Download Chapter pdf

Download Chapter notebook (ipynb)

Mandatory Lesson Feedback Survey

- How to deal with complex classification problems?
- Why is it important to use different classification algorithms?
- What is the best way to find the optimal classifier?
- How can we avoid over-fitting of data?
- How do we evaluate the performance of a classifier?

- Understanding complex training and testing data.
- Comparison of different model classes.
- Explaining the stratified shuffle split.
- Evaluation of classification - the ROC and AUC curves.

Remarks

1. From now on the code will become more complex. When copied, the code should run without errors with the given data sets. (Please report any errors thrown when running the code without modifications).
2. Make a copy of the notebook and start experimenting by modifying part of the code and comparing the outcome. Modifying existing code is one of the successful strategies when learning to programme as a non-programmer.
3. The first resource to consult when facing bugs are the official documentations, be it Python, Numpy, SciKit Learn or other.
4. If you formulate a problem adequately, often there may be good answers on Stack Overflow.
5. Sometimes, simply copying and pasting an error message into the search engine can point you to the solution.

Import functions

```python
from numpy import mgrid, linspace, c_, arange, mean, array
from numpy.random import uniform, seed

from mpl_toolkits import mplot3d
from matplotlib.pyplot import subplots, axes, scatter, xticks, show

from sklearn.datasets import make_circles
```

Challenge

We would like to test several machine learning models' ability to deal with a complicated task. A complicated task is one where the topology of the labelled data is not trivially separable into classes by (hyper)planes, e.g. by a straight line in a scatter plot.

Our example is one class of data organised in a doughnut shape and the other class contained within the first doughnut forming a doughnut-within-a-doughnut.

Here is the function code to create these data, followed by a function call to produce a figure.

```python
def make_torus_3D(n_samples=100, shuffle=True, noise=None, random_state=None,
                 factor=.8):
    """Make a large torus containing a smaller torus in 3d.

    A toy dataset to visualize clustering and classification
    algorithms.

    Read more in the :ref:`User Guide <sample_generators>`.

    Parameters
    ----------
    n_samples : int, optional (default=100)
        The total number of points generated. If odd, the inner circle will
        have one point more than the outer circle.

    shuffle : bool, optional (default=True)
        Whether to shuffle the samples.

    noise : double or None (default=None)
        Standard deviation of Gaussian noise added to the data.

    random_state : int, RandomState instance or None (default)
        Determines random number generation for dataset shuffling and noise.
        Pass an int for reproducible output across multiple function calls.
        See :term:`Glossary <random_state>`.

    factor : 0 < double < 1 (default=.8)
        Scale factor between inner and outer circle.

    Returns
    -------
    X : array of shape [n_samples, 2]
        The generated samples.

    y : array of shape [n_samples]
```

```
        The integer labels (0 or 1) for class membership of each sample.
    """
    from numpy import pi, linspace, cos, sin, append, ones, zeros, hstack, vstack, intp
    from sklearn.utils import check_random_state, shuffle

    if factor >= 1 or factor < 0:
        raise ValueError("'factor' has to be between 0 and 1.")

    n_samples_out = n_samples // 2
    n_samples_in = n_samples - n_samples_out

    co, ao, ci, ai = 3, 1, 3.6, 0.2
    generator = check_random_state(random_state)
    # to not have the first point = last point, we set endpoint=False
    linspace_out = linspace(0, 2 * pi, n_samples_out, endpoint=False)
    linspace_in  = linspace(0, 2 * pi, n_samples_in,  endpoint=False)
    outer_circ_x = (co+ao*cos(linspace_out)) * cos(linspace_out*61.1)
    outer_circ_y = (co+ao*cos(linspace_out)) * sin(linspace_out*61.1)
    outer_circ_z =    ao*sin(linspace_out)

    inner_circ_x = (ci+ai*cos(linspace_in)) * cos(linspace_in*61.1)* factor
    inner_circ_y = (ci+ai*cos(linspace_in)) * sin(linspace_in*61.1) * factor
    inner_circ_z =    ai*sin(linspace_in) * factor

    X = vstack([append(outer_circ_x, inner_circ_x),
                append(outer_circ_y, inner_circ_y),
                append(outer_circ_z, inner_circ_z)]).T

    y = hstack([zeros(n_samples_out, dtype=intp),
                   ones(n_samples_in, dtype=intp)])


    if shuffle:
        X, y = shuffle(X, y, random_state=generator)

    if noise is not None:
        X += generator.normal(scale=noise, size=X.shape)

    return X, y
```

```
RANDOM_STATE  = 12345
seed(RANDOM_STATE)

X, y = make_torus_3D(n_samples=2000, factor=.9, noise=.001, random_state=RANDOM_STATE)

feature_1, feature_2, feature_3 = 0, 1, 2
```

```
ft_min, ft_max = X.min(), X.max()

fig, ax = subplots(figsize=(12, 9))

ax = axes(projection="3d")

im = ax.scatter3D(X[:, feature_1], X[:, feature_2], X[:, feature_3],
    marker='o', s=20, c=y, cmap='bwr');

ax.set_xlabel('Feature 1')
ax.set_ylabel('Feature 2')
ax.set_zlabel('Feature 3')

# Angles to pick the perspective
ax.view_init(30, 50);

show()
```

The challenge here is that the only way to separate the data of the two labels from each other is to find a separating border that lies between the blue and the red doughnut (mathematically: torus) and itself is a torus, i.e. a complex topology. Similarly, one can test to separate one class of data that lie on the surface of a sphere and then have data on another sphere embedded within it. Typically, it is unknown what type of high-dimensional topologies is present in biological data. As such it is not clear at the outset which classification strategy will work best. Let us start with a simpler example.

Traing a variety of machine learning models

`SciKit Learn` provides the means to generate practice datasets with specific qualities. In this section, we will use the `make_circles` function. (see the documentations):

Circular Test Data

```
RANDOM_STATE  = 1234
seed(RANDOM_STATE)

X, y = make_circles(n_samples=500, factor=0.3, noise=.05, random_state=
    RANDOM_STATE)

feature_1, feature_2 = 0, 1
ft_min, ft_max = X.min(), X.max()

print('Shape of X:', X.shape)
```

```
Shape of X: (500, 2)
```

```
fig, ax = subplots(figsize=(10, 5), nrows=1, ncols=2)
ax[0].scatter(X[:, feature_1], X[:, feature_2], c=y, s=4, cmap='bwr');
ax[1].hist(X);

show()
```

The function yields only two features. The reason is that with two features we can visualise the complete state space in the two dimensional scatter plot. The data of both labels are organised along a ring. There is a certain amount of randomness added to create data distributed normally around the ring.

The tricky thing about such a data distribution is that in a standard view of the data, the histogram, the clear state space organisation is not visible. There are e.g. no two distinct mean values of the distributions. Also, while the two features are clearly dependent on each other (as seen in the scatter plot), it is not possible to regress one with the other by means of fits of the type y = f(x).

We will now use different classes of machine learning models to fit to these labelled data.

Classification Algorithms

Different classification algorithms approach problems differently. Let us name the algorithms in `SciKit Learn`.

`SciKit Learn` provides the following algorithms for classification problems:

- Ensemble: Averaging:
  - Random Forest
  - Extra Tree
  - Isolation Forest
  - Bagging
  - Voting
- Boosting:
  - Gradient Boosting
  - AdaBoost
- Decision Trees:
  - Decision Tree
  - Extra Tree
- Nearest Neighbour:
  - K Nearest Neighbour

    - Radius Neighbours
    - Nearest Centroid

- Support Vector Machine:
    - with non-linear kernel:
        - Radial Basis Function (RBF) Polynomial
        - Sigmoid
    - with linear kernel:
        - Linear kernel
    - parametrised with non-linear kernel:
        - Nu-Support Vector Classification
- Neural Networks:
    - Multi-layer Perceptron
    - Gaussian:
        - Gaussian Process
    - Linear Models:
        - Logistic Regression
        - Passive Aggressive
        - Ridge
        - Linear classifiers with Stochastic Gradient Descent
- Baysian:
    - Bernoulli
    - Multinomial
    - Complement

Some of these algorithms require a more in-depth understanding of how they work. To that end, we would only review the performance the ones that are easier to implement and adjust.

AdaBoost

The AdaBoost algorithm is special in that it does not work on its own; instead, it complements another ensemble algorithm (e.g. Random Forest) and *boosts* its performance by weighing the training data through a boosting algorithm. Note that boosting the performance does not necessarily translate into a better fit. This is because boosting algorithms are generally robust against over-fitting, meaning that they always try to produce generalisable models.

Seeding

Most machine learning algorithms rely on random number generation to produce results. Therefore, one simple, but important adjustment is to `seed` the number generator, and thereby making our comparisons more consistent; i.e. ensure that all models use the same set of random numbers. Almost all SciKit Learn models take an argument called `random_state`, which takes an integer number and uses it to seed the random number generator.

Training and Testing

Here is code to import the classifiers from SciKit Learn, fit them to the training data and predict the (complete) state space. The result is plotted below.

```python
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier, GradientBoostingClassifier, AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC, LinearSVC
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier

classifiers = {
    'Random Forest': RandomForestClassifier(random_state=RANDOM_STATE),
    'AdaBoost (Random Forest)': AdaBoostClassifier(RandomForestClassifier(random_state=RANDOM_STATE)),
    'Extra Trees': ExtraTreesClassifier(random_state=RANDOM_STATE),
    'AdaBoost (Extra Tree)': AdaBoostClassifier(ExtraTreesClassifier(random_state=RANDOM_STATE)),
    'Decision Tree': DecisionTreeClassifier(random_state=RANDOM_STATE),
    'SVC (RBF)': SVC(random_state=RANDOM_STATE),
    'SVC (Linear)': LinearSVC(random_state=RANDOM_STATE),
    'Multi-layer Perceptron': MLPClassifier(max_iter=5000, random_state=RANDOM_STATE)
}
```

```python
ft_min, ft_max = -1.5, 1.5

# Constructing (2 grids x 300 rows x 300 cols):
grid_1, grid_2 = mgrid[ft_min:ft_max:.01, ft_min:ft_max:.01]

# We need only the shape for one of the grids (i.e. 300 x  300):
grid_shape = grid_1.shape

# state space grid for testing
new_obs = c_[grid_1.ravel(), grid_2.ravel()]
```

```python
contour_levels = linspace(0, 1, 6)

fig, all_axes = subplots(figsize=[15, 5], ncols=4, nrows=2, sharey=True, sharex=True)
```

```

for ax, (name, clf) in zip(all_axes.ravel(), classifiers.items()):
    clf.fit(X, y)
    y_pred = clf.predict(new_obs)
    y_pred_grid = y_pred.reshape(grid_shape)

    ax.scatter(X[:, feature_1], X[:, feature_2], c=y, s=1, cmap='bwr_r')
    ax.contourf(grid_1, grid_2, y_pred_grid, cmap='gray_r', alpha=.2, levels=contour_levels)
    ax.set_ylim(ft_min, ft_max)
    ax.set_xlim(ft_min, ft_max)
    ax.set_yticks([-1.5, 0, 1.5])
    ax.set_xticks([-1.5, 0, 1.5])
    ax.set_title(name, fontsize=10);

show()
```

Seven of the eight classifiers were able to separate the inner data set from the outer data set successfully. The main difference is that some algorithms ended up with a more rectangular shape of the boundary whereas the others found a more circular form which reflects the original data distribution more closely. One classifier simply fails: SVC (linear): it tries to fit a straight line to separate the classes which in this case is impossible.

The Train-Test Split

We will now modify our workflow to avoid the need to create separate testing data (the typical situation when dealing with recorded data). For this we start with a data set of n labelled samples. Of these n samples, a certain percentage is used for training (using the provided labels) and the rest for testing (withholding the labels). The testing data then do not need to be prepared separately.

The function we use is `train_test_split` from SciKit Learn. A nice feauture of this function is that it tries to preserve the ratio of labels in the split. E.g. if the data contain 70% of `True` and 30 % of `False` labels, the algorithm tries to preserve this ratio in the split as good as possible: around 70% of the training data and of the testing data will have the `True` label.

```
from sklearn.model_selection import train_test_split

X, y = make_circles(n_samples=1000, factor=0.3, noise=.05, random_state=RANDOM_STATE)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.3, random_state=RANDOM_STATE, shuffle=True)

print(X_train.shape, X_test.shape)
```

```
(700, 2) (300, 2)
```

Here is an illustration of the two sets of data. The splitting into testing and training data is done randomly. Picking test data randomly is particularly important for real data as it helps to reduce potential bias in the recording order.

```
fig, ax = subplots(figsize=(7, 6), ncols=2, nrows=2, sharex=True)

ax[0, 0].scatter(X_train[:, feature_1], X_train[:, feature_2], c=
    y_train, s=4, cmap='bwr')
ax[0, 1].scatter(X_test[:, feature_1], X_test[:, feature_2], c=y_test,
    s=4, cmap='bwr')

ax[1, 0].hist(X_train)
ax[1, 1].hist(X_test)
ax[0, 0].set_title('Training data')
ax[0, 1].set_title('Test data')

ax[0, 0].set_ylim(ft_min, ft_max)
ax[0, 1].set_ylim(ft_min, ft_max)
ax[1, 0].set_ylim(0, 100)
ax[1, 1].set_ylim(0, 100);

show()
```

Now we can repeat the training with this split dataset using eight types of models as above.

Before we do so, let us introduce a **scoring** of the performance. The method `.score` takes as input arguments the testing samples and their true labels. It then uses the model predictions to calculate the fraction of labels in the testing data that were predicted correctly.

There are different techniques to evaluate a the performance of a model, but the `.score` method provides a quick, simple, and handy way to assess a model. As far as classification algorithms in SciKit Learn are concerned, the method usually produces the **mean accuracy**, which is between 0 and 1; and the higher the score, the better the fit.

```
fig, all_axes = subplots(figsize=[15, 5], ncols=4, nrows=2, sharey=True
    , sharex=True)

for ax, (name, clf) in zip(all_axes.ravel(), classifiers.items()):
    # Training the model using training data:
    clf.fit(X_train, y_train)

    y_pred = clf.predict(new_obs)
    y_pred_grid = y_pred.reshape(grid_shape)

    # Evaluating the score using test data:
    score = clf.score(X_test, y_test)
```

```python

    # Scattering the test data only:
    ax.scatter(X_test[:, feature_1], X_test[:, feature_2], c=y_test, s=4, cmap='bwr', marker='.')

    ax.contourf(grid_1, grid_2, y_pred_grid, cmap='gray_r', alpha=.2, levels=contour_levels)
#    ax.contourf(grid[0], grid[1], y_pred_grid, cmap='gray_r', alpha=.2, levels=contour_levels)

    ax.set_ylim(ft_min, ft_max)
    ax.set_xlim(ft_min, ft_max)
    ax.set_yticks([-1.5, 0, 1.5])
    ax.set_xticks([-1.5, 0, 1.5])

    label = '{} - Score: {:.2f}'.format(name, score)
    ax.set_title(label , fontsize=10);

show()
```

Here, we only plotted the test data, those that were classified based on the trained model. The gray area shows the result of the classification: within the gray area the prediction is 1 (the red samples) and outside it is 0 (the blue samples). The result is that testing data are classified correctly in all but one of the classifiers, so their performance is 1, or 100 %. This is excellent because it demonstrates that most classifiers are able to deal with embedded topologies.

Let us now repeat the procedure with a higher level of noise to make the task more complicated.

```python
X, y = make_circles(n_samples=1000, factor=.5, noise=.3, random_state=RANDOM_STATE)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.3, random_state=RANDOM_STATE, shuffle=True)

fig, ax = subplots(figsize=(7, 6), ncols=2, nrows=2, sharex=True)

ax[0, 0].scatter(X_train[:, feature_1], X_train[:, feature_2], c=y_train, s=4, cmap='bwr')
ax[0, 1].scatter(X_test[:, feature_1], X_test[:, feature_2], c=y_test, s=4, cmap='bwr')


ax[1, 0].hist(X_train)
ax[1, 1].hist(X_test)
ax[0, 0].set_title('Training data')
ax[0, 1].set_title('Test data')

ax[0, 0].set_ylim(-3, 3)
ax[0, 1].set_ylim(-3, 3)
ax[1, 0].set_ylim(0, 200)
```

```python
ax[1, 1].set_ylim(0, 200);

show()
```

```python
fig, all_axes = subplots(figsize=[15, 5], ncols=4, nrows=2, sharey=True, sharex=True)

for ax, (name, clf) in zip(all_axes.ravel(), classifiers.items()):
    # Training the model using training data:
    clf.fit(X_train, y_train)

    y_pred = clf.predict(new_obs)
    y_pred_grid = y_pred.reshape(grid_shape)

    # Evaluating the score using test data:
    score = clf.score(X_test, y_test)

    # Scattering the test data only:
    ax.scatter(X_test[:, feature_1], X_test[:, feature_2], c=y_test, s=4, cmap='bwr', marker='.')

    ax.contourf(grid_1, grid_2, y_pred_grid, cmap='gray_r', alpha=.2, levels=contour_levels)

    ax.set_ylim(ft_min, ft_max)
    ax.set_xlim(ft_min, ft_max)
    ax.set_yticks([-1.5, 0, 1.5])
    ax.set_xticks([-1.5, 0, 1.5])

    label = '{} - Score: {:.2f}'.format(name, score)
    ax.set_title(label , fontsize=10);

show()
```

Now the data are mixed in the plane and there is no simple way to separate the two classes. We can see in the plots how the algorithms try to cope with their different strategies. One thing that is immediately obvious is that the fitting patterns are different. Particularly, we can see the fragmented outcome of the *decision tree* classifier and the smooth elliptic area found by the *support vector classifier (SVC)* with radial basis functions (RBF) and the neural network (MLP). On a closer look, you may also notice that with ensemble methods in the upper row, the patterns are somewhat disorganised. This is due to the way ensemble methods work: they sample the data randomly and then class them into different categories based on their labels.

If the prediction was made by chance (throwing a dice), one would expect a 50 % score. Thus, the example also shows that the performance depends on the type of problem and that this testing helps to find the optimal classifier.

Never expose the test data

Testing a model on data that is used in training is a methodological mistake. It is therefore vital that the test data is **never, ever** used for training a model at any stage. This is one of the most fundamental principles of machine learning, and its importance cannot be exaggerated. There are numerous examples of people making this mistake one way or another, especially where multiple classification algorithms are used to address a problem.

The Stratified Shuffle Split

One potential bias arises when we try to improve the performance of our models through the change of the so-called **hyperparameters** (instead of using the default parameters as we did so far). We will always receive the optimal output given **the specific test data chosen**. This may lead to overfitting the model on the chosen training and testing data. This can be avoided by choosing different splits into testing and training data and repeating the fit procedure. Doing different splits while preserving the fraction of labels of each class in the original data, the method is called the **stratified shuffle split**.

We first need to import and instantiate the splitter. We set `n_splits` to determine the numober of different splits. `test_size` lets us determine what fraction of samples is used for the testing data.

```python
from sklearn.model_selection import StratifiedShuffleSplit

sss = StratifiedShuffleSplit(random_state=RANDOM_STATE, n_splits=10, test_size=0.3)
```

Let us look at the different splits obtained:

```python
fig, ax = subplots(figsize=[10, 5])

n_splits = sss.n_splits
split_data_indices = sss.split(X=X, y=y)

for index, (tr, tt) in enumerate(split_data_indices):
    indices = X[:, feature_1].copy()
    indices[tt] = 1
    indices[tr] = 0

    # Visualize the results
    x_axis = arange(indices.size)
    y_axis = [index + .5] * indices.size
    ax.scatter(x_axis, y_axis, c=indices, marker='_', lw=10, cmap='coolwarm', vmin=-.2, vmax=1.2)

# Plot the data classes and groups at the end
class_y = [index + 1.5] * indices.size
```

```
ax.scatter(x_axis, class_y, c=y, marker='_', lw=10, cmap='coolwarm')

# Formatting
ylabels = list(range(n_splits))
ylabels.extend(['Data'])

ax.set_yticks(arange(n_splits + 1) + .5)
ax.set_yticklabels(ylabels)
ax.set_xlabel('Sample index')
ax.set_ylabel('SSS iteration');

show()
```

By choosing n_splits=10, we obtained ten different splits that have similarly distributed train and test data subsets from the original data. The fraction of the data set aside for testing is 30 %. The different splits cover the whole data set evenly.

Let us look at the data in state space to check that the classification task is now a real challenge.

```
fig, ax = subplots(figsize=(8, 8))

for train_index, test_index in sss.split(X, y):
    ax.scatter(X[train_index, 0], X[train_index, 1], c=y[train_index], cmap='Set1', s=30, marker='^', alpha=.5)
    ax.scatter(X[test_index, 0], X[test_index, 1], c=y[test_index], cmap='cool', s=30, alpha=.5, marker='*', label='Test');

show()
```

These are the scatter plots of the training (magenta) and testing (blue) data. Here are their distributions:

```
fig, ax = subplots(figsize=(8, 8))

for train_index, test_index in sss.split(X, y):
    ax.hist(X[train_index], color=['magenta', 'red'], alpha=.5, histtype='step')
    ax.hist(X[test_index], color=['cyan', 'blue'], alpha=.4, histtype='step');

show()
```

The distributions differ in height because less data are in the testing test. Otherwise they are similarly centered and spread. Using a number of realisations (instead of just one) we expect to obtain a more accurate and robust result of the training.

We now train our classifiers on these different splits and obtain the respective scores. They will give a robust measure of the classifier's preformance given the data and avoid potential bias due to the selection of specific test data.

```
X, y = make_circles(n_samples=1000, factor=.3, noise=.4, random_state=
    RANDOM_STATE)

split_data_indices = sss.split(X=X, y=y)

score = list()

for train_index, test_index in sss.split(X, y):
    X_s, y_s = X[train_index, :], y[train_index]
    new_obs_s, y_test_s = X[test_index, :], y[test_index]

    score_clf = list()

    for name, clf in classifiers.items():

        clf.fit(X_s, y_s)
        y_pred = clf.predict(new_obs_s)
        score_clf.append(clf.score(new_obs_s, y_test_s))

    score.append(score_clf)

score_mean = mean(score, axis=0)

bins = arange(len(score_mean))

fig, ax = subplots()

ax.bar(bins, score_mean);
ax.set_xticks(arange(0,8)+0.4)
ax.set_xticklabels(classifiers.keys(), rotation=-70);

show()

print(classifiers.keys())
print('Average scores: ')
print(["{0:0.2f}".format(ind) for ind in score_mean])
```

```
dict_keys(['Random Forest', 'AdaBoost (Random Forest)', 'Extra Trees',
    'AdaBoost (Extra Tree)', 'Decision Tree', 'SVC (RBF)', 'SVC (Linear)
    ', 'Multi-layer Perceptron'])
Average scores:
['0.76', '0.76', '0.75', '0.75', '0.70', '0.79', '0.50', '0.78']
```

The result is the average score for the ten splits performed. All results for the noise-contaminated data are now in the seventies.

This is still good given the quality of the data. It appears that the *decision tree* classifier gives the lowest result for this kind of problem, *SVC* scores highest. We have to keep inmind, however, that we are using the classifiers with their default settings. We will later use variation of the so-called hyperparameters to improve the classification score.

Here we have used a for loop to train and test on each of the different splits of the data. SciKit Learn also contains functions that take the stratified shuffle split as an argument, e.g. `permutation_test_score`.

We have now reached a point where we can trust to have a robust and unbiased outcome of the training. Let us now look at more refined ways to quantify the result.

Evaluation: ROC and AUC

There are various measures that may be used to evaluate the performance of a machine learning model. Such measures look at different characteristics, including the goodness of fit and generalisability of a model. Evaluation measures used with regards to classification models include, but are not limited to:

- Receiver Operation Characteristic (ROC) and Area Under the Curve (AUC) - for binary classifiers.
- Accuracy
- Precision
- Recall

There are many other metrics that, depending on the problem, we may use to evaluate a machine learning model. Please see the official documentations for additional information on these measures and their implementation in SciKit Learn.

The quantities we are going to look at are the **Receiver Operation Characteristic (ROC)** and the **Area Under the Curve (AUC)**.

A receiver operation characteristic, often referred to as the **ROC curve**, is a visualisation of the discrimination threshold in a binary classification model. It illustrates the rate of true positives (TPR) against the rate of false positives (FPR) at different thresholds. The aforementioned rates are essentially defined as:

- True Positive Rate (TPR): the sensitivity of the model
- False Positive Rate (FPR): one minus the specificity of the model

This makes ROC a measure of sensitivity versus specificity.

The area under the ROC curve, often referred to as AUC, reduces the information contained within a ROC curve down to a value between 0 and 1, with 1 being a perfect fit. An AUC value of represents any random guess, and values below demonstrate a performance that's even worse than a lucky guess!

Discussion

`SciKit Learn` includes specialist functions called `roc_curve` and `roc_auc_score` to obtain ROC (FPR and TPR values for visualisation) and AUC respectively. Both functions receive as input arguments the test labels (i.e. `y_test`) and the score (probability) associated with each prediction. We obtain the latter measure using one of the following two techniques:

- Decision function: where classification models have a `.decision_function` method that provides us with score associated with each label.
- Probability: where classification models have a `.predict_proba` method that provides us with the probability associated with each prediction (we used it in the Classification Introduction lesson). In this case, however, the results are provided in the form of a two-dimensional array where columns represents different labels (as defined in property). Given that we will only plot ROC curves for binary problems (two labels), we should only use one of these columns. Usually, the second column (the feature representing `True` or **1**) is the one to choose. However, if you notice that the results are unexpectedly bad, you may try the other column just be sure.

We can see that our classifiers now reach different degrees of prediction. The degree can be quantified by the **Area Under the Curve (AUC)**. It refers to the area between the blue ROC curve and the orange diagonal. The area under the ROC curve, often referred to as AUC, reduces the information contained within a ROC curve down to a value between and 0 and 1, with 1 being a perfect fit. An AUC value of 0 represents a random guess, and values below the diagonal demonstrate a performance that's even worse than a guess!

SciKit Learn includes specialist functions called `roc_curve` and `roc_auc_score` to obtain ROC (FPR and TPR values for visualisation) and AUC respectively. Both function receive as input arguments the test labels (i.e. y_score) and the score (probability) associated with each prediction. We obtain the latter measure using one of the following two techniques:

- Decision function: where classification models have a `.decision_function` method that provides us with a score associated with each label.
- Probability: where classification models have a `predict_proba_` method that provides us with the probability associated with each prediction. In this case, however, the results are provided in the form of a two-dimensional array where columns represents different labels (as defined in `.classes` property). Given that we only plot ROC curves for binary problems, we should only use one of these columns. Usually, the second column (the feature representing `True` or **1**) is the one to choose. However, if you notice that the results are unexpectedly bad, you may try the other column just be sure.

```python
from sklearn.metrics import roc_curve, roc_auc_score

```

```
fig, all_axes = subplots(figsize=[15, 10], ncols=4, nrows=2, sharey=True, sharex=True)

for ax, (name, clf) in zip(all_axes.ravel(), classifiers.items()):
    clf.fit(X_train, y_train)

    # Checking whether or not the object has `decision_function`:
    if hasattr(clf, 'decision_function'):
        # If it does:
        y_score = clf.decision_function(X_test)
    else:
        # Otherwise:
        y_score = clf.predict_proba(X_test)[:, feature_2]  # We only need one column.

    # Obtaining the x- and y-axis values for the ROC curve:
    fpr, tpr, thresh = roc_curve(y_test, y_score)

    # Obtaining the AUC value:
    roc_auc = roc_auc_score(y_test, y_score)

    ax.plot(fpr, tpr, lw=2)
    ax.plot([0, 1], [0, 1], lw=1, linestyle='--')

    ax.set_xlabel('False Positive Rate')
    ax.set_ylabel('True Positive Rate')

    label = '{} - AUC: {:.2f}'.format(name, roc_auc)
    ax.set_title(label, fontsize=10)

show()
```

The (orange) diagonal represents predictions of the two labels by a coin toss. To be of value the classifier must reach a ROC curve above the diagonal.

This concludes our first steps into classification with SciKit Learn. There are many more aspects of classification. From a practical point of view, data normalisation and permutation test score as well as the workflow report are important. These will be the topics of our next lesson.

Exercises

End of chapter Exercises

Take the torus-within-a-torus data generator from the **Challenge** above.

1. Create data with three features and a noise level of 0.3.
2. Create a pseudo-3D scatter plot of one of the test data sets to judge the difficulty of the task.

3. Train the above introduced classifiers using the stratified shuffle split to generate 10 sets of testing and training data and obtain the average score for each classifier.
4. Plot the feature importances obtained from the Random Forest classifier to see the contributions of each feature to the outcome.

Note that with 3 or more features it is no longer possible to see the full state space in a plane.

5. Optional: Check how the outcome varies depending on
    - Choice of seed for random number generator
    - Number of data splits
    - Percentage of data withheld for testing

Recommendation

Pick any of the provided (or other) data sets with labels to repeat the above. Feel free to try and do any testing or plotting that you find important. This is not an assignment to get the correct answer. Rather at this stage, we practise to use functionality from scikit-learn to search for structure in the data that helps to achieve the best predictions possible.

```python
from numpy import mgrid, linspace, arange, mean, array
from numpy.random import uniform, seed

from matplotlib.ticker import LinearLocator, FormatStrFormatter
from mpl_toolkits import mplot3d
from matplotlib.pyplot import subplots, axes, scatter, xticks, show
```

```python
def make_torus_3D(n_samples=100, shuffle=True, noise=None, random_state=None,
                  factor=.8):
    """Make a large torus containing a smaller torus in 3d.

    A toy dataset to visualize clustering and classification
    algorithms.

    Read more in the :ref:`User Guide <sample_generators>`.

    Parameters
    ----------
    n_samples : int, optional (default=100)
        The total number of points generated. If odd, the inner circle will
        have one point more than the outer circle.

    shuffle : bool, optional (default=True)
        Whether to shuffle the samples.

    noise : double or None (default=None)
        Standard deviation of Gaussian noise added to the data.

    random_state : int, RandomState instance or None (default)
        Determines random number generation for dataset shuffling and noise.
        Pass an int for reproducible output across multiple function calls.
        See :term:`Glossary <random_state>`.

    factor : 0 < double < 1 (default=.8)
        Scale factor between inner and outer circle.

    Returns
    -------
    X : array of shape [n_samples, 2]
        The generated samples.

    y : array of shape [n_samples]
        The integer labels (0 or 1) for class membership of each sample.
    """
    from numpy import pi, linspace, cos, sin, append, ones, zeros, hstack, vstack, intp
    from sklearn.utils import check_random_state, shuffle

    if factor >= 1 or factor < 0:
        raise ValueError("'factor' has to be between 0 and 1.")

    n_samples_out = n_samples // 2
    n_samples_in = n_samples - n_samples_out

    co, ao, ci, ai = 3, 1, 3.6, 0.2
    generator = check_random_state(random_state)
    # to not have the first point = last point, we set endpoint=False
    linspace_out = linspace(0, 2 * pi, n_samples_out, endpoint=False)
    linspace_in  = linspace(0, 2 * pi, n_samples_in,  endpoint=False)
    outer_circ_x = (co+ao*cos(linspace_out)) * cos(linspace_out*61.1)
    outer_circ_y = (co+ao*cos(linspace_out)) * sin(linspace_out*61.1)
    outer_circ_z =    ao*sin(linspace_out)

    inner_circ_x = (ci+ai*cos(linspace_in)) * cos(linspace_in*61.1)* factor
    inner_circ_y = (ci+ai*cos(linspace_in)) * sin(linspace_in*61.1) * factor
    inner_circ_z =    ai*sin(linspace_in) * factor

    X = vstack([append(outer_circ_x, inner_circ_x),
                append(outer_circ_y, inner_circ_y),
                append(outer_circ_z, inner_circ_z)]).T
```

```python

    y = hstack([zeros(n_samples_out, dtype=intp),
                ones(n_samples_in, dtype=intp)])


    if shuffle:
        X, y = shuffle(X, y, random_state=generator)

    if noise is not None:
        X += generator.normal(scale=noise, size=X.shape)

    return X, y
```

```python
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier, GradientBoostingClassifier, AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC, LinearSVC
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.metrics import roc_curve, roc_auc_score

RANDOM_STATE = 123


classifiers = {
    'Random Forest': RandomForestClassifier(random_state=RANDOM_STATE),
    'AdaBoost (Random Forest)': AdaBoostClassifier(RandomForestClassifier(random_state=RANDOM_STATE)),
    'Extra Trees': ExtraTreesClassifier(random_state=RANDOM_STATE),
    'AdaBoost (Extra Tree)': AdaBoostClassifier(ExtraTreesClassifier(random_state=RANDOM_STATE)),
    'Decision Tree': DecisionTreeClassifier(random_state=RANDOM_STATE),
    'SVC (RBF)': SVC(random_state=RANDOM_STATE),
    'SVC (Linear)': LinearSVC(random_state=RANDOM_STATE),
    'Multi-layer Perceptron': MLPClassifier(max_iter=5000, random_state=RANDOM_STATE)
}
```

Q1 and Q2

```python
seed(RANDOM_STATE)

X, y = make_torus_3D(n_samples=2000, factor=.5, noise=.3, random_state=RANDOM_STATE)

feature_1, feature_2, feature_3 = 0, 1, 2

```

```python
fig, ax = subplots(figsize=(12, 9))
ax.set_visible(False)

ax = axes(projection="3d")

im = ax.scatter3D(X[:, feature_1], X[:, feature_2], X[:, feature_3],
                  marker='o', s=20, c=y, cmap='bwr');

ax.set_xlabel('Feature A')
ax.set_ylabel('Feature B')
ax.set_zlabel('Feature C')

ax.view_init(30, 50);

show()
```

Q3

```python
sss = StratifiedShuffleSplit(random_state=RANDOM_STATE, n_splits=10, test_size=0.3)

split_data_indices = sss.split(X=X, y=y)

score = list()

for train_index, test_index in sss.split(X, y):
    X_s, y_s = X[train_index, :], y[train_index]
    new_obs_s, y_test_s = X[test_index, :], y[test_index]

    score_clf = list()

    for name, clf in classifiers.items():

        clf.fit(X_s, y_s)
        y_pred = clf.predict(new_obs_s)
        score_clf.append(clf.score(new_obs_s, y_test_s))

    score.append(score_clf)

score_mean = mean(score, axis=0)

bins = arange(len(score_mean))

fig, ax = subplots()

ax.bar(bins, score_mean);

show()

```

```
print(classifiers.keys())
print('Average scores: ')
print(["{0:0.2f}".format(ind) for ind in score_mean])
```

```
dict_keys(['Random Forest', 'AdaBoost (Random Forest)', 'Extra Trees', 'AdaBoost (Extra Tree)', 'Decision Tree', 'SVC (RBF)', 'SVC (Linear)', 'Multi-layer Perceptron'])
Average scores:
['0.87', '0.88', '0.87', '0.87', '0.83', '0.89', '0.49', '0.88']
```

```
clf_RF = RandomForestClassifier(random_state=RANDOM_STATE)

clf_RF.fit(X_s, y_s)

y_pred = clf_RF.predict(new_obs_s)

score_RF = clf_RF.score(new_obs_s, y_test_s)

print('Random Forest score:', score_RF)
```

```
Random Forest score: 0.88
```

Q4

```
importances = clf_RF.feature_importances_

template = 'Feature 1: {:.1f}%; Feature 2: {:.1f}%; Feature 3: {:.1f}%'

print(template.format(importances[0]*100, importances[1]*100, importances[2]*100))

bins = arange(importances.shape[0])

fig, ax = subplots()

ax.bar(bins, importances, color=('g', 'm', 'b'));
ax.set_ylabel('Feature Importance', fontsize=16)

xticks(bins, ('Feature 1', 'Feature 2', 'Feature 3'), fontsize=16);

show()
```

```
Feature 1: 31.4%; Feature 2: 33.9%; Feature 3: 34.7%
```

The three features contribute similarly to the outcome.

Keypoints

- Different classification algorithms approach problems differently.
- `train_test_split` function tries to preserve the ratio of labels in the split
- Increasing the level of noise in the data makes the task more complicated.
- The potential bias due to splitting could be avoid using stratified shuffle split.
- `StratifiedShuffleSplit` is a method that uses `n_splits` and `test_size` parameters.