Department of Physics
PH3010 Advanced Skills

Introduction to Machine Learning

1. Introduction

This project will provide an introduction to Machine Learning (ML) using the Python programming package `scikit-learn`. Some basic ideas behind Machine Learning (ML) are introduced in Sec. 2 and in Sec. 3 we focus on an important type of ML algorithm called *classification*. As primary examples, linear classifiers and neural networks are described. In Sec. 4 the Python package `scikit-learn` is presented, which is used for the exercises in Sec. 5.

2. Basic ideas of Machine Learning

The term Machine Learning (ML) refers to algorithms that "learn from data" and make predictions based on what has been learned. In the simplest sense, this means that the algorithm contains adjustable parameters whose values are estimated using data. So formally we can regard the simple fitting of a curve to data as a type of machine learning. For example we could fit a curve

$$f(x; \boldsymbol{\theta}) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 \tag{1}$$

to the data values as shown in Fig. 1. The values of the parameters are estimated (learned) from the measured data values (x_i, y_i), $i = 1, \ldots, N$. The fitted curve can then be used to predict the function at values of x where no data point was measured.

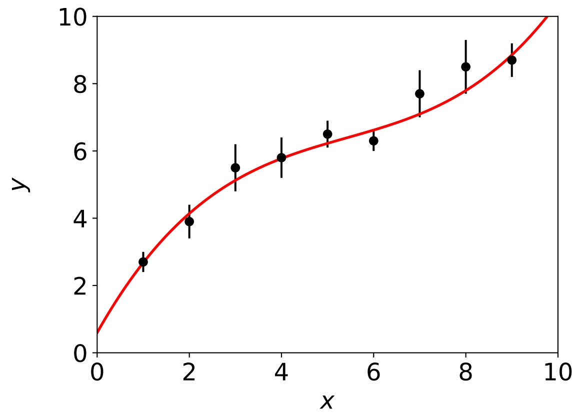

Figure 1: A curve $f(x; \boldsymbol{\theta})$ fitted to measurements y with error bars σ are carried out at known values of a control variable x.

Generally, however, the term Machine Learning refers to situations in which the hypothesized model is very general, e.g., not just a polynomial, and it often contains a very large number of adjustable parameters. Furthermore the quantity we want to predict such as the function $f(x; \boldsymbol{\theta})$ above often depends not only on a single variable x but rather on a large number of quantities, and is thus said to be *multivariate*.

For example, one can extend the least-squares fit of a curve to data points (x_i, y_i) to the fit of a (hyper-)surface to measurements y_i, each carried out at a point $\mathbf{x} = (x_1, \ldots, x_n)$ in an n-dimensional space. For $n = 2$ this would correspond to fitting a surface, as illustrated in Fig. 2. This is often called *multivariate regression*.

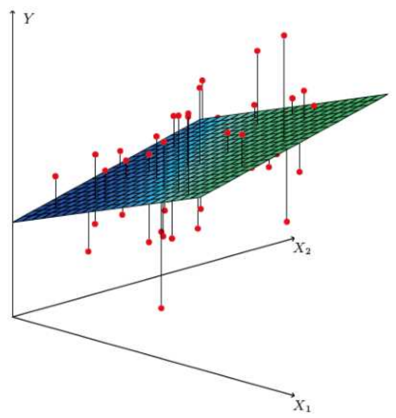

Figure 2: A function $f(x_1, x_2)$ represented as a surface fitted to data points $(x_{1,i}, x_{2,i}, y_i)$ (from Ref. [1]).

In this project we will focus not on regression but rather another Machine Learning task called *classification*, described further in Sec. 3. Machine Learning can be seen as part of or related to a number of other fields, such as Artificial Intelligence, Pattern Recognition, Statistical Learning and Multivariate Analysis. It was developed mainly from Computer Science with important input from Statistics. Collectively these fields are often called *Data Science*. A good introduction to these topics can be found in Ref. [1].

3. Classification

In this section we will introduce a type of machine learning algorithm called *classification*. This is an example of what is called *supervised learning*, whereby the parameters of the algorithm are adjusted using data samples where the true class of the objects in question is known. The basic ideas are presented in Sec. 3.1, and then two important types of classifier are described: linear classifiers in Sec. 3.2 and neural networks in Sec. 3.3. Information on further types of classifiers can be found, e.g., in Ref. [1]. A brief discussion on Boosted Decision Trees can be found in Ref. [2].

3.1. Basic ideas of multivariate classifiers

Suppose we want to distinguish between objects (or "events" or "instances") of two different types. Each object is characterized by a set of n measured quantities or "features", which we write as a *feature vector* $\mathbf{x} = (x_1, \ldots, x_n)$. For example the objects could be fish, and the features are

$x_1 =$ length	$x_4 =$ area of fins
$x_2 =$ width	$x_5 =$ mean spectral reflectance
$x_3 =$ weight	$x_6 = \ldots$

Suppose we scoop up fish in a net that are of two types, and we hire an expert to examine each fish and to assign a *true class label*, e.g., $y = 0$ for sea bass and $y = 1$ for cod. If we consider only two of the features, e.g., (x_1, x_2), then we can display these in a scatter plot as in Fig. 3.

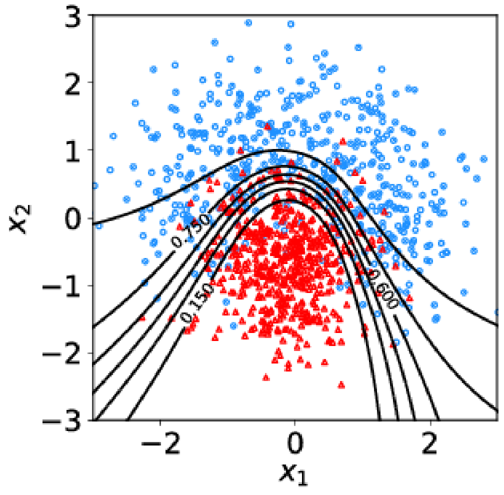

Figure 3: The distribution of (x_1, x_2) for instances of two classes, $y = 0$ (red triangles) and $y = 1$ (blue circles) with possible decision boundaries.

The goal is to determine a decision boundary so that, without the help of the expert, we can classify new fish by seeing where their measured features lie relative to the boundary. The same basic idea holds with an n-dimensional feature vector, in which case the decision boundary is a hypersurface in an n-dimensional space.

The decision boundary is defined in general by an equation of the form

$$t(\mathbf{x}) = t_c \; , \tag{2}$$

where $t(\mathbf{x})$ is a scalar function called a *test statistic* or also a *decision function* and t_c is a constant threshold value. Instances (fish) are classified as being of class 0 or 1 according to whether $t(\mathbf{x})$ is found greater or less than the chosen threshold. Figure 3 shows some possible decision boundaries for several values of t_c.

What then is the best decision boundary? It turns out that there is a well-defined answer for the case where we know the joint probability density functions (pdfs) of the two classes $f(\mathbf{x}|0)$ and $f(\mathbf{x}|1)$.[1] The optimal decision boundary will be a surface of constant

$$t(\mathbf{x}) = \frac{f(\mathbf{x}|1)}{f(\mathbf{x}|0)} \; . \tag{3}$$

This then is called a *Bayes optimal* (or *Neyman-Pearson*) boundary. The problem is that we do not usually have the probability densities $f(\mathbf{x}|0)$ and $f(\mathbf{x}|1)$, but rather only samples of data that correspond to the two classes. For the fish, for example, these were the samples identified by the fish expert. In other cases we might have Monte Carlo model that can be used to generate instances of $\mathbf{x}$ that follow the two densities. In either case we will refer to these data samples as *training data*, i.e., data of the form $(\mathbf{x}_i, y_i)$ with $i = 1, \ldots, N$ where $\mathbf{x}_i$ is the feature vector of the ith instance and y_i is its true class label.

If one does not have access to the true pdfs $f(\mathbf{x}|0)$ and $f(\mathbf{x}|1)$, then how can the optimal decision boundary be found? In general one tries to make some assumption for the functional form of the test statistic $t(\mathbf{x})$ that contains some undetermined parameters. The values of these parameters are then adjusted using the training data so as to result in the best possible separation between the two classes of events. In general this will mean that the corresponding

[1]For a further information on probability densities and their properties, see, e.g, Ref. [3].

decision boundaries should be as close as possible to what would be obtained with a Bayes optimal classifier.

For a given value of the threshold $t_{\rm c}$ one can classify an instance as belonging to class 0 if $t < t_{\rm c}$ and to class 1 if $t \geq t_{\rm c}$. From the distributions of the decision function $t(\mathbf{x})$ for the two classes, $f(t|0)$ and $f(t|1)$, one can then work out the probability for correctly classifying the instances,

$$P(\text{correctly classify type 0}) = P(t < t_{\rm c}|0) = \int_{-\infty}^{t_{\rm c}} f(t|0)\, dt \;, \tag{4}$$

$$P(\text{correctly classify type 1}) = P(t \geq t_{\rm c}|1) = \int_{t_{\rm c}}^{\infty} f(t|1)\, dt \;. \tag{5}$$

Furthermore if a real data sample consists of fractions π_0 and $\pi_1 = 1 - \pi_0$ of instances of type 0 and 1, respectively, then one can find from *Bayes' theorem* (see, e.g., Ref. [3]) the probability for an instance to belong to one of the classes given a value t of the decision function,

$$P(0|t) = \frac{f(t|0)\pi_0}{f(t|0)\pi_0 + f(t|1)\pi_1} \;, \tag{6}$$

$$P(1|t) = \frac{f(t|1)\pi_1}{f(t|0)\pi_0 + f(t|1)\pi_1} \;. \tag{7}$$

In a similar way one can find the probability for an instance to be of type 0 or 1 given that it is found on one or the other side of the decision boundary, i.e.,

$$P(0|t < t_{\rm c}) = \frac{P(t < t_{\rm c}|0)\pi_0}{P(t < t_{\rm c}|0)\pi_0 + P(t < t_{\rm c}|1)\pi_1} \;, \tag{8}$$

$$P(1|t \geq t_{\rm c}) = \frac{P(t \geq t_{\rm c}|0)\pi_0}{P(t \geq t_{\rm c}|0)\pi_0 + P(t \geq t_{\rm c}|1)\pi_1} \;. \tag{9}$$

3.2. Linear classifiers

A simple *Ansatz* for the form of the test statistic is a linear function of the components of the n-dimensional feature vector $\mathbf{x} = (x_1, \ldots x_n)$, also referred to below as the "input variables". That is, we define the statistic $t(\mathbf{x})$ as

$$t(\mathbf{x}) = \sum_{i=1}^{n} w_i x_i \;, \tag{10}$$

where the *weights* $\mathbf{w} = (w_1, \ldots, w_n)$ are parameters that we adjust to achieve the best possible decision boundary. To make the problem well defined we must specify what we mean by "best" decision boundary. One possibility is to maximize the quantity

$$J(\mathbf{w}) = \frac{(E[t|0] - E[t|1])^2}{V[t|0] + V[t|1]} \;, \tag{11}$$

where $E[t|0]$ and $E[t|1]$ are the expectation values (means) of t and $V[t|0]$ and $V[t|1]$ are the corresponding variances for instances of class 0 and 1, respectively. By adjusting the weights to maximize $J(\mathbf{w})$ one obtains a Fisher linear discriminant. The weights are only determined up to a multiplicative constant and can be shown to be given by (see, e.g., Ref. [3])

$$\mathbf{w} \propto W^{-1}(E[\mathbf{x}|0] - E[\mathbf{x}|1]) \ , \tag{12}$$

where the matrix W is the sum of the covariances for the two classes, i.e.,

$$W_{ij} = \mathrm{cov}[x_i, x_j|0] + \mathrm{cov}[x_i, x_j|1] \ . \tag{13}$$

Once the weights are fixed, one can treat the function $t(\mathbf{x})$ as a fixed function of the random variable $\mathbf{x}$. For each new instance $\mathbf{x}$ one obtains a value of t, and so the distribution of t can be displayed as a histogram.

A two-dimensional example is shown in Fig. 4(a), with decision boundaries corresponding to several values of the constant t_{c}. Figure 4(b) shows the distribution of of the decision function t for the two classes.

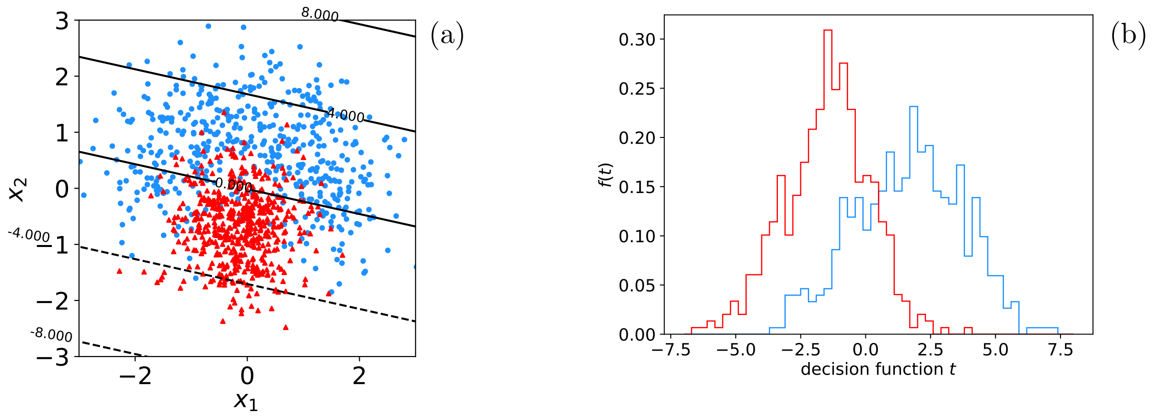

Figure 4: (a) Distributions of instances (x_1, x_2) for two classes with decision boundaries corresponding to a linear Fisher discriminant; (b) distributions of the corresponding decision function $t(\mathbf{x})$ for the two classes.

From the histograms shown in Fig. 4(b) one can determine the distributions of the statistic t, $f(t|0)$ and $f(t|1)$, and from the corresponding classification probabilities can be found using the formulae given in Sec. 3.1.

3.3. Neural networks

One can easily see from the scatter plot shown in Fig. 4(a) that a linear decision boundary is not the best possible choice. Better classification probabilities would clearly result from a nonlinear boundary as in Fig. 3. To obtain such a nonlinear boundary, the decision function $t(\mathbf{x})$ must be a nonlinear function of the input variables $\mathbf{x}$. One way of constructing such a function is with a neural network.

A simple type of neural network called a *single layer perceptron* is can be defined as

$$t(\mathbf{x}) = h\left(w_0 + \sum_{i=1}^{n} w_i x_i\right) \ , \tag{14}$$

where $h(\,)$ is called the *activation function.* That is, the argument of the activation function is the same as the linear combination of variables used in the linear discriminant, but here also has an adjustable offset w_0. Often one chooses for h a *logistic sigmoid* function, defined as

$$h(u) = \frac{1}{1 + e^{-u}} \; . \tag{15}$$

A sketch of the logistic sigmoid is shown in Fig. 5.

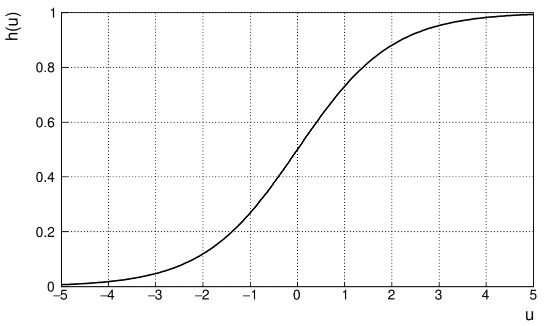

Figure 5: A sigmoid activation function $h(u)$.

The structure of the test statistic $t(\mathbf{x})$ is often represented with a graph as in Fig. 6(a). The input variables $\mathbf{x} = (x_1, \ldots, x_n)$ are shown as nodes on the left; the function's output is on the right. The lines from the output to input nodes represent the connections, quantified by the corresponding weight parameters $w_1, \ldots, w_n$.

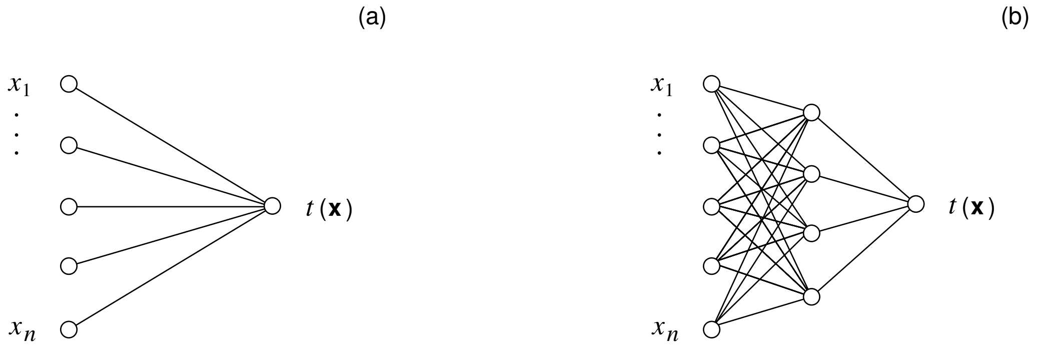

Figure 6: Simple examples of neural networks: (a) a single-layer perceptron and (b) a multi-layer perceptron with a single hidden layer.

The sigmoid function is monotonic and thus has a unique inverse h^{-1}. Therefore for the single-layer perceptron, the surface described by $t(\mathbf{x}) = t_{\rm c}$ is the same as what one finds from

$$h^{-1}(t(\mathbf{x})) = w_0 + \sum_{i=1}^{n} w_i x_i = h^{-1}(t_{\rm c}) \; . \tag{16}$$

That is, the linear combination of the input variables is still equal to a constant and therefore one finds a linear decision boundary. We can, however, obtain a nonlinear boundary by defining a *multi-layer perceptron.* To do this, we first define a set of functions $\boldsymbol{\varphi} = (\varphi_1(\mathbf{x}), \ldots, \varphi_m(\mathbf{x}))$, according to

$$\varphi_i(\mathbf{x}) = h\left(w_{i0}^{(1)} + \sum_{j=1}^{m} w_{ij}^{(1)} x_j\right) , \tag{17}$$

where the $w_{ij}^{(1)}$ are the weights that relate φ_i to the input variables. The functions $\varphi_i(\mathbf{x})$ are said to constitute a *hidden layer* and are indicated by the middle column of nodes in Fig. 6(b). The final decision function is determined by treating the $\boldsymbol{\varphi}$ as if they were now the input variables, i.e.,

$$t(\mathbf{x}) = h\left(w_0^{(2)} + \sum_{j=1}^{m} w_j^{(2)} \varphi_j(\mathbf{x})\right) , \tag{18}$$

where $w_j^{(2)}$ are the weights that connect the output $t(\mathbf{x})$ to the previous layer of the network, here the hidden layer. The decision boundaries shown earlier in Fig. 3 are in fact from a multi-layer perceptron with a single hidden layer.

In constructing a neural network the analyst must decide on its *architecture*, i.e. the number of hidden layers and the number nodes in each of these layers. This is an active topic of research and a detailed discussion is beyond the scope of this project. There is a theorem according to which one can obtain a Bayes optimal decision boundary for a sufficiently large number of nodes in a single hidden layer (see, e.g., [4]). Nevertheless the number of required nodes may be very large and this may entail significant computational difficulties. In recent years there has been significant progress made with *deep neural networks*, where here "deep" refers to a large number (e.g., 5 to 10 or more) of hidden layers. These form the basis of *deep learning*, and have demonstrated significant advantages over "shallow" networks with a single hidden layer.

To train the neural network, i.e., to find the optimal values of the weights, one generally minimizes a *loss function* of the form

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{N} |t(\mathbf{x}_i) - y_i|^2 , \tag{19}$$

where $\mathbf{w}$ represents all of the weights, and one sums for all training events the square of the difference between the decision function $t(\mathbf{x}_i)$ and the true class label y_i. This way, the weights will be adjusted such that an event of types 0 or 1 will return a value of $t(\mathbf{x})$ near zero or one, respectively.

For a complex neural network with many weights, minimizing the loss function can be computationally very difficult and has its own extensive literature. Important algorithms including *error backpropagation* and *stochastic gradient descent*; further details can be found, e.g., in Ref. [5].

One of the most important properties of a neural network or indeed any machine learning algorithm is its ability to correctly classify unseen data, i.e., data that has not been used as part of the training. As an example of the type of difficulty involved, suppose one were to design an algorithm with a very large number of adjustable parameters and thus a very flexible decision boundary, as shown in Fig. 7(a).

Because of its ability to twist and turn, the boundary has succeeded in correctly classifying all of the events in the training sample. But if one takes the same boundary and applies it to a statistically independent sample of events (a *test sample*) as shown in Fig. 7(b), then the

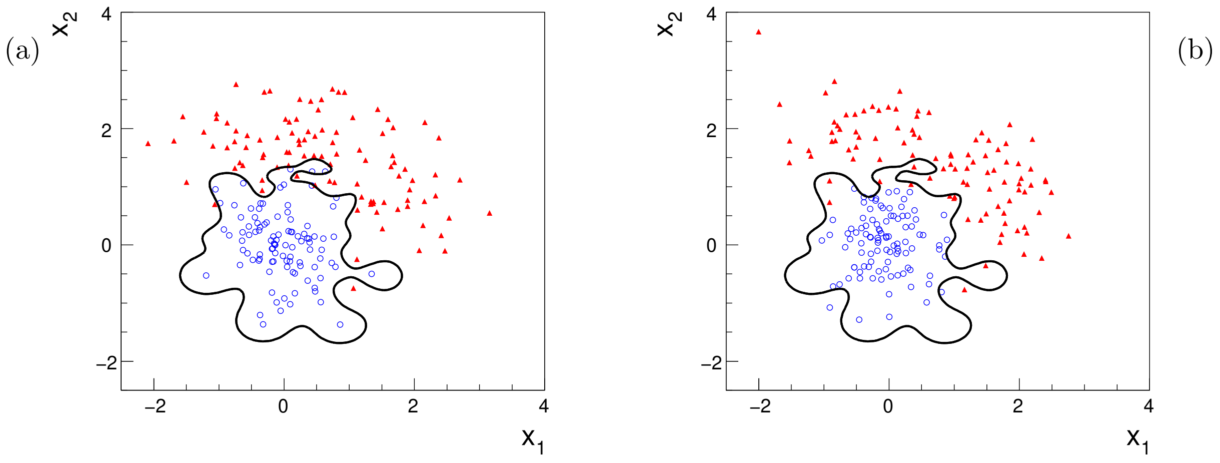

Figure 7: Scatter plot of events of two types and the decision boundary determined by a particularly flexible classifier. Plot (a) shows the events used to train the classifier, and (b) shows an independent sample of test data.

performance is significantly worse. The nonlinear features needed to get every event on the right side of the boundary in the training sample actually degrade the performance for the test sample. This is an example of what is called *overtraining*. To avoid this, one can use the test sample to evaluate the classifier's performance and to choose its architecture and in this way fix the level of flexibility of the decision boundary.

4. Using `scikit-learn`

In this project several examples of multivariate classifiers will be explored using the Python package `scikit-learn`, available from the website `scikit-learn.org`. It is built on the Python packages NumPy and SciPy, and as usual plots can be made with MatPlotLib. So at the start of your code you need to include the appropriate libraries, as shown in the sample code in Appendix A.

To use a given classifier supported by `scikit-learn` you need to include its corresponding package. The usual practice is to import only those that are needed. For example, for a multi-layer perceptron your code would contain the line

```
from sklearn.neural_network import MLPClassifier
```

For a list of the various classifiers in `scikit-learn` see the documents on `scikit-learn.org`, specifically the very useful sample program at

```
scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html
```

We will do an example with data corresponding to events of two types: signal ($y = 1$, blue) and background ($y = 0$, red). Each event is characterized by three values, i.e., $\mathbf{x} = (x_1, x_2, x_3)$. The three components are correlated in a nontrivial way; for x_1 and x_2 this can be seen in the scatter plots shown earlier such as Fig 3. The marginal distributions of each of the three components are shown in Fig. 8.

The sample program `simpleClassifier.py` in Appendix A shows how to read in the data from two files, one for each event type, `signal.txt` and `background.txt`. The signal

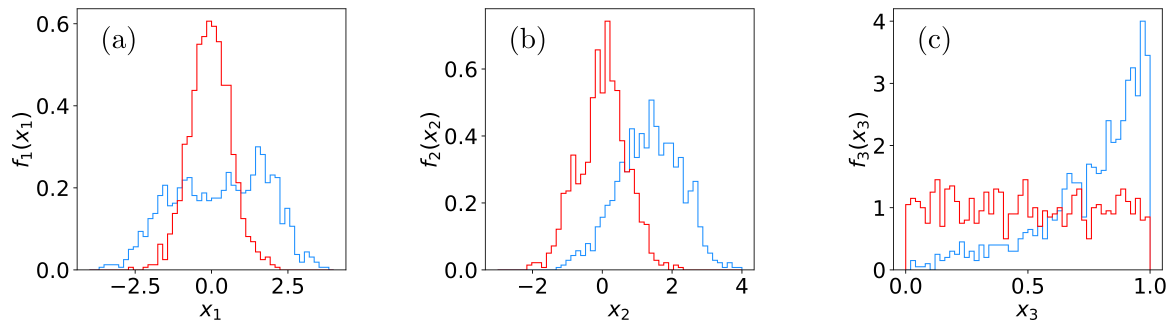

Figure 8: Marginal distributions of the three components of the feature vector $\mathbf{x} = (x_1, x_2, x_3)$ for events of the two classes: signal ($y = 1$, blue) and background ($y = 0$, red).

events are assigned a target value of 1 and the background events a value 0. The arrays of values for signal and background events are then concatenated into a single long array, which is then split by random assignment of events into two disjoint sets, one for training the classifier and the other for evaluating its performance.

The code as given in `simpleClassifier.py` contains the line

```
X = X[:,0:2]                      # at first, only use x1 and x2
```

The purpose of this line is to use only the first two components, x_1 and x_2, of the three-dimensional feature vector so that one can make a scatter plot of the two values and visualise more easily the resulting decision boundary. After having made the two-dimensional plots one would comment out this line so as to use all of the components of the feature vector.

The architecture and other parameters of the multilayer perceptron are defined by the line

```
clf = MLPClassifier(hidden_layer_sizes=(5,), activation='tanh',
                    max_iter=2000, random_state=0)
```

There are several choices for the activation function, including tanh, logistic (i.e., a sigmoid function) and relu (rectified linear unit). To define the number of hidden layers of given sizes one uses the `hidden_layer_sizes` argument. For example,

```
hidden_layer_sizes=(10,10,10)
```

gives three hidden layers each having ten nodes. Note that for a single hidden layer, Python requires an extra comma, e.g., `hidden_layer_sizes=(5,)` for a single hidden layer with five nodes. The classifier is trained, i.e., the values of the weights of the neural network are fitted using the training data by the line

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

One can then evaluate the prediction accuracy of the classifier using

```
y_pred = clf.predict(X_test)
print 'classification accuracy = ', metrics.accuracy_score(y_test, y_pred)
```

This uses the `predict_proba` function with a default threshold of 0.5. Alternative one can use an arbitrary threshold, e.g., $t_{\rm c} = 0.3$, with

```
y_pred = (clf.predict_proba(X_test)[:,1] >= 0.3).astype(bool)
```

or with classifiers where a decision function has been defined one can replace in the line above `predict_proba` by `decision_function`. Or one can simply loop over the events and evaluate the classifier's decision function at an arbitrary point. For example, using the two-dimensional example one could evaluate $t(x_1, x_2)$ at the point $x = 0.37$ and $x_2 = 2.46$ using

```
xpt = np.array([0.37, 2.46]).reshape((1,-1))   # make numpy array
t = clf.predict_proba(xpt)[0, 1]
```

Note for most classifiers available in `scikit-learn` one can use the `decision_function` to get $t(\mathbf{x})$. Exceptionally, for the multilayer perceptron this function has not been implemented, but one has instead the closely related function `predict_proba`, which gives the probability for an event to be of type 0 or 1 according to Eqs. (6) and (7).

5. Project exercises

Exercise 1 (warm up):

1(a) Run the program simpleClassifier.py and describe the output. It is set up to use at first only the first two components, x_1 and x_2, so that the results can be displayed as a scatter plot.

1(b) Change the program to use all three input variables by removing the line `X = X[:,0:2]`. You will also have to side-step the code that makes the scatter plot.

1(c) Change numbers of hidden layers and nodes; try to find the maximum possible classification accuracy. Note if the number of requested layers/nodes gets too large, it will not be possible to train (find the minimum of the loss function).

1(e) For the best architecture that you find, use the test sample to produce histogram of the network output (see sample code).

Exercise 2: In this exercise you will experiment with different types of classifiers.

2(a) Using the scikit-learn documentation and the program `plot_classifier_comparison` mentioned above, implement a linear classifier (class `LinearDiscriminantAnalysis`) using all three input variables.

2(b) Make a histogram of the classifier output; compare its performance to your best neural network.

2(c) By consulting the documentation and sample program, implement and investigate the performance of: (i) a K-Nearest Neighbour Classifier (KNeighborsClassifier), (ii) a Support Vector Machine (SVC), (iii) a Boosted Decision Tree (AdaBoostClassifier). Provide a brief description of these classifiers in your report.

2(d) For at least one of the classifiers implemented, plot the classification error rate as a function of its complexity (i.e., flexibility of the decision boundary). For example, for the K-Nearest Neighbour algorithm, vary K; for the Support Vector Machine, use the radial basis function (Gaussian) kernel, and plot the error rate as a function of the cost parameter C for several values γ; for a Boosted Decision Tree vary the number of boosting iterations. See the `scikit-learn` documentation for a full explanation of the parameters.

Exercise 3: Using the samples of training events above, suppose that event type 0 is regarded as "background", i.e., events of some known type, and type 1 results from a hypothetical new process, “signal”, whose existence in Nature has not yet been established. In an experiment a certain total number of events is produced, which we will model as a random variable following a Poisson distribution.

We can count the events and measure for each the feature vector $\mathbf{x}$, but we do not know whether a given event is signal or background. Nevertheless we can use the training samples corresponding to the two known data types to construct a test statistic $t(\mathbf{x})$ that will be, say, low for background events and high for signal events. We can then count the number of events n found with $t(\mathbf{x})$ in the signal-like region, $t(\mathbf{x}) \geq t_\mathrm{c}$ for some threshold or “cut” value t_c. If this number of events is found to be too large than what can be explained by the background-only hypothesis, then we can claim discovery of the signal.

Suppose that the expected total number of background events is $b_\mathrm{tot} = 100$ and the expected number for the hypothetical signal process is $s_\mathrm{tot} = 10$. The expected numbers of events having $t(\mathbf{x}) \geq t_\mathrm{c}$ are given by

$$b = b_\mathrm{tot} P(t \geq t_\mathrm{c}|0) \; , \tag{20}$$

$$s = s_\mathrm{tot} P(t \geq t_\mathrm{c}|1) \; . \tag{21}$$

To find the probabilities of $t \geq t_\mathrm{c}$ for the given event types one can loop over the events of the two types, evaluate $t(\mathbf{x})$ for each event, and count the number on each side of the threshold.

3(a) Train classifiers using the samples of training data. Make a plot of s and b as a function of t_c for the linear classifier and at least one of the others.

3(b) In the search for the signal process, we can count the number of events found with $t \geq t_\mathrm{c}$ and use this to test the hypothesis that all of the events are of the background type. The number of events found n will follow a Poisson distribution with mean $s+b$, i.e., the probability of n given a certain threshold t_c is

$$P(n|t \geq t_\mathrm{c}) = \frac{(s+b)^n}{n!} e^{-(s+b)} \; . \tag{22}$$

To establish the existence of the signal process we can test and try to reject the hypothesis that $s = 0$ (the “background-only hypothesis”). If, for example, we select all of the events regardless of t (i.e., $t_\mathrm{cut} \to -\infty$)), then $s + b = 0 + 100 = 100$. If we were to see, say, 103 events, then this would be consistent with having $s = 0$, since the value of n fluctuates as a Poisson distributed quantity with mean $b = 100$ and thus with a standard deviation $\sigma_n = \sqrt{b} = 10$. On the other hand, if we saw $n = 150$, then we would be more confident that some additional process (such as the signal) is contributing to the background.

An estimate of s based on data can be found using $\hat{s} = n - b$ (here the hat denotes that this quantity is an *estimator* for the parameter s). That is, if we find $n = 120$ then the estimated signal strength is $\hat{s} = n - b = 20$. We can quantify the statistical significance of the observed signal by comparing $\hat{s}$ to the standard deviation of the number of background events, which is a Poisson distributed variable with mean b and therefore standard deviation $\sqrt{b}$. A measure of significance can therefore be defined as

$$Z = \frac{\hat{s}}{\sqrt{b}} = \frac{n-b}{\sqrt{b}} \; . \tag{23}$$

One can understand this formula as a measure of the signal’s size compared to the standard deviation of the number of background events, and is thus often quoted as a number of "sigmas". If, say, $Z = 5$, then the observed signal rate is five times greater than the expected level of fluctuation in the background, and thus the background-only hypothesis is strongly disfavoured.

In the planning phase of the experiment it is useful to have a measure of how statistically significant an apparent signal may be if in fact the nominal signal model is true. To quantify this we can give the expected value of Z under assumption that the mean of n is $s + b$ for a hypothesized value of s. That is, the expected significance is

$$\langle Z \rangle = \frac{\langle n \rangle - b}{\sqrt{b}} = \frac{s + b - b}{\sqrt{b}} = \frac{s}{\sqrt{b}} \,. \tag{24}$$

The formula (24) for the expected significance is in fact an approximation that breaks down for small b. A better approximation that takes into effect the Poisson nature of the data is

$$\langle Z \rangle = \sqrt{2\left((s+b)\ln\left(1 + \frac{s}{b}\right) - s\right)} \,. \tag{25}$$

The motivation behind Eq. (25) is given in Ref. [6] and goes beyond the scope of this project. Nevertheless you can use the formula and compare the results you get between Eqs. (24) and (25).

To design an experiment to search for the presence of signal, one must determine an optimal value for t_c. To do this, make a plot of $\langle Z \rangle$ using the formulae above (Eq. (25) is preferred). Find the value of t_c for which this is maximum and the corresponding $\langle Z \rangle$.

In addition to the cut value t_c, all aspects of the analysis, e.g., the choice of classifier, its hyperparameters, input variables, etc., can be found by optimizing something that measures the expected "quality" of the result, which in this case can be taken as the expected discovery significance $\langle Z \rangle$. Try therefore to repeat the optimization procedure above by varying the classifier and/or its hyperparameters, with the goal of finding the highest possible expected discovery significance.

References

[1] Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani, *An Introduction to Statistical Learning with Applications in R*, Springer, 2013; `http://www-bcf.usc.edu/~gareth/ISL/`

[2] G. Cowan, *Topics in statistical data analysis for high energy physics*, Lectures given at the 2009 European School of High-Energy Physics, Bautzen, Germany, 14-27 June 2009, CERN Yellow Report CERN-2010-002, pp.197-218; arXiv:1012.3589 (2010).

[3] G. Cowan, *Statistical Data Analysis*, Oxford University Press, 1998.

[4] M. Leshno, V. Lin, A. Pinkus and S. Schocken, *Multilayer Feedforward Networks With a Nonpolynomial Activation Function Can Approximate Any Function*, Neural Networks 6 (1993) 861-867.

[5] Michael A. Nielsen, *Neural Networks and Deep Learning*, Determination Press, 2015; `neuralnetworksanddeeplearning.com`

[6] G. Cowan, K. Cranmer, E. Gross and O. Vitells, *Asymptotic formulae for likelihood-based tests of new physics*, Eur. Phys. J. C 71 (2011) 1554.

A. Python code

Program `simpleClassifier.py` for multivariate classification.

```python
#  simpleClassifier.py
#  G. Cowan / RHUL Physics / October 2017
#  Simple program to illustrate classification with scikit-learn

import scipy as sp
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics

#  read the data in from files,
#  assign target values 1 for signal, 0 for background
sigData = np.loadtxt('signal.txt')
nSig = sigData.shape[0]
sigTargets = np.ones(nSig)

bkgData = np.loadtxt('background.txt')
nBkg = bkgData.shape[0]
bkgTargets = np.zeros(nBkg)

# concatenate arrays into data X and targets y
X = np.concatenate((sigData,bkgData),0)
X = X[:,0:2]                    # at first, only use x1 and x2
y = np.concatenate((sigTargets, bkgTargets))
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=1)

# create classifier object and train
clf = MLPClassifier(hidden_layer_sizes=(5,), activation='tanh',
                    max_iter=2000, random_state=0)
clf.fit(X_train, y_train)

# evaluate its accuracy using the test data
y_pred = clf.predict(X_test)
print ('classification accuracy = ', metrics.accuracy_score(y_test, y_pred))

# make a scatter plot
fig, ax = plt.subplots(1,1)
plt.gcf().subplots_adjust(bottom=0.15)
plt.gcf().subplots_adjust(left=0.15)
ax.set_xlim((-2.5,3.5))
ax.set_ylim((-2,4))
x0,x1 = ax.get_xlim()
y0,y1 = ax.get_ylim()
ax.set_aspect(abs(x1-x0)/abs(y1-y0))       # make square plot
xtick_spacing = 0.5
ytick_spacing = 2.0
ax.yaxis.set_major_locator(ticker.MultipleLocator(xtick_spacing))
ax.yaxis.set_major_locator(ticker.MultipleLocator(ytick_spacing))
plt.scatter(sigData[:,0], sigData[:,1], s=3, color='dodgerblue', marker='o')
plt.scatter(bkgData[:,0], bkgData[:,1], s=3, color='red', marker='o')

# add decision boundary to scatter plot
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 = .01  # step size in the mesh
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
# depending on classifier call predict_proba or decision_function
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
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]
```

```
Z = Z.reshape(xx.shape)
plt.contour(xx, yy, Z, 1, colors='k')
plt.xlabel(r'$x_{1}$', labelpad=0)
plt.ylabel(r'$x_{2}$', labelpad=15)
plt.savefig("scatterplot.pdf", format='pdf')

# make histogram of decision function
plt.figure()                                        # new window
matplotlib.rcParams.update({'font.size':14})        # set all font sizes
# depending on classifier call predict_proba or decision_function
if hasattr(clf, "decision_function"):
    tTest = clf.decision_function(X_test)
else:
    tTest = clf.predict_proba(X_test)[:,1]
tBkg = tTest[y_test==0]
tSig = tTest[y_test==1]
nBins = 50
tMin = np.floor(np.min(tTest))
tMax = np.ceil(np.max(tTest))
bins = np.linspace(tMin, tMax, nBins+1)
plt.xlabel('decision function $t$', labelpad=3)
plt.ylabel('$f(t)$', labelpad=3)
n, bins, patches = plt.hist(tSig, bins=bins, density=True, histtype='step', fill=False, color='dodgerblue')
n, bins, patches = plt.hist(tBkg, bins=bins, density=True, histtype='step', fill=False, color='red', alpha=0.5)
plt.savefig("decision_function_hist.pdf", format='pdf')

plt.show()
```

G. Cowan
RHUL Physics
Version 1.2 / November 2019