Artificial Neural Networks

Perceptron

The Perceptron is a very simple linear binary classifier. It basically maps and input vector x to a binary output $f(x)$.

Given a weight vector w, the Perceptron's classfication rule is: $f(x) = 1$ if $w \cdot x + b > 0$ or $f(x) = 0$ otherwise. Here, b is a bias value which is responsible for shifting the Perceptron's hyperplane away from the origin.

Practical example

First we do all necessary imports.

In [1]:

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

from matplotlib.colors import ListedColormap

from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.linear_model import Perceptron
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Setting random seed.
seed = 10
```

The most simple examples for Perceptron are the basic logic operations, such as: AND, OR and XOR.

The AND operation is defined as:

x_0	x_1		y
0	0		0
1	0		0
0	1		0
1	1		1

Below we run the Perceptron to learn the logical AND.

In [2]:

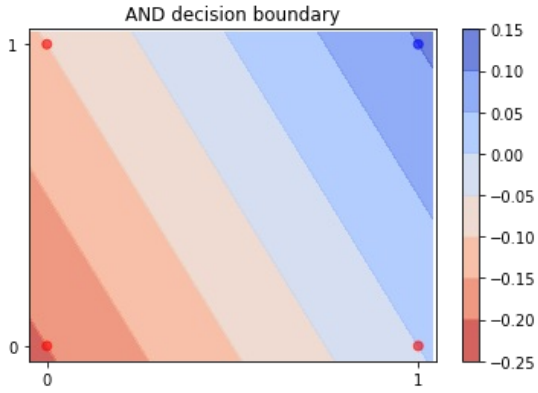
```
# Setting the input samples.
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]],
             dtype=np.double)

# Setting the expected outputs.
y_AND = np.array([0, 0, 0, 1])

# Creating and training a Perceptron.
p = Perceptron(random_state=seed, eta0=0.1, max_iter=1000)
p.fit(X, y_AND)

# Obtaining f(x) scores.
pred_scores = p.decision_function(X)
print("Perceptron's AND scores: {}".format(pred_scores))
```

```
Perceptron's AND scores: [ -2.00000000e-01  -1.00000000e-01  -2.77555756e-17   1.00000000e-01]
```

Then, we plot the Perceptron's decision boundary. The colorbar to the left shows the scores achieved by $w \cdot x + b$. Each point color indicates a different class (blue = 1, red = 0).

In [3]:

```
# Method to plot the Perceptron's decision boundary.
# This code is based on http://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html
def plot_decision_boundary(classifier, X, y, title):
    xmin, xmax = np.min(X[:, 0]) - 0.05, np.max(X[:, 0]) + 0.05
    ymin, ymax = np.min(X[:, 1]) - 0.05, np.max(X[:, 1]) + 0.05
    step = 0.01

    cm = plt.cm.coolwarm_r
    #cm = plt.cm.RdBu

    thr = 0.0
    xx, yy = np.meshgrid(np.arange(xmin - thr, xmax + thr, step), np.arange(ymin - thr, ymax + thr, step))

    if hasattr(classifier, 'decision_function'):
        Z = classifier.decision_function(np.hstack((xx.ravel()[:, np.newaxis], yy.ravel()[:, np.newaxis])))
    else:
        Z = classifier.predict_proba(np.hstack((xx.ravel()[:, np.newaxis], yy.ravel()[:, np.newaxis])))[:, 1]

    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z, cmap=cm, alpha=0.8)
    plt.colorbar()

    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=ListedColormap(['#FF0000', '#0000FF']), alpha=0.6)

    plt.xlim(xmin, xmax)
    plt.ylim(ymin, ymax)

    plt.xticks((0.0, 1.0))
    plt.yticks((0.0, 1.0))

    plt.title(title)

# Plotting Perceptron decision boundary.
# The colorbar shows the scores obtained for f(x).
plot_decision_boundary(p, X, y_AND, 'AND decision boundary')
plt.show()
```



The OR operation is defined as:

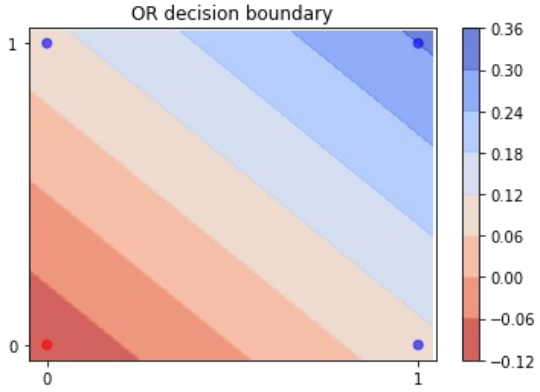
x_0	x_1	y
0	0	0
1	0	1
0	1	1
1	1	1

Below we run the Perceptron to the logical OR, print its achieved scores and plot its decision boundary.

In [4]:

```
y_OR = np.array([0, 1, 1, 1])
p.fit(X, y_OR)

# Obtaining f(x) scores.
pred_scores = p.decision_function(X)
print("Perceptron's OR scores: {}".format(pred_scores))

plot_decision_boundary(p, X, y_OR, 'OR decision boundary')
plt.show()
```

```
Perceptron's OR scores: [-0.1  0.1  0.1  0.3]
```



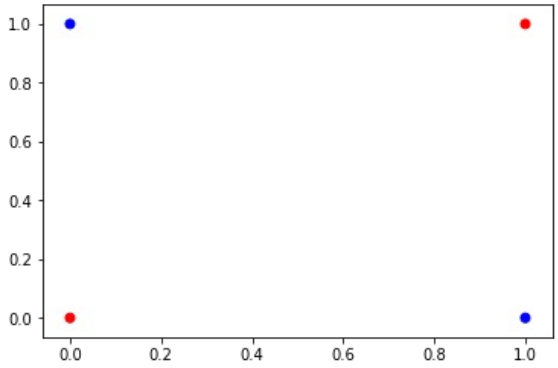
Finally, we analyze the XOR operation, which is defined as:

x_0	x_1	y
0	0	0
1	0	1
0	1	1
1	1	0

Below we plot XOR.

In [5]:

```
y_XOR = np.array([0, 1, 1, 0])
plt.scatter(X[:, 0], X[:, 1], c=y_XOR, cmap=ListedColormap(['#FF0000', '#0000FF']), alpha=1.0)
plt.show()
```



Clearly, this is not a linear separable problem. In other words, it is not possible to separate the two classes with a single hyperplane.

This kind of problem motivates us to use Multilayer Perceptrons (MLPs), which are shown in the sequence.

Multilayer Perceptron (MLP)

A MLP is a neural network which is composed by at least three different layers: an input layer, a hidden layer and an output layer. Except for the input layer, the remaining ones are composed by Perceptrons with nonlinear activation functions (e.g., sigmoid or tanh).

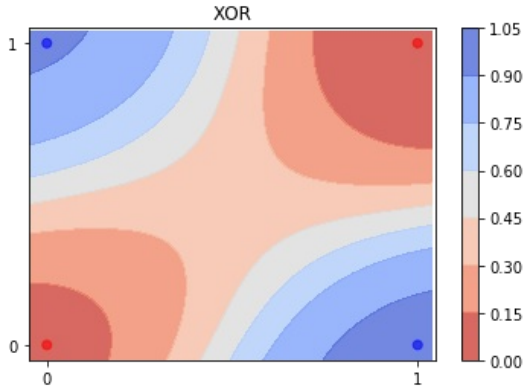
MLPs are usually trained using the backpropagation algorithm and are able to deal with not linearly separable problems.

Below we present an example for the XOR problem.

Practical example - XOR

In [6]:

```
# Creating a MLPClassifier.
# hidden_layer_sizes receive a tuple where each position i indicates the number of neurons
# in the ith hidden layer
# activation specifies the activation function (other options are: 'identity', 'logistic' and 'relu')
# max_iter indicates the maximum number of training iterations
# There are other parameters which can also be changed.
# See http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html
mlp = MLPClassifier(hidden_layer_sizes=(5,),
                    activation='tanh',
                    max_iter=10000,
                    random_state=seed)

# Training and plotting the decision boundary.
mlp.fit(X, y_XOR)
plot_decision_boundary(mlp, X, y_XOR, 'XOR')
plt.show()

pred = mlp.predict_proba(X)
print("MLP's XOR probabilities:\n[class0, class1]\n{}".format(pred))
```




```
MLP's XOR probabilities:
[class0, class1]
[[ 0.90713158  0.09286842]
 [ 0.0837964   0.9162036 ]
 [ 0.04619978  0.95380022]
 [ 0.95695933  0.04304067]]
```

Practical example - Breast Cancer

First of all, we load the dataset, encode its labels as int values and split it into training and test sets.

In [7]:

```
# Loading Breast Cancer dataset.
data = pd.read_csv('data/breast_cancer.csv')

# Creating a LabelEncoder and transforming the dataset labels.
le = LabelEncoder()
y = le.fit_transform(data['diagnosis'].values)

# Extracting the instances data.
X = data.drop('diagnosis', axis=1).values

# Splitting into train and test sets.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.34, stratify=y, random_state=seed)
```

Then, we create, train and test a MLPClassifier.

In [8]:

```
mlp = MLPClassifier(hidden_layer_sizes=(10,),
                    activation='tanh',
                    max_iter=10000,
                    random_state=seed)

mlp.fit(X_train, y_train)
y_pred = mlp.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("MLP's accuracy score: {}".format(accuracy))
```

```
MLP's accuracy score: 0.6288659793814433
```

We can observe that its accuracy score was rather low.

Unfortunately, MLPs are very sensitive to different feature scales. So, it is normally necessary to normalize or rescale the input data.

In [9]:

```
# Creating a StandardScaler. This object normalizes features to zero mean and unit variance.
scaler = StandardScaler()
scaler.fit(X_train)

# Normalizing train and test data.
X_train_scaled, X_test_scaled = scaler.transform(X_train), scaler.transform(X_test)

# Training MLP with normalized data.
mlp.fit(X_train_scaled, y_train)

# Testing MLP with normalized data.
y_pred = mlp.predict(X_test_scaled)
accuracy = accuracy_score(y_test, y_pred)
print("MLP's accuracy score: {}".format(accuracy))
```

```
MLP's accuracy score: 0.979381443298969
```