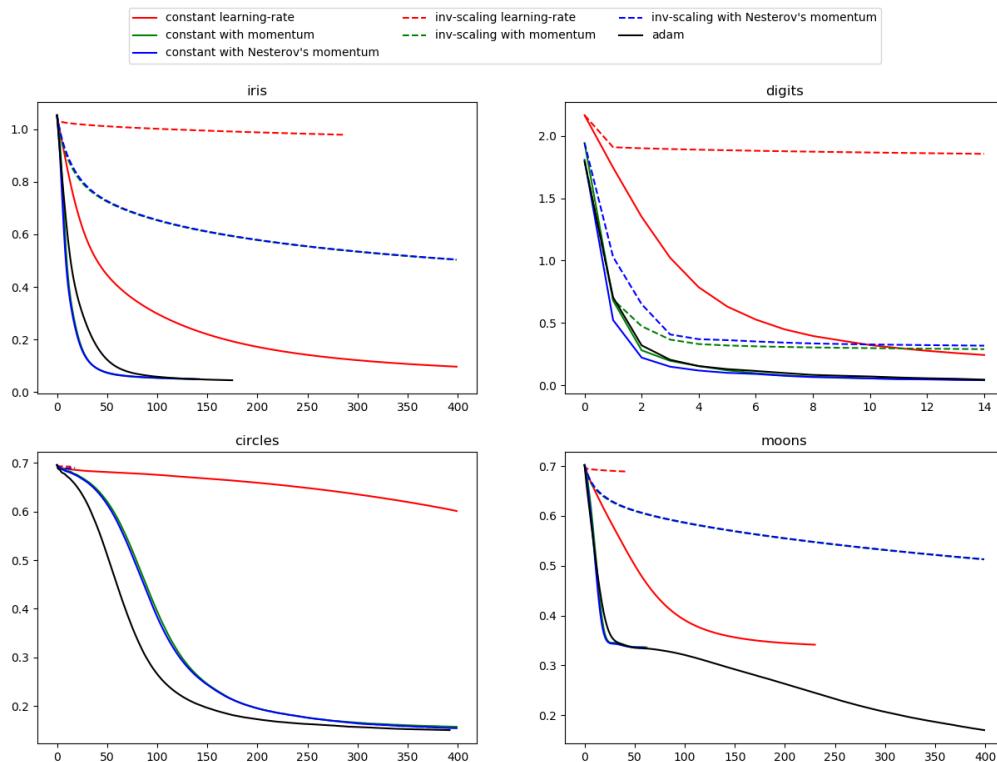
**Total running time of the script:** ( 0 minutes 7.509 seconds)

**Note:** Click [here](#) to download the full example code

#### 5.24.4 Compare Stochastic learning strategies for MLPClassifier

This example visualizes some training loss curves for different stochastic learning strategies, including SGD and Adam. Because of time-constraints, we use several small datasets, for which L-BFGS might be more suitable. The general trend shown in these examples seems to carry over to larger datasets, however.

Note that those results can be highly dependent on the value of `learning_rate_init`.



Out:

```
learning on dataset iris
training: constant learning-rate
Training set score: 0.980000
Training set loss: 0.096950
training: constant with momentum
Training set score: 0.980000
Training set loss: 0.049530
training: constant with Nesterov's momentum
```

```
Training set score: 0.980000
Training set loss: 0.049540
training: inv-scaling learning-rate
Training set score: 0.360000
Training set loss: 0.978444
training: inv-scaling with momentum
Training set score: 0.860000
Training set loss: 0.503452
training: inv-scaling with Nesterov's momentum
Training set score: 0.860000
Training set loss: 0.504185
training: adam
Training set score: 0.980000
Training set loss: 0.045311

learning on dataset digits
training: constant learning-rate
Training set score: 0.956038
Training set loss: 0.243802
training: constant with momentum
Training set score: 0.992766
Training set loss: 0.041297
training: constant with Nesterov's momentum
Training set score: 0.993879
Training set loss: 0.042898
training: inv-scaling learning-rate
Training set score: 0.638843
Training set loss: 1.855465
training: inv-scaling with momentum
Training set score: 0.912632
Training set loss: 0.290584
training: inv-scaling with Nesterov's momentum
Training set score: 0.909293
Training set loss: 0.318387
training: adam
Training set score: 0.991653
Training set loss: 0.045934

learning on dataset circles
training: constant learning-rate
Training set score: 0.840000
Training set loss: 0.601052
training: constant with momentum
Training set score: 0.940000
Training set loss: 0.157334
training: constant with Nesterov's momentum
Training set score: 0.940000
Training set loss: 0.154453
training: inv-scaling learning-rate
Training set score: 0.500000
Training set loss: 0.692470
training: inv-scaling with momentum
Training set score: 0.500000
Training set loss: 0.689143
training: inv-scaling with Nesterov's momentum
Training set score: 0.500000
Training set loss: 0.689751
training: adam
```

```
Training set score: 0.940000
Training set loss: 0.150527

learning on dataset moons
training: constant learning-rate
Training set score: 0.850000
Training set loss: 0.341523
training: constant with momentum
Training set score: 0.850000
Training set loss: 0.336188
training: constant with Nesterov's momentum
Training set score: 0.850000
Training set loss: 0.335919
training: inv-scaling learning-rate
Training set score: 0.500000
Training set loss: 0.689015
training: inv-scaling with momentum
Training set score: 0.830000
Training set loss: 0.512595
training: inv-scaling with Nesterov's momentum
Training set score: 0.830000
Training set loss: 0.513034
training: adam
Training set score: 0.930000
Training set loss: 0.170087
```

```
print(__doc__)

import warnings

import matplotlib.pyplot as plt

from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import MinMaxScaler
from sklearn import datasets
from sklearn.exceptions import ConvergenceWarning

different learning rate schedules and momentum parameters
params = [{"solver": "sgd", "learning_rate": "constant", "momentum": 0,
 "learning_rate_init": 0.2},
 {"solver": "sgd", "learning_rate": "constant", "momentum": .9,
 "nesterovs_momentum": False, "learning_rate_init": 0.2},
 {"solver": "sgd", "learning_rate": "constant", "momentum": .9,
 "nesterovs_momentum": True, "learning_rate_init": 0.2},
 {"solver": "sgd", "learning_rate": "invscaling", "momentum": 0,
 "learning_rate_init": 0.2},
 {"solver": "sgd", "learning_rate": "invscaling", "momentum": .9,
 "nesterovs_momentum": True, "learning_rate_init": 0.2},
 {"solver": "sgd", "learning_rate": "invscaling", "momentum": .9,
 "nesterovs_momentum": False, "learning_rate_init": 0.2},
 {"solver": "adam", "learning_rate_init": 0.01}]

labels = ["constant learning-rate", "constant with momentum",
```

```

"constant with Nesterov's momentum",
"inv-scaling learning-rate", "inv-scaling with momentum",
"inv-scaling with Nesterov's momentum", "adam"]

plot_args = [{'c': 'red', 'linestyle': '-'},
 {'c': 'green', 'linestyle': '-'},
 {'c': 'blue', 'linestyle': '-'},
 {'c': 'red', 'linestyle': '--'},
 {'c': 'green', 'linestyle': '--'},
 {'c': 'blue', 'linestyle': '--'},
 {'c': 'black', 'linestyle': '-'}]

def plot_on_dataset(X, y, ax, name):
 # for each dataset, plot learning for each learning strategy
 print("\nlearning on dataset %s" % name)
 ax.set_title(name)

 X = MinMaxScaler().fit_transform(X)
 mlps = []
 if name == "digits":
 # digits is larger but converges fairly quickly
 max_iter = 15
 else:
 max_iter = 400

 for label, param in zip(labels, params):
 print("training: %s" % label)
 mlp = MLPClassifier(verbose=0, random_state=0,
 max_iter=max_iter, **param)

 # some parameter combinations will not converge as can be seen on the
 # plots so they are ignored here
 with warnings.catch_warnings():
 warnings.filterwarnings("ignore", category=ConvergenceWarning,
 module="sklearn")
 mlp.fit(X, y)

 mlps.append(mlp)
 print("Training set score: %f" % mlp.score(X, y))
 print("Training set loss: %f" % mlp.loss_)
 for mlp, label, args in zip(mlps, labels, plot_args):
 ax.plot(mlp.loss_curve_, label=label, **args)

fig, axes = plt.subplots(2, 2, figsize=(15, 10))
load / generate some toy datasets
iris = datasets.load_iris()
digits = datasets.load_digits()
data_sets = [(iris.data, iris.target),
 (digits.data, digits.target),
 datasets.make_circles(noise=0.2, factor=0.5, random_state=1),
 datasets.make_moons(noise=0.3, random_state=0)]

for ax, data, name in zip(axes.ravel(), data_sets, ['iris', 'digits',
 'circles', 'moons']):
 plot_on_dataset(*data, ax=ax, name=name)

```

```
fig.legend(ax.get_lines(), labels, ncol=3, loc="upper center")
plt.show()
```

Total running time of the script: ( 0 minutes 4.379 seconds)

## 5.25 Preprocessing

Examples concerning the `sklearn.preprocessing` module.

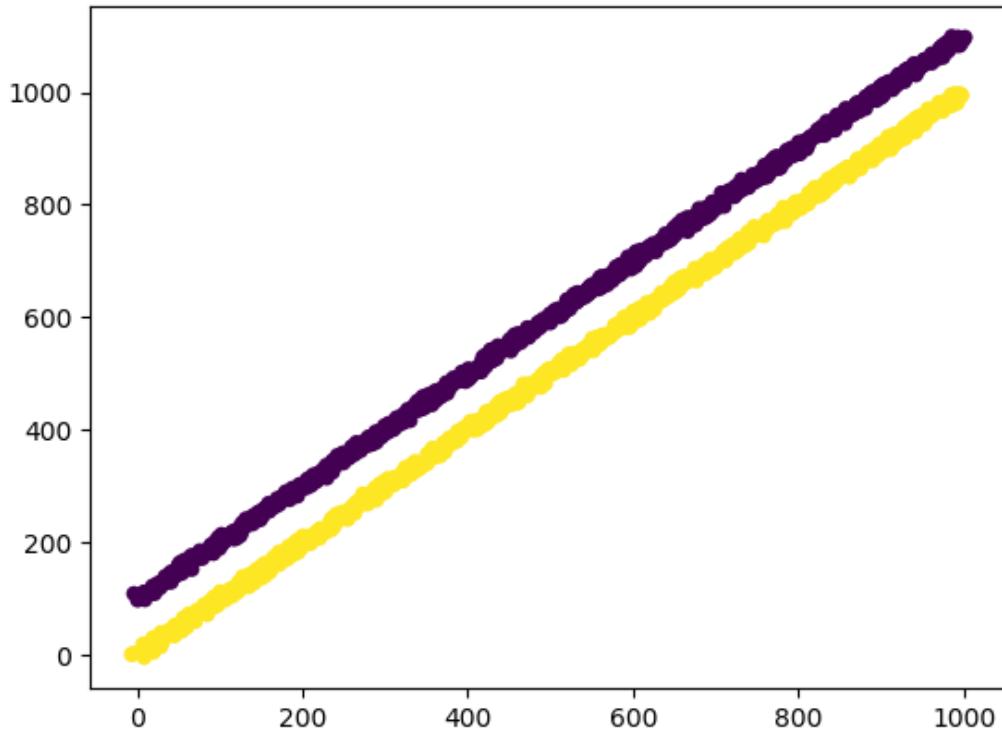
---

**Note:** Click [here](#) to download the full example code

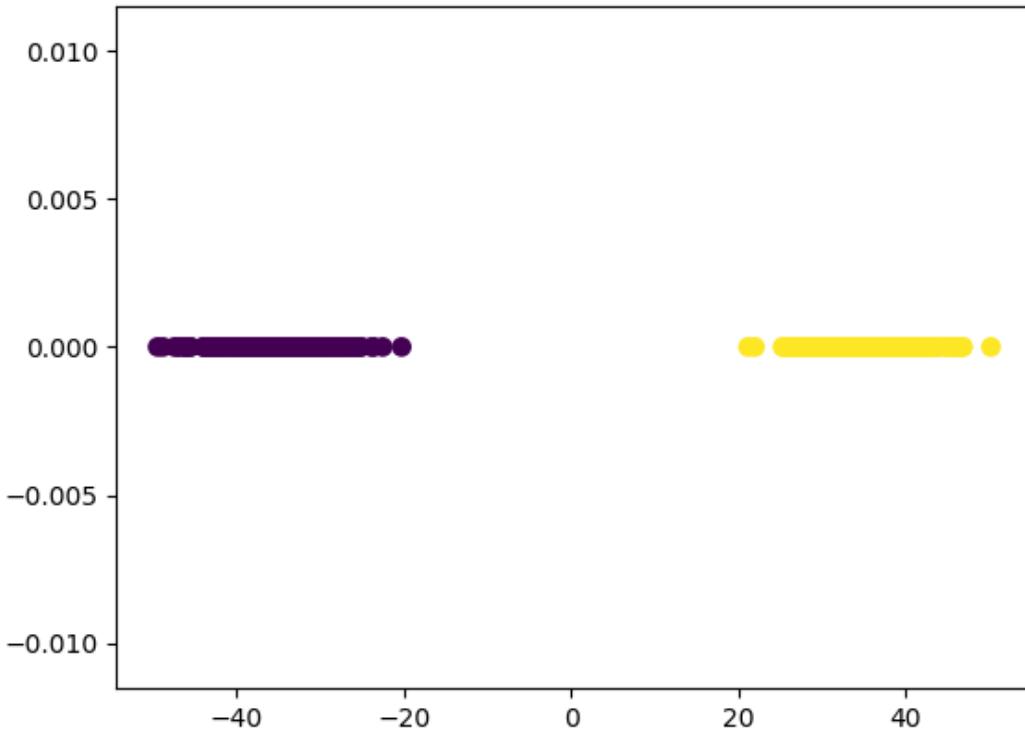
---

### 5.25.1 Using FunctionTransformer to select columns

Shows how to use a function transformer in a pipeline. If you know your dataset's first principle component is irrelevant for a classification task, you can use the FunctionTransformer to select all but the first column of the PCA transformed data.



•



```
import matplotlib.pyplot as plt
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import FunctionTransformer

def _generate_vector(shift=0.5, noise=15):
 return np.arange(1000) + (np.random.rand(1000) - shift) * noise

def generate_dataset():
 """
 This dataset is two lines with a slope ~ 1, where one has
 a y offset of ~100
 """
 return np.vstack((
 np.vstack((
 _generate_vector(),
 _generate_vector() + 100,
)).T,
 np.vstack((
 _generate_vector(),
 _generate_vector(),
)).T,
))
```

```
) , np.hstack((np.zeros(1000) , np.ones(1000)))

def all_but_first_column(X):
 return X[:, 1:]

def drop_first_component(X, y):
 """
 Create a pipeline with PCA and the column selector and use it to
 transform the dataset.
 """
 pipeline = make_pipeline(
 PCA(), FunctionTransformer(all_but_first_column),
)
 X_train, X_test, y_train, y_test = train_test_split(X, y)
 pipeline.fit(X_train, y_train)
 return pipeline.transform(X_test), y_test

if __name__ == '__main__':
 X, y = generate_dataset()
 lw = 0
 plt.figure()
 plt.scatter(X[:, 0], X[:, 1], c=y, lw=lw)
 plt.figure()
 X_transformed, y_transformed = drop_first_component(*generate_dataset())
 plt.scatter(
 X_transformed[:, 0],
 np.zeros(len(X_transformed)),
 c=y_transformed,
 lw=lw,
 s=60
)
 plt.show()
```

**Total running time of the script:** ( 0 minutes 0.028 seconds)

---

**Note:** Click [here](#) to download the full example code

---

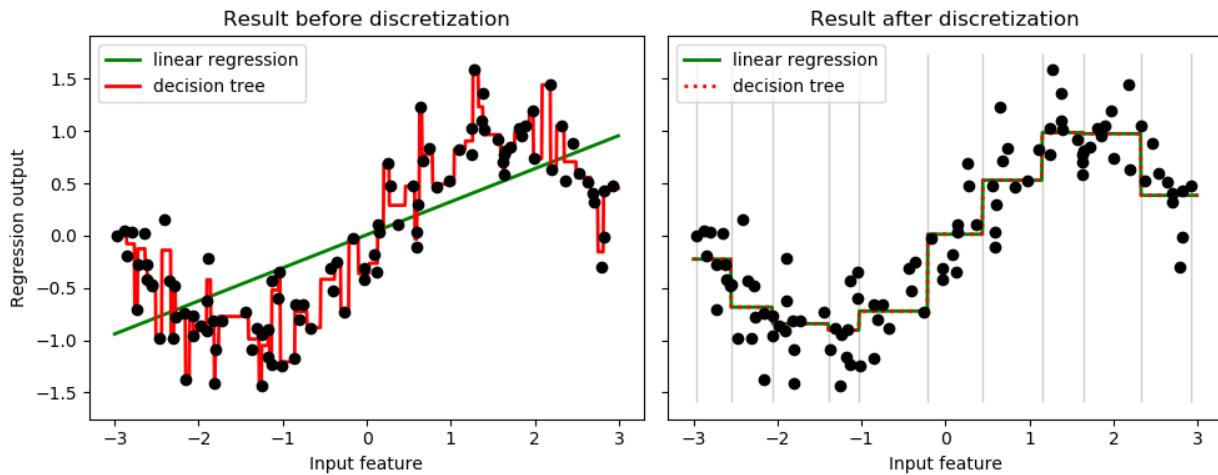
## 5.25.2 Using KBinsDiscretizer to discretize continuous features

The example compares prediction result of linear regression (linear model) and decision tree (tree based model) with and without discretization of real-valued features.

As is shown in the result before discretization, linear model is fast to build and relatively straightforward to interpret, but can only model linear relationships, while decision tree can build a much more complex model of the data. One way to make linear model more powerful on continuous data is to use discretization (also known as binning). In the example, we discretize the feature and one-hot encode the transformed data. Note that if the bins are not reasonably wide, there would appear to be a substantially increased risk of overfitting, so the discretizer parameters should usually be tuned under cross validation.

After discretization, linear regression and decision tree make exactly the same prediction. As features are constant within each bin, any model must predict the same value for all points within a bin. Compared with the result before

discretization, linear model become much more flexible while decision tree gets much less flexible. Note that binning features generally has no beneficial effect for tree-based models, as these models can learn to split up the data anywhere.



```
Author: Andreas Müller
Hanmin Qin <qinhanmin2005@sina.com>
License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import KBinsDiscretizer
from sklearn.tree import DecisionTreeRegressor

print(__doc__)

construct the dataset
rnd = np.random.RandomState(42)
X = rnd.uniform(-3, 3, size=100)
y = np.sin(X) + rnd.normal(size=len(X)) / 3
X = X.reshape(-1, 1)

transform the dataset with KBinsDiscretizer
enc = KBinsDiscretizer(n_bins=10, encode='onehot')
X_binned = enc.fit_transform(X)

predict with original dataset
fig, (ax1, ax2) = plt.subplots(ncols=2, sharey=True, figsize=(10, 4))
line = np.linspace(-3, 3, 1000, endpoint=False).reshape(-1, 1)
reg = LinearRegression().fit(X, y)
ax1.plot(line, reg.predict(line), linewidth=2, color='green',
 label="linear regression")
reg = DecisionTreeRegressor(min_samples_split=3, random_state=0).fit(X, y)
ax1.plot(line, reg.predict(line), linewidth=2, color='red',
 label="decision tree")
ax1.plot(X[:, 0], y, 'o', c='k')
ax1.legend(loc="best")
ax1.set_ylabel("Regression output")
ax1.set_xlabel("Input feature")
ax1.set_title("Result before discretization")

predict with discretized dataset
X_binned = enc.transform(X)
reg = LinearRegression().fit(X_binned, y)
ax2.plot(line, reg.predict(line), linewidth=2, color='green',
 label="linear regression")
reg = DecisionTreeRegressor(min_samples_split=3, random_state=0).fit(X_binned, y)
ax2.plot(line, reg.predict(line), linewidth=2, color='red',
 label="decision tree")
ax2.set_title("Result after discretization")
```

```
predict with transformed dataset
line_binned = enc.transform(line)
reg = LinearRegression().fit(X_binned, y)
ax2.plot(line, reg.predict(line_binned), linewidth=2, color='green',
 linestyle='--', label='linear regression')
reg = DecisionTreeRegressor(min_samples_split=3,
 random_state=0).fit(X_binned, y)
ax2.plot(line, reg.predict(line_binned), linewidth=2, color='red',
 linestyle=':', label='decision tree')
ax2.plot(X[:, 0], y, 'o', c='k')
ax2.vlines(enc.bin_edges_[0], *plt.gca().get_ylim(), linewidth=1, alpha=.2)
ax2.legend(loc="best")
ax2.set_xlabel("Input feature")
ax2.set_title("Result after discretization")

plt.tight_layout()
plt.show()
```

**Total running time of the script:** ( 0 minutes 0.122 seconds)

---

**Note:** Click [here](#) to download the full example code

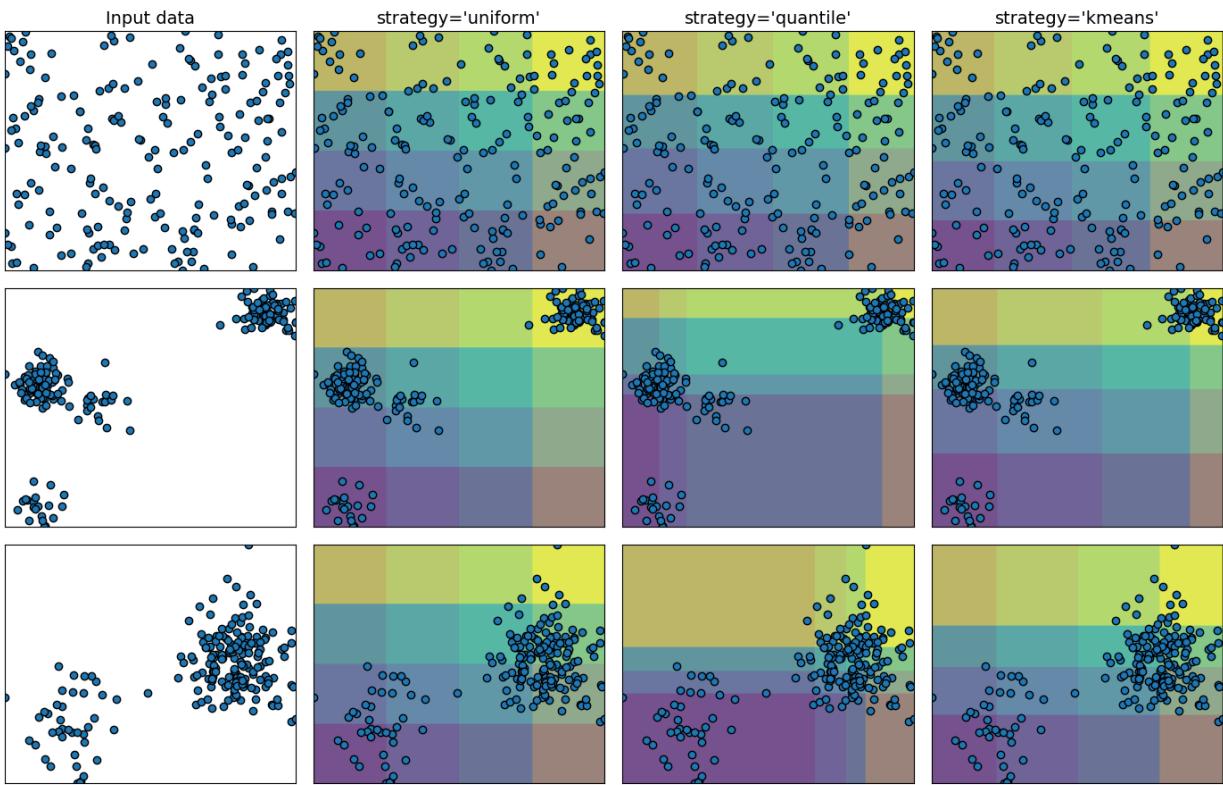
---

### 5.25.3 Demonstrating the different strategies of KBinsDiscretizer

This example presents the different strategies implemented in KBinsDiscretizer:

- ‘uniform’: The discretization is uniform in each feature, which means that the bin widths are constant in each dimension.
- ‘quantile’: The discretization is done on the quantiled values, which means that each bin has approximately the same number of samples.
- ‘kmeans’: The discretization is based on the centroids of a KMeans clustering procedure.

The plot shows the regions where the discretized encoding is constant.



```
Author: Tom Dupré la Tour
License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.preprocessing import KBinsDiscretizer
from sklearn.datasets import make_blobs

print(__doc__)

strategies = ['uniform', 'quantile', 'kmeans']

n_samples = 200
centers_0 = np.array([[0, 0], [0, 5], [2, 4], [8, 8]])
centers_1 = np.array([[0, 0], [3, 1]])

construct the datasets
random_state = 42
X_list = [
 np.random.RandomState(random_state).uniform(-3, 3, size=(n_samples, 2)),
 make_blobs(n_samples=[n_samples // 10, n_samples * 4 // 10,
 n_samples // 10, n_samples * 4 // 10],
 cluster_std=0.5, centers=centers_0,
 random_state=random_state)[0],
 make_blobs(n_samples=[n_samples // 5, n_samples * 4 // 5],
 cluster_std=0.5, centers=centers_1,
 random_state=random_state)[0],
]

```

```
figure = plt.figure(figsize=(14, 9))
i = 1
for ds_cnt, X in enumerate(X_list):

 ax = plt.subplot(len(X_list), len(strategies) + 1, i)
 ax.scatter(X[:, 0], X[:, 1], edgecolors='k')
 if ds_cnt == 0:
 ax.set_title("Input data", size=14)

 xx, yy = np.meshgrid(
 np.linspace(X[:, 0].min(), X[:, 0].max(), 300),
 np.linspace(X[:, 1].min(), X[:, 1].max(), 300))
 grid = np.c_[xx.ravel(), yy.ravel()]

 ax.set_xlim(xx.min(), xx.max())
 ax.set_ylim(yy.min(), yy.max())
 ax.set_xticks(())
 ax.set_yticks(())

 i += 1
 # transform the dataset with KBinsDiscretizer
 for strategy in strategies:
 enc = KBinsDiscretizer(n_bins=4, encode='ordinal', strategy=strategy)
 enc.fit(X)
 grid_encoded = enc.transform(grid)

 ax = plt.subplot(len(X_list), len(strategies) + 1, i)

 # horizontal stripes
 horizontal = grid_encoded[:, 0].reshape(xx.shape)
 ax.contourf(xx, yy, horizontal, alpha=.5)
 # vertical stripes
 vertical = grid_encoded[:, 1].reshape(xx.shape)
 ax.contourf(xx, yy, vertical, alpha=.5)

 ax.scatter(X[:, 0], X[:, 1], edgecolors='k')
 ax.set_xlim(xx.min(), xx.max())
 ax.set_ylim(yy.min(), yy.max())
 ax.set_xticks(())
 ax.set_yticks(())
 if ds_cnt == 0:
 ax.set_title("strategy='%" + str(strategy) + "'", size=14)

 i += 1

plt.tight_layout()
plt.show()
```

**Total running time of the script:** ( 0 minutes 0.890 seconds)

---

**Note:** Click [here](#) to download the full example code

---

## 5.25.4 Importance of Feature Scaling

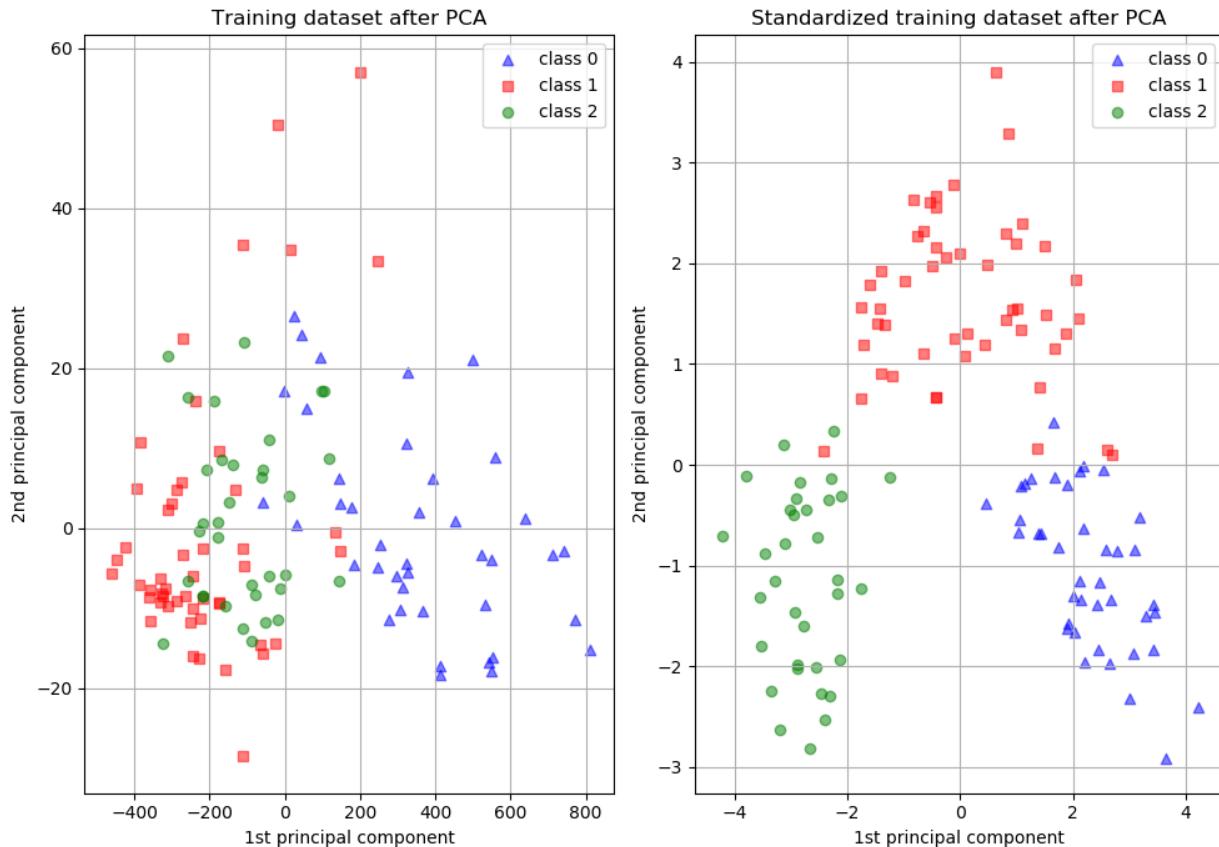
Feature scaling through standardization (or Z-score normalization) can be an important preprocessing step for many machine learning algorithms. Standardization involves rescaling the features such that they have the properties of a standard normal distribution with a mean of zero and a standard deviation of one.

While many algorithms (such as SVM, K-nearest neighbors, and logistic regression) require features to be normalized, intuitively we can think of Principle Component Analysis (PCA) as being a prime example of when normalization is important. In PCA we are interested in the components that maximize the variance. If one component (e.g. human height) varies less than another (e.g. weight) because of their respective scales (meters vs. kilos), PCA might determine that the direction of maximal variance more closely corresponds with the ‘weight’ axis, if those features are not scaled. As a change in height of one meter can be considered much more important than the change in weight of one kilogram, this is clearly incorrect.

To illustrate this, PCA is performed comparing the use of data with `StandardScaler` applied, to unscaled data. The results are visualized and a clear difference noted. The 1st principal component in the unscaled set can be seen. It can be seen that feature #13 dominates the direction, being a whole two orders of magnitude above the other features. This is contrasted when observing the principal component for the scaled version of the data. In the scaled version, the orders of magnitude are roughly the same across all the features.

The dataset used is the Wine Dataset available at UCI. This dataset has continuous features that are heterogeneous in scale due to differing properties that they measure (i.e alcohol content, and malic acid).

The transformed data is then used to train a naive Bayes classifier, and a clear difference in prediction accuracies is observed wherein the dataset which is scaled before PCA vastly outperforms the unscaled version.



Out:

```
Prediction accuracy for the normal test dataset with PCA
81.48%

Prediction accuracy for the standardized test dataset with PCA
98.15%

PC 1 without scaling:
[1.76e-03 -8.36e-04 1.55e-04 -5.31e-03 2.02e-02 1.02e-03 1.53e-03
-1.12e-04 6.31e-04 2.33e-03 1.54e-04 7.43e-04 1.00e+00]

PC 1 with scaling:
[0.13 -0.26 -0.01 -0.23 0.16 0.39 0.42 -0.28 0.33 -0.11 0.3 0.38
 0.28]
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.naive_bayes import GaussianNB
from sklearn import metrics
import matplotlib.pyplot as plt
from sklearn.datasets import load_wine
from sklearn.pipeline import make_pipeline
print(__doc__)

Code source: Tyler Lanigan <tylerlanigan@gmail.com>
Sebastian Raschka <mail@sebastianraschka.com>

License: BSD 3 clause

RANDOM_STATE = 42
FIG_SIZE = (10, 7)

features, target = load_wine(return_X_y=True)

Make a train/test split using 30% test size
X_train, X_test, y_train, y_test = train_test_split(features, target,
 test_size=0.30,
 random_state=RANDOM_STATE)

Fit to data and predict using pipelined GNB and PCA.
unscaled_clf = make_pipeline(PCA(n_components=2), GaussianNB())
unscaled_clf.fit(X_train, y_train)
pred_test = unscaled_clf.predict(X_test)

Fit to data and predict using pipelined scaling, GNB and PCA.
std_clf = make_pipeline(StandardScaler(), PCA(n_components=2), GaussianNB())
std_clf.fit(X_train, y_train)
pred_test_std = std_clf.predict(X_test)

Show prediction accuracies in scaled and unscaled data.
```

```

print('\nPrediction accuracy for the normal test dataset with PCA')
print(' {:.2%}\n'.format(metrics.accuracy_score(y_test, pred_test)))

print('\nPrediction accuracy for the standardized test dataset with PCA')
print(' {:.2%}\n'.format(metrics.accuracy_score(y_test, pred_test_std)))

Extract PCA from pipeline
pca = unscaled_clf.named_steps['pca']
pca_std = std_clf.named_steps['pca']

Show first principal components
print('\nPC 1 without scaling:\n', pca.components_[0])
print('\nPC 1 with scaling:\n', pca_std.components_[0])

Use PCA without and with scale on X_train data for visualization.
X_train_transformed = pca.transform(X_train)
scaler = std_clf.named_steps['standardscaler']
X_train_std_transformed = pca_std.transform(scaler.transform(X_train))

visualize standardized vs. untouched dataset with PCA performed
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=FIG_SIZE)

for l, c, m in zip(range(0, 3), ('blue', 'red', 'green'), ('^', 's', 'o')):
 ax1.scatter(X_train_transformed[y_train == l, 0],
 X_train_transformed[y_train == l, 1],
 color=c,
 label='class %s' % l,
 alpha=0.5,
 marker=m
)

for l, c, m in zip(range(0, 3), ('blue', 'red', 'green'), ('^', 's', 'o')):
 ax2.scatter(X_train_std_transformed[y_train == l, 0],
 X_train_std_transformed[y_train == l, 1],
 color=c,
 label='class %s' % l,
 alpha=0.5,
 marker=m
)

ax1.set_title('Training dataset after PCA')
ax2.set_title('Standardized training dataset after PCA')

for ax in (ax1, ax2):
 ax.set_xlabel('1st principal component')
 ax.set_ylabel('2nd principal component')
 ax.legend(loc='upper right')
 ax.grid()

plt.tight_layout()
plt.show()

```

**Total running time of the script:** ( 0 minutes 0.109 seconds)

---

**Note:** Click [here](#) to download the full example code

---

## 5.25.5 Map data to a normal distribution

This example demonstrates the use of the Box-Cox and Yeo-Johnson transforms through `preprocessing.PowerTransformer` to map data from various distributions to a normal distribution.

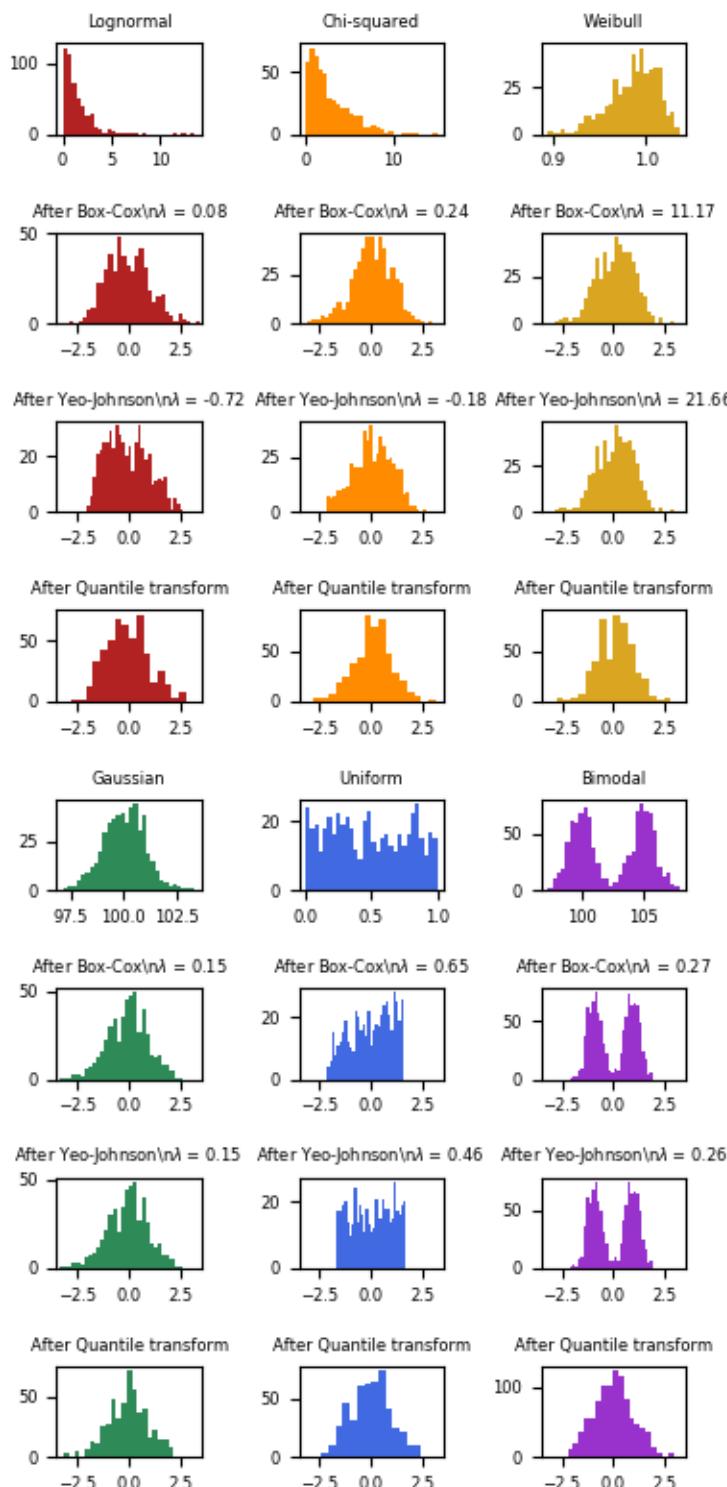
The power transform is useful as a transformation in modeling problems where homoscedasticity and normality are desired. Below are examples of Box-Cox and Yeo-Johnson applied to six different probability distributions: Lognormal, Chi-squared, Weibull, Gaussian, Uniform, and Bimodal.

Note that the transformations successfully map the data to a normal distribution when applied to certain datasets, but are ineffective with others. This highlights the importance of visualizing the data before and after transformation.

Also note that even though Box-Cox seems to perform better than Yeo-Johnson for lognormal and chi-squared distributions, keep in mind that Box-Cox does not support inputs with negative values.

For comparison, we also add the output from `preprocessing.QuantileTransformer`. It can force any arbitrary distribution into a gaussian, provided that there are enough training samples (thousands). Because it is a non-parametric method, it is harder to interpret than the parametric ones (Box-Cox and Yeo-Johnson).

On “small” datasets (less than a few hundred points), the quantile transformer is prone to overfitting. The use of the power transform is then recommended.



```
Author: Eric Chang <ericchang2017@u.northwestern.edu>
Nicolas Hug <contact@nicolas-hug.com>
License: BSD 3 clause

import numpy as np
```

```
import matplotlib.pyplot as plt

from sklearn.preprocessing import PowerTransformer
from sklearn.preprocessing import QuantileTransformer
from sklearn.model_selection import train_test_split

print(__doc__)

N_SAMPLES = 1000
FONT_SIZE = 6
BINS = 30

rng = np.random.RandomState(304)
bc = PowerTransformer(method='box-cox')
yj = PowerTransformer(method='yeo-johnson')
n_quantiles is set to the training set size rather than the default value
to avoid a warning being raised by this example
qt = QuantileTransformer(n_quantiles=500, output_distribution='normal',
 random_state=rng)
size = (N_SAMPLES, 1)

lognormal distribution
X_lognormal = rng.lognormal(size=size)

chi-squared distribution
df = 3
X_chisq = rng.chisquare(df=df, size=size)

weibull distribution
a = 50
X_weibull = rng.weibull(a=a, size=size)

gaussian distribution
loc = 100
X_gaussian = rng.normal(loc=loc, size=size)

uniform distribution
X_uniform = rng.uniform(low=0, high=1, size=size)

bimodal distribution
loc_a, loc_b = 100, 105
X_a, X_b = rng.normal(loc=loc_a, size=size), rng.normal(loc=loc_b, size=size)
X_bimodal = np.concatenate([X_a, X_b], axis=0)

create plots
distributions = [
 ('Lognormal', X_lognormal),
 ('Chi-squared', X_chisq),
 ('Weibull', X_weibull),
 ('Gaussian', X_gaussian),
 ('Uniform', X_uniform),
 ('Bimodal', X_bimodal)
]
```

```

colors = ['firebrick', 'darkorange', 'goldenrod',
 'seagreen', 'royalblue', 'darkorchid']

fig, axes = plt.subplots(nrows=8, ncols=3, figsize=plt.figaspect(2))
axes = axes.flatten()
axes_idxs = [(0, 3, 6, 9), (1, 4, 7, 10), (2, 5, 8, 11), (12, 15, 18, 21),
 (13, 16, 19, 22), (14, 17, 20, 23)]
axes_list = [(axes[i], axes[j], axes[k], axes[l])
 for (i, j, k, l) in axes_idxs]

for distribution, color, axes in zip(distributions, colors, axes_list):
 name, X = distribution
 X_train, X_test = train_test_split(X, test_size=.5)

 # perform power transforms and quantile transform
 X_trans_bc = bc.fit(X_train).transform(X_test)
 lmbda_bc = round(bc.lambdas_[0], 2)
 X_trans_yj = yj.fit(X_train).transform(X_test)
 lmbda_yj = round(yj.lambdas_[0], 2)
 X_trans_qt = qt.fit(X_train).transform(X_test)

 ax_original, ax_bc, ax_yj, ax_qt = axes

 ax_original.hist(X_train, color=color, bins=BINS)
 ax_original.set_title(name, fontsize=FONT_SIZE)
 ax_original.tick_params(axis='both', which='major', labelsize=FONT_SIZE)

 for ax, X_trans, meth_name, lmbda in zip(
 (ax_bc, ax_yj, ax_qt),
 (X_trans_bc, X_trans_yj, X_trans_qt),
 ('Box-Cox', 'Yeo-Johnson', 'Quantile transform'),
 (lmbda_bc, lmbda_yj, None)):
 ax.hist(X_trans, color=color, bins=BINS)
 title = 'After {}'.format(meth_name)
 if lmbda is not None:
 title += r'\nλ = {}'.format(lmbda)
 ax.set_title(title, fontsize=FONT_SIZE)
 ax.tick_params(axis='both', which='major', labelsize=FONT_SIZE)
 ax.set_xlim([-3.5, 3.5])

plt.tight_layout()
plt.show()

```

**Total running time of the script:** ( 0 minutes 1.536 seconds)

---

**Note:** Click [here](#) to download the full example code

---

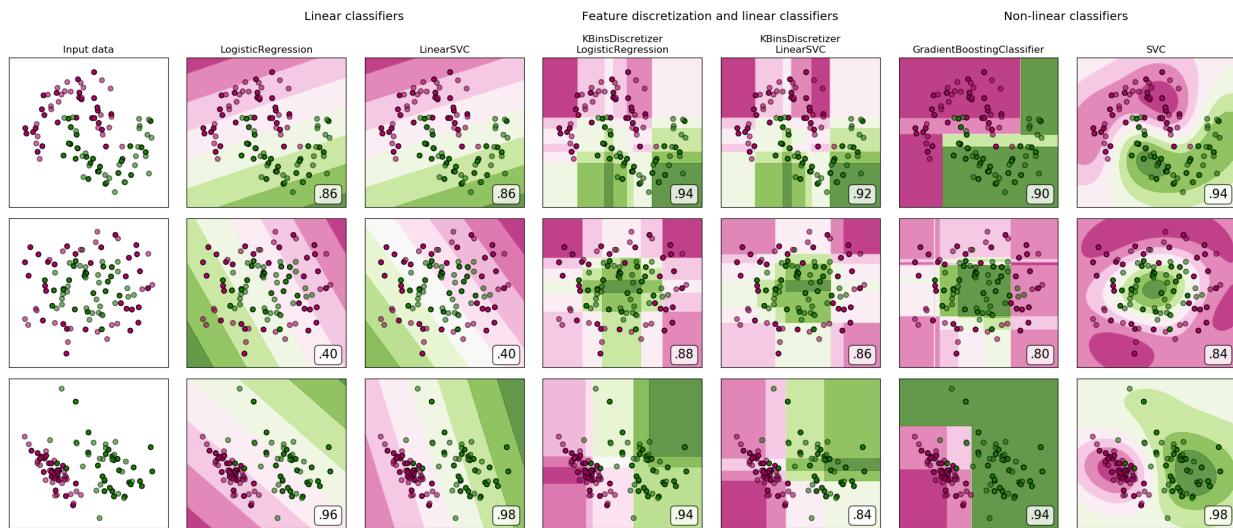
## 5.25.6 Feature discretization

A demonstration of feature discretization on synthetic classification datasets. Feature discretization decomposes each feature into a set of bins, here equally distributed in width. The discrete values are then one-hot encoded, and given to a linear classifier. This preprocessing enables a non-linear behavior even though the classifier is linear.

On this example, the first two rows represent linearly non-separable datasets (moons and concentric circles) while the third is approximately linearly separable. On the two linearly non-separable datasets, feature discretization largely increases the performance of linear classifiers. On the linearly separable dataset, feature discretization decreases the performance of linear classifiers. Two non-linear classifiers are also shown for comparison.

This example should be taken with a grain of salt, as the intuition conveyed does not necessarily carry over to real datasets. Particularly in high-dimensional spaces, data can more easily be separated linearly. Moreover, using feature discretization and one-hot encoding increases the number of features, which easily lead to overfitting when the number of samples is small.

The plots show training points in solid colors and testing points semi-transparent. The lower right shows the classification accuracy on the test set.



Out:

```
dataset 0

LogisticRegression: 0.86
LinearSVC: 0.86
KBinsDiscretizer + LogisticRegression: 0.94
KBinsDiscretizer + LinearSVC: 0.92
GradientBoostingClassifier: 0.90
SVC: 0.94

dataset 1

LogisticRegression: 0.40
LinearSVC: 0.40
KBinsDiscretizer + LogisticRegression: 0.88
KBinsDiscretizer + LinearSVC: 0.86
GradientBoostingClassifier: 0.80
SVC: 0.84

dataset 2

LogisticRegression: 0.96
LinearSVC: 0.98
KBinsDiscretizer + LogisticRegression: 0.94
KBinsDiscretizer + LinearSVC: 0.84
GradientBoostingClassifier: 0.94
```

```
SVC: 0.98
```

```
Code source: Tom Dupré la Tour
Adapted from plot_classifier_comparison by Gaël Varoquaux and Andreas Müller
#
License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import KBinsDiscretizer
from sklearn.svm import SVC, LinearSVC
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.utils.testing import ignore_warnings
from sklearn.exceptions import ConvergenceWarning

print(__doc__)

h = .02 # step size in the mesh

def get_name(estimator):
 name = estimator.__class__.__name__
 if name == 'Pipeline':
 name = [get_name(est[1]) for est in estimator.steps]
 name = ' + '.join(name)
 return name

list of (estimator, param_grid), where param_grid is used in GridSearchCV
classifiers = [
 (LogisticRegression(solver='lbfgs', random_state=0), {
 'C': np.logspace(-2, 7, 10)
 }),
 (LinearSVC(random_state=0), {
 'C': np.logspace(-2, 7, 10)
 }),
 (make_pipeline(
 KBinsDiscretizer(encode='onehot'),
 LogisticRegression(solver='lbfgs', random_state=0)), {
 'kbinsdiscretizer_n_bins': np.arange(2, 10),
 'logisticregression_C': np.logspace(-2, 7, 10),
 }),
 (make_pipeline(
 KBinsDiscretizer(encode='onehot'), LinearSVC(random_state=0)), {
 'kbinsdiscretizer_n_bins': np.arange(2, 10),
 'linearsvc_C': np.logspace(-2, 7, 10),
 })
]
```

```

 }),
(GradientBoostingClassifier(n_estimators=50, random_state=0), {
 'learning_rate': np.logspace(-4, 0, 10)
}),
(SVC(random_state=0, gamma='scale'), {
 'C': np.logspace(-2, 7, 10)
}),
]
]

names = [get_name(e) for e, g in classifiers]

n_samples = 100
datasets = [
 make_moons(n_samples=n_samples, noise=0.2, random_state=0),
 make_circles(n_samples=n_samples, noise=0.2, factor=0.5, random_state=1),
 make_classification(n_samples=n_samples, n_features=2, n_redundant=0,
 n_informative=2, random_state=2,
 n_clusters_per_class=1)
]

fig, axes = plt.subplots(nrows=len(datasets), ncols=len(classifiers) + 1,
 figsize=(21, 9))

cm = plt.cm.PiYG
cm_bright = ListedColormap(['#b30065', '#178000'])

iterate over datasets
for ds_cnt, (X, y) in enumerate(datasets):
 print('\ndataset %d' % ds_cnt)

 # preprocess dataset, split into training and test part
 X = StandardScaler().fit_transform(X)
 X_train, X_test, y_train, y_test = train_test_split(
 X, y, test_size=.5, random_state=42)

 # create the grid for background colors
 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))

 # plot the dataset first
 ax = axes[ds_cnt, 0]
 if ds_cnt == 0:
 ax.set_title("Input data")
 # plot the training points
 ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
 edgecolors='k')
 # and testing points
 ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, alpha=0.6,
 edgecolors='k')
 ax.set_xlim(xx.min(), xx.max())
 ax.set_ylim(yy.min(), yy.max())
 ax.set_xticks(())
 ax.set_yticks(())

 # iterate over classifiers
 for est_idx, (name, (estimator, param_grid)) in \

```

```

 enumerate(zip(names, classifiers)):
 ax = axes[ds_cnt, est_idx + 1]

 clf = GridSearchCV(estimator=estimator, param_grid=param_grid, cv=5,
 iid=False)
 with ignore_warnings(category=ConvergenceWarning):
 clf.fit(X_train, y_train)
 score = clf.score(X_test, y_test)
 print('%s: %.2f' % (name, score))

 # plot the decision boundary. For that, we will assign a color to each
 # point in the mesh [x_min, x_max]*[y_min, y_max].
 if hasattr(clf, "decision_function"):
 Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
 else:
 Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

 # put the result into a color plot
 Z = Z.reshape(xx.shape)
 ax.contourf(xx, yy, Z, cmap=cm, alpha=.8)

 # plot the training points
 ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
 edgecolors='k')
 # and testing points
 ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright,
 edgecolors='k', alpha=0.6)
 ax.set_xlim(xx.min(), xx.max())
 ax.set_ylim(yy.min(), yy.max())
 ax.set_xticks(())
 ax.set_yticks(())

 if ds_cnt == 0:
 ax.set_title(name.replace(' + ', '\n'))
 ax.text(0.95, 0.06, ('%.2f' % score).lstrip('0'), size=15,
 bbox=dict(boxstyle='round', alpha=0.8, facecolor='white'),
 transform=ax.transAxes, horizontalalignment='right')

plt.tight_layout()

Add subtitles above the figure
plt.subplots_adjust(top=0.90)
suptitles = [
 'Linear classifiers',
 'Feature discretization and linear classifiers',
 'Non-linear classifiers',
]
for i, suptitle in zip([1, 3, 5], suptitles):
 ax = axes[0, i]
 ax.text(1.05, 1.25, suptitle, transform=ax.transAxes,
 horizontalalignment='center', size='x-large')
plt.show()

```

**Total running time of the script:** ( 0 minutes 16.451 seconds)

---

**Note:** Click [here](#) to download the full example code

---

## 5.25.7 Compare the effect of different scalers on data with outliers

Feature 0 (median income in a block) and feature 5 (number of households) of the California housing dataset have very different scales and contain some very large outliers. These two characteristics lead to difficulties to visualize the data and, more importantly, they can degrade the predictive performance of many machine learning algorithms. Unscaled data can also slow down or even prevent the convergence of many gradient-based estimators.

Indeed many estimators are designed with the assumption that each feature takes values close to zero or more importantly that all features vary on comparable scales. In particular, metric-based and gradient-based estimators often assume approximately standardized data (centered features with unit variances). A notable exception are decision tree-based estimators that are robust to arbitrary scaling of the data.

This example uses different scalers, transformers, and normalizers to bring the data within a pre-defined range.

Scalers are linear (or more precisely affine) transformers and differ from each other in the way to estimate the parameters used to shift and scale each feature.

QuantileTransformer provides non-linear transformations in which distances between marginal outliers and inliers are shrunk. PowerTransformer provides non-linear transformations in which data is mapped to a normal distribution to stabilize variance and minimize skewness.

Unlike the previous transformations, normalization refers to a per sample transformation instead of a per feature transformation.

The following code is a bit verbose, feel free to jump directly to the analysis of the [results](#).

```
Author: Raghav RV <rvraghav93@gmail.com>
Guillaume Lemaitre <g.lemaitre58@gmail.com>
Thomas Unterthiner
License: BSD 3 clause

import numpy as np

import matplotlib as mpl
from matplotlib import pyplot as plt
from matplotlib import cm

from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import minmax_scale
from sklearn.preprocessing import MaxAbsScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import Normalizer
from sklearn.preprocessing import QuantileTransformer
from sklearn.preprocessing import PowerTransformer

from sklearn.datasets import fetch_california_housing

print(__doc__)

dataset = fetch_california_housing()
X_full, y_full = dataset.data, dataset.target

Take only 2 features to make visualization easier
```

```

Feature of 0 has a long tail distribution.
Feature 5 has a few but very large outliers.

X = X_full[:, [0, 5]]

distributions = [
 ('Unscaled data', X),
 ('Data after standard scaling',
 StandardScaler().fit_transform(X)),
 ('Data after min-max scaling',
 MinMaxScaler().fit_transform(X)),
 ('Data after max-abs scaling',
 MaxAbsScaler().fit_transform(X)),
 ('Data after robust scaling',
 RobustScaler(quantile_range=(25, 75)).fit_transform(X)),
 ('Data after power transformation (Yeo-Johnson)',
 PowerTransformer(method='yeo-johnson').fit_transform(X)),
 ('Data after power transformation (Box-Cox)',
 PowerTransformer(method='box-cox').fit_transform(X)),
 ('Data after quantile transformation (gaussian pdf)',
 QuantileTransformer(output_distribution='normal')
 .fit_transform(X)),
 ('Data after quantile transformation (uniform pdf)',
 QuantileTransformer(output_distribution='uniform'
 .fit_transform(X)),
 ('Data after sample-wise L2 normalizing',
 Normalizer().fit_transform(X)),
]

scale the output between 0 and 1 for the colorbar
y = minmax_scale(y_full)

plasma does not exist in matplotlib < 1.5
cmap = getattr(cm, 'plasma_r', cm.hot_r)

def create_axes(title, figsize=(16, 6)):
 fig = plt.figure(figsize=figsize)
 fig.suptitle(title)

 # define the axis for the first plot
 left, width = 0.1, 0.22
 bottom, height = 0.1, 0.7
 bottom_h = height + 0.15
 left_h = left + width + 0.02

 rect_scatter = [left, bottom, width, height]
 rect_histx = [left, bottom_h, width, 0.1]
 rect_histy = [left_h, bottom, 0.05, height]

 ax_scatter = plt.axes(rect_scatter)
 ax_histx = plt.axes(rect_histx)
 ax_histy = plt.axes(rect_histy)

 # define the axis for the zoomed-in plot
 left = width + left + 0.2
 left_h = left + width + 0.02

 rect_scatter = [left, bottom, width, height]

```

```

rect_histx = [left, bottom_h, width, 0.1]
rect_histy = [left_h, bottom, 0.05, height]

ax_scatter_zoom = plt.axes(rect_scatter)
ax_histx_zoom = plt.axes(rect_histx)
ax_histy_zoom = plt.axes(rect_histy)

define the axis for the colorbar
left, width = width + left + 0.13, 0.01

rect_colorbar = [left, bottom, width, height]
ax_colorbar = plt.axes(rect_colorbar)

return ((ax_scatter, ax_histy, ax_histx),
 (ax_scatter_zoom, ax_histy_zoom, ax_histx_zoom),
 ax_colorbar)

def plot_distribution(axes, X, y, hist_nbins=50, title="",
 x0_label="", x1_label ""):
 ax, hist_X1, hist_X0 = axes

 ax.set_title(title)
 ax.set_xlabel(x0_label)
 ax.set_ylabel(x1_label)

 # The scatter plot
 colors = cmap(y)
 ax.scatter(X[:, 0], X[:, 1], alpha=0.5, marker='o', s=5, lw=0, c=colors)

 # Removing the top and the right spine for aesthetics
 # make nice axis layout
 ax.spines['top'].set_visible(False)
 ax.spines['right'].set_visible(False)
 ax.get_xaxis().tick_bottom()
 ax.get_yaxis().tick_left()
 ax.spines['left'].set_position(('outward', 10))
 ax.spines['bottom'].set_position(('outward', 10))

 # Histogram for axis X1 (feature 5)
 hist_X1.set_ylim(ax.get_ylim())
 hist_X1.hist(X[:, 1], bins=hist_nbins, orientation='horizontal',
 color='grey', ec='grey')
 hist_X1.axis('off')

 # Histogram for axis X0 (feature 0)
 hist_X0.set_xlim(ax.get_xlim())
 hist_X0.hist(X[:, 0], bins=hist_nbins, orientation='vertical',
 color='grey', ec='grey')
 hist_X0.axis('off')

```

Two plots will be shown for each scaler/normalizer/transformer. The left figure will show a scatter plot of the full data set while the right figure will exclude the extreme values considering only 99 % of the data set, excluding marginal outliers. In addition, the marginal distributions for each feature will be shown on the side of the scatter plot.

```

def make_plot(item_idx):
 title, X = distributions[item_idx]
 ax_zoom_out, ax_zoom_in, ax_colorbar = create_axes(title)

```

```

axarr = (ax_zoom_out, ax_zoom_in)
plot_distribution(axarr[0], X, y, hist_nbins=200,
 x0_label="Median Income",
 x1_label="Number of households",
 title="Full data")

zoom-in
zoom_in_percentile_range = (0, 99)
cutoffs_X0 = np.percentile(X[:, 0], zoom_in_percentile_range)
cutoffs_X1 = np.percentile(X[:, 1], zoom_in_percentile_range)

non_outliers_mask = (
 np.all(X > [cutoffs_X0[0], cutoffs_X1[0]], axis=1) &
 np.all(X < [cutoffs_X0[1], cutoffs_X1[1]], axis=1))
plot_distribution(axarr[1], X[non_outliers_mask], y[non_outliers_mask],
 hist_nbins=50,
 x0_label="Median Income",
 x1_label="Number of households",
 title="Zoom-in")

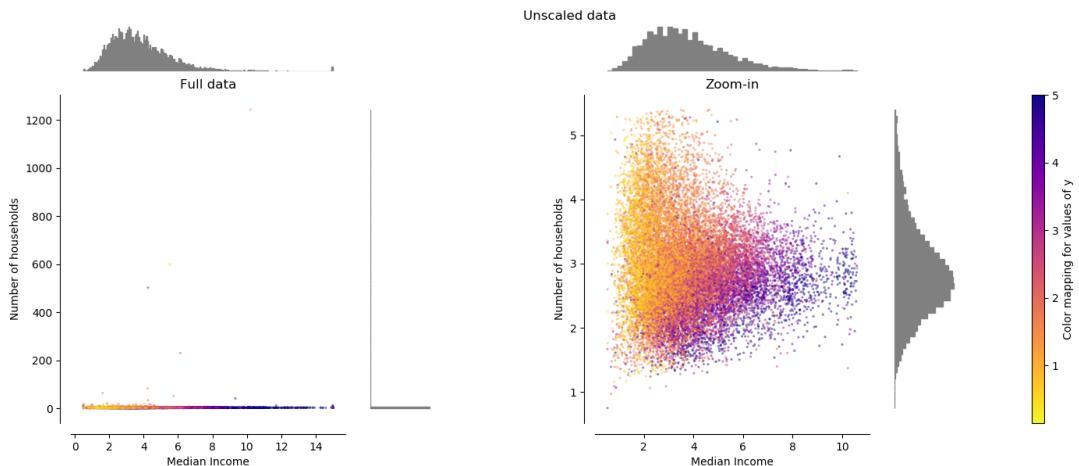
norm = mpl.colors.Normalize(y_full.min(), y_full.max())
mpl.colorbar.ColorbarBase(ax_colorbar, cmap=cmap,
 norm=norm, orientation='vertical',
 label='Color mapping for values of y')

```

## Original data

Each transformation is plotted showing two transformed features, with the left plot showing the entire dataset, and the right zoomed-in to show the dataset without the marginal outliers. A large majority of the samples are compacted to a specific range, [0, 10] for the median income and [0, 6] for the number of households. Note that there are some marginal outliers (some blocks have more than 1200 households). Therefore, a specific pre-processing can be very beneficial depending of the application. In the following, we present some insights and behaviors of those pre-processing methods in the presence of marginal outliers.

```
make_plot(0)
```

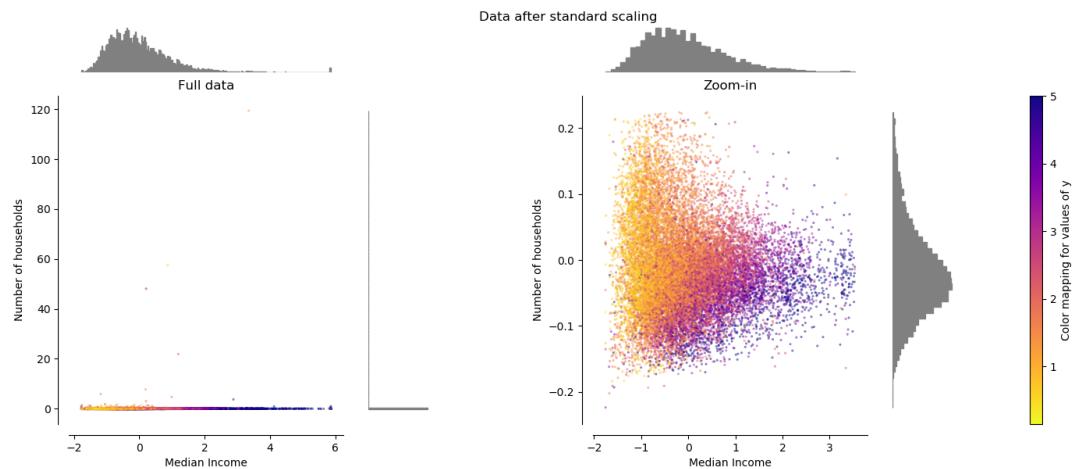


## StandardScaler

`StandardScaler` removes the mean and scales the data to unit variance. However, the outliers have an influence when computing the empirical mean and standard deviation which shrink the range of the feature values as shown in the left figure below. Note in particular that because the outliers on each feature have different magnitudes, the spread of the transformed data on each feature is very different: most of the data lie in the [-2, 4] range for the transformed median income feature while the same data is squeezed in the smaller [-0.2, 0.2] range for the transformed number of households.

`StandardScaler` therefore cannot guarantee balanced feature scales in the presence of outliers.

```
make_plot(1)
```

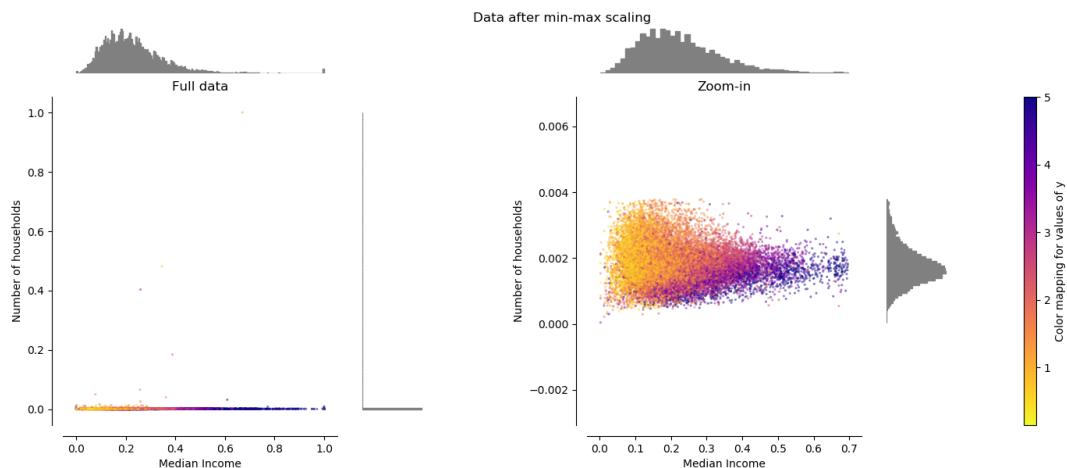


## MinMaxScaler

`MinMaxScaler` rescales the data set such that all feature values are in the range [0, 1] as shown in the right panel below. However, this scaling compresses all inliers in the narrow range [0, 0.005] for the transformed number of households.

As `StandardScaler`, `MinMaxScaler` is very sensitive to the presence of outliers.

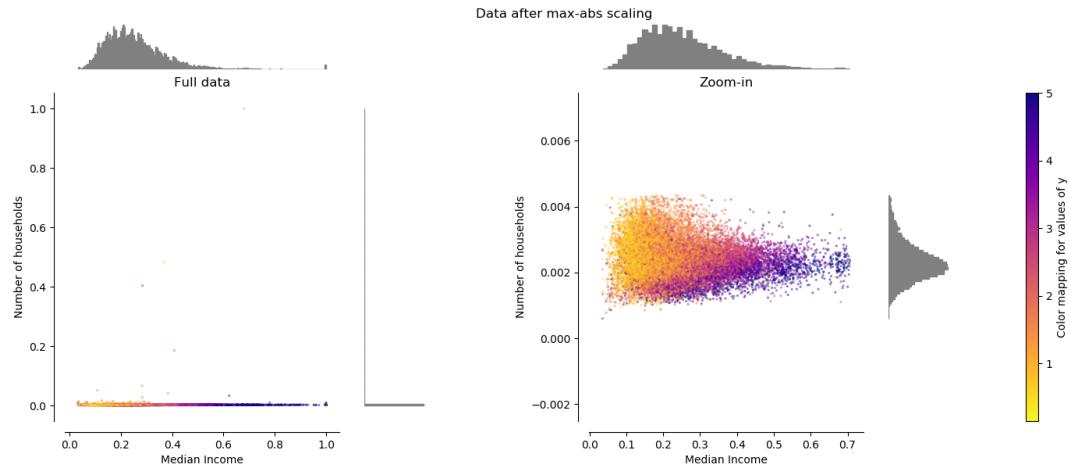
```
make_plot(2)
```



## MaxAbsScaler

`MaxAbsScaler` differs from the previous scaler such that the absolute values are mapped in the range [0, 1]. On positive only data, this scaler behaves similarly to `MinMaxScaler` and therefore also suffers from the presence of large outliers.

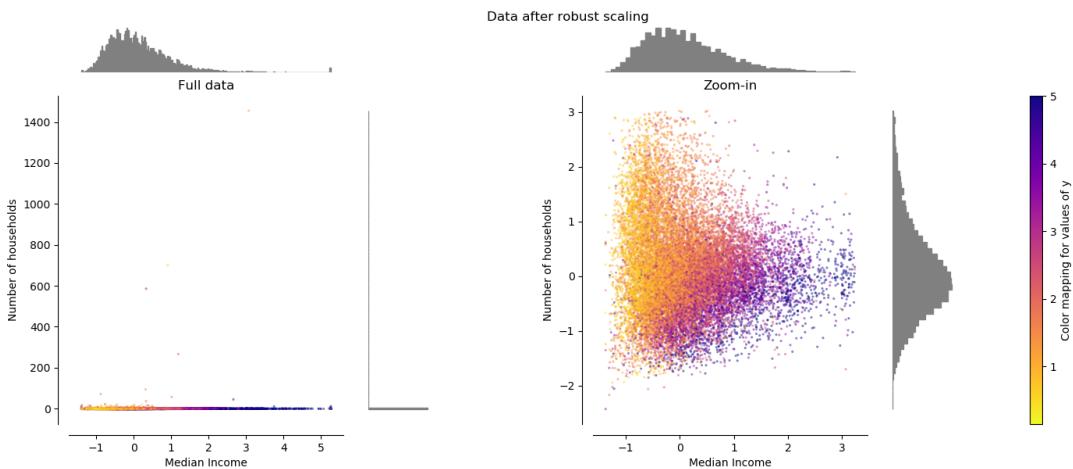
```
make_plot(3)
```



## RobustScaler

Unlike the previous scalers, the centering and scaling statistics of this scaler are based on percentiles and are therefore not influenced by a few number of very large marginal outliers. Consequently, the resulting range of the transformed feature values is larger than for the previous scalers and, more importantly, are approximately similar: for both features most of the transformed values lie in a [-2, 3] range as seen in the zoomed-in figure. Note that the outliers themselves are still present in the transformed data. If a separate outlier clipping is desirable, a non-linear transformation is required (see below).

```
make_plot(4)
```

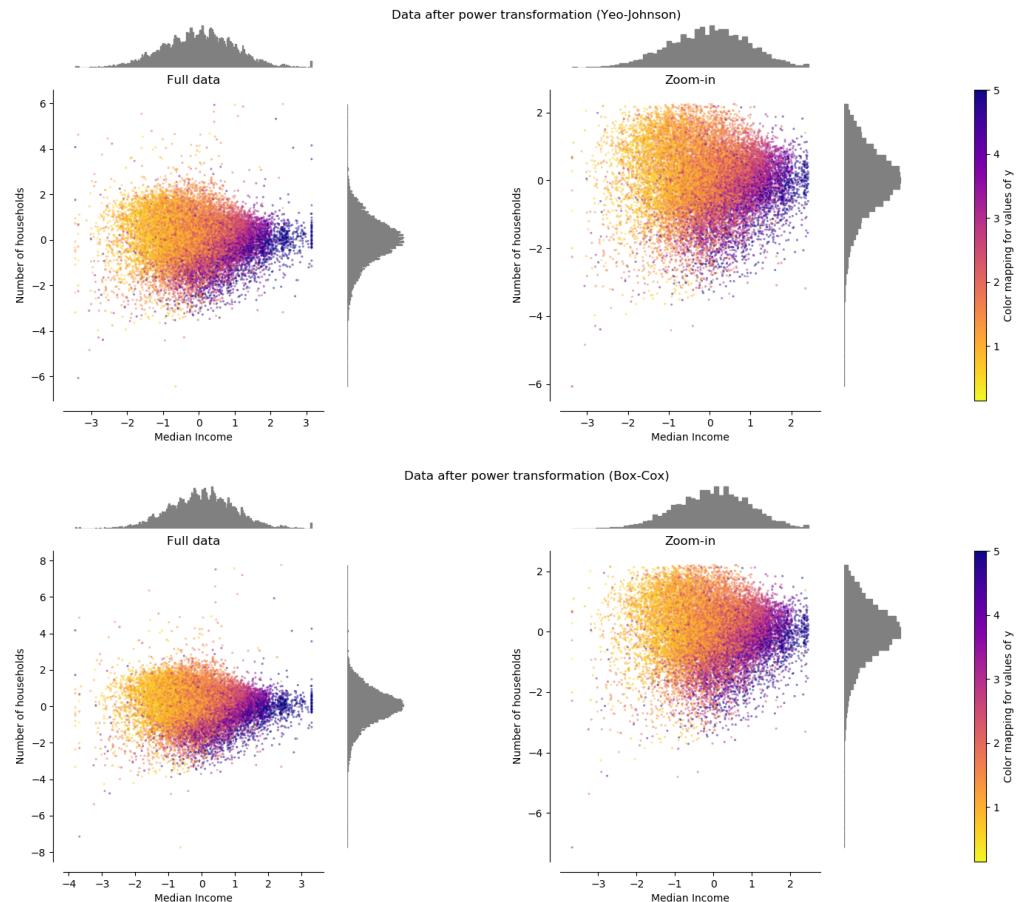


## PowerTransformer

`PowerTransformer` applies a power transformation to each feature to make the data more Gaussian-like. Cur-

rently, `PowerTransformer` implements the Yeo-Johnson and Box-Cox transforms. The power transform finds the optimal scaling factor to stabilize variance and minimize skewness through maximum likelihood estimation. By default, `PowerTransformer` also applies zero-mean, unit variance normalization to the transformed output. Note that Box-Cox can only be applied to strictly positive data. Income and number of households happen to be strictly positive, but if negative values are present the Yeo-Johnson transformed is to be preferred.

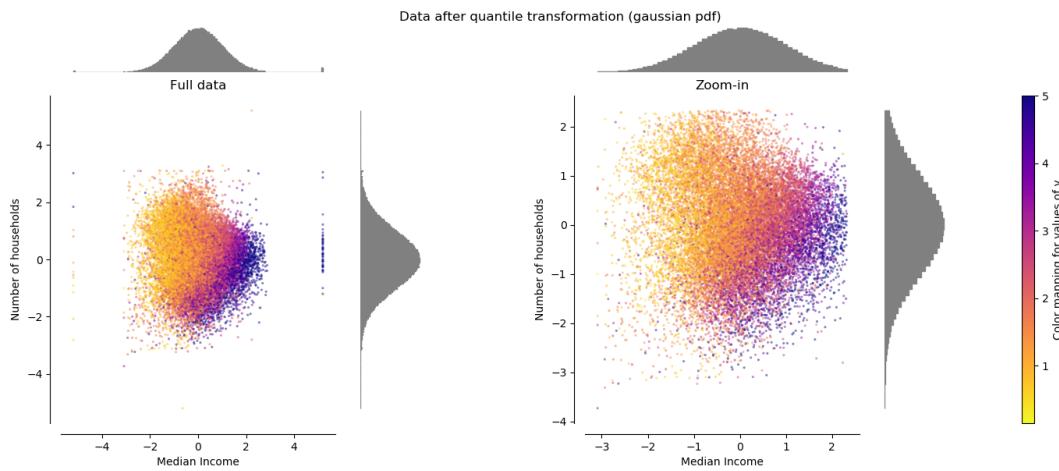
```
make_plot(5)
make_plot(6)
```



### QuantileTransformer (Gaussian output)

`QuantileTransformer` has an additional `output_distribution` parameter allowing to match a Gaussian distribution instead of a uniform distribution. Note that this non-parametric transformer introduces saturation artifacts for extreme values.

```
make_plot(7)
```

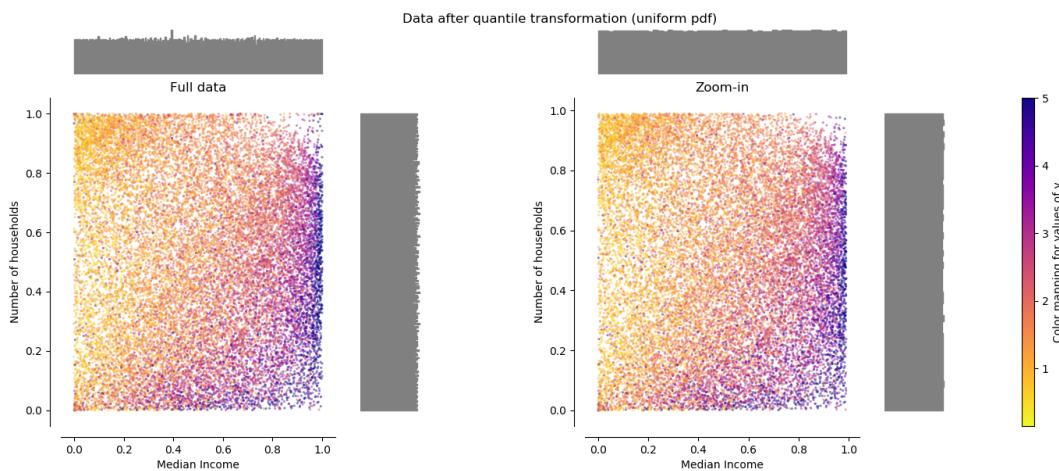


### QuantileTransformer (uniform output)

`QuantileTransformer` applies a non-linear transformation such that the probability density function of each feature will be mapped to a uniform distribution. In this case, all the data will be mapped in the range [0, 1], even the outliers which cannot be distinguished anymore from the inliers.

As `RobustScaler`, `QuantileTransformer` is robust to outliers in the sense that adding or removing outliers in the training set will yield approximately the same transformation on held out data. But contrary to `RobustScaler`, `QuantileTransformer` will also automatically collapse any outlier by setting them to the a priori defined range boundaries (0 and 1).

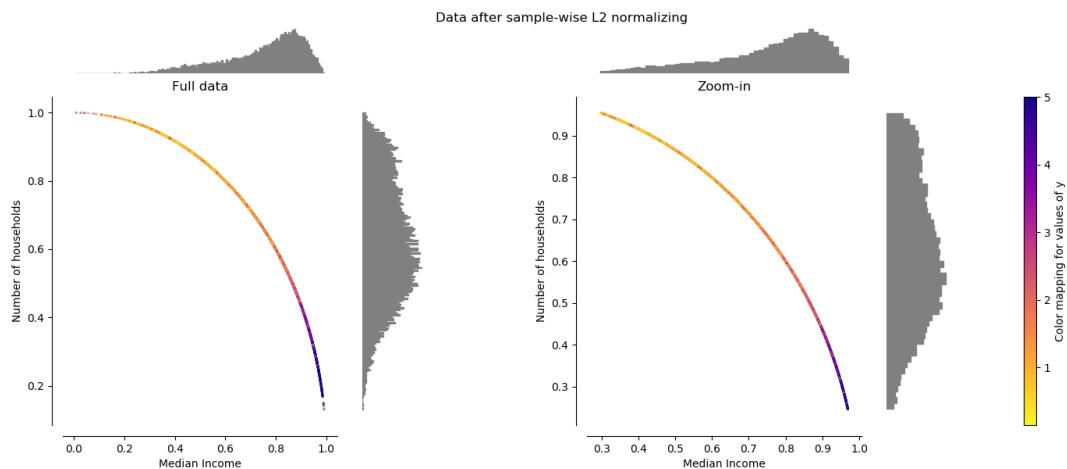
```
make_plot(8)
```



### Normalizer

The `Normalizer` rescales the vector for each sample to have unit norm, independently of the distribution of the samples. It can be seen on both figures below where all samples are mapped onto the unit circle. In our example the two selected features have only positive values; therefore the transformed data only lie in the positive quadrant. This would not be the case if some original features had a mix of positive and negative values.

```
make_plot(9)
plt.show()
```



**Total running time of the script:** ( 0 minutes 4.599 seconds)

## 5.26 Semi Supervised Classification

Examples concerning the `sklearn.semi_supervised` module.

---

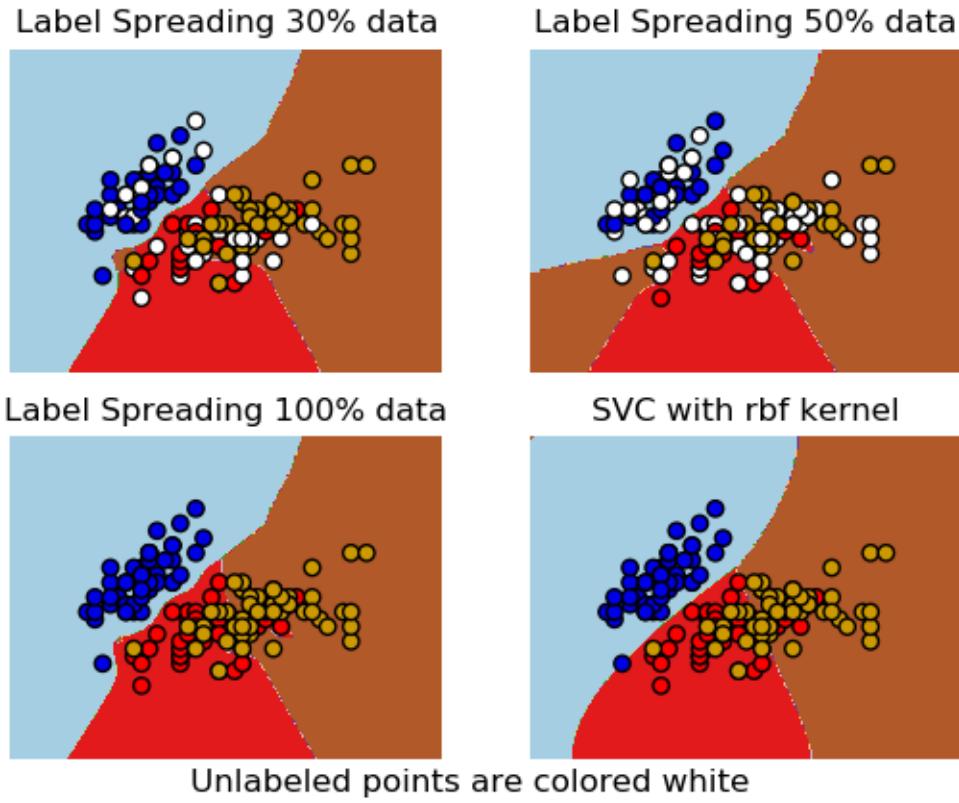
**Note:** Click [here](#) to download the full example code

---

### 5.26.1 Decision boundary of label propagation versus SVM on the Iris dataset

Comparison for decision boundary generated on iris dataset between Label Propagation and SVM.

This demonstrates Label Propagation learning a good boundary even with a small amount of labeled data.



```

print(__doc__)

Authors: Clay Woolam <clay@woolam.org>
License: BSD

import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn import svm
from sklearn.semi_supervised import label_propagation

rng = np.random.RandomState(0)

iris = datasets.load_iris()

X = iris.data[:, :2]
y = iris.target

step size in the mesh
h = .02

y_30 = np.copy(y)
y_30[rng.rand(len(y)) < 0.3] = -1
y_50 = np.copy(y)
y_50[rng.rand(len(y)) < 0.5] = -1
we create an instance of SVM and fit out data. We do not scale our

```

```
data since we want to plot the support vectors
ls30 = (label_propagation.LabelSpreading().fit(X, y_30),
 y_30)
ls50 = (label_propagation.LabelSpreading().fit(X, y_50),
 y_50)
ls100 = (label_propagation.LabelSpreading().fit(X, y), y)
rbf_svc = (svm.SVC(kernel='rbf', gamma=.5).fit(X, y), y)

create a mesh to plot in
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
 np.arange(y_min, y_max, h))

title for the plots
titles = ['Label Spreading 30% data',
 'Label Spreading 50% data',
 'Label Spreading 100% data',
 'SVC with rbf kernel']

color_map = {-1: (1, 1, 1), 0: (0, 0, .9), 1: (1, 0, 0), 2: (.8, .6, 0)}

for i, (clf, y_train) in enumerate((ls30, ls50, ls100, rbf_svc)):
 # Plot the decision boundary. For that, we will assign a color to each
 # point in the mesh [x_min, x_max]x[y_min, y_max].
 plt.subplot(2, 2, i + 1)
 Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

 # Put the result into a color plot
 Z = Z.reshape(xx.shape)
 plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
 plt.axis('off')

 # Plot also the training points
 colors = [color_map[y] for y in y_train]
 plt.scatter(X[:, 0], X[:, 1], c=colors, edgecolors='black')

 plt.title(titles[i])

plt.suptitle("Unlabeled points are colored white", y=0.1)
plt.show()
```

**Total running time of the script:** ( 0 minutes 0.915 seconds)

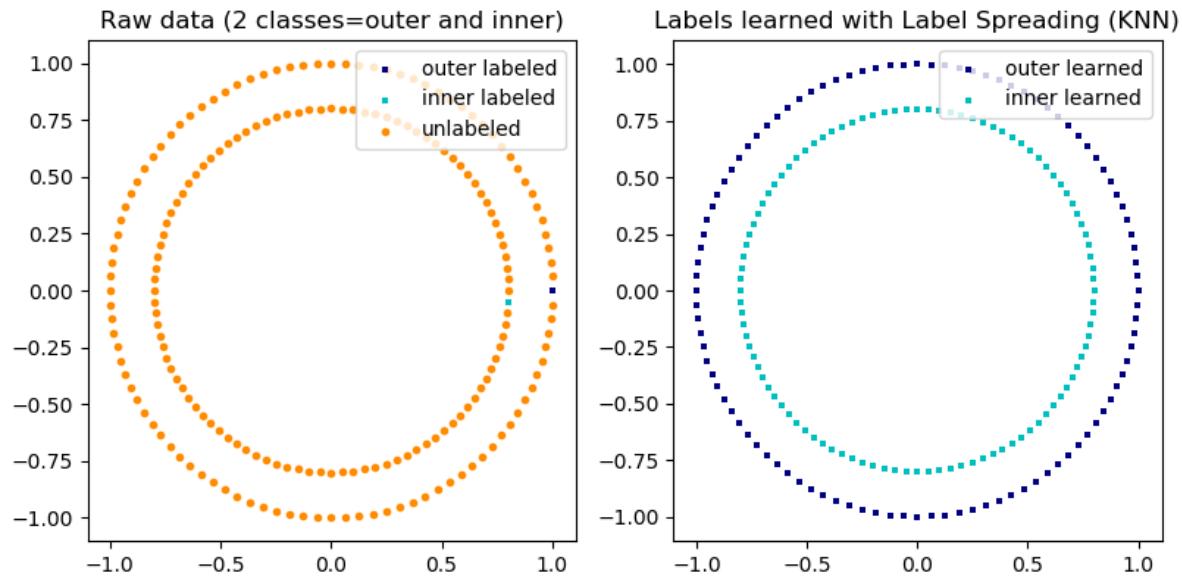
---

**Note:** Click [here](#) to download the full example code

---

## 5.26.2 Label Propagation learning a complex structure

Example of LabelPropagation learning a complex internal structure to demonstrate “manifold learning”. The outer circle should be labeled “red” and the inner circle “blue”. Because both label groups lie inside their own distinct shape, we can see that the labels propagate correctly around the circle.



```

print(__doc__)

Authors: Clay Woolam <clay@woolam.org>
Andreas Mueller <amueller@ais.uni-bonn.de>
License: BSD

import numpy as np
import matplotlib.pyplot as plt
from sklearn.semi_supervised import label_propagation
from sklearn.datasets import make_circles

generate ring with inner box
n_samples = 200
X, y = make_circles(n_samples=n_samples, shuffle=False)
outer, inner = 0, 1
labels = np.full(n_samples, -1.)
labels[0] = outer
labels[-1] = inner

######
Learn with LabelSpreading
label_spread = label_propagation.LabelSpreading(kernel='knn', alpha=0.8)
label_spread.fit(X, labels)

#####
Plot output labels
output_labels = label_spread.transduction_
plt.figure(figsize=(8.5, 4))
plt.subplot(1, 2, 1)
plt.scatter(X[labels == outer, 0], X[labels == outer, 1], color='navy',
 marker='s', lw=0, label="outer labeled", s=10)
plt.scatter(X[labels == inner, 0], X[labels == inner, 1], color='c',
 marker='s', lw=0, label='inner labeled', s=10)
plt.scatter(X[labels == -1, 0], X[labels == -1, 1], color='darkorange',
 marker='.', label='unlabeled')
plt.legend(scatterpoints=1, shadow=False, loc='upper right')
plt.title("Raw data (2 classes=outer and inner)")


```

```
plt.subplot(1, 2, 2)
output_label_array = np.asarray(output_labels)
outer_numbers = np.where(output_label_array == outer)[0]
inner_numbers = np.where(output_label_array == inner)[0]
plt.scatter(X[outer_numbers, 0], X[outer_numbers, 1], color='navy',
 marker='s', lw=0, s=10, label="outer learned")
plt.scatter(X[inner_numbers, 0], X[inner_numbers, 1], color='c',
 marker='s', lw=0, s=10, label="inner learned")
plt.legend(scatterpoints=1, shadow=False, loc='upper right')
plt.title("Labels learned with Label Spreading (KNN)")

plt.subplots_adjust(left=0.07, bottom=0.07, right=0.93, top=0.92)
plt.show()
```

**Total running time of the script:** ( 0 minutes 0.031 seconds)

---

**Note:** Click [here](#) to download the full example code

---

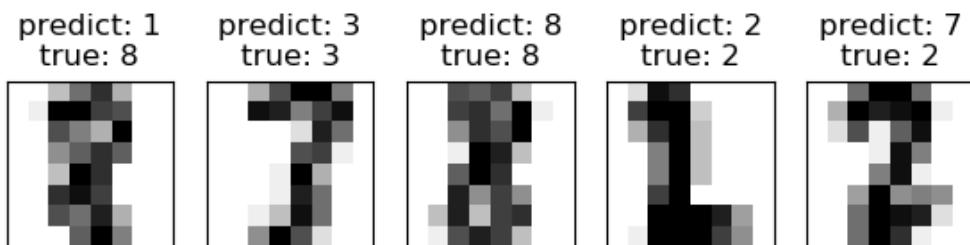
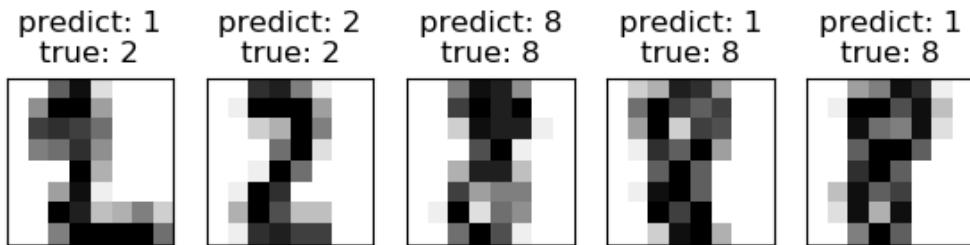
### 5.26.3 Label Propagation digits: Demonstrating performance

This example demonstrates the power of semisupervised learning by training a Label Spreading model to classify handwritten digits with sets of very few labels.

The handwritten digit dataset has 1797 total points. The model will be trained using all points, but only 30 will be labeled. Results in the form of a confusion matrix and a series of metrics over each class will be very good.

At the end, the top 10 most uncertain predictions will be shown.

### Learning with small amount of labeled data



Out:

```
Label Spreading model: 40 labeled & 300 unlabeled points (340 total)
 precision recall f1-score support
```

|   |      |      |      |    |
|---|------|------|------|----|
| 0 | 1.00 | 1.00 | 1.00 | 27 |
| 1 | 0.82 | 1.00 | 0.90 | 37 |
| 2 | 1.00 | 0.86 | 0.92 | 28 |
| 3 | 1.00 | 0.80 | 0.89 | 35 |
| 4 | 0.92 | 1.00 | 0.96 | 24 |
| 5 | 0.74 | 0.94 | 0.83 | 34 |
| 6 | 0.89 | 0.96 | 0.92 | 25 |
| 7 | 0.94 | 0.89 | 0.91 | 35 |
| 8 | 1.00 | 0.68 | 0.81 | 31 |
| 9 | 0.81 | 0.88 | 0.84 | 24 |

```
accuracy 0.90 300
```

```
macro avg 0.91 0.90 0.90 300
weighted avg 0.91 0.90 0.90 300
```

Confusion matrix

```
[[27 0 0 0 0 0 0 0 0]
 [0 37 0 0 0 0 0 0 0]
 [0 1 24 0 0 0 2 1 0]
 [0 0 0 28 0 5 0 1 0]
 [0 0 0 0 24 0 0 0 0]
 [0 0 0 0 0 32 0 0 2]]
```

```
[0 0 0 0 0 1 24 0 0 0]
[0 0 0 0 1 3 0 31 0 0]
[0 7 0 0 0 0 1 0 21 2]
[0 0 0 0 1 2 0 0 0 21]]
```

```
print(__doc__)

Authors: Clay Woolam <clay@woolam.org>
License: BSD

import numpy as np
import matplotlib.pyplot as plt

from scipy import stats

from sklearn import datasets
from sklearn.semi_supervised import label_propagation

from sklearn.metrics import confusion_matrix, classification_report

digits = datasets.load_digits()
rng = np.random.RandomState(2)
indices = np.arange(len(digits.data))
rng.shuffle(indices)

X = digits.data[indices[:340]]
y = digits.target[indices[:340]]
images = digits.images[indices[:340]]

n_total_samples = len(y)
n_labeled_points = 40

indices = np.arange(n_total_samples)

unlabeled_set = indices[n_labeled_points:]

######
Shuffle everything around
y_train = np.copy(y)
y_train[unlabeled_set] = -1

#####
Learn with LabelSpreading
lp_model = label_propagation.LabelSpreading(gamma=.25, max_iter=20)
lp_model.fit(X, y_train)
predicted_labels = lp_model.transduction_[unlabeled_set]
true_labels = y[unlabeled_set]

cm = confusion_matrix(true_labels, predicted_labels, labels=lp_model.classes_)

print("Label Spreading model: %d labeled & %d unlabeled points (%d total)" %
 (n_labeled_points, n_total_samples - n_labeled_points, n_total_samples))
```

```

print(classification_report(true_labels, predicted_labels))

print("Confusion matrix")
print(cm)

##########
Calculate uncertainty values for each transduced distribution
pred_entropies = stats.distributions.entropy(lp_model.label_distributions_.T)

#####
Pick the top 10 most uncertain labels
uncertainty_index = np.argsort(pred_entropies)[-10:]

#####
Plot
f = plt.figure(figsize=(7, 5))
for index, image_index in enumerate(uncertainty_index):
 image = images[image_index]

 sub = f.add_subplot(2, 5, index + 1)
 sub.imshow(image, cmap=plt.cm.gray_r)
 plt.xticks([])
 plt.yticks([])
 sub.set_title('predict: %i\ntrue: %i' % (
 lp_model.transduction_[image_index], y[image_index]))

f.suptitle('Learning with small amount of labeled data')
plt.show()

```

**Total running time of the script:** ( 0 minutes 0.225 seconds)

---

**Note:** Click [here](#) to download the full example code

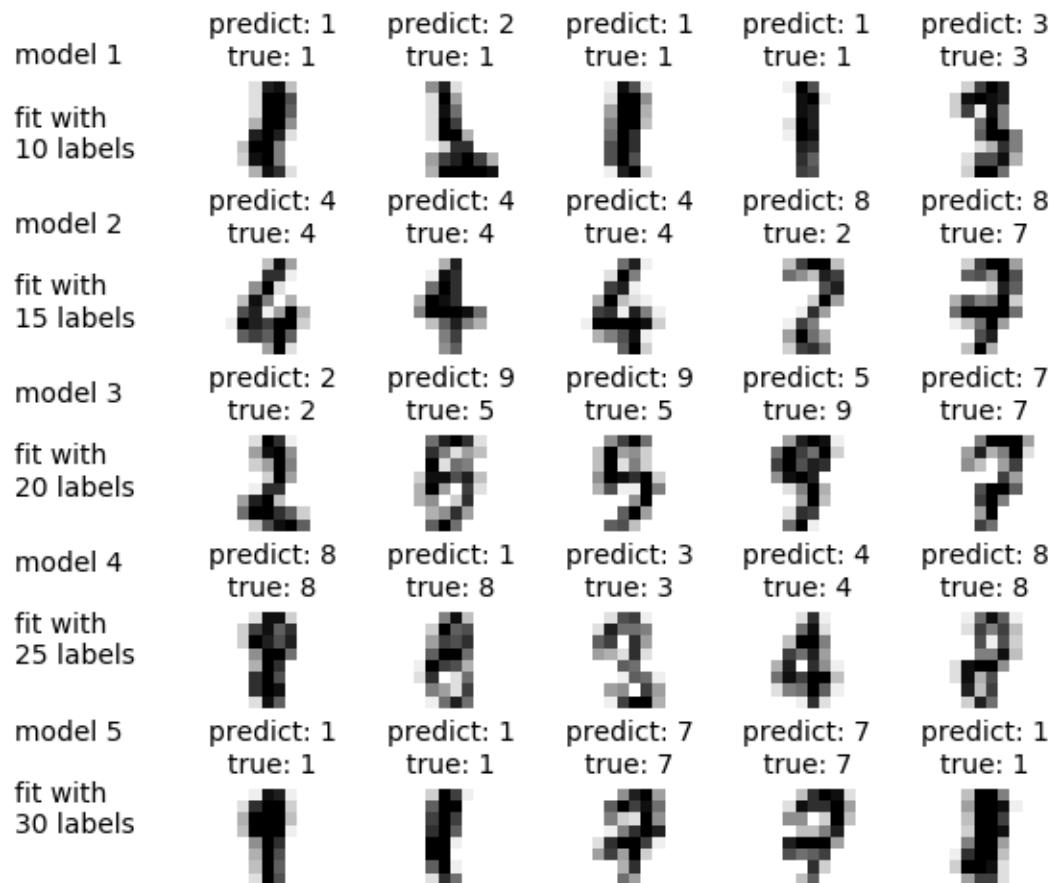
---

## 5.26.4 Label Propagation digits active learning

Demonstrates an active learning technique to learn handwritten digits using label propagation.

We start by training a label propagation model with only 10 labeled points, then we select the top five most uncertain points to label. Next, we train with 15 labeled points (original 10 + 5 new ones). We repeat this process four times to have a model trained with 30 labeled examples. Note you can increase this to label more than 30 by changing `max_iterations`. Labeling more than 30 can be useful to get a sense for the speed of convergence of this active learning technique.

A plot will appear showing the top 5 most uncertain digits for each iteration of training. These may or may not contain mistakes, but we will train the next model with their true labels.



Out:

| Iteration 0                                                                                                                                |                                                               |        |          |         |
|--------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------|--------|----------|---------|
|                                                                                                                                            | Label Spreading model: 40 labeled & 290 unlabeled (330 total) |        |          |         |
|                                                                                                                                            | precision                                                     | recall | f1-score | support |
| 0                                                                                                                                          | 1.00                                                          | 1.00   | 1.00     | 22      |
| 1                                                                                                                                          | 0.78                                                          | 0.69   | 0.73     | 26      |
| 2                                                                                                                                          | 0.93                                                          | 0.93   | 0.93     | 29      |
| 3                                                                                                                                          | 1.00                                                          | 0.89   | 0.94     | 27      |
| 4                                                                                                                                          | 0.92                                                          | 0.96   | 0.94     | 23      |
| 5                                                                                                                                          | 0.96                                                          | 0.70   | 0.81     | 33      |
| 6                                                                                                                                          | 0.97                                                          | 0.97   | 0.97     | 35      |
| 7                                                                                                                                          | 0.94                                                          | 0.91   | 0.92     | 33      |
| 8                                                                                                                                          | 0.62                                                          | 0.89   | 0.74     | 28      |
| 9                                                                                                                                          | 0.73                                                          | 0.79   | 0.76     | 34      |
| accuracy                                                                                                                                   |                                                               |        | 0.87     | 290     |
| macro avg                                                                                                                                  | 0.89                                                          | 0.87   | 0.87     | 290     |
| weighted avg                                                                                                                               | 0.88                                                          | 0.87   | 0.87     | 290     |
| Confusion matrix                                                                                                                           |                                                               |        |          |         |
| [[22  0  0  0  0  0  0  0  0  0]<br>[ 0 18  2  0  0  0  1  0  5  0]<br>[ 0  0 27  0  0  0  0  0  2  0]<br>[ 0  0  0 24  0  0  0  0  3  0]] |                                                               |        |          |         |

```
[0 1 0 0 22 0 0 0 0 0]
[0 0 0 0 23 0 0 0 10]
[0 1 0 0 0 34 0 0 0]
[0 0 0 0 0 0 30 3 0]
[0 3 0 0 0 0 0 25 0]
[0 0 0 0 2 1 0 2 27]]
```

**Iteration 1**

Label Spreading model: 45 labeled & 285 unlabeled (330 total)

|              | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| 0            | 1.00      | 1.00   | 1.00     | 22      |
| 1            | 0.79      | 1.00   | 0.88     | 22      |
| 2            | 1.00      | 0.93   | 0.96     | 29      |
| 3            | 1.00      | 1.00   | 1.00     | 26      |
| 4            | 0.92      | 0.96   | 0.94     | 23      |
| 5            | 0.96      | 0.70   | 0.81     | 33      |
| 6            | 1.00      | 0.97   | 0.99     | 35      |
| 7            | 0.94      | 0.91   | 0.92     | 33      |
| 8            | 0.77      | 0.86   | 0.81     | 28      |
| 9            | 0.73      | 0.79   | 0.76     | 34      |
| accuracy     |           |        | 0.90     | 285     |
| macro avg    | 0.91      | 0.91   | 0.91     | 285     |
| weighted avg | 0.91      | 0.90   | 0.90     | 285     |

**Confusion matrix**

```
[[22 0 0 0 0 0 0 0 0]
 [0 22 0 0 0 0 0 0 0]
 [0 0 27 0 0 0 0 0 2]
 [0 0 0 26 0 0 0 0 0]
 [0 1 0 0 22 0 0 0 0]
 [0 0 0 0 23 0 0 0 10]
 [0 1 0 0 0 34 0 0 0]
 [0 0 0 0 0 0 30 3 0]
 [0 4 0 0 0 0 0 24 0]
 [0 0 0 0 2 1 0 2 27]]
```

**Iteration 2**

Label Spreading model: 50 labeled & 280 unlabeled (330 total)

|              | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| 0            | 1.00      | 1.00   | 1.00     | 22      |
| 1            | 0.85      | 1.00   | 0.92     | 22      |
| 2            | 1.00      | 1.00   | 1.00     | 28      |
| 3            | 1.00      | 1.00   | 1.00     | 26      |
| 4            | 0.87      | 1.00   | 0.93     | 20      |
| 5            | 0.96      | 0.70   | 0.81     | 33      |
| 6            | 1.00      | 0.97   | 0.99     | 35      |
| 7            | 0.94      | 1.00   | 0.97     | 32      |
| 8            | 0.92      | 0.86   | 0.89     | 28      |
| 9            | 0.73      | 0.79   | 0.76     | 34      |
| accuracy     |           |        | 0.92     | 280     |
| macro avg    | 0.93      | 0.93   | 0.93     | 280     |
| weighted avg | 0.93      | 0.92   | 0.92     | 280     |

**Confusion matrix**

```
[[22 0 0 0 0 0 0 0 0]
 [0 22 0 0 0 0 0 0 0]]
```

```
[0 0 28 0 0 0 0 0 0 0]
[0 0 0 26 0 0 0 0 0 0]
[0 0 0 0 20 0 0 0 0 0]
[0 0 0 0 0 23 0 0 0 10]
[0 1 0 0 0 34 0 0 0 0]
[0 0 0 0 0 0 32 0 0 0]
[0 3 0 0 1 0 0 0 24 0]
[0 0 0 0 2 1 0 2 2 27]]
```

Iteration 3

Label Spreading model: 55 labeled &amp; 275 unlabeled (330 total)

|              | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| 0            | 1.00      | 1.00   | 1.00     | 22      |
| 1            | 0.85      | 1.00   | 0.92     | 22      |
| 2            | 1.00      | 1.00   | 1.00     | 27      |
| 3            | 1.00      | 1.00   | 1.00     | 26      |
| 4            | 0.87      | 1.00   | 0.93     | 20      |
| 5            | 0.96      | 0.87   | 0.92     | 31      |
| 6            | 1.00      | 0.97   | 0.99     | 35      |
| 7            | 1.00      | 1.00   | 1.00     | 31      |
| 8            | 0.92      | 0.86   | 0.89     | 28      |
| 9            | 0.88      | 0.85   | 0.86     | 33      |
| accuracy     |           |        | 0.95     | 275     |
| macro avg    | 0.95      | 0.95   | 0.95     | 275     |
| weighted avg | 0.95      | 0.95   | 0.95     | 275     |

Confusion matrix

```
[[22 0 0 0 0 0 0 0 0 0]
 [0 22 0 0 0 0 0 0 0 0]
 [0 0 27 0 0 0 0 0 0 0]
 [0 0 0 26 0 0 0 0 0 0]
 [0 0 0 0 20 0 0 0 0 0]
 [0 0 0 0 0 27 0 0 0 4]
 [0 1 0 0 0 34 0 0 0 0]
 [0 0 0 0 0 0 31 0 0 0]
 [0 3 0 0 1 0 0 0 24 0]
 [0 0 0 0 2 1 0 0 2 28]]
```

Iteration 4

Label Spreading model: 60 labeled &amp; 270 unlabeled (330 total)

|              | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| 0            | 1.00      | 1.00   | 1.00     | 22      |
| 1            | 0.96      | 1.00   | 0.98     | 22      |
| 2            | 1.00      | 0.96   | 0.98     | 27      |
| 3            | 0.96      | 1.00   | 0.98     | 25      |
| 4            | 0.86      | 1.00   | 0.93     | 19      |
| 5            | 0.96      | 0.87   | 0.92     | 31      |
| 6            | 1.00      | 0.97   | 0.99     | 35      |
| 7            | 1.00      | 1.00   | 1.00     | 31      |
| 8            | 0.92      | 0.96   | 0.94     | 25      |
| 9            | 0.88      | 0.85   | 0.86     | 33      |
| accuracy     |           |        | 0.96     | 270     |
| macro avg    | 0.95      | 0.96   | 0.96     | 270     |
| weighted avg | 0.96      | 0.96   | 0.96     | 270     |

Confusion matrix

```
[[22 0 0 0 0 0 0 0 0 0]
[0 22 0 0 0 0 0 0 0 0]
[0 0 26 1 0 0 0 0 0 0]
[0 0 0 25 0 0 0 0 0 0]
[0 0 0 0 19 0 0 0 0 0]
[0 0 0 0 0 27 0 0 0 4]
[0 1 0 0 0 0 34 0 0 0]
[0 0 0 0 0 0 0 31 0 0]
[0 0 0 0 1 0 0 0 24 0]
[0 0 0 0 2 1 0 0 2 28]]
```

```
print(__doc__)

Authors: Clay Woolam <clay@woolam.org>
License: BSD

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

from sklearn import datasets
from sklearn.semi_supervised import label_propagation
from sklearn.metrics import classification_report, confusion_matrix

digits = datasets.load_digits()
rng = np.random.RandomState(0)
indices = np.arange(len(digits.data))
rng.shuffle(indices)

X = digits.data[indices[:330]]
y = digits.target[indices[:330]]
images = digits.images[indices[:330]]

n_total_samples = len(y)
n_labeled_points = 40
max_iterations = 5

unlabeled_indices = np.arange(n_total_samples)[n_labeled_points:]
f = plt.figure()

for i in range(max_iterations):
 if len(unlabeled_indices) == 0:
 print("No unlabeled items left to label.")
 break
 y_train = np.copy(y)
 y_train[unlabeled_indices] = -1

 lp_model = label_propagation.LabelSpreading(gamma=0.25, max_iter=20)
 lp_model.fit(X, y_train)

 predicted_labels = lp_model.transduction_[unlabeled_indices]
 true_labels = y[unlabeled_indices]
```

```

cm = confusion_matrix(true_labels, predicted_labels,
 labels=lp_model.classes_)

print("Iteration %i %s" % (i, 70 * "_"))
print("Label Spreading model: %d labeled & %d unlabeled (%d total)"
 % (n_labeled_points, n_total_samples - n_labeled_points,
 n_total_samples))

print(classification_report(true_labels, predicted_labels))

print("Confusion matrix")
print(cm)

compute the entropies of transduced label distributions
pred_entropies = stats.distributions.entropy(
 lp_model.label_distributions_.T)

select up to 5 digit examples that the classifier is most uncertain about
uncertainty_index = np.argsort(pred_entropies)[::-1]
uncertainty_index = uncertainty_index[
 np.in1d(uncertainty_index, unlabeled_indices)][:5]

keep track of indices that we get labels for
delete_indices = np.array([], dtype=int)

for more than 5 iterations, visualize the gain only on the first 5
if i < 5:
 f.text(.05, (1 - (i + 1) * .183),
 "model %d\nfit with\n%d labels" %
 ((i + 1), i * 5 + 10), size=10)
 for index, image_index in enumerate(uncertainty_index):
 image = images[image_index]

 # for more than 5 iterations, visualize the gain only on the first 5
 if i < 5:
 sub = f.add_subplot(5, 5, index + 1 + (5 * i))
 sub.imshow(image, cmap=plt.cm.gray_r, interpolation='none')
 sub.set_title("predict: %i\ntrue: %i" %
 lp_model.transduction_[image_index], size=10)
 sub.axis('off')

 # labeling 5 points, remote from labeled set
 delete_index, = np.where(unlabeled_indices == image_index)
 delete_indices = np.concatenate((delete_indices, delete_index))

unlabeled_indices = np.delete(unlabeled_indices, delete_indices)
n_labeled_points += len(uncertainty_index)

f.suptitle("Active learning with Label Propagation.\nRows show 5 most "
 "uncertain labels to learn with the next model.", y=1.15)
plt.subplots_adjust(left=0.2, bottom=0.03, right=0.9, top=0.9, wspace=0.2,
 hspace=0.85)
plt.show()

```

**Total running time of the script:** ( 0 minutes 0.575 seconds)

## 5.27 Support Vector Machines

Examples concerning the `sklearn.svm` module.

---

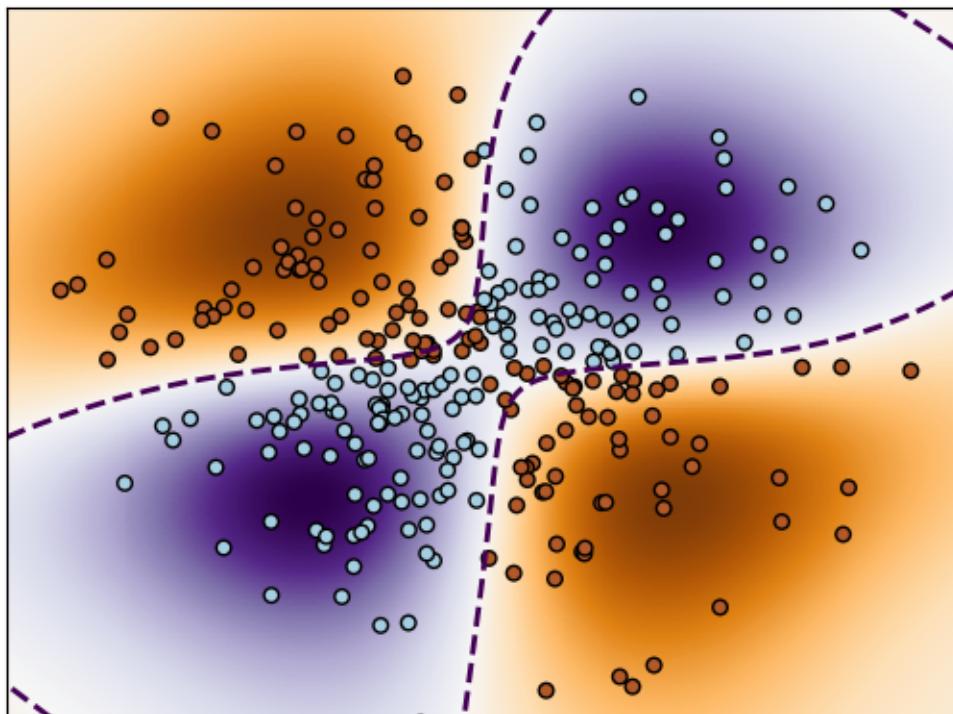
**Note:** Click [here](#) to download the full example code

---

### 5.27.1 Non-linear SVM

Perform binary classification using non-linear SVC with RBF kernel. The target to predict is a XOR of the inputs.

The color map illustrates the decision function learned by the SVC.



```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm

xx, yy = np.meshgrid(np.linspace(-3, 3, 500),
 np.linspace(-3, 3, 500))
np.random.seed(0)
X = np.random.randn(300, 2)
Y = np.logical_xor(X[:, 0] > 0, X[:, 1] > 0)
```

```
fit the model
clf = svm.NuSVC(gamma='auto')
clf.fit(X, Y)

plot the decision function for each datapoint on the grid
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.imshow(Z, interpolation='nearest',
 extent=(xx.min(), xx.max(), yy.min(), yy.max()), aspect='auto',
 origin='lower', cmap=plt.cm.PuOr_r)
contours = plt.contour(xx, yy, Z, levels=[0], linewidths=2,
 linestyles='dashed')
plt.scatter(X[:, 0], X[:, 1], s=30, c=Y, cmap=plt.cm.Paired,
 edgecolors='k')
plt.xticks(())
plt.yticks(())
plt.axis([-3, 3, -3, 3])
plt.show()
```

**Total running time of the script:** ( 0 minutes 1.073 seconds)

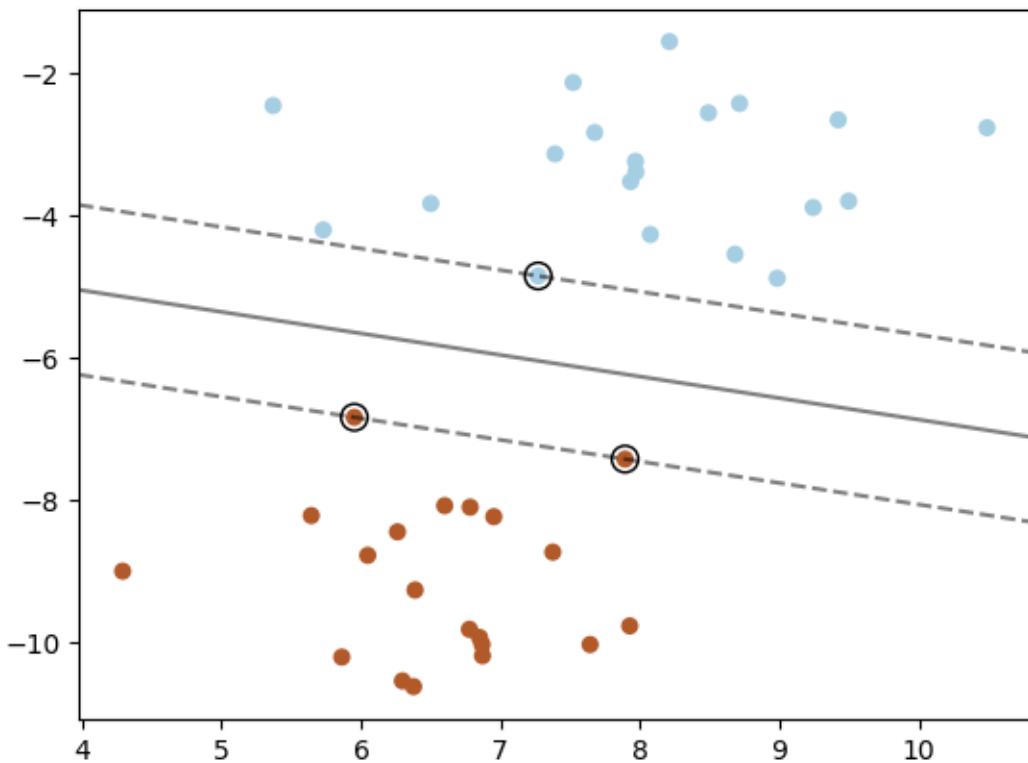
---

**Note:** Click [here](#) to download the full example code

---

## 5.27.2 SVM: Maximum margin separating hyperplane

Plot the maximum margin separating hyperplane within a two-class separable dataset using a Support Vector Machine classifier with linear kernel.



```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn.datasets import make_blobs

we create 40 separable points
X, y = make_blobs(n_samples=40, centers=2, random_state=6)

fit the model, don't regularize for illustration purposes
clf = svm.SVC(kernel='linear', C=1000)
clf.fit(X, y)

plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Paired)

plot the decision function
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()

create grid to evaluate model
xx = np.linspace(xlim[0], xlim[1], 30)
yy = np.linspace(ylim[0], ylim[1], 30)
YY, XX = np.meshgrid(yy, xx)

```

```

xy = np.vstack([XX.ravel(), YY.ravel()]).T
Z = clf.decision_function(xy).reshape(XX.shape)

plot decision boundary and margins
ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
 linestyles=['--', '--', '--'])
plot support vectors
ax.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=100,
 linewidth=1, facecolors='none', edgecolors='k')
plt.show()

```

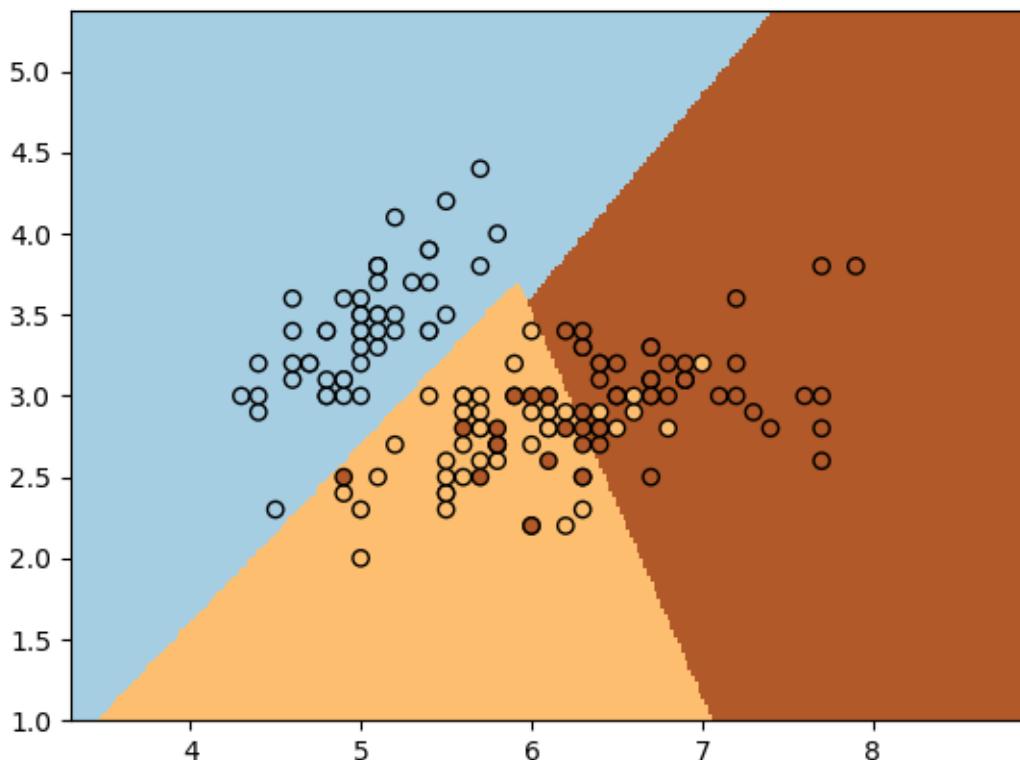
**Total running time of the script:** ( 0 minutes 0.021 seconds)

**Note:** Click [here](#) to download the full example code

### 5.27.3 SVM with custom kernel

Simple usage of Support Vector Machines to classify a sample. It will plot the decision surface and the support vectors.

3-Class classification using Support Vector Machine with custom kernel



```

print(__doc__)

import numpy as np

```

```

import matplotlib.pyplot as plt
from sklearn import svm, datasets

import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2] # we only take the first two features. We could
 # avoid this ugly slicing by using a two-dim dataset
Y = iris.target

def my_kernel(X, Y):
 """
 We create a custom kernel:

 (2 0)
 k(X, Y) = X () Y.T
 (0 1)
 """
 M = np.array([[2, 0], [0, 1.0]])
 return np.dot(np.dot(X, M), Y.T)

h = .02 # step size in the mesh

we create an instance of SVM and fit out data.
clf = svm.SVC(kernel=my_kernel)
clf.fit(X, Y)

Plot the decision boundary. For that, we will assign a color to each
point in the mesh [x_min, x_max]x[y_min, y_max].
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = clf.predict(xx.ravel(), yy.ravel())

Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)

Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=Y, cmap=plt.cm.Paired, edgecolors='k')
plt.title('3-Class classification using Support Vector Machine with custom'
 ' kernel')
plt.axis('tight')
plt.show()

```

**Total running time of the script:** ( 0 minutes 0.110 seconds)

---

**Note:** Click [here](#) to download the full example code

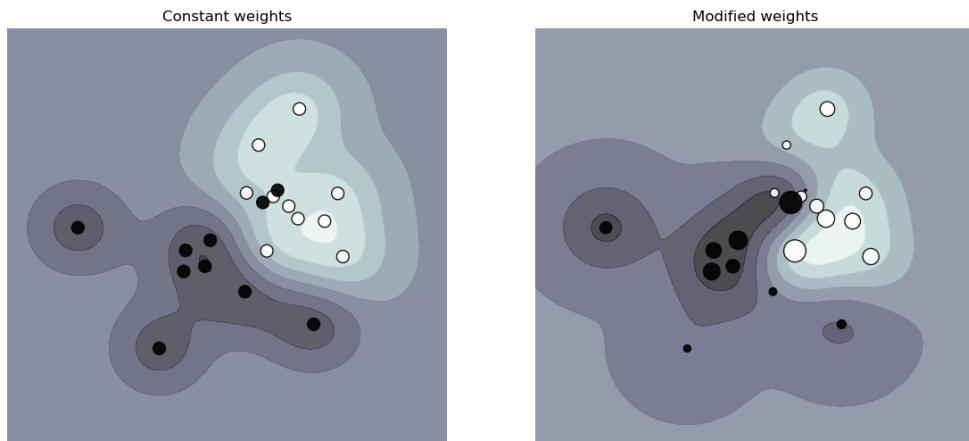
---

## 5.27.4 SVM: Weighted samples

Plot decision function of a weighted dataset, where the size of points is proportional to its weight.

The sample weighting rescales the C parameter, which means that the classifier puts more emphasis on getting these

points right. The effect might often be subtle. To emphasize the effect here, we particularly weight outliers, making the deformation of the decision boundary very visible.



```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm

def plot_decision_function(classifier, sample_weight, axis, title):
 # plot the decision function
 xx, yy = np.meshgrid(np.linspace(-4, 5, 500), np.linspace(-4, 5, 500))

 Z = classifier.decision_function(np.c_[xx.ravel(), yy.ravel()])
 Z = Z.reshape(xx.shape)

 # plot the line, the points, and the nearest vectors to the plane
 axis.contourf(xx, yy, Z, alpha=0.75, cmap=plt.cm.bone)
 axis.scatter(X[:, 0], X[:, 1], c=y, s=100 * sample_weight, alpha=0.9,
 cmap=plt.cm.bone, edgecolors='black')

 axis.axis('off')
 axis.set_title(title)

we create 20 points
np.random.seed(0)
X = np.r_[np.random.randn(10, 2) + [1, 1], np.random.randn(10, 2)]
y = [1] * 10 + [-1] * 10
sample_weight_last_ten = abs(np.random.randn(len(X)))
sample_weight_constant = np.ones(len(X))
and bigger weights to some outliers
sample_weight_last_ten[15:] *= 5
sample_weight_last_ten[9] *= 15

for reference, first fit without sample weights

fit the model
```

```

clf_weights = svm.SVC(gamma=1)
clf_weights.fit(X, y, sample_weight=sample_weight_last_ten)

clf_no_weights = svm.SVC(gamma=1)
clf_no_weights.fit(X, y)

fig, axes = plt.subplots(1, 2, figsize=(14, 6))
plot_decision_function(clf_no_weights, sample_weight_constant, axes[0],
 "Constant weights")
plot_decision_function(clf_weights, sample_weight_last_ten, axes[1],
 "Modified weights")

plt.show()

```

**Total running time of the script:** ( 0 minutes 0.349 seconds)

---

**Note:** Click [here](#) to download the full example code

---

## 5.27.5 SVM: Separating hyperplane for unbalanced classes

Find the optimal separating hyperplane using an SVC for classes that are unbalanced.

We first find the separating plane with a plain SVC and then plot (dashed) the separating hyperplane with automatically correction for unbalanced classes.

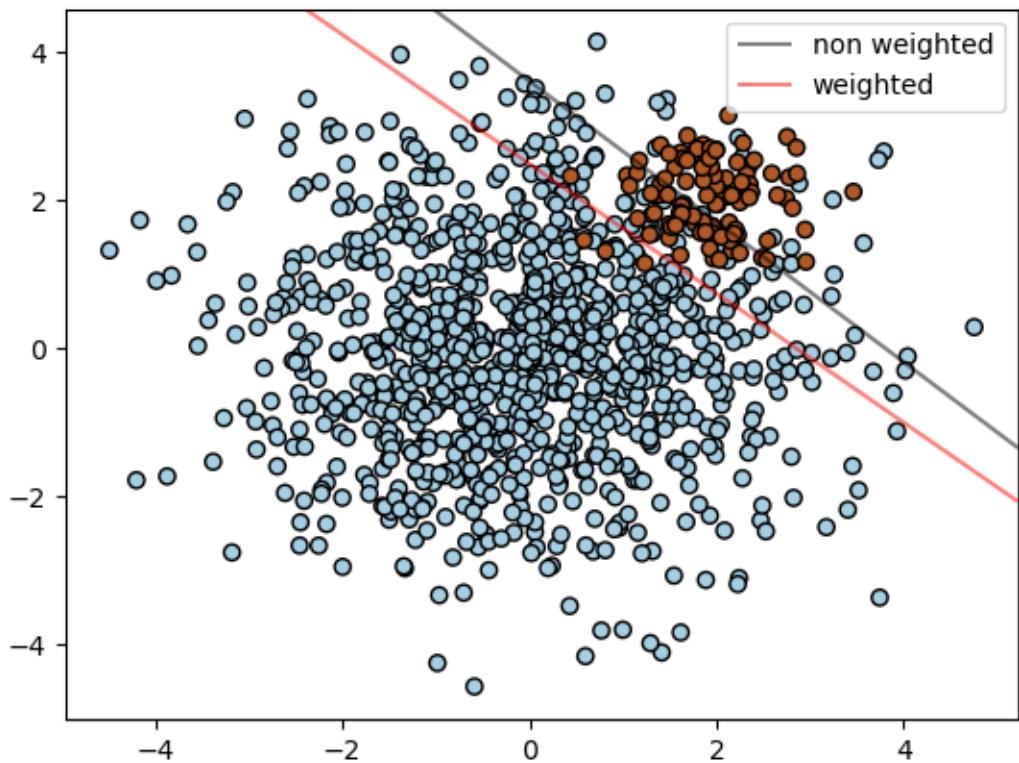
---

**Note:** This example will also work by replacing `SVC(kernel="linear")` with `SGDClassifier(loss="hinge")`. Setting the `loss` parameter of the `SGDClassifier` equal to hinge will yield behaviour such as that of a SVC with a linear kernel.

For example try instead of the SVC:

```
clf = SGDClassifier(n_iter=100, alpha=0.01)
```

---



```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn.datasets import make_blobs

we create two clusters of random points
n_samples_1 = 1000
n_samples_2 = 100
centers = [[0.0, 0.0], [2.0, 2.0]]
clusters_std = [1.5, 0.5]
X, y = make_blobs(n_samples=[n_samples_1, n_samples_2],
 centers=centers,
 cluster_std=clusters_std,
 random_state=0, shuffle=False)

fit the model and get the separating hyperplane
clf = svm.SVC(kernel='linear', C=1.0)
clf.fit(X, y)

fit the model and get the separating hyperplane using weighted classes
wclf = svm.SVC(kernel='linear', class_weight={1: 10})
wclf.fit(X, y)

plot the samples
```

```

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired, edgecolors='k')

plot the decision functions for both classifiers
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()

create grid to evaluate model
xx = np.linspace(xlim[0], xlim[1], 30)
yy = np.linspace(ylim[0], ylim[1], 30)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T

get the separating hyperplane
Z = clf.decision_function(xy).reshape(XX.shape)

plot decision boundary and margins
a = ax.contour(XX, YY, Z, colors='k', levels=[0], alpha=0.5, linestyles=['-'])

get the separating hyperplane for weighted classes
Z = wclf.decision_function(xy).reshape(XX.shape)

plot decision boundary and margins for weighted classes
b = ax.contour(XX, YY, Z, colors='r', levels=[0], alpha=0.5, linestyles=['-'])

plt.legend([a.collections[0], b.collections[0]], ["non weighted", "weighted"],
 loc="upper right")
plt.show()

```

**Total running time of the script:** ( 0 minutes 0.034 seconds)

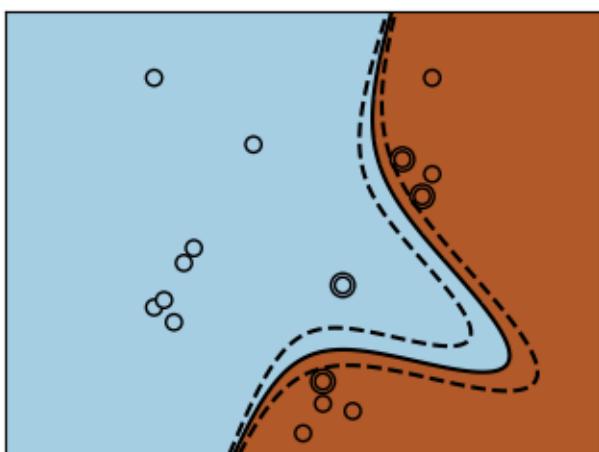
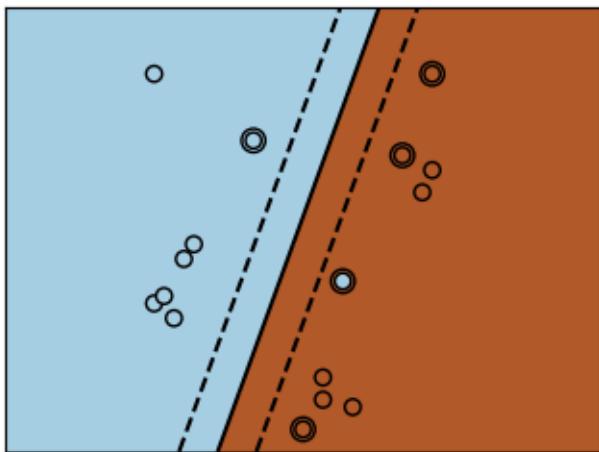
---

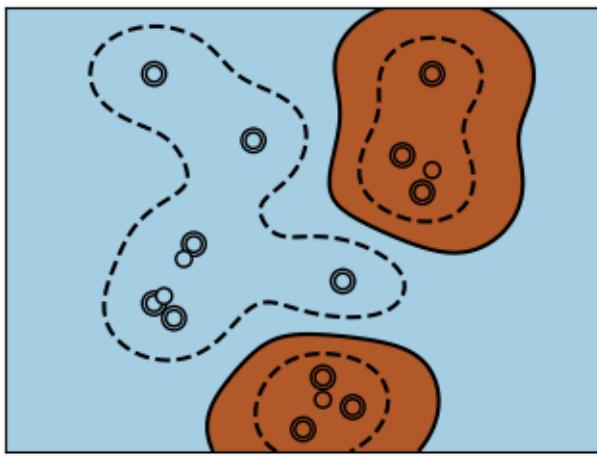
**Note:** Click [here](#) to download the full example code

---

## 5.27.6 SVM-Kernels

Three different types of SVM-Kernels are displayed below. The polynomial and RBF are especially useful when the data-points are not linearly separable.





```

print(__doc__)

Code source: Gaël Varoquaux
License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm

Our dataset and targets
X = np.c_[(.4, -.7),
 (-1.5, -1),
 (-1.4, -.9),
 (-1.3, -1.2),
 (-1.1, -.2),
 (-1.2, -.4),
 (-.5, 1.2),
 (-1.5, 2.1),
 (1, 1),
 # --
 (1.3, .8),
 (1.2, .5),
 (.2, -2),
 (.5, -2.4),
 (.2, -2.3),
 (0, -2.7),
 (1.3, 2.1)].T
Y = [0] * 8 + [1] * 8

figure number
fignum = 1

fit the model
for kernel in ('linear', 'poly', 'rbf'):
 clf = svm.SVC(kernel=kernel, gamma=2)
 clf.fit(X, Y)

```

```
plot the line, the points, and the nearest vectors to the plane
plt.figure(fignum, figsize=(4, 3))
plt.clf()

plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=80,
 facecolors='none', zorder=10, edgecolors='k')
plt.scatter(X[:, 0], X[:, 1], c=Y, zorder=10, cmap=plt.cm.Paired,
 edgecolors='k')

plt.axis('tight')
x_min = -3
x_max = 3
y_min = -3
y_max = 3

XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]
Z = clf.decision_function(np.c_[XX.ravel(), YY.ravel()])

Put the result into a color plot
Z = Z.reshape(XX.shape)
plt.figure(fignum, figsize=(4, 3))
plt.pcolormesh(XX, YY, Z > 0, cmap=plt.cm.Paired)
plt.contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '- ', '--'],
 levels=[-.5, 0, .5])

plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)

plt.xticks(())
plt.yticks(())
fignum = fignum + 1
plt.show()
```

**Total running time of the script:** ( 0 minutes 0.096 seconds)

---

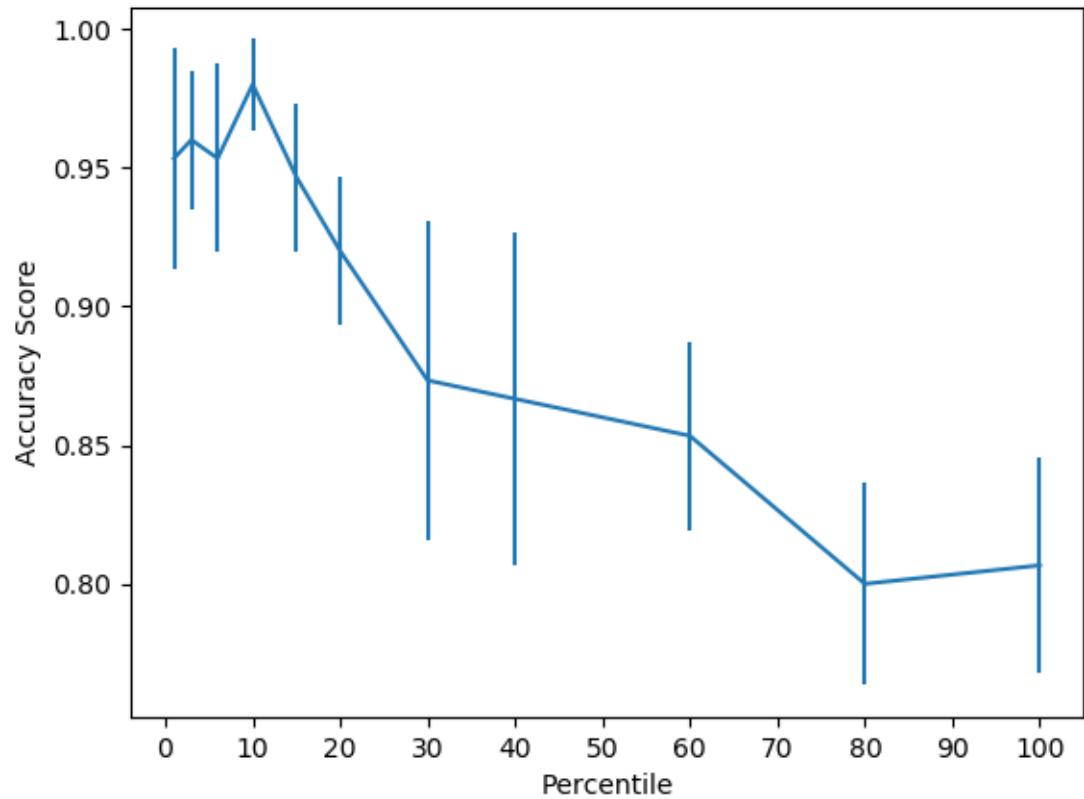
**Note:** Click [here](#) to download the full example code

---

## 5.27.7 SVM-Anova: SVM with univariate feature selection

This example shows how to perform univariate feature selection before running a SVC (support vector classifier) to improve the classification scores. We use the iris dataset (4 features) and add 36 non-informative features. We can find that our model achieves best performance when we select around 10% of features.

## Performance of the SVM-Anova varying the percentile of features selected



```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.feature_selection import SelectPercentile, chi2
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC

Import some data to play with
Add non-informative features
X, y = load_iris(return_X_y=True)
np.random.seed(0)
X = np.hstack((X, 2 * np.random.random((X.shape[0], 36))))

Create a feature-selection transform, a scaler and an instance of SVM that we
combine together to have an full-blown estimator
clf = Pipeline([('anova', SelectPercentile(chi2)),
 ('scaler', StandardScaler()),
 ('svc', SVC(gamma="auto"))])
```

```

Plot the cross-validation score as a function of percentile of features
score_means = list()
score_stds = list()
percentiles = (1, 3, 6, 10, 15, 20, 30, 40, 60, 80, 100)

for percentile in percentiles:
 clf.set_params(anova_percentile=percentile)
 this_scores = cross_val_score(clf, X, y, cv=5)
 score_means.append(this_scores.mean())
 score_stds.append(this_scores.std())

plt.errorbar(percentiles, score_means, np.array(score_stds))
plt.title(
 'Performance of the SVM-Anova varying the percentile of features selected')
plt.xticks(np.linspace(0, 100, 11, endpoint=True))
plt.xlabel('Percentile')
plt.ylabel('Accuracy Score')
plt.axis('tight')
plt.show()
```

**Total running time of the script:** ( 0 minutes 0.199 seconds)

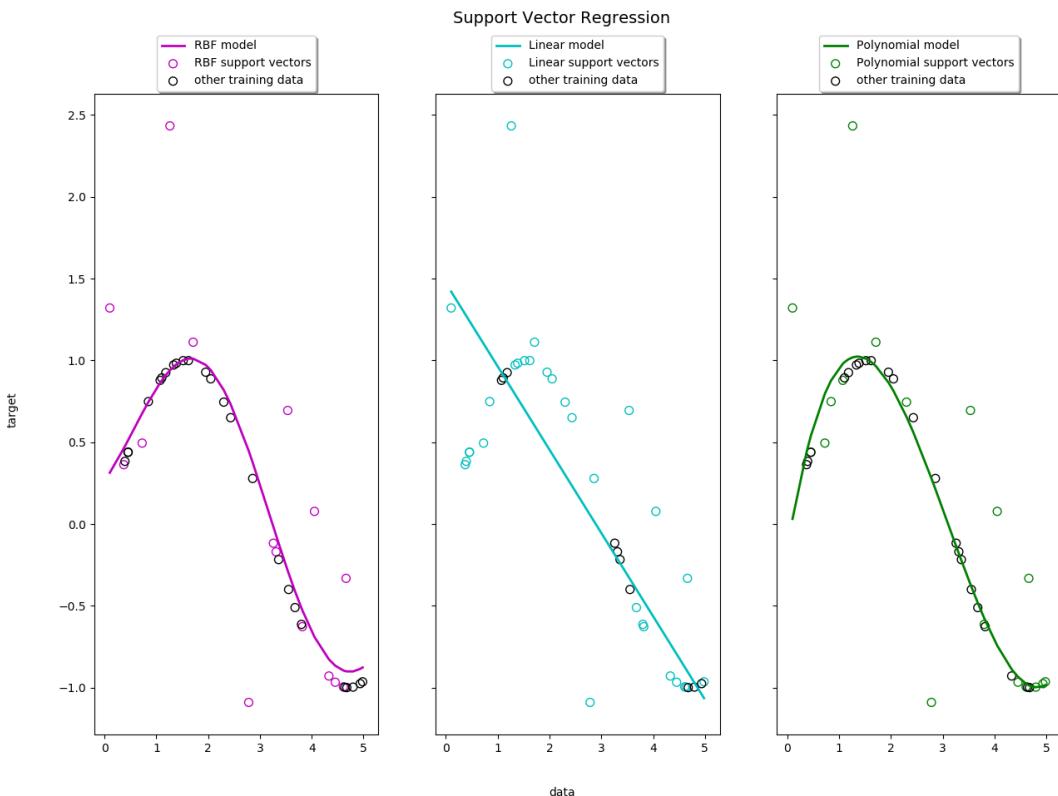
---

**Note:** Click [here](#) to download the full example code

---

## 5.27.8 Support Vector Regression (SVR) using linear and non-linear kernels

Toy example of 1D regression using linear, polynomial and RBF kernels.



```

print(__doc__)

import numpy as np
from sklearn.svm import SVR
import matplotlib.pyplot as plt

#####
Generate sample data
X = np.sort(5 * np.random.rand(40, 1), axis=0)
y = np.sin(X).ravel()

#####
Add noise to targets
y[::5] += 3 * (0.5 - np.random.rand(8))

#####
Fit regression model
svr_rbf = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=.1)
svr_lin = SVR(kernel='linear', C=100, gamma='auto')
svr_poly = SVR(kernel='poly', C=100, gamma='auto', degree=3, epsilon=.1,
 coef0=1)

#####
Look at the results
lw = 2

svrs = [svr_rbf, svr_lin, svr_poly]
kernel_label = ['RBF', 'Linear', 'Polynomial']
model_color = ['m', 'c', 'g']

```

```

fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(15, 10), sharey=True)
for ix, svr in enumerate(svrs):
 axes[ix].plot(X, svr.fit(X, y).predict(X), color=model_color[ix], lw=lw,
 label='{} model'.format(kernel_label[ix]))
 axes[ix].scatter(X[svr.support_], y[svr.support_], facecolor="none",
 edgecolor=model_color[ix], s=50,
 label='{} support vectors'.format(kernel_label[ix]))
 axes[ix].scatter(X[np.setdiff1d(np.arange(len(X)), svr.support_)],
 y[np.setdiff1d(np.arange(len(X)), svr.support_)],
 facecolor="none", edgecolor="k", s=50,
 label='other training data')
 axes[ix].legend(loc='upper center', bbox_to_anchor=(0.5, 1.1),
 ncol=1, fancybox=True, shadow=True)

fig.text(0.5, 0.04, 'data', ha='center', va='center')
fig.text(0.06, 0.5, 'target', ha='center', va='center', rotation='vertical')
fig.suptitle("Support Vector Regression", fontsize=14)
plt.show()

```

**Total running time of the script:** ( 0 minutes 3.104 seconds)

---

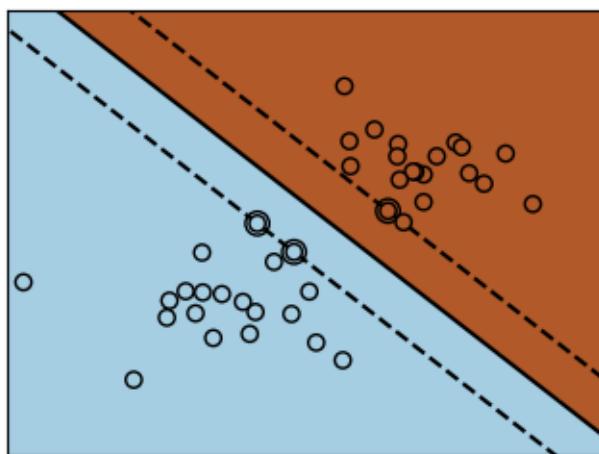
**Note:** Click [here](#) to download the full example code

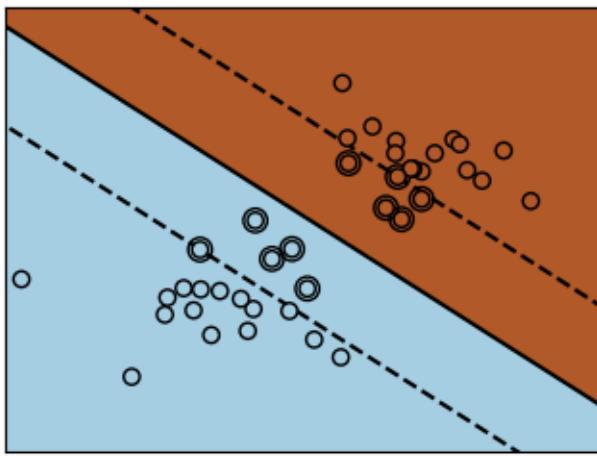
---

### 5.27.9 SVM Margins Example

The plots below illustrate the effect the parameter  $C$  has on the separation line. A large value of  $C$  basically tells our model that we do not have that much faith in our data's distribution, and will only consider points close to line of separation.

A small value of  $C$  includes more/all the observations, allowing the margins to be calculated using all the data in the area.





```

print(__doc__)

Code source: Gaël Varoquaux
Modified for documentation by Jaques Grobler
License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm

we create 40 separable points
np.random.seed(0)
X = np.r_[np.random.randn(20, 2) - [2, 2], np.random.randn(20, 2) + [2, 2]]
Y = [0] * 20 + [1] * 20

figure number
fignum = 1

fit the model
for name, penalty in (('unreg', 1), ('reg', 0.05)):

 clf = svm.SVC(kernel='linear', C=penalty)
 clf.fit(X, Y)

 # get the separating hyperplane
 w = clf.coef_[0]
 a = -w[0] / w[1]
 xx = np.linspace(-5, 5)
 yy = a * xx - (clf.intercept_[0]) / w[1]

 # plot the parallels to the separating hyperplane that pass through the
 # support vectors (margin away from hyperplane in direction
 # perpendicular to hyperplane). This is sqrt(1+a^2) away vertically in
 # 2-d.
 margin = 1 / np.sqrt(np.sum(clf.coef_ ** 2))
 yy_down = yy - np.sqrt(1 + a ** 2) * margin
 yy_up = yy + np.sqrt(1 + a ** 2) * margin

 plt.plot(xx, yy, 'k-')
 plt.plot(xx, yy_down, 'k--')
 plt.plot(xx, yy_up, 'k--')

 plt.scatter(X[Y == 0, 0], X[Y == 0, 1], c='blue')
 plt.scatter(X[Y == 1, 0], X[Y == 1, 1], c='orange')

 plt.title("Support Vector Machine with %s penalty" % name)
 plt.xlabel("Feature 1")
 plt.ylabel("Feature 2")
 plt.show()

```

```
plot the line, the points, and the nearest vectors to the plane
plt.figure(fignum, figsize=(4, 3))
plt.clf()
plt.plot(xx, yy, 'k-')
plt.plot(xx, yy_down, 'k--')
plt.plot(xx, yy_up, 'k--')

plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=80,
 facecolors='none', zorder=10, edgecolors='k')
plt.scatter(X[:, 0], X[:, 1], c=Y, zorder=10, cmap=plt.cm.Paired,
 edgecolors='k')

plt.axis('tight')
x_min = -4.8
x_max = 4.2
y_min = -6
y_max = 6

XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]
Z = clf.predict(np.c_[XX.ravel(), YY.ravel()])

Put the result into a color plot
Z = Z.reshape(XX.shape)
plt.figure(fignum, figsize=(4, 3))
plt.pcolormesh(XX, YY, Z, cmap=plt.cm.Paired)

plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)

plt.xticks(())
plt.yticks(())
fignum = fignum + 1

plt.show()
```

**Total running time of the script:** ( 0 minutes 0.061 seconds)

---

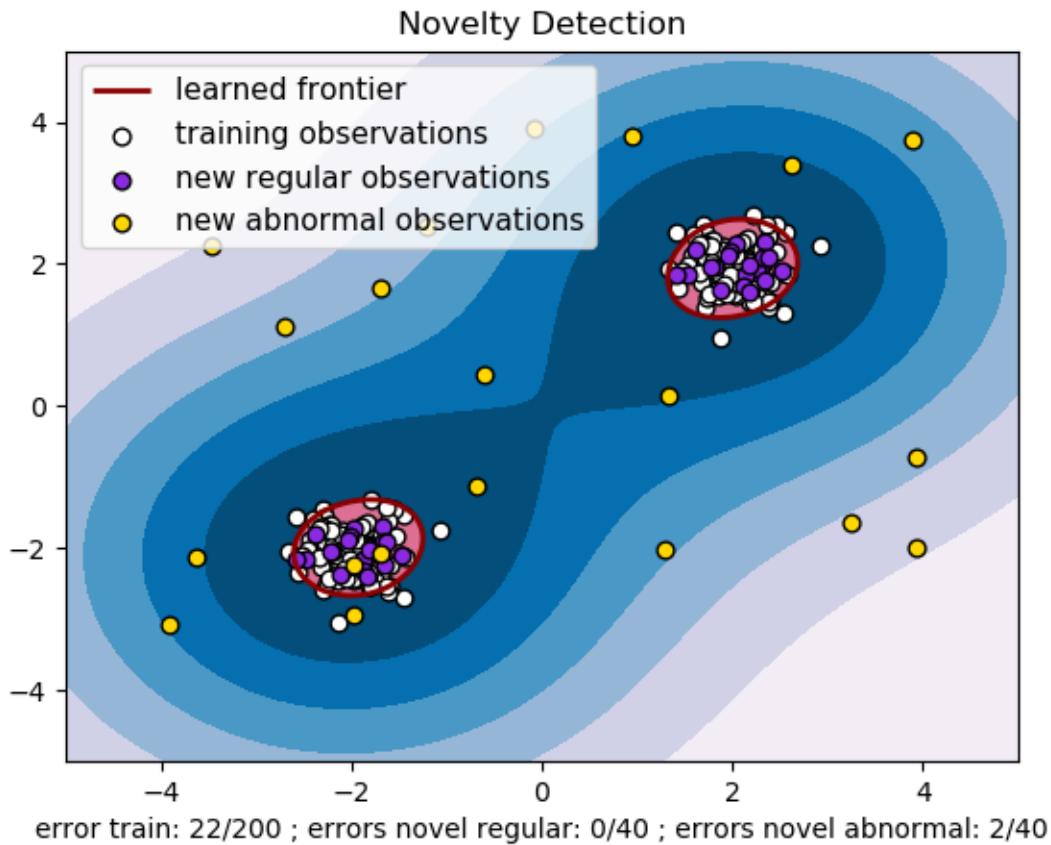
**Note:** Click [here](#) to download the full example code

---

## 5.27.10 One-class SVM with non-linear kernel (RBF)

An example using a one-class SVM for novelty detection.

*One-class SVM* is an unsupervised algorithm that learns a decision function for novelty detection: classifying new data as similar or different to the training set.



```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.font_manager
from sklearn import svm

xx, yy = np.meshgrid(np.linspace(-5, 5, 500), np.linspace(-5, 5, 500))
Generate train data
X = 0.3 * np.random.randn(100, 2)
X_train = np.r_[X + 2, X - 2]
Generate some regular novel observations
X = 0.3 * np.random.randn(20, 2)
X_test = np.r_[X + 2, X - 2]
Generate some abnormal novel observations
X_outliers = np.random.uniform(low=-4, high=4, size=(20, 2))

fit the model
clf = svm.OneClassSVM(nu=0.1, kernel="rbf", gamma=0.1)
clf.fit(X_train)

y_pred_train = clf.predict(X_train)
y_pred_test = clf.predict(X_test)
y_pred_outliers = clf.predict(X_outliers)
n_error_train = y_pred_train[y_pred_train == -1].size
n_error_test = y_pred_test[y_pred_test == -1].size
n_error_outliers = y_pred_outliers[y_pred_outliers == 1].size

```

```
plot the line, the points, and the nearest vectors to the plane
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.title("Novelty Detection")
plt.contourf(xx, yy, Z, levels=np.linspace(Z.min(), 0, 7), cmap=plt.cm.PuBu)
a = plt.contour(xx, yy, Z, levels=[0], linewidths=2, colors='darkred')
plt.contourf(xx, yy, Z, levels=[0, Z.max()], colors='palevioletred')

s = 40
b1 = plt.scatter(X_train[:, 0], X_train[:, 1], c='white', s=s, edgecolors='k')
b2 = plt.scatter(X_test[:, 0], X_test[:, 1], c='blueviolet', s=s,
 edgecolors='k')
c = plt.scatter(X_outliers[:, 0], X_outliers[:, 1], c='gold', s=s,
 edgecolors='k')
plt.axis('tight')
plt.xlim((-5, 5))
plt.ylim((-5, 5))
plt.legend([a.collections[0], b1, b2, c],
 ["learned frontier", "training observations",
 "new regular observations", "new abnormal observations"],
 loc="upper left",
 prop=matplotlib.font_manager.FontProperties(size=11))
plt.xlabel(
 "error train: %d/200 ; errors novel regular: %d/40 ; "
 "errors novel abnormal: %d/40"
 % (n_error_train, n_error_test, n_error_outliers))
plt.show()
```

**Total running time of the script:** ( 0 minutes 0.196 seconds)

---

**Note:** Click [here](#) to download the full example code

---

### 5.27.11 Plot different SVM classifiers in the iris dataset

Comparison of different linear SVM classifiers on a 2D projection of the iris dataset. We only consider the first 2 features of this dataset:

- Sepal length
- Sepal width

This example shows how to plot the decision surface for four SVM classifiers with different kernels.

The linear models `LinearSVC()` and `SVC(kernel='linear')` yield slightly different decision boundaries. This can be a consequence of the following differences:

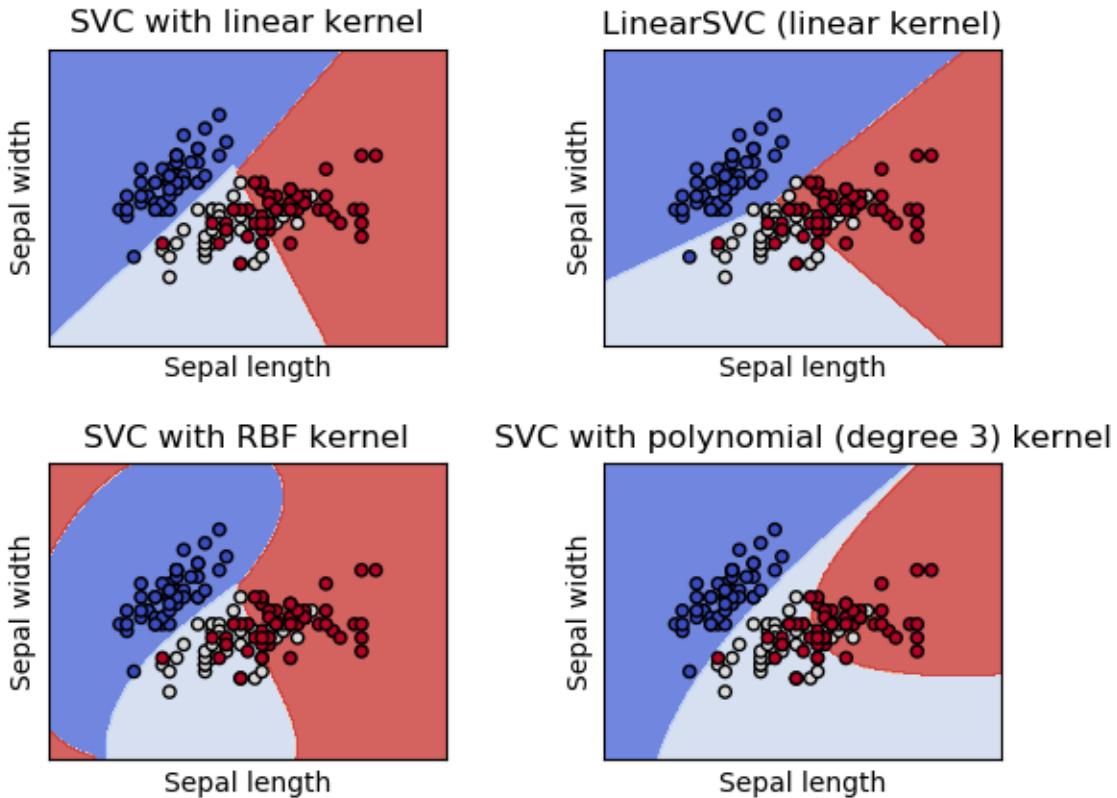
- `LinearSVC` minimizes the squared hinge loss while `SVC` minimizes the regular hinge loss.
- `LinearSVC` uses the One-vs-All (also known as One-vs-Rest) multiclass reduction while `SVC` uses the One-vs-One multiclass reduction.

Both linear models have linear decision boundaries (intersecting hyperplanes) while the non-linear kernel models (polynomial or Gaussian RBF) have more flexible non-linear decision boundaries with shapes that depend on the kind of kernel and its parameters.

---

**Note:** while plotting the decision function of classifiers for toy 2D datasets can help get an intuitive understanding of their respective expressive power, be aware that those intuitions don't always generalize to more realistic high-dimensional problems.

---



```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets

def make_meshgrid(x, y, h=.02):
 """Create a mesh of points to plot in

 Parameters

 x: data to base x-axis meshgrid on
 y: data to base y-axis meshgrid on
 h: stepsize for meshgrid, optional

 Returns

 xx, yy : ndarray
 """

```

```

x_min, x_max = x.min() - 1, x.max() + 1
y_min, y_max = y.min() - 1, y.max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
 np.arange(y_min, y_max, h))
return xx, yy

def plot_contours(ax, clf, xx, yy, **params):
 """Plot the decision boundaries for a classifier.

 Parameters

 ax: matplotlib axes object
 clf: a classifier
 xx: meshgrid ndarray
 yy: meshgrid ndarray
 params: dictionary of params to pass to contourf, optional
 """
 Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
 Z = Z.reshape(xx.shape)
 out = ax.contourf(xx, yy, Z, **params)
 return out

import some data to play with
iris = datasets.load_iris()
Take the first two features. We could avoid this by using a two-dim dataset
X = iris.data[:, :2]
y = iris.target

we create an instance of SVM and fit out data. We do not scale our
data since we want to plot the support vectors
C = 1.0 # SVM regularization parameter
models = (svm.SVC(kernel='linear', C=C),
 svm.LinearSVC(C=C, max_iter=10000),
 svm.SVC(kernel='rbf', gamma=0.7, C=C),
 svm.SVC(kernel='poly', degree=3, gamma='auto', C=C))
models = (clf.fit(X, y) for clf in models)

title for the plots
titles = ('SVC with linear kernel',
 'LinearSVC (linear kernel)',
 'SVC with RBF kernel',
 'SVC with polynomial (degree 3) kernel')

Set-up 2x2 grid for plotting.
fig, sub = plt.subplots(2, 2)
plt.subplots_adjust(wspace=0.4, hspace=0.4)

X0, X1 = X[:, 0], X[:, 1]
xx, yy = make_meshgrid(X0, X1)

for clf, title, ax in zip(models, titles, sub.flatten()):
 plot_contours(ax, clf, xx, yy,
 cmap=plt.cm.coolwarm, alpha=0.8)
 ax.scatter(X0, X1, c=y, cmap=plt.cm.coolwarm, s=20, edgecolors='k')
 ax.set_xlim(xx.min(), xx.max())
 ax.set_ylim(yy.min(), yy.max())

```

```

ax.set_xlabel('Sepal length')
ax.set_ylabel('Sepal width')
ax.set_xticks(())
ax.set_yticks(())
ax.set_title(title)

plt.show()

```

**Total running time of the script:** ( 0 minutes 0.481 seconds)

---

**Note:** Click [here](#) to download the full example code

---

### 5.27.12 Scaling the regularization parameter for SVCs

The following example illustrates the effect of scaling the regularization parameter when using *Support Vector Machines* for *classification*. For SVC classification, we are interested in a risk minimization for the equation:

$$C \sum_{i=1,n} \mathcal{L}(f(x_i), y_i) + \Omega(w)$$

where

- $C$  is used to set the amount of regularization
- $\mathcal{L}$  is a loss function of our samples and our model parameters.
- $\Omega$  is a penalty function of our model parameters

If we consider the loss function to be the individual error per sample, then the data-fit term, or the sum of the error for each sample, will increase as we add more samples. The penalization term, however, will not increase.

When using, for example, *cross validation*, to set the amount of regularization with  $C$ , there will be a different amount of samples between the main problem and the smaller problems within the folds of the cross validation.

Since our loss function is dependent on the amount of samples, the latter will influence the selected value of  $C$ . The question that arises is How do we optimally adjust  $C$  to account for the different amount of training samples?

The figures below are used to illustrate the effect of scaling our  $C$  to compensate for the change in the number of samples, in the case of using an  $l_1$  penalty, as well as the  $l_2$  penalty.

#### **$l_1$ -penalty case**

In the  $l_1$  case, theory says that prediction consistency (i.e. that under given hypothesis, the estimator learned predicts as well as a model knowing the true distribution) is not possible because of the bias of the  $l_1$ . It does say, however, that model consistency, in terms of finding the right set of non-zero parameters as well as their signs, can be achieved by scaling  $C$ .

#### **$l_2$ -penalty case**

The theory says that in order to achieve prediction consistency, the penalty parameter should be kept constant as the number of samples grow.

## Simulations

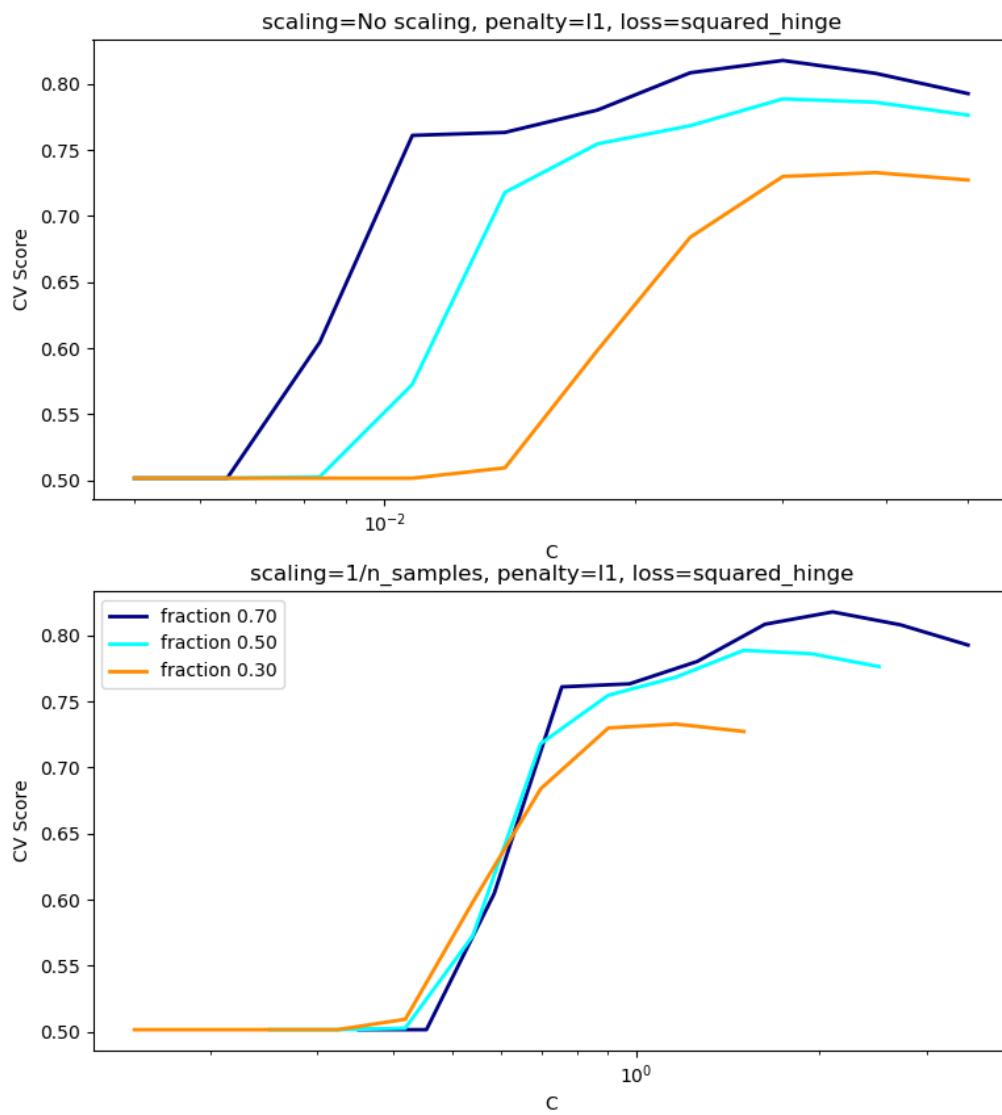
The two figures below plot the values of  $C$  on the x-axis and the corresponding cross-validation scores on the y-axis, for several different fractions of a generated data-set.

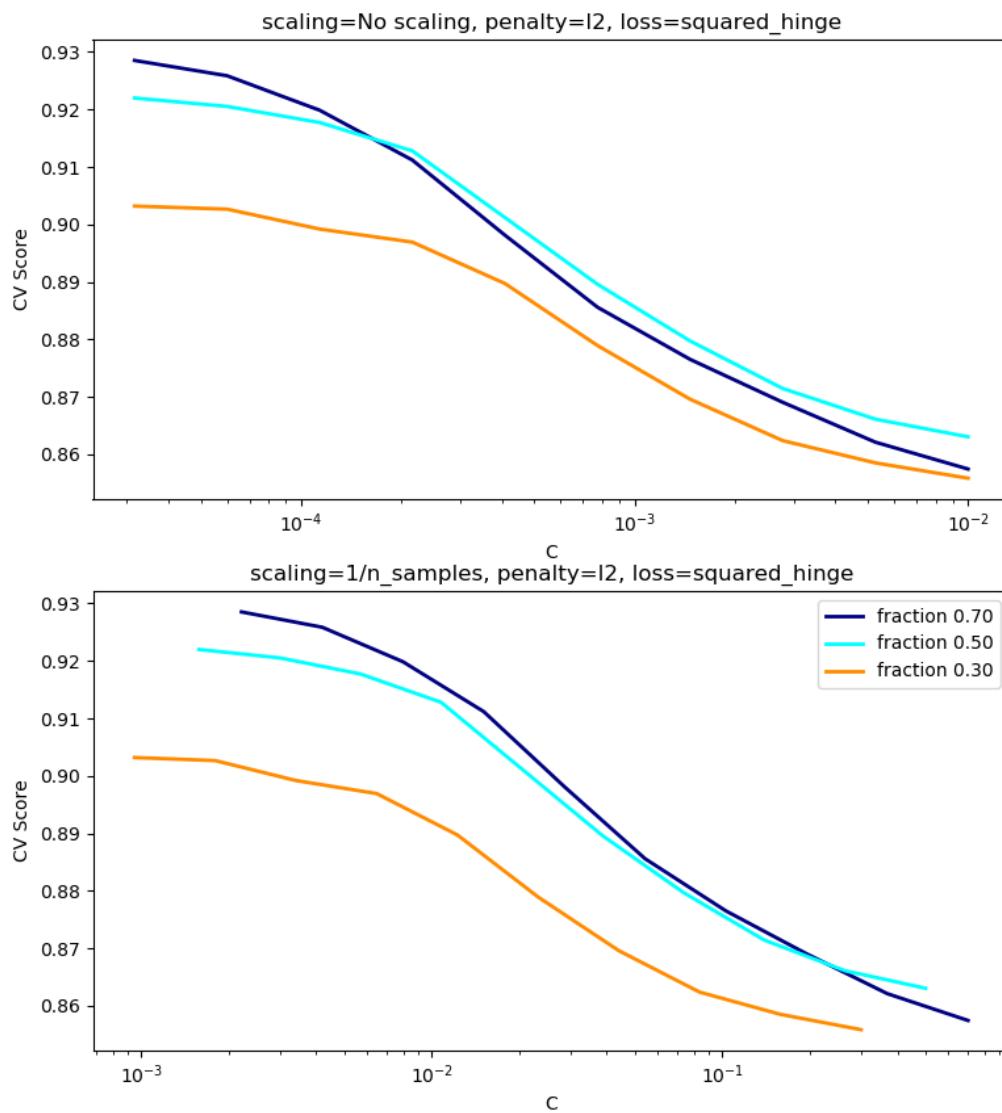
In the  $l_1$  penalty case, the cross-validation-error correlates best with the test-error, when scaling our  $C$  with the number of samples,  $n$ , which can be seen in the first figure.

For the  $l_2$  penalty case, the best result comes from the case where  $C$  is not scaled.

**Note:**

Two separate datasets are used for the two different plots. The reason behind this is the  $l_1$  case works better on sparse data, while  $l_2$  is better suited to the non-sparse case.





```

print(__doc__)

Author: Andreas Mueller <amueller@ais.uni-bonn.de>
Jaques Grobler <jaques.grobler@inria.fr>
License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt

from sklearn.svm import LinearSVC
from sklearn.model_selection import ShuffleSplit

```

```

from sklearn.model_selection import GridSearchCV
from sklearn.utils import check_random_state
from sklearn import datasets

rnd = check_random_state(1)

set up dataset
n_samples = 100
n_features = 300

11 data (only 5 informative features)
X_1, y_1 = datasets.make_classification(n_samples=n_samples,
 n_features=n_features, n_informative=5,
 random_state=1)

12 data: non sparse, but less features
y_2 = np.sign(.5 - rnd.rand(n_samples))
X_2 = rnd.randn(n_samples, n_features // 5) + y_2[:, np.newaxis]
X_2 += 5 * rnd.randn(n_samples, n_features // 5)

clf_sets = [(LinearSVC(penalty='11', loss='squared_hinge', dual=False,
 tol=1e-3),
 np.logspace(-2.3, -1.3, 10), X_1, y_1),
 (LinearSVC(penalty='12', loss='squared_hinge', dual=True,
 tol=1e-4),
 np.logspace(-4.5, -2, 10), X_2, y_2)]

colors = ['navy', 'cyan', 'darkorange']
lw = 2

for clf, cs, X, y in clf_sets:
 # set up the plot for each regressor
 fig, axes = plt.subplots(nrows=2, sharey=True, figsize=(9, 10))

 for k, train_size in enumerate(np.linspace(0.3, 0.7, 3)[::-1]):
 param_grid = dict(C=cs)
 # To get nice curve, we need a large number of iterations to
 # reduce the variance
 grid = GridSearchCV(clf, refit=False, param_grid=param_grid,
 cv=ShuffleSplit(train_size=train_size,
 test_size=.3,
 n_splits=250, random_state=1))
 grid.fit(X, y)
 scores = grid.cv_results_['mean_test_score']

 scales = [(1, 'No scaling'),
 ((n_samples * train_size), '1/n_samples'),
]

 for ax, (scaler, name) in zip(axes, scales):
 ax.set_xlabel('C')
 ax.set_ylabel('CV Score')
 grid_cs = cs * float(scaler) # scale the C's
 ax.semilogx(grid_cs, scores, label="fraction %.2f" %
 train_size, color=colors[k], lw=lw)
 ax.set_title('scaling=%s, penalty=%s, loss=%s' %
 (name, clf.penalty, clf.loss))

```

```
plt.legend(loc="best")
plt.show()
```

**Total running time of the script:** ( 0 minutes 14.575 seconds)

---

**Note:** Click [here](#) to download the full example code

---

### 5.27.13 RBF SVM parameters

This example illustrates the effect of the parameters `gamma` and `C` of the Radial Basis Function (RBF) kernel SVM.

Intuitively, the `gamma` parameter defines how far the influence of a single training example reaches, with low values meaning ‘far’ and high values meaning ‘close’. The `gamma` parameters can be seen as the inverse of the radius of influence of samples selected by the model as support vectors.

The `C` parameter trades off correct classification of training examples against maximization of the decision function’s margin. For larger values of `C`, a smaller margin will be accepted if the decision function is better at classifying all training points correctly. A lower `C` will encourage a larger margin, therefore a simpler decision function, at the cost of training accuracy. In other words “`C`“ behaves as a regularization parameter in the SVM.

The first plot is a visualization of the decision function for a variety of parameter values on a simplified classification problem involving only 2 input features and 2 possible target classes (binary classification). Note that this kind of plot is not possible to do for problems with more features or target classes.

The second plot is a heatmap of the classifier’s cross-validation accuracy as a function of `C` and `gamma`. For this example we explore a relatively large grid for illustration purposes. In practice, a logarithmic grid from  $10^{-3}$  to  $10^3$  is usually sufficient. If the best parameters lie on the boundaries of the grid, it can be extended in that direction in a subsequent search.

Note that the heat map plot has a special colorbar with a midpoint value close to the score values of the best performing models so as to make it easy to tell them apart in the blink of an eye.

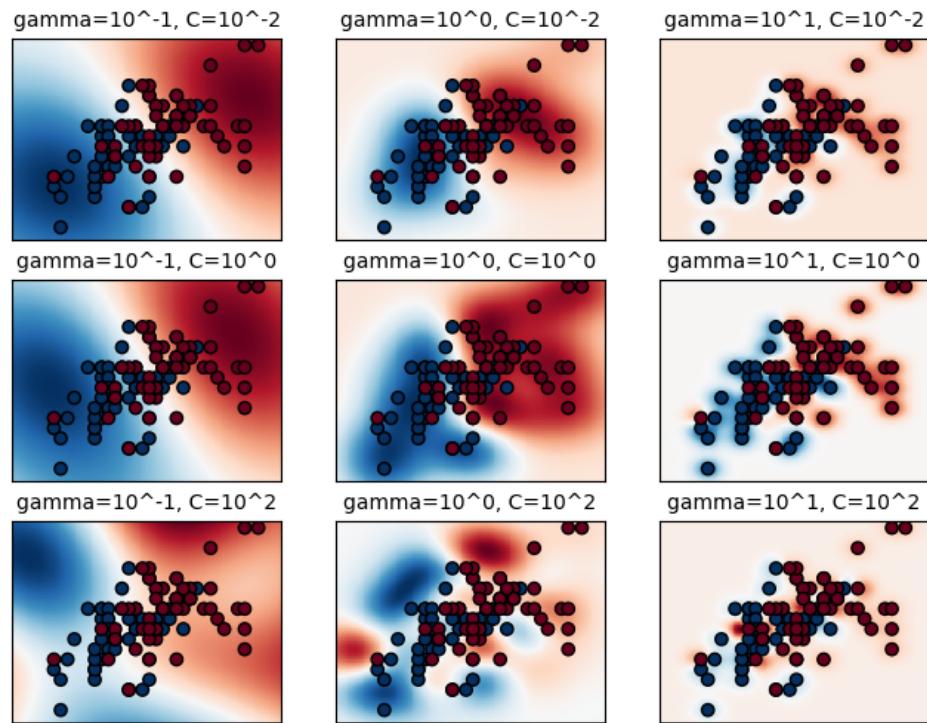
The behavior of the model is very sensitive to the `gamma` parameter. If `gamma` is too large, the radius of the area of influence of the support vectors only includes the support vector itself and no amount of regularization with `C` will be able to prevent overfitting.

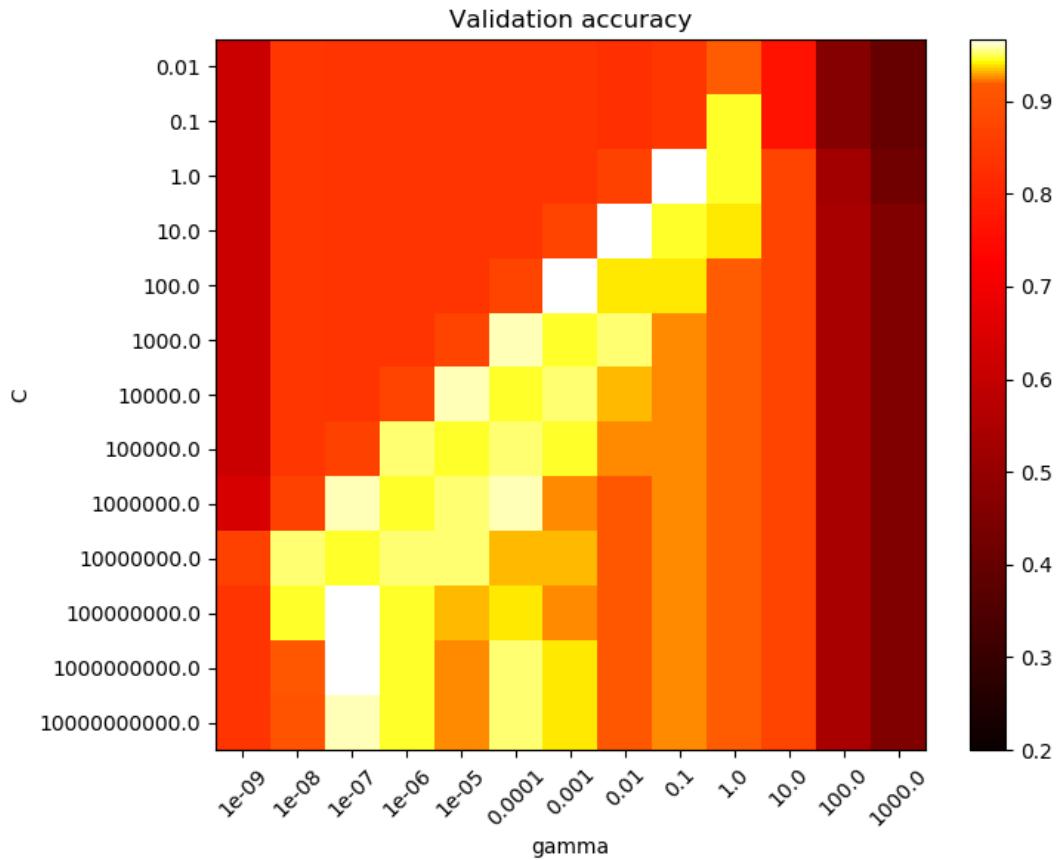
When `gamma` is very small, the model is too constrained and cannot capture the complexity or “shape” of the data. The region of influence of any selected support vector would include the whole training set. The resulting model will behave similarly to a linear model with a set of hyperplanes that separate the centers of high density of any pair of two classes.

For intermediate values, we can see on the second plot that good models can be found on a diagonal of `C` and `gamma`. Smooth models (lower `gamma` values) can be made more complex by increasing the importance of classifying each point correctly (larger `C` values) hence the diagonal of good performing models.

Finally one can also observe that for some intermediate values of `gamma` we get equally performing models when `C` becomes very large: it is not necessary to regularize by enforcing a larger margin. The radius of the RBF kernel alone acts as a good structural regularizer. In practice though it might still be interesting to simplify the decision function with a lower value of `C` so as to favor models that use less memory and that are faster to predict.

We should also note that small differences in scores results from the random splits of the cross-validation procedure. Those spurious variations can be smoothed out by increasing the number of CV iterations `n_splits` at the expense of compute time. Increasing the value number of `C_range` and `gamma_range` steps will increase the resolution of the hyper-parameter heat map.





Out:

```
The best parameters are {'C': 1.0, 'gamma': 0.1} with a score of 0.97
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import Normalize

from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.model_selection import GridSearchCV

Utility function to move the midpoint of a colormap to be around
the values of interest.

class MidpointNormalize(Normalize):
 pass
```

```

def __init__(self, vmin=None, vmax=None, midpoint=None, clip=False):
 self.midpoint = midpoint
 Normalize.__init__(self, vmin, vmax, clip)

def __call__(self, value, clip=None):
 x, y = [self.vmin, self.midpoint, self.vmax], [0, 0.5, 1]
 return np.ma.masked_array(np.interp(value, x, y))

######
Load and prepare data set
#
dataset for grid search

iris = load_iris()
X = iris.data
y = iris.target

Dataset for decision function visualization: we only keep the first two
features in X and sub-sample the dataset to keep only 2 classes and
make it a binary classification problem.

X_2d = X[:, :2]
X_2d = X_2d[y > 0]
y_2d = y[y > 0]
y_2d -= 1

It is usually a good idea to scale the data for SVM training.
We are cheating a bit in this example in scaling all of the data,
instead of fitting the transformation on the training set and
just applying it on the test set.

scaler = StandardScaler()
X = scaler.fit_transform(X)
X_2d = scaler.fit_transform(X_2d)

######
Train classifiers
#
For an initial search, a logarithmic grid with basis
10 is often helpful. Using a basis of 2, a finer
tuning can be achieved but at a much higher cost.

C_range = np.logspace(-2, 10, 13)
gamma_range = np.logspace(-9, 3, 13)
param_grid = dict(gamma=gamma_range, C=C_range)
cv = StratifiedShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
grid = GridSearchCV(SVC(), param_grid=param_grid, cv=cv)
grid.fit(X, y)

print("The best parameters are %s with a score of %0.2f"
 % (grid.best_params_, grid.best_score_))

Now we need to fit a classifier for all parameters in the 2d version
(we use a smaller set of parameters here because it takes a while to train)

C_2d_range = [1e-2, 1, 1e2]
gamma_2d_range = [1e-1, 1, 1e1]
classifiers = []

```

```

for C in C_2d_range:
 for gamma in gamma_2d_range:
 clf = SVC(C=C, gamma=gamma)
 clf.fit(X_2d, y_2d)
 classifiers.append((C, gamma, clf))

##########
Visualization
#
draw visualization of parameter effects

plt.figure(figsize=(8, 6))
xx, yy = np.meshgrid(np.linspace(-3, 3, 200), np.linspace(-3, 3, 200))
for (k, (C, gamma, clf)) in enumerate(classifiers):
 # evaluate decision function in a grid
 Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
 Z = Z.reshape(xx.shape)

 # visualize decision function for these parameters
 plt.subplot(len(C_2d_range), len(gamma_2d_range), k + 1)
 plt.title("gamma=%d, C=%d" % (np.log10(gamma), np.log10(C)),
 size='medium')

 # visualize parameter's effect on decision function
 plt.pcolormesh(xx, yy, -Z, cmap=plt.cm.RdBu)
 plt.scatter(X_2d[:, 0], X_2d[:, 1], c=y_2d, cmap=plt.cm.RdBu_r,
 edgecolors='k')
 plt.xticks(())
 plt.yticks(())
 plt.axis('tight')

scores = grid.cv_results_['mean_test_score'].reshape(len(C_range),
 len(gamma_range))

Draw heatmap of the validation accuracy as a function of gamma and C
#
The score are encoded as colors with the hot colormap which varies from dark
red to bright yellow. As the most interesting scores are all located in the
0.92 to 0.97 range we use a custom normalizer to set the mid-point to 0.92 so
as to make it easier to visualize the small variations of score values in the
interesting range while not brutally collapsing all the low score values to
the same color.

plt.figure(figsize=(8, 6))
plt.subplots_adjust(left=.2, right=0.95, bottom=0.15, top=0.95)
plt.imshow(scores, interpolation='nearest', cmap=plt.cm.hot,
 norm=MidpointNormalize(vmin=0.2, midpoint=0.92))
plt.xlabel('gamma')
plt.ylabel('C')
plt.colorbar()
plt.xticks(np.arange(len(gamma_range)), gamma_range, rotation=45)
plt.yticks(np.arange(len(C_range)), C_range)
plt.title('Validation accuracy')
plt.show()

```

**Total running time of the script:** ( 0 minutes 3.597 seconds)

## 5.28 Working with text documents

Examples concerning the `sklearn.feature_extraction.text` module.

---

**Note:** Click [here](#) to download the full example code

---

### 5.28.1 FeatureHasher and DictVectorizer Comparison

Compares FeatureHasher and DictVectorizer by using both to vectorize text documents.

The example demonstrates syntax and speed only; it doesn't actually do anything useful with the extracted vectors. See the example scripts `{document_classification_20newsgroups,clustering}.py` for actual learning on text documents.

A discrepancy between the number of terms reported for DictVectorizer and for FeatureHasher is to be expected due to hash collisions.

Out:

```
Usage: /home/circleci/project/examples/text/plot_hashing_vs_dict_vectorizer.py [n_
→features_for_hashing]
 The default number of features is 2**18.

Loading 20 newsgroups training data
3803 documents - 6.245MB

DictVectorizer
done in 0.979095s at 6.378MB/s
Found 47928 unique terms

FeatureHasher on frequency dicts
done in 0.816599s at 7.647MB/s
Found 43873 unique terms

FeatureHasher on raw tokens
done in 0.935579s at 6.675MB/s
Found 43873 unique terms
```

```
Author: Lars Buitinck
License: BSD 3 clause
from collections import defaultdict
import re
import sys
from time import time

import numpy as np

from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction import DictVectorizer, FeatureHasher
```

```
def n_nonzero_columns(X):
 """Returns the number of non-zero columns in a CSR matrix X."""
 return len(np.unique(X.nonzero()[1]))

def tokens(doc):
 """Extract tokens from doc.

 This uses a simple regex to break strings into tokens. For a more
 principled approach, see CountVectorizer or TfidfVectorizer.
 """
 return (tok.lower() for tok in re.findall(r"\w+", doc))

def token_freqs(doc):
 """Extract a dict mapping tokens from doc to their frequencies."""
 freq = defaultdict(int)
 for tok in tokens(doc):
 freq[tok] += 1
 return freq

categories = [
 'alt.atheism',
 'comp.graphics',
 'comp.sys.ibm.pc.hardware',
 'misc.forsale',
 'rec.autos',
 'sci.space',
 'talk.religion.misc',
]
Uncomment the following line to use a larger set (11k+ documents)
categories = None

print(__doc__)
print("Usage: %s [n_features_for_hashing]" % sys.argv[0])
print(" The default number of features is 2**18.")
print()

try:
 n_features = int(sys.argv[1])
except IndexError:
 n_features = 2 ** 18
except ValueError:
 print("not a valid number of features: %r" % sys.argv[1])
 sys.exit(1)

print("Loading 20 newsgroups training data")
raw_data = fetch_20newsgroups(subset='train', categories=categories).data
data_size_mb = sum(len(s.encode('utf-8')) for s in raw_data) / 1e6
print("%d documents - %.3fMB" % (len(raw_data), data_size_mb))
print()

print("DictVectorizer")
t0 = time()
vectorizer = DictVectorizer()
vectorizer.fit_transform(token_freqs(d) for d in raw_data)
```

```

duration = time() - t0
print("done in %fs at %0.3fMB/s" % (duration, data_size_mb / duration))
print("Found %d unique terms" % len(vectorizer.get_feature_names()))
print()

print("FeatureHasher on frequency dicts")
t0 = time()
hasher = FeatureHasher(n_features=n_features)
X = hasher.transform(token_freqs(d) for d in raw_data)
duration = time() - t0
print("done in %fs at %0.3fMB/s" % (duration, data_size_mb / duration))
print("Found %d unique terms" % n_nonzero_columns(X))
print()

print("FeatureHasher on raw tokens")
t0 = time()
hasher = FeatureHasher(n_features=n_features, input_type="string")
X = hasher.transform(tokens(d) for d in raw_data)
duration = time() - t0
print("done in %fs at %0.3fMB/s" % (duration, data_size_mb / duration))
print("Found %d unique terms" % n_nonzero_columns(X))

```

**Total running time of the script:** ( 0 minutes 3.036 seconds)

---

**Note:** Click [here](#) to download the full example code

---

## 5.28.2 Clustering text documents using k-means

This is an example showing how the scikit-learn can be used to cluster documents by topics using a bag-of-words approach. This example uses a `scipy.sparse` matrix to store the features instead of standard numpy arrays.

Two feature extraction methods can be used in this example:

- `TfidfVectorizer` uses a in-memory vocabulary (a python dict) to map the most frequent words to features indices and hence compute a word occurrence frequency (sparse) matrix. The word frequencies are then reweighted using the Inverse Document Frequency (IDF) vector collected feature-wise over the corpus.
- `HashingVectorizer` hashes word occurrences to a fixed dimensional space, possibly with collisions. The word count vectors are then normalized to each have l2-norm equal to one (projected to the euclidean unit-ball) which seems to be important for k-means to work in high dimensional space.

`HashingVectorizer` does not provide IDF weighting as this is a stateless model (the fit method does nothing). When IDF weighting is needed it can be added by pipelining its output to a `TfidfTransformer` instance.

Two algorithms are demoed: ordinary k-means and its more scalable cousin minibatch k-means.

Additionally, latent semantic analysis can also be used to reduce dimensionality and discover latent patterns in the data.

It can be noted that k-means (and minibatch k-means) are very sensitive to feature scaling and that in this case the IDF weighting helps improve the quality of the clustering by quite a lot as measured against the “ground truth” provided by the class label assignments of the 20 newsgroups dataset.

This improvement is not visible in the Silhouette Coefficient which is small for both as this measure seem to suffer from the phenomenon called “Concentration of Measure” or “Curse of Dimensionality” for high dimensional datasets

such as text data. Other measures such as V-measure and Adjusted Rand Index are information theoretic based evaluation scores: as they are only based on cluster assignments rather than distances, hence not affected by the curse of dimensionality.

Note: as k-means is optimizing a non-convex objective function, it will likely end up in a local optimum. Several runs with independent random init might be necessary to get a good convergence.

Out:

```
Usage: plot_document_clustering.py [options]

Options:
-h, --help show this help message and exit
--lsa=N_COMPONENTS Preprocess documents with latent semantic analysis.
--no-minibatch Use ordinary k-means algorithm (in batch mode).
--no-idf Disable Inverse Document Frequency feature weighting.
--use-hashing Use a hashing feature vectorizer
--n-features=N_FEATURES Maximum number of features (dimensions) to extract
 from text.
--verbose Print progress reports inside k-means algorithm.

Loading 20 newsgroups dataset for categories:
['alt.atheism', 'talk.religion.misc', 'comp.graphics', 'sci.space']
3387 documents
4 categories

Extracting features from the training dataset using a sparse vectorizer
done in 0.691398s
n_samples: 3387, n_features: 10000

Clustering sparse data with MiniBatchKMeans(batch_size=1000, init_size=1000, n_
→clusters=4, n_init=1,
 verbose=False)
done in 0.057s

Homogeneity: 0.359
Completeness: 0.440
V-measure: 0.396
Adjusted Rand-Index: 0.253
Silhouette Coefficient: 0.007

Top terms per cluster:
Cluster 0: henry alaska toronto moon zoo spencer aurora space nsmca zoology
Cluster 1: com graphics university posting host nntp know uk article cs
Cluster 2: god com sandvik people keith morality sgi kent livesey jesus
Cluster 3: space nasa access gov digex pat shuttle hst orbit net
```

```
Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>
Lars Buitinck
License: BSD 3 clause
from sklearn.datasets import fetch_20newsgroups
from sklearn.decomposition import TruncatedSVD
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import HashingVectorizer
```

```

from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import Normalizer
from sklearn import metrics

from sklearn.cluster import KMeans, MiniBatchKMeans

import logging
from optparse import OptionParser
import sys
from time import time

import numpy as np

Display progress logs on stdout
logging.basicConfig(level=logging.INFO,
 format='%(asctime)s %(levelname)s %(message)s')

parse commandline arguments
op = OptionParser()
op.add_option("--lsa",
 dest="n_components", type="int",
 help="Preprocess documents with latent semantic analysis.")
op.add_option("--no-minibatch",
 action="store_false", dest="minibatch", default=True,
 help="Use ordinary k-means algorithm (in batch mode).")
op.add_option("--no-idf",
 action="store_false", dest="use_idf", default=True,
 help="Disable Inverse Document Frequency feature weighting.")
op.add_option("--use-hashing",
 action="store_true", default=False,
 help="Use a hashing feature vectorizer")
op.add_option("--n-features", type=int, default=10000,
 help="Maximum number of features (dimensions) "
 "to extract from text.")
op.add_option("--verbose",
 action="store_true", dest="verbose", default=False,
 help="Print progress reports inside k-means algorithm.")

print(__doc__)
op.print_help()

def is_interactive():
 return not hasattr(sys.modules['__main__'], '__file__')

work-around for Jupyter notebook and IPython console
argv = [] if is_interactive() else sys.argv[1:]
(opts, args) = op.parse_args(argv)
if len(args) > 0:
 op.error("this script takes no arguments.")
 sys.exit(1)

#####
Load some categories from the training set

```

```
categories = [
 'alt.atheism',
 'talk.religion.misc',
 'comp.graphics',
 'sci.space',
]
Uncomment the following to do the analysis on all the categories
categories = None

print("Loading 20 newsgroups dataset for categories:")
print(categories)

dataset = fetch_20newsgroups(subset='all', categories=categories,
 shuffle=True, random_state=42)

print("%d documents" % len(dataset.data))
print("%d categories" % len(dataset.target_names))
print()

labels = dataset.target
true_k = np.unique(labels).shape[0]

print("Extracting features from the training dataset "
 "using a sparse vectorizer")
t0 = time()
if opts.use_hashing:
 if opts.use_idf:
 # Perform an IDF normalization on the output of HashingVectorizer
 hasher = HashingVectorizer(n_features=opts.n_features,
 stop_words='english', alternate_sign=False,
 norm=None, binary=False)
 vectorizer = make_pipeline(hasher, TfidfTransformer())
 else:
 vectorizer = HashingVectorizer(n_features=opts.n_features,
 stop_words='english',
 alternate_sign=False, norm='l2',
 binary=False)
else:
 vectorizer = TfidfVectorizer(max_df=0.5, max_features=opts.n_features,
 min_df=2, stop_words='english',
 use_idf=opts.use_idf)
X = vectorizer.fit_transform(dataset.data)

print("done in %fs" % (time() - t0))
print("n_samples: %d, n_features: %d" % X.shape)
print()

if opts.n_components:
 print("Performing dimensionality reduction using LSA")
 t0 = time()
 # Vectorizer results are normalized, which makes KMeans behave as
 # spherical k-means for better results. Since LSA/SVD results are
 # not normalized, we have to redo the normalization.
 svd = TruncatedSVD(opts.n_components)
 normalizer = Normalizer(copy=False)
 lsa = make_pipeline(svd, normalizer)

 X = lsa.fit_transform(X)
```

```

print("done in %fs" % (time() - t0))

explained_variance = svd.explained_variance_ratio_.sum()
print("Explained variance of the SVD step: {}%".format(
 int(explained_variance * 100)))

print()

##########
Do the actual clustering

if opts.minibatch:
 km = MiniBatchKMeans(n_clusters=true_k, init='k-means++', n_init=1,
 init_size=1000, batch_size=1000, verbose=opts.verbose)
else:
 km = KMeans(n_clusters=true_k, init='k-means++', max_iter=100, n_init=1,
 verbose=opts.verbose)

print("Clustering sparse data with %s" % km)
t0 = time()
km.fit(X)
print("done in %0.3fs" % (time() - t0))
print()

print("Homogeneity: %0.3f" % metrics.homogeneity_score(labels, km.labels_))
print("Completeness: %0.3f" % metrics.completeness_score(labels, km.labels_))
print("V-measure: %0.3f" % metrics.v_measure_score(labels, km.labels_))
print("Adjusted Rand-Index: %.3f"
 % metrics.adjusted_rand_score(labels, km.labels_))
print("Silhouette Coefficient: %0.3f"
 % metrics.silhouette_score(X, km.labels_, sample_size=1000))

print()

if not opts.use_hashing:
 print("Top terms per cluster:")

 if opts.n_components:
 original_space_centroids = svd.inverse_transform(km.cluster_centers_)
 order_centroids = original_space_centroids.argsort()[:, ::-1]
 else:
 order_centroids = km.cluster_centers_.argsort()[:, ::-1]

 terms = vectorizer.get_feature_names()
 for i in range(true_k):
 print("Cluster %d:" % i, end=' ')
 for ind in order_centroids[i, :10]:
 print(' %s' % terms[ind], end=' ')
 print()

```

**Total running time of the script:** ( 0 minutes 1.137 seconds)

---

**Note:** Click [here](#) to download the full example code

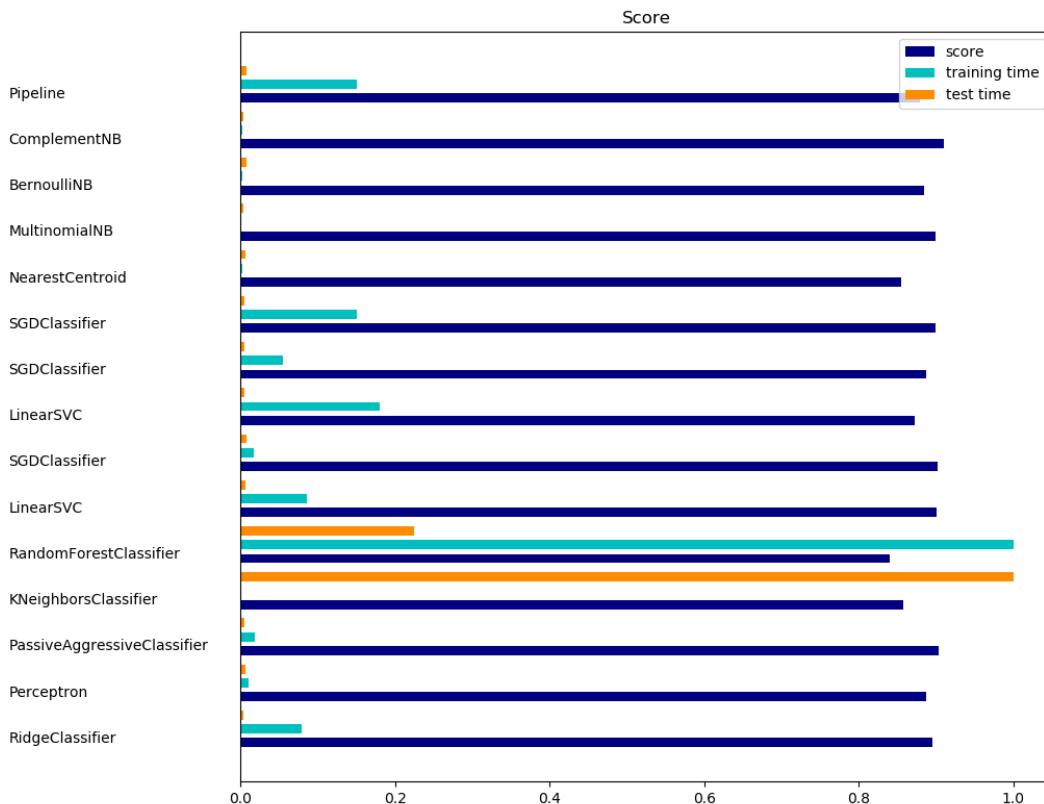
---

### 5.28.3 Classification of text documents using sparse features

This is an example showing how scikit-learn can be used to classify documents by topics using a bag-of-words approach. This example uses a `scipy.sparse` matrix to store the features and demonstrates various classifiers that can efficiently handle sparse matrices.

The dataset used in this example is the 20 newsgroups dataset. It will be automatically downloaded, then cached.

The bar plot indicates the accuracy, training time (normalized) and test time (normalized) of each classifier.



Out:

```
Usage: plot_document_classification_20newsgroups.py [options]

Options:
-h, --help show this help message and exit
--report Print a detailed classification report.
--chi2_select=SELECT_CHI2
 Select some number of features using a chi-squared
 test
--confusion_matrix Print the confusion matrix.
--top10 Print ten most discriminative terms per class for
 every classifier.
--all_categories Whether to use all categories or not.
--use_hashing Use a hashing vectorizer.
--n_features=N_FEATURES
 n_features when using the hashing vectorizer.
--filtered Remove newsgroup information that is easily overfit:
 headers, signatures, and quoting.
```

```
Loading 20 newsgroups dataset for categories:
['alt.atheism', 'talk.religion.misc', 'comp.graphics', 'sci.space']
data loaded
2034 documents - 3.980MB (training set)
1353 documents - 2.867MB (test set)
4 categories

Extracting features from the training data using a sparse vectorizer
done in 0.412178s at 9.655MB/s
n_samples: 2034, n_features: 33809

Extracting features from the test data using the same vectorizer
done in 0.351330s at 8.162MB/s
n_samples: 1353, n_features: 33809

=====
Ridge Classifier

Training:
RidgeClassifier(solver='sag', tol=0.01)
train time: 0.132s
test time: 0.001s
accuracy: 0.896
dimensionality: 33809
density: 1.000000

=====
Perceptron

Training:
Perceptron(max_iter=50)
train time: 0.017s
test time: 0.002s
accuracy: 0.888
dimensionality: 33809
density: 0.255302

=====
Passive-Aggressive

Training:
PassiveAggressiveClassifier(max_iter=50)
train time: 0.031s
test time: 0.002s
accuracy: 0.904
dimensionality: 33809
density: 0.694674

=====
kNN

Training:
KNeighborsClassifier(n_neighbors=10)
train time: 0.002s
test time: 0.317s
```

```
accuracy: 0.858
```

```
=====
```

```
Random forest
```

---

```
Training:
```

```
RandomForestClassifier(n_estimators=100)
train time: 1.671s
test time: 0.071s
accuracy: 0.840
```

```
=====
```

```
L2 penalty
```

---

```
Training:
```

```
LinearSVC(dual=False, tol=0.001)
train time: 0.145s
test time: 0.002s
accuracy: 0.900
dimensionality: 33809
density: 1.000000
```

---

```
Training:
```

```
SGDClassifier(max_iter=50)
train time: 0.030s
test time: 0.002s
accuracy: 0.902
dimensionality: 33809
density: 0.579380
```

```
=====
```

```
L1 penalty
```

---

```
Training:
```

```
LinearSVC(dual=False, penalty='l1', tol=0.001)
train time: 0.301s
test time: 0.002s
accuracy: 0.873
dimensionality: 33809
density: 0.005553
```

---

```
Training:
```

```
SGDClassifier(max_iter=50, penalty='l1')
train time: 0.093s
test time: 0.002s
accuracy: 0.887
dimensionality: 33809
density: 0.022901
```

```
=====
```

```
Elastic-Net penalty
```

---

```
Training:
SGDClassifier(max_iter=50, penalty='elasticnet')
train time: 0.252s
test time: 0.002s
accuracy: 0.899
dimensionality: 33809
density: 0.187472
```

```
=====
NearestCentroid (aka Rocchio classifier)
```

---

```
Training:
NearestCentroid()
train time: 0.004s
test time: 0.002s
accuracy: 0.855
```

```
=====
Naive Bayes
```

---

```
Training:
MultinomialNB(alpha=0.01)
train time: 0.003s
test time: 0.001s
accuracy: 0.899
dimensionality: 33809
density: 1.000000
```

---

```
Training:
BernoulliNB(alpha=0.01)
train time: 0.004s
test time: 0.003s
accuracy: 0.884
dimensionality: 33809
density: 1.000000
```

---

```
Training:
ComplementNB(alpha=0.1)
train time: 0.004s
test time: 0.001s
accuracy: 0.911
dimensionality: 33809
density: 1.000000
```

```
=====
LinearSVC with L1-based feature selection
```

---

```
Training:
Pipeline(steps=[('feature_selection',
 SelectFromModel(estimator=LinearSVC(dual=False, penalty='l1',
 tol=0.001))),
 ('classification', LinearSVC())])
```

```
train time: 0.252s
test time: 0.002s
accuracy: 0.880
```

```
Author: Peter Prettenhofer <peter.prettenhofer@gmail.com>
Olivier Grisel <olivier.grisel@ensta.org>
Mathieu Blondel <mathieu@mblondel.org>
Lars Buitinck
License: BSD 3 clause

import logging
import numpy as np
from optparse import OptionParser
import sys
from time import time
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import HashingVectorizer
from sklearn.feature_selection import SelectFromModel
from sklearn.feature_selection import SelectKBest, chi2
from sklearn.linear_model import RidgeClassifier
from sklearn.pipeline import Pipeline
from sklearn.svm import LinearSVC
from sklearn.linear_model import SGDClassifier
from sklearn.linear_model import Perceptron
from sklearn.linear_model import PassiveAggressiveClassifier
from sklearn.naive_bayes import BernoulliNB, ComplementNB, MultinomialNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neighbors import NearestCentroid
from sklearn.ensemble import RandomForestClassifier
from sklearn.utils.extmath import density
from sklearn import metrics

Display progress logs on stdout
logging.basicConfig(level=logging.INFO,
 format='%(asctime)s %(levelname)s %(message)s')

parse commandline arguments
op = OptionParser()
op.add_option("--report",
 action="store_true", dest="print_report",
 help="Print a detailed classification report.")
op.add_option("--chi2_select",
 action="store", type="int", dest="select_chi2",
 help="Select some number of features using a chi-squared test")
op.add_option("--confusion_matrix",
 action="store_true", dest="print_cm",
 help="Print the confusion matrix.")
op.add_option("--top10",
 action="store_true", dest="print_top10",
```

```

 help="Print ten most discriminative terms per class"
 " for every classifier.")
op.add_option("--all_categories",
 action="store_true", dest="all_categories",
 help="Whether to use all categories or not.")
op.add_option("--use_hashing",
 action="store_true",
 help="Use a hashing vectorizer.")
op.add_option("--n_features",
 action="store", type=int, default=2 ** 16,
 help="n_features when using the hashing vectorizer.")
op.add_option("--filtered",
 action="store_true",
 help="Remove newsgroup information that is easily overfit:
 "headers, signatures, and quoting.")

def is_interactive():
 return not hasattr(sys.modules['__main__'], '__file__')

work-around for Jupyter notebook and IPython console
argv = [] if is_interactive() else sys.argv[1:]
(opts, args) = op.parse_args(argv)
if len(args) > 0:
 op.error("this script takes no arguments.")
 sys.exit(1)

print(__doc__)
op.print_help()
print()

#####
Load some categories from the training set
if opts.all_categories:
 categories = None
else:
 categories = [
 'alt.atheism',
 'talk.religion.misc',
 'comp.graphics',
 'sci.space',
]

if opts.filtered:
 remove = ('headers', 'footers', 'quotes')
else:
 remove = ()

print("Loading 20 newsgroups dataset for categories:")
print(categories if categories else "all")

data_train = fetch_20newsgroups(subset='train', categories=categories,
 shuffle=True, random_state=42,
 remove=remove)

data_test = fetch_20newsgroups(subset='test', categories=categories,

```

```
shuffle=True, random_state=42,
remove=remove)

print('data loaded')

order of labels in `target_names` can be different from `categories`
target_names = data_train.target_names

def size_mb(docs):
 return sum(len(s.encode('utf-8')) for s in docs) / 1e6

data_train_size_mb = size_mb(data_train.data)
data_test_size_mb = size_mb(data_test.data)

print("%d documents - %.3fMB (training set)" % (
 len(data_train.data), data_train_size_mb))
print("%d documents - %.3fMB (test set)" % (
 len(data_test.data), data_test_size_mb))
print("%d categories" % len(target_names))
print()

split a training set and a test set
y_train, y_test = data_train.target, data_test.target

print("Extracting features from the training data using a sparse vectorizer")
t0 = time()
if opts.use_hashing:
 vectorizer = HashingVectorizer(stop_words='english', alternate_sign=False,
 n_features=opts.n_features)
 X_train = vectorizer.transform(data_train.data)
else:
 vectorizer = TfidfVectorizer(sublinear_tf=True, max_df=0.5,
 stop_words='english')
 X_train = vectorizer.fit_transform(data_train.data)
duration = time() - t0
print("done in %fs at %.3fMB/s" % (duration, data_train_size_mb / duration))
print("n_samples: %d, n_features: %d" % X_train.shape)
print()

print("Extracting features from the test data using the same vectorizer")
t0 = time()
X_test = vectorizer.transform(data_test.data)
duration = time() - t0
print("done in %fs at %.3fMB/s" % (duration, data_test_size_mb / duration))
print("n_samples: %d, n_features: %d" % X_test.shape)
print()

mapping from integer feature name to original token string
if opts.use_hashing:
 feature_names = None
else:
 feature_names = vectorizer.get_feature_names()

if opts.select_chi2:
 print("Extracting %d best features by a chi-squared test" %
 opts.select_chi2)
 t0 = time()
```

```

ch2 = SelectKBest(chi2, k=opts.select_chi2)
X_train = ch2.fit_transform(X_train, y_train)
X_test = ch2.transform(X_test)
if feature_names:
 # keep selected feature names
 feature_names = [feature_names[i] for i
 in ch2.get_support(indices=True)]
print("done in %fs" % (time() - t0))
print()

if feature_names:
 feature_names = np.asarray(feature_names)

def trim(s):
 """Trim string to fit on terminal (assuming 80-column display)"""
 return s if len(s) <= 80 else s[:77] + "..."

Benchmark classifiers
Benchmark classifiers
def benchmark(clf):
 print('_' * 80)
 print("Training: ")
 print(clf)
 t0 = time()
 clf.fit(X_train, y_train)
 train_time = time() - t0
 print("train time: %0.3fs" % train_time)

 t0 = time()
 pred = clf.predict(X_test)
 test_time = time() - t0
 print("test time: %0.3fs" % test_time)

 score = metrics.accuracy_score(y_test, pred)
 print("accuracy: %0.3f" % score)

 if hasattr(clf, 'coef_'):
 print("dimensionality: %d" % clf.coef_.shape[1])
 print("density: %f" % density(clf.coef_))

 if opts.print_top10 and feature_names is not None:
 print("top 10 keywords per class:")
 for i, label in enumerate(target_names):
 top10 = np.argsort(clf.coef_[i])[-10:]
 print(trim("%s: %s" % (label, " ".join(feature_names[top10]))))
 print()

 if opts.print_report:
 print("classification report:")
 print(metrics.classification_report(y_test, pred,
 target_names=target_names))

 if opts.print_cm:
 print("confusion matrix:")
 print(metrics.confusion_matrix(y_test, pred))

```

```
print()
clf_descr = str(clf).split('(')[0]
return clf_descr, score, train_time, test_time

results = []
for clf, name in (
 (RidgeClassifier(tol=1e-2, solver="sag"), "Ridge Classifier"),
 (Perceptron(max_iter=50, tol=1e-3), "Perceptron"),
 (PassiveAggressiveClassifier(max_iter=50, tol=1e-3),
 "Passive-Aggressive"),
 (KNeighborsClassifier(n_neighbors=10), "kNN"),
 (RandomForestClassifier(n_estimators=100), "Random forest")):
 print('=' * 80)
 print(name)
 results.append(benchmark(clf))

for penalty in ["l2", "l1"]:
 print('=' * 80)
 print("%s penalty" % penalty.upper())
 # Train Liblinear model
 results.append(benchmark(LinearSVC(penalty=penalty, dual=False,
 tol=1e-3)))

 # Train SGD model
 results.append(benchmark(SGDClassifier(alpha=.0001, max_iter=50,
 penalty=penalty)))

Train SGD with Elastic Net penalty
print('=' * 80)
print("Elastic-Net penalty")
results.append(benchmark(SGDClassifier(alpha=.0001, max_iter=50,
 penalty="elasticnet")))

Train NearestCentroid without threshold
print('=' * 80)
print("NearestCentroid (aka Rocchio classifier)")
results.append(benchmark(NearestCentroid()))

Train sparse Naive Bayes classifiers
print('=' * 80)
print("Naive Bayes")
results.append(benchmark(MultinomialNB(alpha=.01)))
results.append(benchmark(BernoulliNB(alpha=.01)))
results.append(benchmark(ComplementNB(alpha=.1)))

print('=' * 80)
print("LinearSVC with L1-based feature selection")
The smaller C, the stronger the regularization.
The more regularization, the more sparsity.
results.append(benchmark(Pipeline([
 ('feature_selection', SelectFromModel(LinearSVC(penalty="l1", dual=False,
 tol=1e-3))),
 ('classification', LinearSVC(penalty="l2"))])))

make some plots

indices = np.arange(len(results))
```

```

results = [[x[i] for x in results] for i in range(4)]

clf_names, score, training_time, test_time = results
training_time = np.array(training_time) / np.max(training_time)
test_time = np.array(test_time) / np.max(test_time)

plt.figure(figsize=(12, 8))
plt.title("Score")
plt.barh(indices, score, .2, label="score", color='navy')
plt.barh(indices + .3, training_time, .2, label="training time",
 color='c')
plt.barh(indices + .6, test_time, .2, label="test time", color='darkorange')
plt.yticks(())
plt.legend(loc='best')
plt.subplots_adjust(left=.25)
plt.subplots_adjust(top=.95)
plt.subplots_adjust(bottom=.05)

for i, c in zip(indices, clf_names):
 plt.text(-.3, i, c)

plt.show()

```

**Total running time of the script:** ( 0 minutes 4.728 seconds)

## 5.29 Decision Trees

Examples concerning the `sklearn.tree` module.

---

**Note:** Click [here](#) to download the full example code

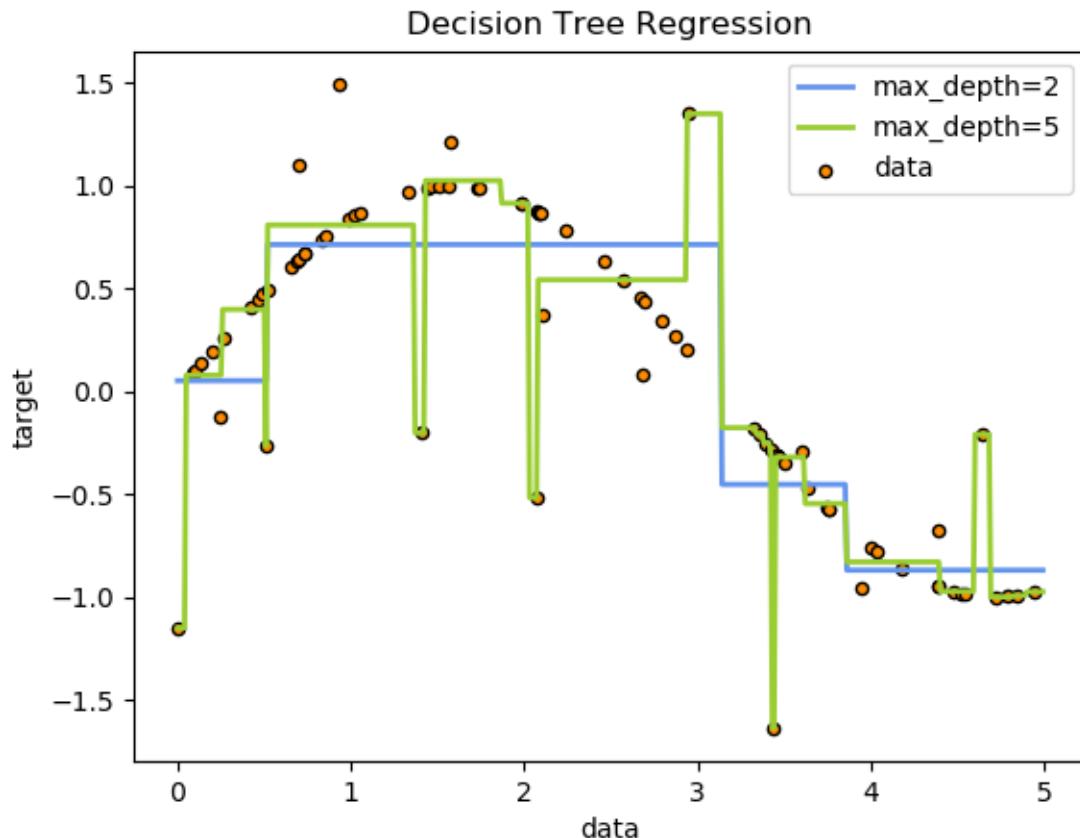
---

### 5.29.1 Decision Tree Regression

A 1D regression with decision tree.

The `decision trees` is used to fit a sine curve with addition noisy observation. As a result, it learns local linear regressions approximating the sine curve.

We can see that if the maximum depth of the tree (controlled by the `max_depth` parameter) is set too high, the decision trees learn too fine details of the training data and learn from the noise, i.e. they overfit.



```

print(__doc__)

Import the necessary modules and libraries
import numpy as np
from sklearn.tree import DecisionTreeRegressor
import matplotlib.pyplot as plt

Create a random dataset
rng = np.random.RandomState(1)
X = np.sort(5 * rng.rand(80, 1), axis=0)
y = np.sin(X).ravel()
y[::5] += 3 * (0.5 - rng.rand(16))

Fit regression model
regr_1 = DecisionTreeRegressor(max_depth=2)
regr_2 = DecisionTreeRegressor(max_depth=5)
regr_1.fit(X, y)
regr_2.fit(X, y)

Predict
X_test = np.arange(0.0, 5.0, 0.01)[:, np.newaxis]
y_1 = regr_1.predict(X_test)
y_2 = regr_2.predict(X_test)

Plot the results
plt.figure()

```

```
plt.scatter(X, y, s=20, edgecolor="black",
 c="darkorange", label="data")
plt.plot(X_test, y_1, color="cornflowerblue",
 label="max_depth=2", linewidth=2)
plt.plot(X_test, y_2, color="yellowgreen", label="max_depth=5", linewidth=2)
plt.xlabel("data")
plt.ylabel("target")
plt.title("Decision Tree Regression")
plt.legend()
plt.show()
```

**Total running time of the script:** ( 0 minutes 0.124 seconds)

---

**Note:** Click [here](#) to download the full example code

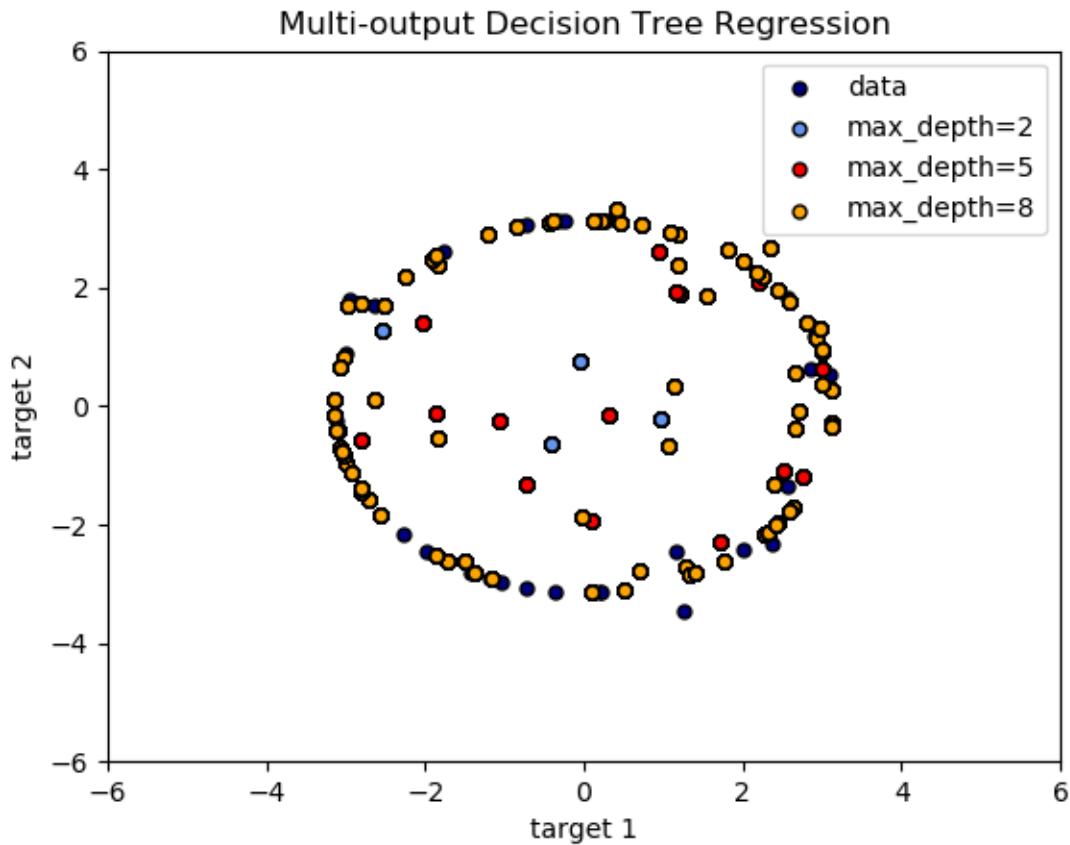
---

## 5.29.2 Multi-output Decision Tree Regression

An example to illustrate multi-output regression with decision tree.

The [decision trees](#) is used to predict simultaneously the noisy x and y observations of a circle given a single underlying feature. As a result, it learns local linear regressions approximating the circle.

We can see that if the maximum depth of the tree (controlled by the `max_depth` parameter) is set too high, the decision trees learn too fine details of the training data and learn from the noise, i.e. they overfit.



```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor

Create a random dataset
rng = np.random.RandomState(1)
X = np.sort(200 * rng.rand(100, 1) - 100, axis=0)
y = np.array([np.pi * np.sin(X).ravel(), np.pi * np.cos(X).ravel()]).T
y[:, ::5] += (0.5 - rng.rand(20, 2))

Fit regression model
regr_1 = DecisionTreeRegressor(max_depth=2)
regr_2 = DecisionTreeRegressor(max_depth=5)
regr_3 = DecisionTreeRegressor(max_depth=8)
regr_1.fit(X, y)
regr_2.fit(X, y)
regr_3.fit(X, y)

Predict
X_test = np.arange(-100.0, 100.0, 0.01)[:, np.newaxis]
y_1 = regr_1.predict(X_test)
y_2 = regr_2.predict(X_test)
y_3 = regr_3.predict(X_test)

```

```
Plot the results
plt.figure()
s = 25
plt.scatter(y[:, 0], y[:, 1], c="navy", s=s,
 edgecolor="black", label="data")
plt.scatter(y_1[:, 0], y_1[:, 1], c="cornflowerblue", s=s,
 edgecolor="black", label="max_depth=2")
plt.scatter(y_2[:, 0], y_2[:, 1], c="red", s=s,
 edgecolor="black", label="max_depth=5")
plt.scatter(y_3[:, 0], y_3[:, 1], c="orange", s=s,
 edgecolor="black", label="max_depth=8")
plt.xlim([-6, 6])
plt.ylim([-6, 6])
plt.xlabel("target 1")
plt.ylabel("target 2")
plt.title("Multi-output Decision Tree Regression")
plt.legend(loc="best")
plt.show()
```

**Total running time of the script:** ( 0 minutes 0.109 seconds)

---

**Note:** Click [here](#) to download the full example code

---

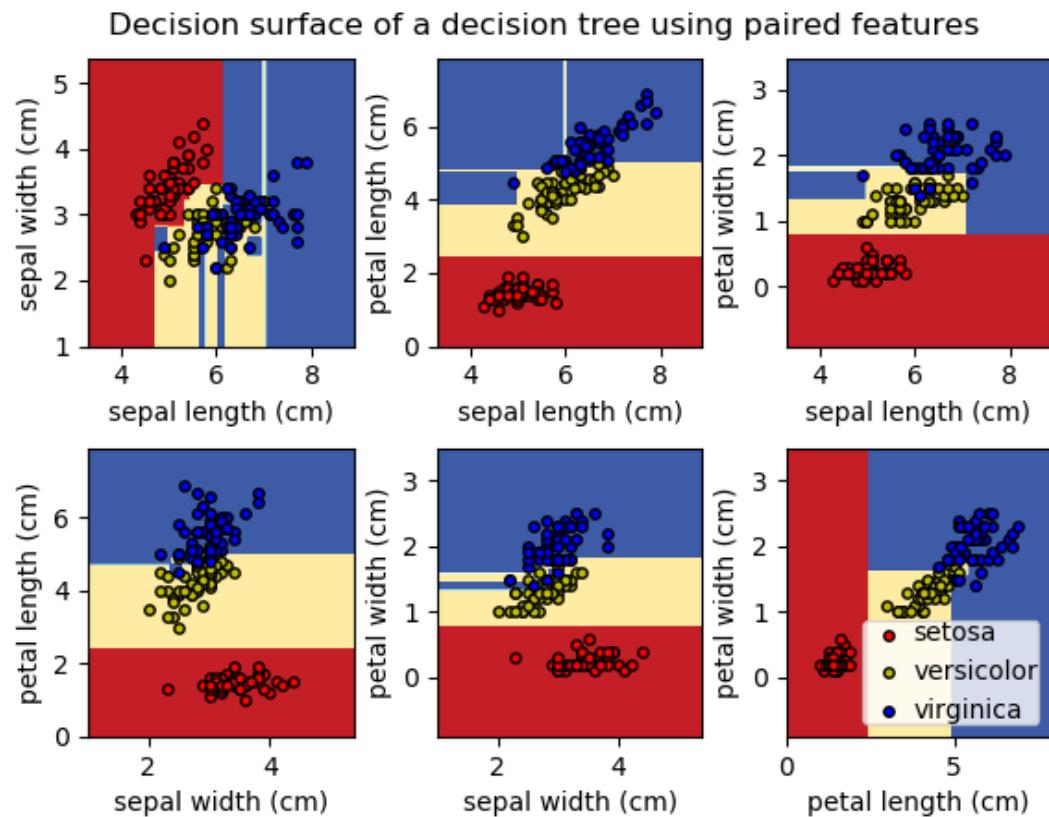
### 5.29.3 Plot the decision surface of a decision tree on the iris dataset

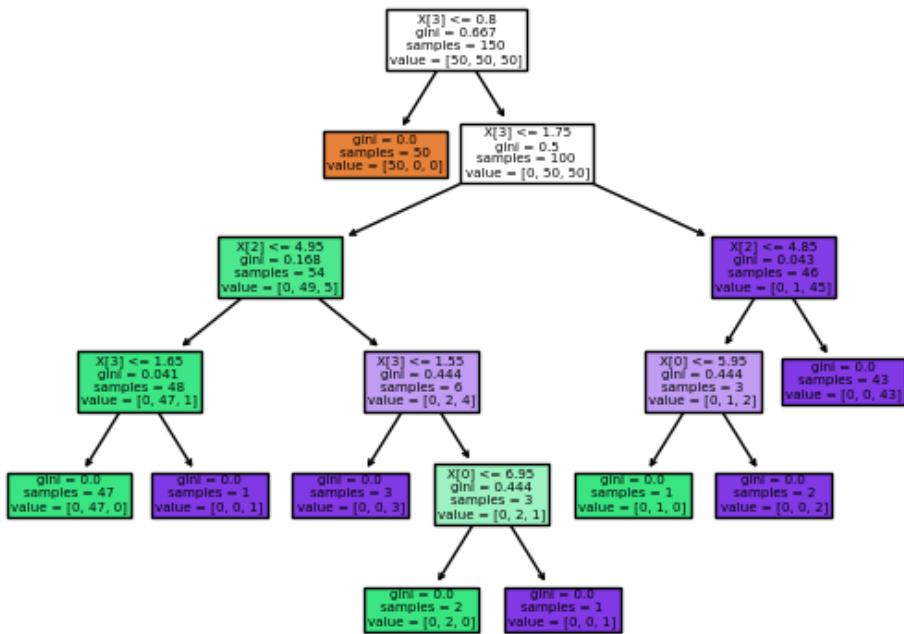
Plot the decision surface of a decision tree trained on pairs of features of the iris dataset.

See [decision tree](#) for more information on the estimator.

For each pair of iris features, the decision tree learns decision boundaries made of combinations of simple thresholding rules inferred from the training samples.

We also show the tree structure of a model built on all of the features.





```

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree

Parameters
n_classes = 3
plot_colors = "ryb"
plot_step = 0.02

Load data
iris = load_iris()

for pairidx, pair in enumerate([[0, 1], [0, 2], [0, 3],
 [1, 2], [1, 3], [2, 3]]):
 # We only take the two corresponding features
 X = iris.data[:, pair]
 y = iris.target

 # Train
 clf = DecisionTreeClassifier().fit(X, y)

 # Plot the decision boundary
 plt.subplot(2, 3, pairidx + 1)

```

```
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
 np.arange(y_min, y_max, plot_step))
plt.tight_layout(h_pad=0.5, w_pad=0.5, pad=2.5)

Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
cs = plt.contourf(xx, yy, Z, cmap=plt.cm.RdYlBu)

plt.xlabel(iris.feature_names[pair[0]])
plt.ylabel(iris.feature_names[pair[1]])

Plot the training points
for i, color in zip(range(n_classes), plot_colors):
 idx = np.where(y == i)
 plt.scatter(X[idx, 0], X[idx, 1], c=color, label=iris.target_names[i],
 cmap=plt.cm.RdYlBu, edgecolor='black', s=15)

plt.suptitle("Decision surface of a decision tree using paired features")
plt.legend(loc='lower right', borderpad=0, handletextpad=0)
plt.axis("tight")

plt.figure()
clf = DecisionTreeClassifier().fit(iris.data, iris.target)
plot_tree(clf, filled=True)
plt.show()
```

**Total running time of the script:** ( 0 minutes 0.934 seconds)

---

**Note:** Click [here](#) to download the full example code

---

## 5.29.4 Understanding the decision tree structure

The decision tree structure can be analysed to gain further insight on the relation between the features and the target to predict. In this example, we show how to retrieve:

- the binary tree structure;
- the depth of each node and whether or not it's a leaf;
- the nodes that were reached by a sample using the `decision_path` method;
- the leaf that was reached by a sample using the `apply` method;
- the rules that were used to predict a sample;
- the decision path shared by a group of samples.

Out:

```
The binary tree structure has 5 nodes and has the following tree structure:
node=0 test node: go to node 1 if X[:, 3] <= 0.800000011920929 else to node 2.
 node=1 leaf node.
 node=2 test node: go to node 3 if X[:, 2] <= 4.950000047683716 else to node 4.
 node=3 leaf node.
```

```

node=4 leaf node.

Rules used to predict sample 0:
decision id node 0 : (X_test[0, 3] (= 2.4) > 0.800000011920929)
decision id node 2 : (X_test[0, 2] (= 5.1) > 4.950000047683716)

The following samples [0, 1] share the node [0 2] in the tree
It is 40.0 % of all nodes.

```

```

import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

estimator = DecisionTreeClassifier(max_leaf_nodes=3, random_state=0)
estimator.fit(X_train, y_train)

The decision estimator has an attribute called tree_ which stores the entire
tree structure and allows access to low level attributes. The binary tree
tree_ is represented as a number of parallel arrays. The i-th element of each
array holds information about the node `i`. Node 0 is the tree's root. NOTE:
Some of the arrays only apply to either leaves or split nodes, resp. In this
case the values of nodes of the other type are arbitrary!
#
Among those arrays, we have:
- left_child, id of the left child of the node
- right_child, id of the right child of the node
- feature, feature used for splitting the node
- threshold, threshold value at the node
#
Using those arrays, we can parse the tree structure:

n_nodes = estimator.tree_.node_count
children_left = estimator.tree_.children_left
children_right = estimator.tree_.children_right
feature = estimator.tree_.feature
threshold = estimator.tree_.threshold

The tree structure can be traversed to compute various properties such
as the depth of each node and whether or not it is a leaf.
node_depth = np.zeros(shape=n_nodes, dtype=np.int64)
is_leaves = np.zeros(shape=n_nodes, dtype=bool)
stack = [(0, -1)] # seed is the root node id and its parent depth
while len(stack) > 0:
 node_id, parent_depth = stack.pop()

```

```

node_depth[node_id] = parent_depth + 1

If we have a test node
if (children_left[node_id] != children_right[node_id]):
 stack.append((children_left[node_id], parent_depth + 1))
 stack.append((children_right[node_id], parent_depth + 1))
else:
 is_leaves[node_id] = True

print("The binary tree structure has %s nodes and has "
 "the following tree structure:"
 % n_nodes)
for i in range(n_nodes):
 if is_leaves[i]:
 print("%snode=%s leaf node." % (node_depth[i] * "\t", i))
 else:
 print("%snode=%s test node: go to node %s if X[:, %s] <= %s else to "
 "node %s."
 % (node_depth[i] * "\t",
 i,
 children_left[i],
 feature[i],
 threshold[i],
 children_right[i],
))
print()

First let's retrieve the decision path of each sample. The decision_path
method allows to retrieve the node indicator functions. A non zero element of
indicator matrix at the position (i, j) indicates that the sample i goes
through the node j.

node_indicator = estimator.decision_path(X_test)

Similarly, we can also have the leaves ids reached by each sample.

leave_id = estimator.apply(X_test)

Now, it's possible to get the tests that were used to predict a sample or
a group of samples. First, let's make it for the sample.

sample_id = 0
node_index = node_indicator.indices[node_indicator.indptr[sample_id]:node_indicator.indptr[sample_id + 1]]

print('Rules used to predict sample %s: ' % sample_id)
for node_id in node_index:
 if leave_id[sample_id] == node_id:
 continue

 if (X_test[sample_id, feature[node_id]] <= threshold[node_id]):
 threshold_sign = "<="
 else:
 threshold_sign = ">"

 print("decision id node %s : (X_test[%s, %s] (= %s) %s %s)" %
 % (node_id,
 sample_id,

```

```
 feature[node_id],
 X_test[sample_id, feature[node_id]],
 threshold_sign,
 threshold[node_id]))\n\n# For a group of samples, we have the following common node.\nsample_ids = [0, 1]\ncommon_nodes = (node_indicator.toarray() [sample_ids].sum(axis=0) ==\n len(sample_ids))\n\ncommon_node_id = np.arange(n_nodes) [common_nodes]\n\nprint("\nThe following samples %s share the node %s in the tree"\n % (sample_ids, common_node_id))\nprint("It is %s %% of all nodes." % (100 * len(common_node_id) / n_nodes,))
```

**Total running time of the script:** ( 0 minutes 0.003 seconds)

