APA-L6-python

September 6, 2018

1 APA Laboratori 6 - MLP and the RBF

```
In [1]: # Uncomment to upgrade packages
        # !pip install pandas --upgrade
        # !pip install numpy --upgrade
        # !pip install scipy --upgrade
        # !pip install statsmodels --upgrade
        # !pip install scikit-learn --upgrade
        %load_ext autoreload
```

```
In [2]: #%matplotlib notebook
        import numpy as np
        import matplotlib.pyplot as plt
        import seaborn as sn
        import pandas as pd
        from IPython.core.interactiveshell import InteractiveShell
        pd.set_option('precision', 3)
        InteractiveShell.ast_node_interactivity = "all"
```

```
In [3]: # Extra imports
        from sklearn.metrics import confusion_matrix,\
                        classification_report, accuracy_score
        from pandas import read_csv
        from sklearn.model_selection import train_test_split
        from statsmodels.genmod.generalized_linear_model import GLM
        from statsmodels.genmod.families.family import Binomial
        from statsmodels.tools.tools import add_constant
        from sklearn.preprocessing import StandardScaler
        from sklearn.neural_network import MLPClassifier
        from graphviz import Digraph
        from sklearn.model_selection import GridSearchCV
        from numpy.random import normal
        from scipy.special import expit as logistic
        from numpy.random import uniform
        from sklearn.cluster import KMeans
        from sklearn.linear_model import RidgeCV, Ridge
```

```
In [4]: def confusion(true, pred, classes):
            """
            Function for pretty printing confusion matrices
            """
            cm =pd.DataFrame(confusion_matrix(true, pred), index=classes,
                             columns=classes)
            cm.index.name = 'Actual'
            cm.columns.name = 'Predicted'
            return cm

        def graphMLP(vars,layers,intercepts):
            """
            Function for plotting the weights of a mlp
            """
            f = Digraph('')
            f.attr(rankdir='LR')
            for i,l in enumerate(layers):
                if i==0:
                    for j in range(l.shape[1]):
                        for k, v in enumerate(vars):
                            f.edge(v, 'L%dN%d'%(i,j), label=str(l[k,j]))
                    f.node('ILI', shape='doublecircle')
                    for k in range(intercepts[i].shape[0]):
                        f.edge('ILI',
                               'L%dN%d'%(i,k),
                               label=str(intercepts[i][k]))
                else:
                    for j in range(l.shape[1]):
                        for k in range(layers[i-1].shape[1]):
                            f.edge('L%dN%d'%(i-1,k),
                                   'L%dN%d'%(i,j),
                                   label=str(l[k,j]))
                    f.node('L%dI'%(i-1), shape='doublecircle')
                    for k in range(intercepts[i].shape[0]):
                        f.edge('L%dI'%(i-1),
                               'L%dN%d'%(i,k),
                               label=str(intercepts[i][k]))

            return f

In [5]: np.random.seed(4567)
```

Suppose we are interested in how variables, such as
GRE (Graduate Record Exam scores)
GPA (Grade Point Average) and
rank (prestige of the undergraduate institution)
affect admission into a graduate school.

The target variable, admit/don't admit, is a binary variable, which we want to characterize and, if possible, to predict (a model)

```
In [6]: Admis = read_csv("Admissions.csv", delimiter=',')
```

view the first few rows of the data

```
In [7]: Admis.head()
```

```
Out[7]:    admit  gre   gpa  rank
        0      0  380  3.61     3
        1      1  660  3.67     3
        2      1  800  4.00     1
        3      1  640  3.19     4
        4      0  520  2.93     4
```

We will treat all the variables gre and gpa as continuous.

The variable rank takes on the values 1 through 4, so we can fairly treat it as numerical (although, in rigour, it is ordinal)

```
In [8]: Admis.describe()

        N = Admis.shape[0]
```

```
Out[8]:          admit      gre      gpa     rank
        count  400.000  400.000  400.000  400.000
        mean     0.318  587.700    3.390    2.485
        std      0.466  115.517    0.381    0.944
        min      0.000  220.000    2.260    1.000
        25%      0.000  520.000    3.130    2.000
        50%      0.000  580.000    3.395    2.000
        75%      1.000  660.000    3.670    3.000
        max      1.000  800.000    4.000    4.000
```

We first split the available data into learning and test sets, selecting randomly 2/3 and 1/3 of the data We do this for a honest estimation of prediction performance

```
In [9]: np.random.seed(63)
        X_train, X_test, y_train, y_test =\
                train_test_split(Admis.loc[:,'gre':], Admis.admit, test_size=0.33)
```

We start using logistic regression (a linear classifier)

```
In [10]: model = GLM(y_train, add_constant(X_train), family=Binomial())
         result = model.fit()
         result.summary()
         print('AIC=', result.aic)
```

```
Out[10]: <class 'statsmodels.iolib.summary.Summary'>
         """
                          Generalized Linear Model Regression Results
         ==============================================================================
         Dep. Variable:                  admit   No. Observations:                  268
```

```
Model:                            GLM   Df Residuals:                      264
Model Family:                Binomial   Df Model:                            3
Link Function:                  logit   Scale:                          1.0000
Method:                          IRLS   Log-Likelihood:                -151.65
Date:                Thu, 06 Sep 2018   Deviance:                       303.30
Time:                        14:52:50   Pearson chi2:                     265.
No. Iterations:                     4   Covariance Type:             nonrobust
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -3.3285      1.441     -2.309      0.021      -6.153      -0.503
gre            0.0025      0.001      1.842      0.066      -0.000       0.005
gpa            0.7622      0.420      1.815      0.069      -0.061       1.585
rank          -0.6534      0.157     -4.168      0.000      -0.961      -0.346
==============================================================================
"""
```

```
AIC= 311.2952954945122
```

As we can see gre has a very low weight, so we can test if eliminating that variable we can obtain an equivalent model

```
In [11]: model = GLM(y_train, add_constant(X_train.loc[:,'gpa':]), family=Binomial())
         result = model.fit()
         result.summary()
         print('AIC=', result.aic)
```

```
Out[11]: <class 'statsmodels.iolib.summary.Summary'>
         """
                         Generalized Linear Model Regression Results
         ==============================================================================
         Dep. Variable:                  admit   No. Observations:                  268
         Model:                            GLM   Df Residuals:                      265
         Model Family:                Binomial   Df Model:                            2
         Link Function:                  logit   Scale:                          1.0000
         Method:                          IRLS   Log-Likelihood:                -153.38
         Date:                Thu, 06 Sep 2018   Deviance:                       306.75
         Time:                        14:52:50   Pearson chi2:                     271.
         No. Iterations:                     4   Covariance Type:             nonrobust
         ==============================================================================
                          coef    std err          z      P>|z|      [0.025      0.975]
         ------------------------------------------------------------------------------
         const         -2.8166      1.400     -2.012      0.044      -5.560      -0.073
         gpa            1.0625      0.389      2.735      0.006       0.301       1.824
         rank          -0.6703      0.156     -4.308      0.000      -0.975      -0.365
         ==============================================================================
         """
```

```
AIC= 312.7544764268358
```

The new model has one variable less and the error (residual deviance) is virtually the same (311.2 vs 312.7)

Interpretation of the coefficients

```
In [12]: result.params
```

```
Out[12]: const    -2.817
         gpa       1.062
         rank     -0.670
         dtype: float64
```

Calculation of apparent error in the training

```
In [13]: P=0.5

         pred = result.predict(add_constant(X_train.loc[:,'gpa':]))
         lab_tr = [1 if i>=P else 0 for i in pred]
         confusion(y_train,lab_tr, ['noadmit','admit'])

         (1-accuracy_score(y_train,lab_tr))*100
```

```
Out[13]: Predicted  noadmit  admit
         Actual
         noadmit        168     13
         admit           60     27
```

```
Out[13]: 27.23880597014925
```

we get a learning error which is quite high (~27%)

Estimation of prediction error using the test set

```
In [14]: pred = result.predict(add_constant(X_test.loc[:,'gpa':]))
         lab_tr = [1 if i>=P else 0 for i in pred]
         confusion(y_test,lab_tr, ['noadmit','admit'])
         (1-accuracy_score(y_test,lab_tr))*100
```

```
Out[14]: Predicted  noadmit  admit
         Actual
         noadmit         85      7
         admit           33      7
```

```
Out[14]: 30.303030303030297
```

we get a prediction error of ~30%

Now we switch to non-linear modelling with a MLP

In contrast to the nnet package from the R version, the MLP implementation from scikit learn is fully capable and has more features but we will restrict the model to only one hidden later

The basic parameters are 'hidden_layer_sizes' and 'alpha' (the regularization constant, lambda)

It buils a MLP with one output neuron (just two classes), with the logistic function and uses the cross-entropy as error function

Let's start by standardizing of inputs, this is important to avoid network 'stagnation' (premature convergence)

```
In [15]: Admis.loc[:,'gre':] = StandardScaler().fit_transform(Admis.loc[:,'gre':])
         Admis.head()
         X_train, X_test, y_train, y_test =\
               train_test_split(Admis.loc[:,'gre':], Admis.admit, test_size=0.33)
```

```
Out[15]:    admit    gre    gpa   rank
         0      0 -1.800  0.579  0.546
         1      1  0.627  0.737  0.546
         2      1  1.840  1.605 -1.574
         3      1  0.453 -0.526  1.606
         4      0 -0.587 -1.210  1.606
```

To illustrate the first results, we just fit a MLP with 2 hidden neurons

```
In [16]: model_nnet = MLPClassifier(hidden_layer_sizes=[2],
                                    alpha=0,
                                    activation='logistic',
                                    max_iter=200,
                                    solver='lbfgs')
         model_nnet.fit(X_train,y_train);
```

This is the final value of the error function (also known as fitting criterion)

```
In [17]: model_nnet.loss_
```

```
Out[17]: 0.5616482088928606
```

Now look at the weights

```
In [18]: model_nnet.coefs_
         model_nnet.intercepts_
```

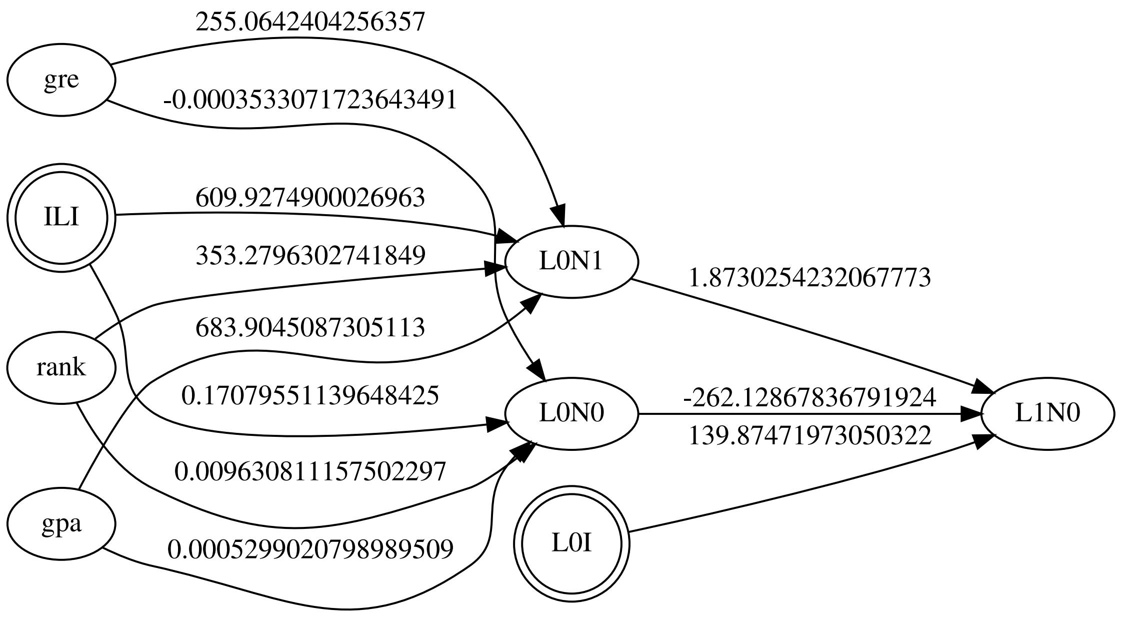
```
Out[18]: [array([[-3.53307172e-04,  2.55064240e+02],
                 [ 5.29902080e-04,  6.83904509e+02],
                 [ 9.63081116e-03,  3.53279630e+02]]), array([[-262.12867837],
                 [   1.87302542]])]
```

```
Out[18]: [array([1.70795511e-01, 6.09927490e+02]), array([139.87471973])]
```

It is a bit clearer if we draw the network

```
In [19]: graphMLP(Admis.columns[1:], model_nnet.coefs_, model_nnet.intercepts_)
```

```
Out[19]:
```



i1,i2,i3 are the 3 inputs, h1, h2 are the two hidden neurons, b is the bias (offset)

As you can see, some weights are large (two orders of magnitude larger then others) This is no good, since it makes the model unstable (i.e., small changes in some inputs may entail significant changes in the network, because of the large weights)

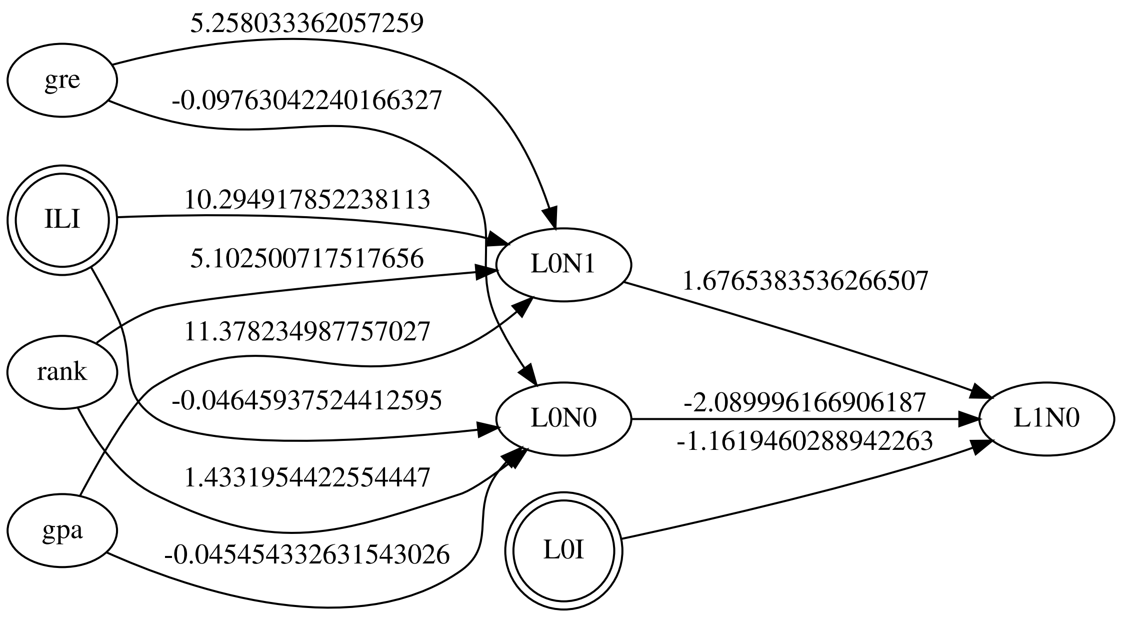
One way to avoid this is by regularizing the learning process:

```
In [20]: model_nnet = MLPClassifier(hidden_layer_sizes=[2],
                                    alpha=0.01,
                                    activation='logistic',
                                    max_iter=200,
                                    solver='lbfgs')

         model_nnet.fit(X_train,y_train);
```

notice the big difference

```
In [21]: model_nnet.coefs_
         model_nnet.intercepts_

Out[21]: [array([[-0.09763042,  5.25803336],
                 [-0.04545433, 11.37823499],
                 [ 1.43319544,  5.10250072]]), array([[-2.08999617],
                 [ 1.67653835]])]

Out[21]: [array([-0.04645938, 10.29491785]), array([-1.16194603])]

In [22]: graphMLP(Admis.columns[1:], model_nnet.coefs_, model_nnet.intercepts_)
```

```
Out[22]:
```



Now let's compute the training error

```
In [23]: pred = model_nnet.predict(X_train)
         (1-accuracy_score(y_train,pred))*100
```

```
Out[23]: 30.22388059701493
```

And the corresponding test error

```
In [24]: pred = model_nnet.predict(X_test)
         (1-accuracy_score(y_test,pred))*100
```

```
Out[24]: 24.242424242424242
```

We get 24.24%, so it seems that the MLP helps a little bit; however, we need to work harder

We are going to do the modelling in a principled way now. Using 10x10 CV to select the best combination of 'size' and 'decay'

Just by curiosity, let me show you that we can fit any dataset (in the sense of reducing the training error):

```
In [25]: model_nnet = MLPClassifier(hidden_layer_sizes=[30],
                                    alpha=0,
                                    activation='logistic',
                                    max_iter=500,
                                    solver='lbfgs')
         model_nnet.fit(X_train,y_train);
```

```
In [26]: pred = model_nnet.predict(X_train)

         confusion(y_train,pred, ['noadmit','admit'])
         (1-accuracy_score(y_train,pred))*100

Out[26]: Predicted  noadmit  admit
         Actual
         noadmit        180      3
         admit            4     81

Out[26]: 2.6119402985074647
```

And the corresponding test error

```
In [27]: pred = model_nnet.predict(X_test)
         confusion(y_test,pred, ['noadmit','admit'])
         (1-accuracy_score(y_test,pred))*100

Out[27]: Predicted  noadmit  admit
         Actual
         noadmit         60     30
         admit           28     14

Out[27]: 43.939393939393945
```

that's it: we got a training error around 2%, but it is illusory ... the test error is larger than before (around 40%); The relevant comparison is between 2% and 40%, this large gap is an indication of overfitting

Scikit learn has specific functions for parameter search so we can tune the parameters of a model.

We are going to use a grid search that will use a cross validation strategy to evaluate the results for each combination of parameters. At the end the best model will be returned

In order to find the best network architecture, we are going to explore two methods:

1. Explore different numbers of hidden units in one hidden layer, with no regularization
2. Fix a large number of hidden units in one hidden layer, and explore different regularization values (recommended)

doing both (explore different numbers of hidden units AND regularization values) is usually a waste of computing resources (but notice that it would admit it)

Let's start with 1.

set desired sizes

```
In [28]: sizes = [2*i for i in range(1,11)]
         sizes

Out[28]: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

```
In [29]: model_nnet = MLPClassifier(alpha=0,
                                    activation='logistic',
                                    max_iter=500,
                                    solver='lbfgs')
         trc = GridSearchCV(estimator=model_nnet,
                            param_grid ={'hidden_layer_sizes':sizes},
                            cv=10,
                            return_train_score=True)
         model_10CV = trc.fit(X_train,y_train)
         model_10CV.best_score_
```

```
Out[29]: 0.6716417910447762
```

```
In [30]: pd.DataFrame(model_10CV.cv_results_).\
                 loc[:,['param_hidden_layer_sizes','mean_test_score',
                        'std_test_score','rank_test_score' ]]
```

```
Out[30]:   param_hidden_layer_sizes  mean_test_score  std_test_score  rank_test_score
         0                        2            0.657           0.049                2
         1                        4            0.623           0.081                6
         2                        6            0.631           0.064                4
         3                        8            0.672           0.057                1
         4                       10            0.631           0.072                4
         5                       12            0.646           0.115                3
         6                       14            0.593           0.087                8
         7                       16            0.552           0.086               10
         8                       18            0.604           0.070                7
         9                       20            0.582           0.066                9
```

and the best model found

```
In [31]: model_10CV.best_params_
```

```
Out[31]: {'hidden_layer_sizes': 8}
```

The results are quite disappointing ...
Now method 2.

```
In [32]: decays = [10**i for i in np.arange(-3,0,0.1)]
         decays
```

```
Out[32]: [0.001,
          0.0012589254117941675,
          0.001584893192461114,
          0.0019952623149688807,
          0.002511886431509582,
          0.0031622776601683824,
          0.003981071705534978,
          0.00501187233627273,
          0.006309573444801942,
          0.00794328234724283,
          0.010000000000000021,
          0.012589254117941701,
          0.015848931924611172,
          0.01995262314968885,
          0.025118864315095874,
          0.03162277660168389,
          0.03981071705534985,
          0.0501187233627274,
          0.06309573444801955,
          0.07943282347242846,
          0.10000000000000041,
          0.12589254117941726,
          0.15848931924611206,
          0.1995262314968889,
          0.25118864315095923,
          0.31622776601683955,
          0.39810717055349937,
          0.501187233627275,
          0.6309573444801969,
          0.7943282347242863]
```

WARNING: this takes a few minutes

```
In [33]: model_nnet = MLPClassifier(alpha=0,
                                    activation='logistic',
                                    hidden_layer_sizes=20,
                                    max_iter=500,
                                    solver='lbfgs')
         trc = GridSearchCV(estimator=model_nnet,
                            param_grid ={'alpha':decays},
                            cv=10,
                            return_train_score=True)
         model_10CV = trc.fit(X_train,y_train)
         model_10CV.best_score_

Out[33]: 0.6791044776119403

In [34]: pd.DataFrame(model_10CV.cv_results_).\
                 loc[:,['param_alpha','mean_test_score', 'std_test_score','rank_test_score' ]]

Out[34]:    param_alpha  mean_test_score  std_test_score  rank_test_score
         0        0.001            0.582           0.093               23
         1      0.00126            0.608           0.037               13
         2      0.00158            0.597           0.055               19
         3        0.002            0.582           0.095               23
         4      0.00251            0.593           0.071               21
         5      0.00316            0.575           0.092               27
```

```
6     0.00398        0.556        0.064          29
7     0.00501        0.582        0.068          23
8     0.00631        0.608        0.085          13
9     0.00794        0.601        0.108          17
10       0.01        0.578        0.084          26
11     0.0126        0.590        0.091          22
12     0.0158        0.545        0.059          30
13       0.02        0.601        0.099          17
14     0.0251        0.597        0.097          19
15     0.0316        0.608        0.084          13
16     0.0398        0.575        0.106          27
17     0.0501        0.608        0.093          13
18     0.0631        0.646        0.086          11
19     0.0794        0.675        0.068           2
20        0.1        0.664        0.090           7
21      0.126        0.638        0.090          12
22      0.158        0.660        0.087           8
23        0.2        0.675        0.054           2
24      0.251        0.675        0.055           2
25      0.316        0.679        0.056           1
26      0.398        0.660        0.039           8
27      0.501        0.675        0.047           2
28      0.631        0.668        0.039           6
29      0.794        0.660        0.041           8
```

```
In [35]: model_10CV.best_params_
```

```
Out[35]: {'alpha': 0.31622776601683955}
```

The results are a bit better; we should choose the model with the lowest 10x10CV error overall, in this case it corresponds to 20 hidden neurons, with a decay of 0.3162278

So what remains is to predict the test set with our final model

```
In [36]: pred = model_10CV.predict(X_test)

         (1-accuracy_score(y_test,pred))*100
```

```
Out[36]: 32.57575757575758
```

We get ~32% after all this work; it seems that the information in this dataset is not enough to accurately predict admittance. Note that ...

... upon looking at the confusion matrix for the predictions ...

```
In [37]: confusion(y_test,pred, ['noadmit','admit'])
```

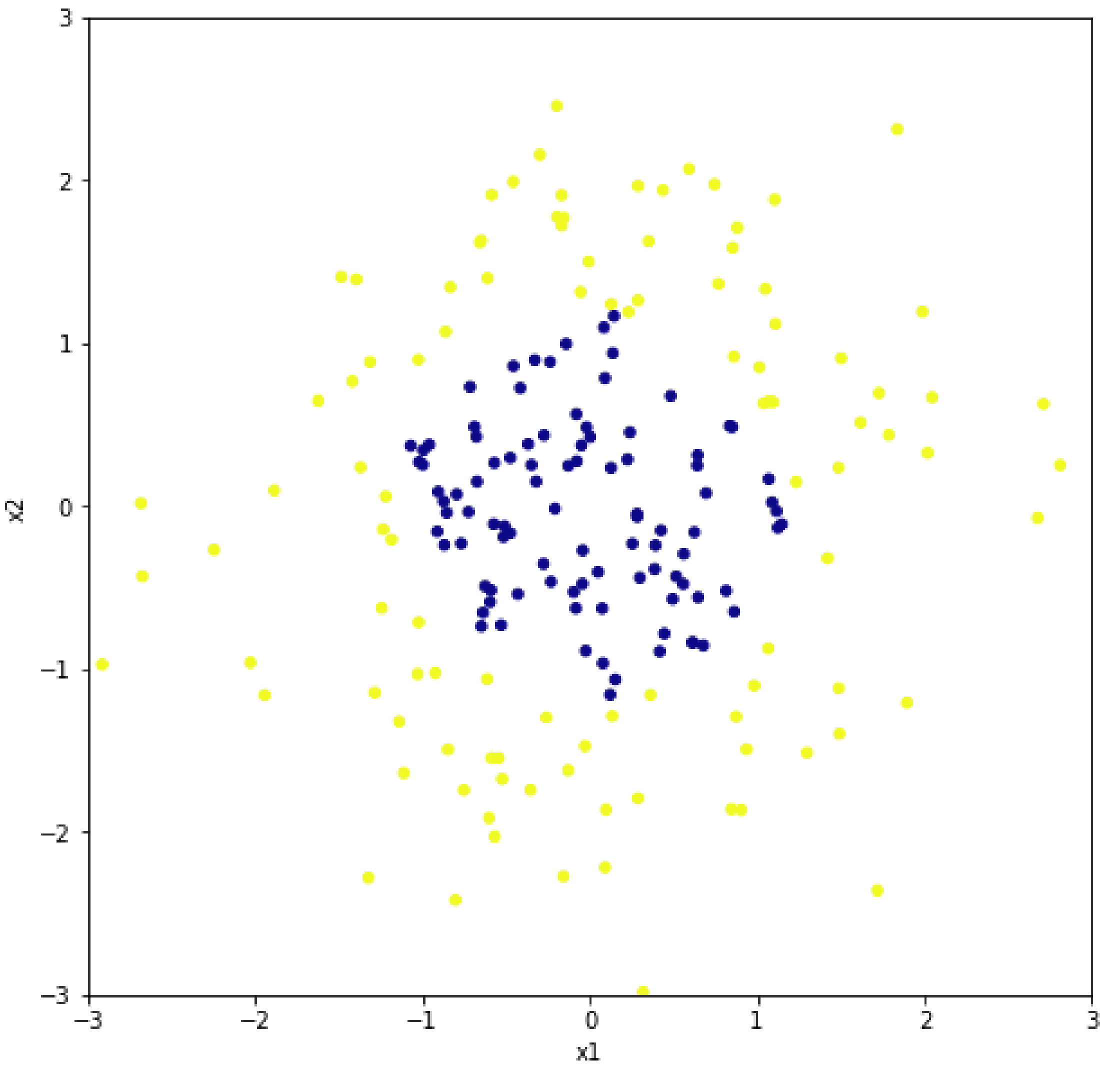
```
Out[37]: Predicted  noadmit  admit
         Actual
         noadmit         84      6
         admit           37      5
```

it clearly suggests that quite a lot of people is getting accepted when they should not, given their gre, gpa and rank It is very likely that other (subjective?) factors are being taken into account, that are not in the dataset

---

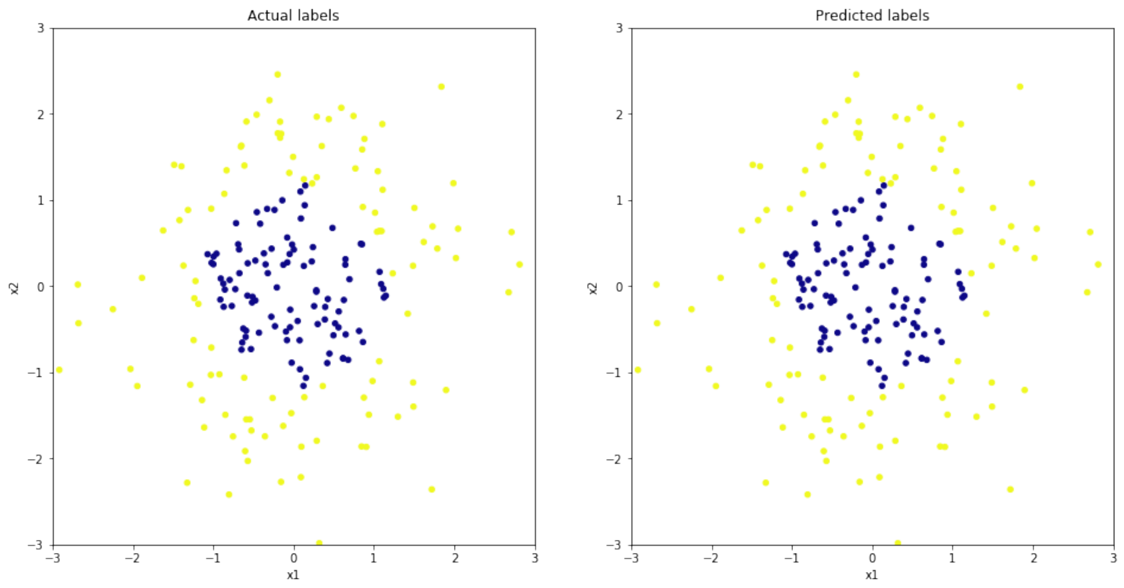
1.1 Multilayer Perceptron Example 2: circular artificial 2D data

```
In [38]: np.random.seed(3)
         p = 2
         N = 200

         x = normal(size=N*p).reshape(-1,2)
         y = (x[:,0]**2+x[:,1]**2 > 1.4).astype('int')

         mydata = pd.DataFrame({'x1':x[:,0],'x2':x[:,1],'y':y})
         mydata

         mydata.plot.scatter(x='x1',y='x2',c='y',
                             colormap='plasma',
                             figsize=(8,8),
                             colorbar=False,
                             xlim=[-3,3],
                             ylim=[-3,3]);
```



Let's use one hidden layer, 3 hidden units, no regularization and the error function "cross-entropy" In this case it is not necessary to standardize because they variables already are (they have been generated from a distribution with mean 0 and standard deviation 1).

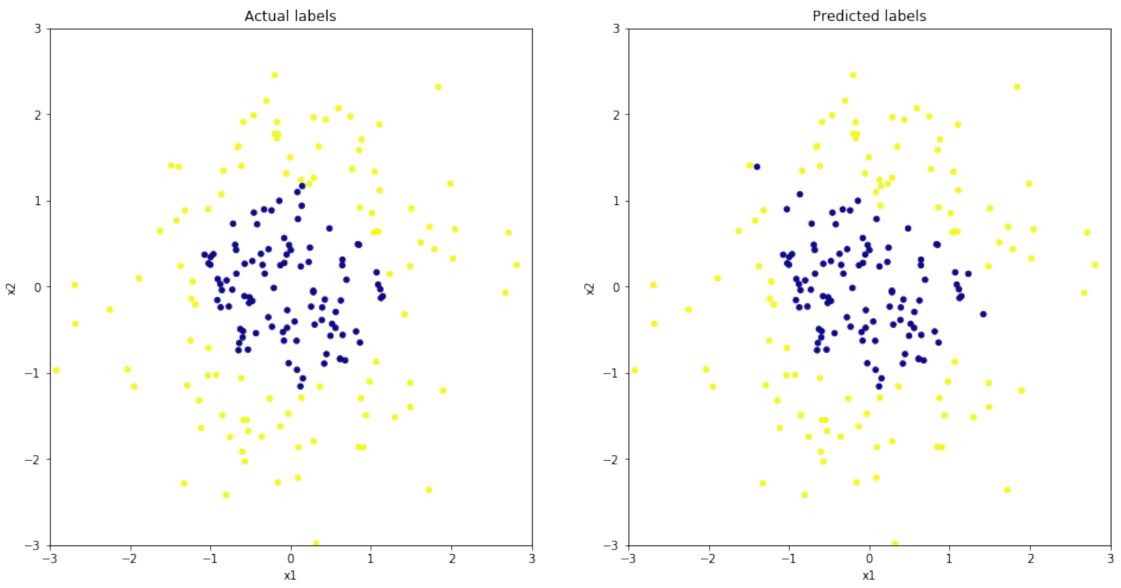
```
In [39]: np.random.seed(30)
         nn1= MLPClassifier(alpha=0,
                            activation='logistic',
                            hidden_layer_sizes=3,
                            max_iter=2000,
                            solver='lbfgs')
         nn1.fit(mydata.loc[:,'x1':'x2'],mydata.y);
         mydata['yhat']=nn1.predict(mydata.loc[:,'x1':'x2']);

In [40]: fig, (ax1,ax2) = plt.subplots(1,2,figsize=(16,8))
         mydata.plot.scatter(x='x1',y='x2',c='y',
                             colormap='plasma',
                             colorbar=False,xlim=[-3,3],
                             ylim=[-3,3],
                             ax=ax1,
                             title='Actual labels')
         mydata.plot.scatter(x='x1',y='x2',c='yhat',
                             colormap='plasma',
                             colorbar=False,
                             xlim=[-3,3],
                             ylim=[-3,3],
                             ax=ax2,
                             title='Predicted labels');
```



```
In [41]: confusion(mydata.y,mydata.yhat,[0,1])

Out[41]: Predicted   0    1
         Actual
         0          96    0
         1           0  104
```

Excellent, indeed
Let's execute it again, this time wth a different random seed

```
In [42]: np.random.seed(3)
         nn1= MLPClassifier(alpha=0,
                               activation='logistic',
                               hidden_layer_sizes=3,
                               max_iter=2000,
                               solver='lbfgs')
         nn1.fit(mydata.loc[:,'x1':'x2'],mydata.y);
         mydata['yhat']=nn1.predict(mydata.loc[:,'x1':'x2']);
         fig, (ax1,ax2) = plt.subplots(1,2,figsize=(16,8))
         mydata.plot.scatter(x='x1',y='x2',c='y',
                             colormap='plasma',
                             colorbar=False,
                             xlim=[-3,3],
                             ylim=[-3,3],
                             ax=ax1,
                             title='Actual labels')
         mydata.plot.scatter(x='x1',y='x2',c='yhat',
                             colormap='plasma',
                             colorbar=False,
                             xlim=[-3,3],
                             ylim=[-3,3],
                             ax=ax2,
                             title='Predicted labels');
```



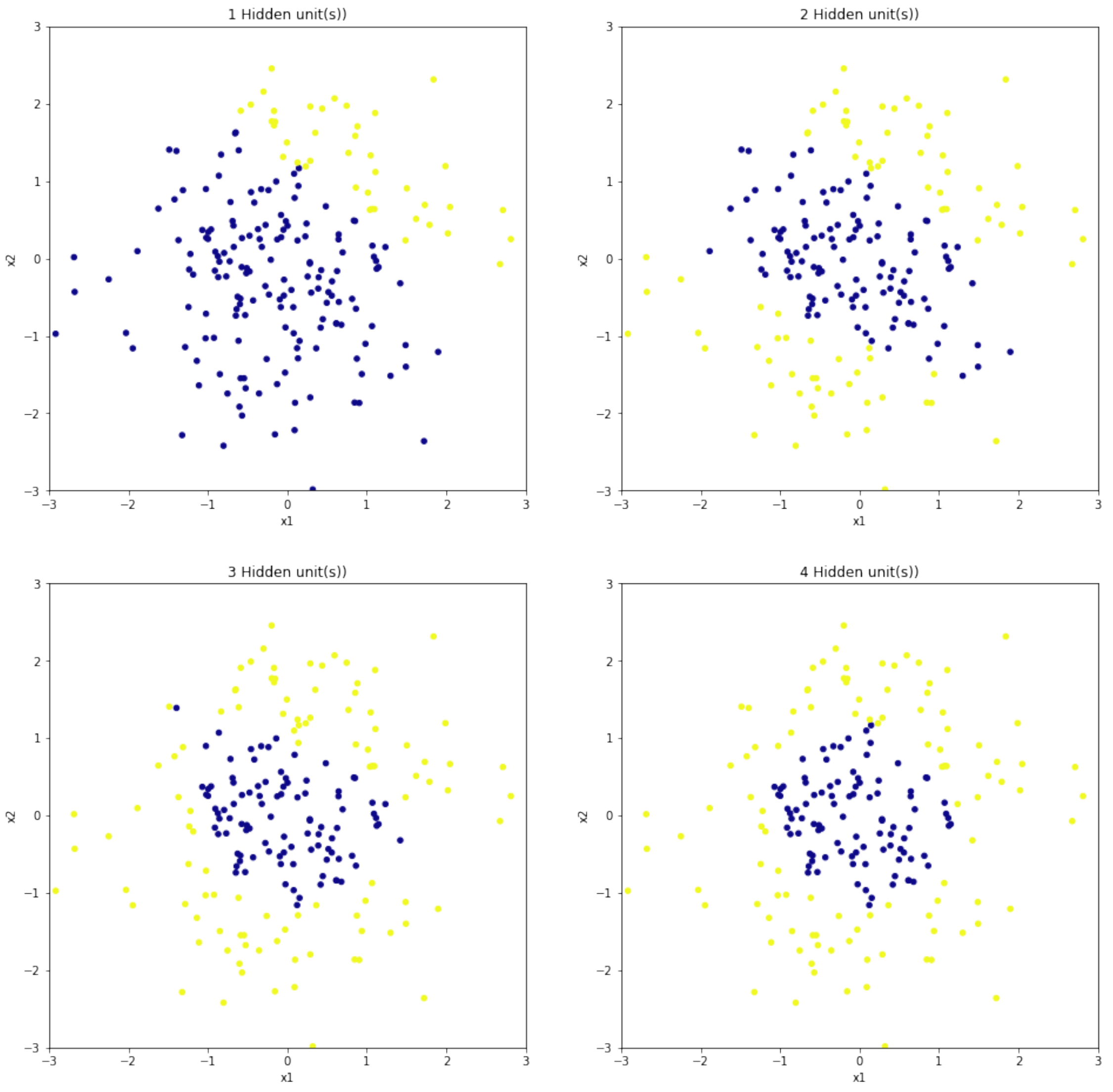
```
In [43]: confusion(mydata.y,mydata.yhat, [0,1])
```

```
Out[43]: Predicted   0   1
         Actual
         0          93   3
         1           5  99
```

we see that the optimizer does not always find a good solution, even with the right number of neurons

How many hidden units do we need?

```
In [44]: fig, ax = plt.subplots(2,2,figsize=(16,16))
         for i,a in zip(range(1,5),ax.ravel()):
             np.random.seed(3)
             nn1= MLPClassifier(alpha=0,
                                activation='logistic',
                                hidden_layer_sizes=i,
                                max_iter=2000,
                                solver='lbfgs')
             nn1.fit(mydata.loc[:,'x1':'x2'],mydata.y);
             mydata['yhat']=nn1.predict(mydata.loc[:,'x1':'x2']);
             mydata.plot.scatter(x='x1',y='x2',c='yhat',
                                 colormap='plasma',
                                 colorbar=False,
                                 xlim=[-3,3],
                                 ylim=[-3,3],
                                 ax=a,
                                 title='%d Hidden unit(s))'%i)
         0;
```



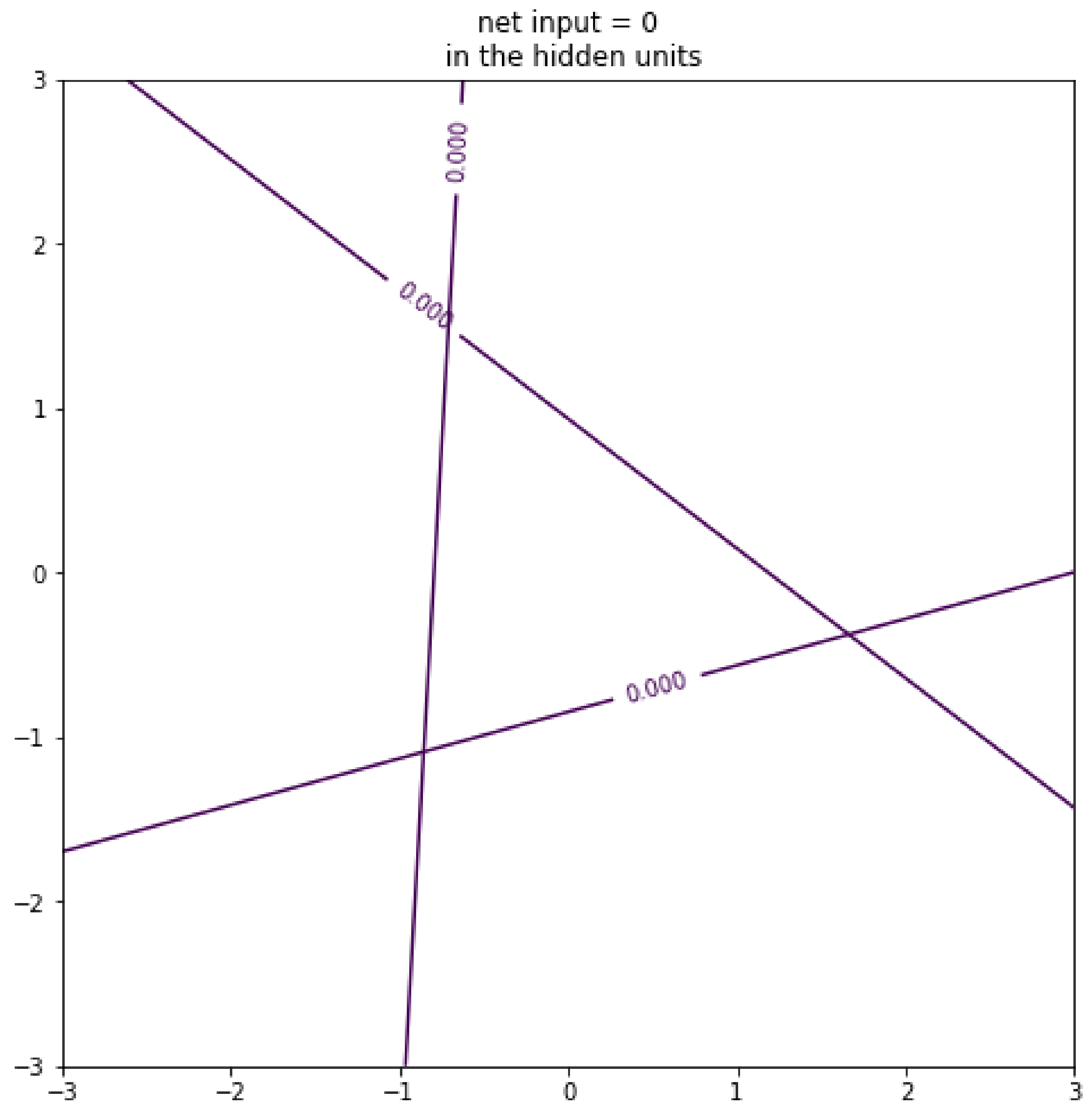
Let’s find out which function has been learned exactly, with 3 units

```
In [45]: np.random.seed(3)
         nn1= MLPClassifier(alpha=0.01,activation='logistic',
                            hidden_layer_sizes=3,
                            max_iter=3000,solver='lbfgs')
         nn1.fit(mydata.loc[:,'x1':'x2'],mydata.y);
```

```
In [46]: x1grid = np.linspace(-3.0, 3.0, 200)
         x2grid = np.linspace(-3.0, 3.0, 200)
         X,Y = np.meshgrid(x1grid, x2grid)
         Ones = np.ones((200,200))
```

```
In [47]: v1 = (X*nn1.coefs_[0][0,0] + Y * nn1.coefs_[0][1,0] + Ones * nn1.intercepts_[0][0])
         v2 = (X*nn1.coefs_[0][0,1] + Y * nn1.coefs_[0][1,1] + Ones * nn1.intercepts_[0][1])
         v3 = (X*nn1.coefs_[0][0,2] + Y * nn1.coefs_[0][1,2] + Ones * nn1.intercepts_[0][2])
```

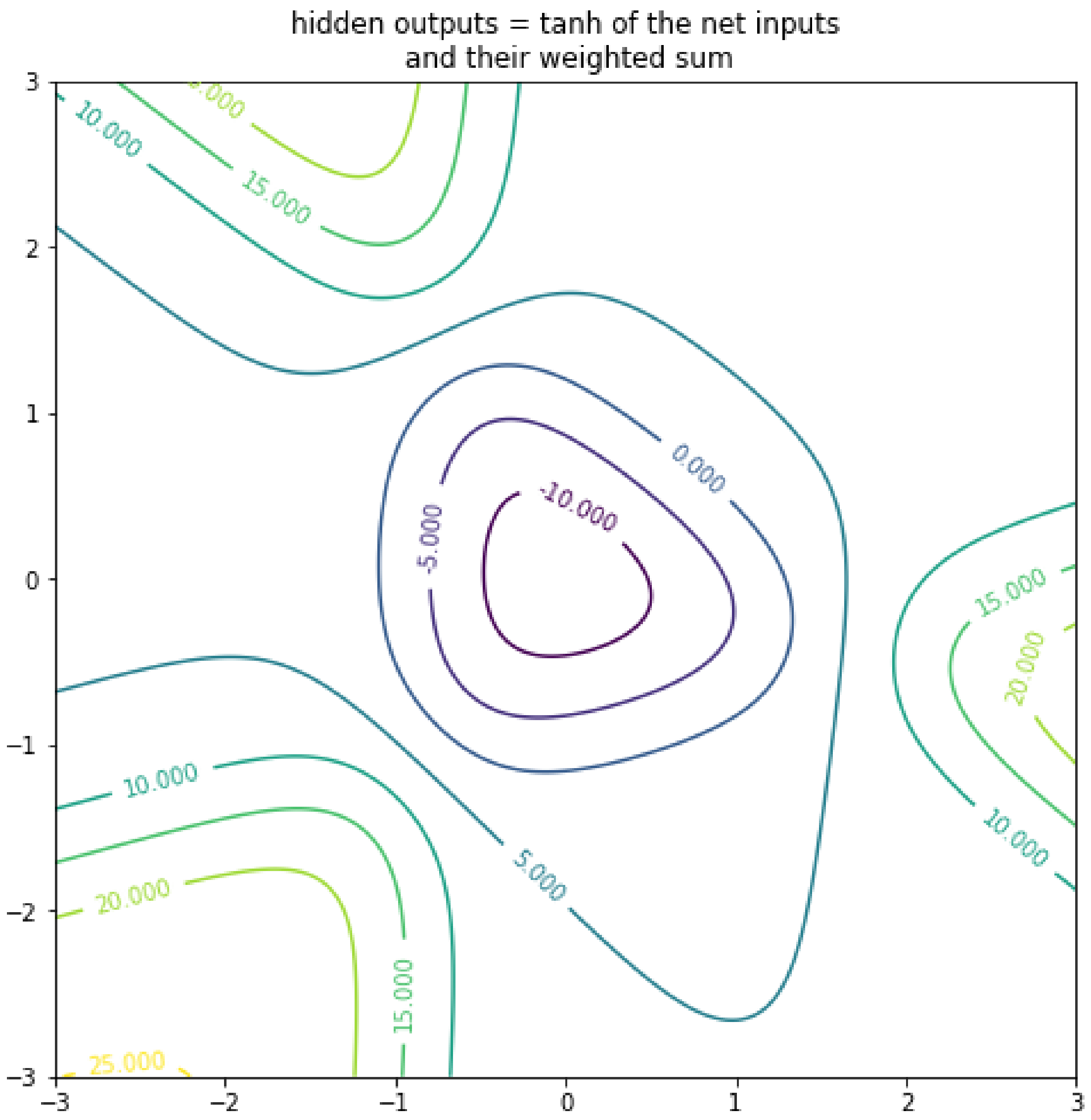
```
In [48]: fig, ax = plt.subplots(figsize=(8,8))
         CS = ax.contour(X, Y, v1,levels=0)
         plt.clabel(CS, inline=1, fontsize=10)
         CS = ax.contour(X, Y, v2,levels=0)
         plt.clabel(CS, inline=1, fontsize=10)
         CS = ax.contour(X, Y, v3,levels=0)
         plt.clabel(CS, inline=1, fontsize=10)
         plt.title('net input = 0\n in the hidden units');
```



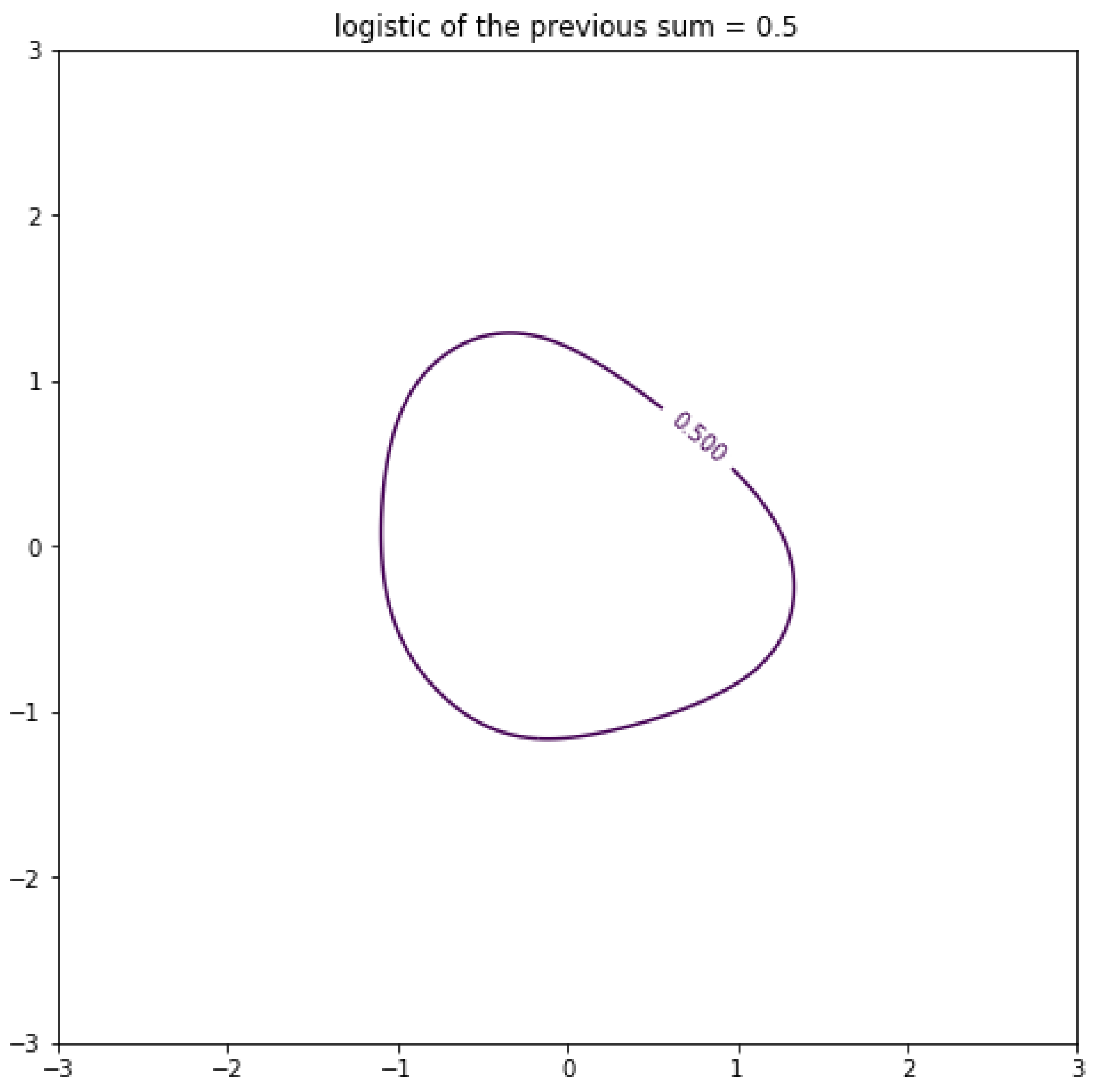
this is the logistic function, used by nnet() for the hidden neurons, and for the output neurons in two-class classification problems

```
In [49]: z = nn1.intercepts_[1][0] + nn1.coefs_[1][0] * logistic(v1) +\
             nn1.coefs_[1][1] * logistic(v2) + nn1.coefs_[1][2] * logistic(v3)
```

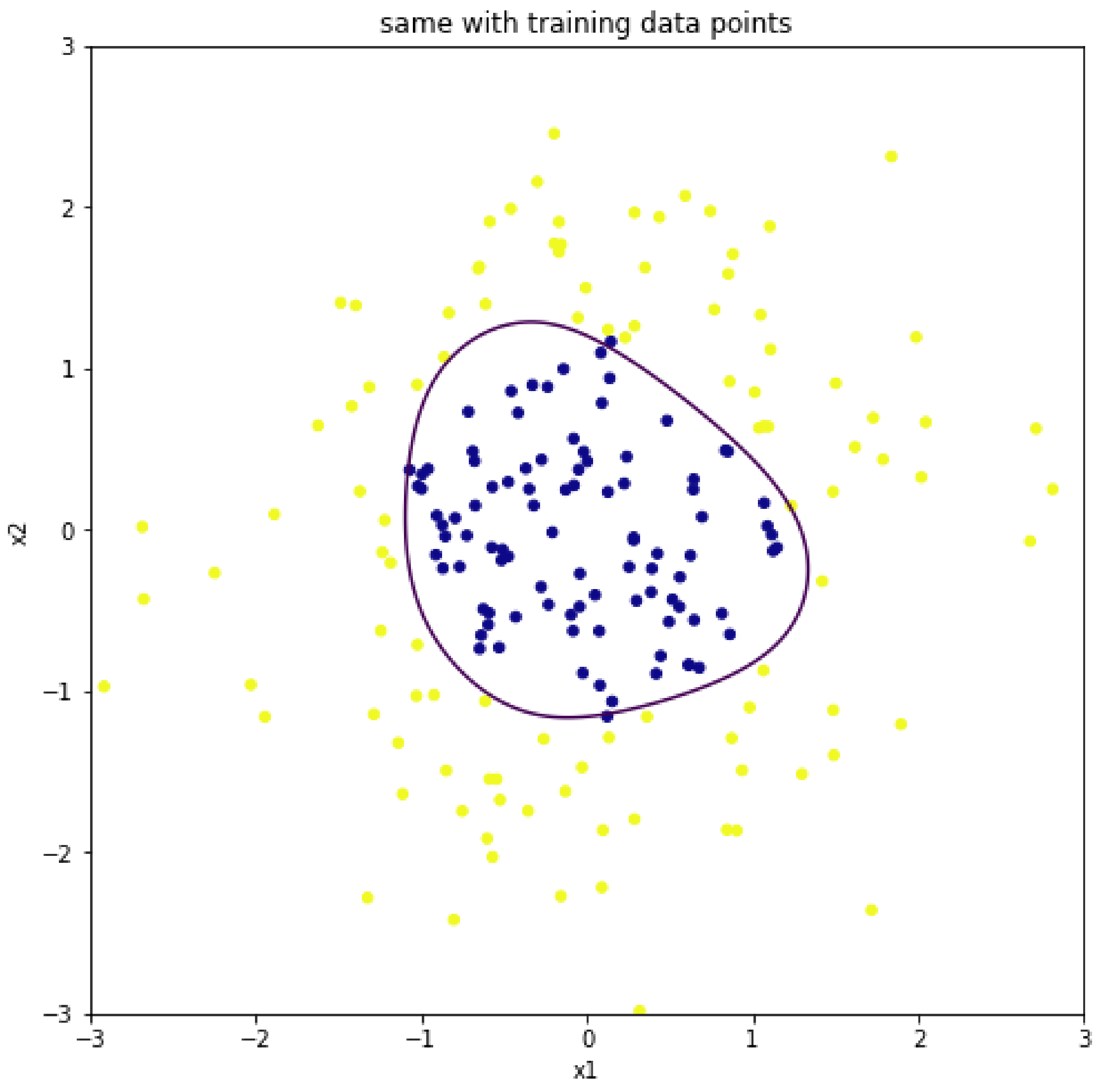
```
fig, ax = plt.subplots(figsize=(8,8))

CS = ax.contour(X, Y, z)
plt.clabel(CS, fontsize=10)
plt.title('hidden outputs = tanh of the net inputs\n and their weighted sum');
```



```
In [50]: fig, ax = plt.subplots(figsize=(8,8))
         CS = ax.contour(X, Y, logistic(z),levels=0.5)
         plt.clabel(CS, fontsize=10)
         plt.title('logistic of the previous sum = 0.5');
```



```
In [51]: ax =mydata.plot.scatter(x='x1',y='x2',c='y',
                                 colormap='plasma',figsize=(8,8),
                                 colorbar=False,
                                 xlim=[-3,3],ylim=[-3,3],
                                 title='same with training data points')
         CS = ax.contour(X, Y, logistic(z),levels=0.5)
```



If you prefer a more visual demo of how ANN work, you can play with the Google's tensorflow playground http://playground.tensorflow.org

---

1.2 Radial Basis Function Network Example: regression of a 1D function

We are going to do all the computations "by hand"

Let us depart from the following function in the (a,b) interval

```
In [52]: def myf(x): return (1 + x - 2*x**2) * np.exp(-x**2)
```

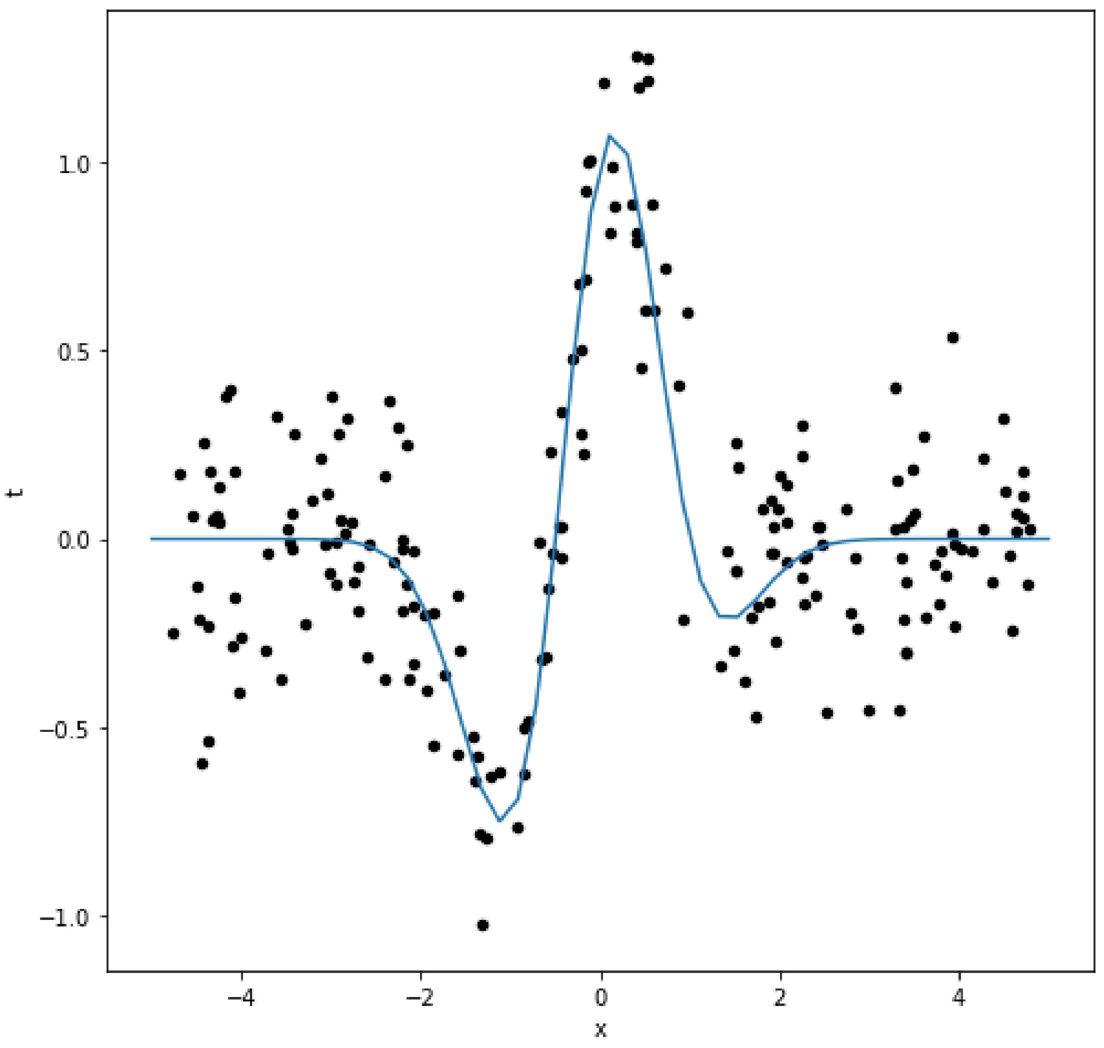
We are going to model this function in the interval (-5,5)

```
In [53]: np.random.seed(3)

         N = 200
         a=-5
         b=5

         def myf_data(N,a,b):
             x = uniform(a,b,N)
             return pd.DataFrame({'x':x,
                                  't': myf(x) + normal(scale=0.2,size=N)})
         d = myf_data(N,a,b)

         d.describe()
```

```
Out[53]:              x        t
         count  200.000  200.000
         mean    -0.020    0.031
         std      2.837    0.407
         min     -4.760   -1.023
         25%     -2.449   -0.207
         50%     -0.170   -0.015
         75%      2.280    0.188
         max      4.780    1.284
```

The black points are the data, the blue line is the true underlying function

```
In [54]: ax = d.plot.scatter(x='x',y='t',figsize=(8,8),c='black');
         ax.plot(np.linspace(-5,5), myf(np.linspace(-5,5)));
```



Create a large test data too for future use; notice that the generation mechanism is the same

```
In [55]: N_test=2000
         d_test = myf_data(N_test,a,b)
```

Function to compute a PHI (N x M) design matrix, without the Phi_0(x) = 1 column; m.i, h.i are the centers and variances (sigmas) of the neurons, respectively

```
In [56]: def PHI(x,m_i,h_i):
           N = x.shape[0]
           M = m_i.shape[0]
           phis = np.zeros((M,N))
           for i in range(M):
             phis[i:] = np.exp(-(x - m_i[i])**2/(2*h_i[i]))
           return phis.T
```

We find the centers and variances for each neuron using k-means; since this clustering algorithm is non-deterministic (because the initial centers are random), we do it 'NumKmeans' times

```
In [57]: NumKmeans = 10
```

We set a rather large number of hidden units (= basis functions) M as a function of data size (the sqrt is just a heuristic!) because we are going to try different regularizers

```
In [58]: M = int(np.floor(np.sqrt(N)))
```

```
In [59]: m = []
         h = []
         data_Kmeans=np.array(d.x).reshape(-1, 1)

         for j in range(1,NumKmeans+1):
             # Find the centers m.i with k-means
             km = KMeans(n_clusters=M)
             km.fit(data_Kmeans)
             m.append(km.cluster_centers_[:,0])

             # Obtain the variances h_i as a function of the m_i
             th = np.zeros(M)
             for i in range(M):
                 nind = data_Kmeans[km.labels_==i].shape[0]
                 th[i] = np.sum(np.abs(data_Kmeans[km.labels_==i]-m[j-1][i]))/nind
                 if th[i]==0:
                     th[i]=1
             h.append(th)
         0;
```

Now for each k-means we get the hidden-to-output weights by solving a regularized least-squares problem (standard ridge regression), very much as we did in previous labs

The difference is that now we perform ridge regression on the PHI matrix (that is, on the new regressors given by the hidden neurons), not on the original inputs ...

... and find the best lambda with using GCV across all choices of basis functions (the NumKmeans clusterings)

```
In [60]: lambdes = 10**np.arange(-3,1.5,0.1)

         errors = []
         bestLambdes = []

         for num in range(1,NumKmeans+1):
             m_i = m[num-1]
             h_i = h[num-1]
             myPHI = PHI(d.x,m_i,h_i)
             ridgecv = RidgeCV(alphas=lambdes,normalize=True)
             ridgecv.fit(myPHI,d.t)
             ridge=Ridge(alpha=ridgecv.alpha_,normalize=True).fit(myPHI,d.t)
```

```
            errors.append(ridge.score(myPHI,d.t))
            bestLambdes.append(ridgecv.alpha_)

        0;
```

Now we obtain the best model among the tested ones

```
In [61]: bestIndex = np.argmin(errors)
         bestLambda =bestLambdes[bestIndex]
         m_i = m[bestIndex]
         h_i = h[bestIndex]
```

we see that this problem needs a lot of regularization! This makes sense if you take a look at how the data is generated (the previous plot): the noise level is very high relative to the signal

We also see that the best lambda fluctuates (since the data changes due to the clustering, but the order of magnitude is quite stable

```
In [62]: bestLambdes
```

```
Out[62]: [0.0031622776601683824,
          0.00794328234724283,
          0.01995262314968885,
          0.01995262314968885,
          0.015848931924611172,
          0.01995262314968885,
          0.0050118723362727273,
          0.01995262314968885,
          0.00794328234724283,
          0.01995262314968885]
```

We now create the final model:

```
In [63]: myRBF=Ridge(alpha=bestLambda,normalize=True).fit(PHI (d.x,m_i,h_i),d.t)
```

these are the final hidden-to-output weights: note how small they are (here is where we regularize)

```
In [64]: weights = np.hstack((np.array(myRBF.intercept_), myRBF.coef_))
         pd.DataFrame(weights, index=['w_%d'%i for i in range(weights.shape[0])])
```

```
Out[64]:          0
         w_0  -0.113
         w_1   0.200
         w_2   0.131
         w_3   0.102
         w_4  -0.376
         w_5   0.062
         w_6   0.880
         w_7   0.074
```

```
w_8   0.136
w_9   0.768
w_10  0.050
w_11  0.075
w_12 -0.075
w_13 -0.489
w_14 -0.148
```

It remains to calculate the prediction on the test data

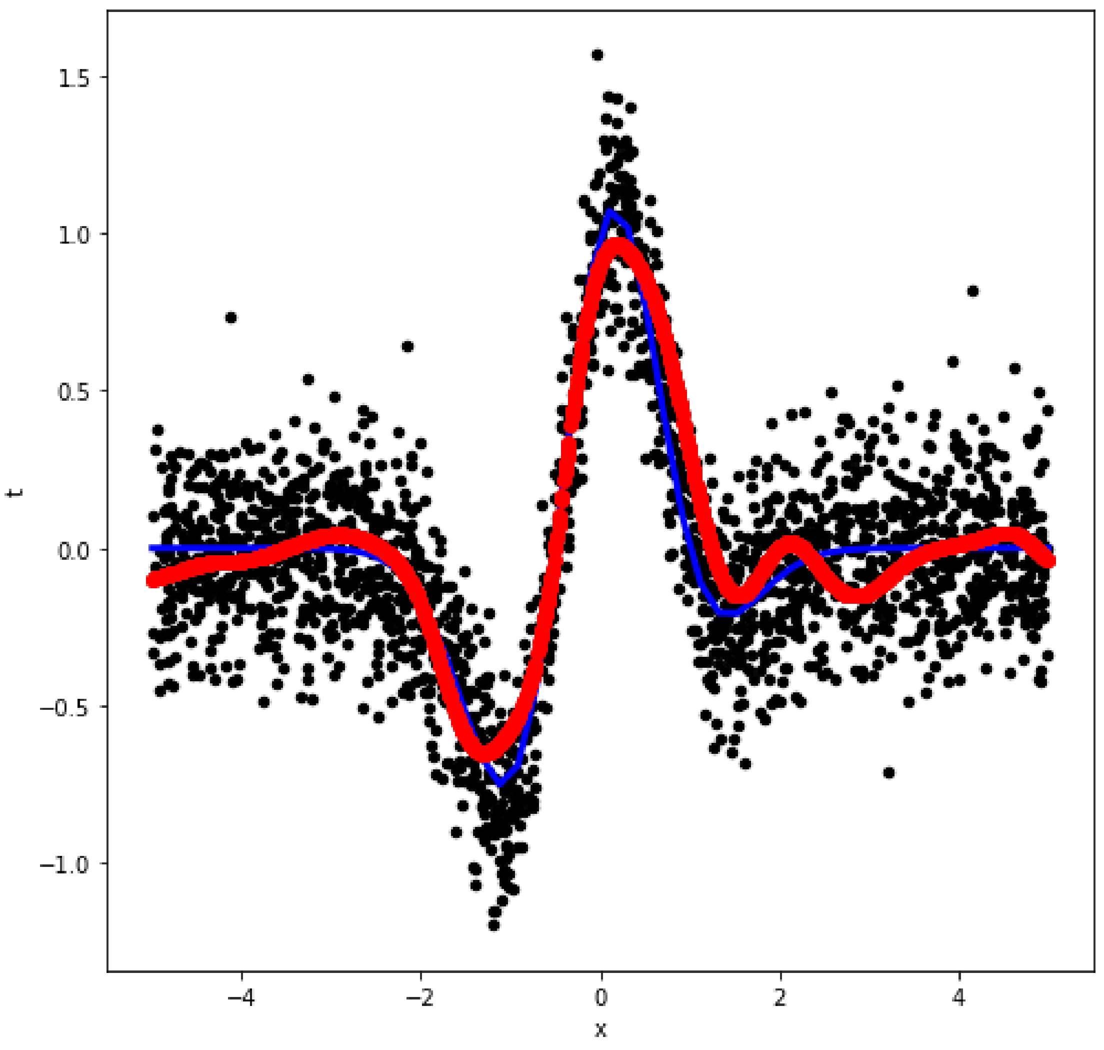
```
In [65]: test_PHI = np.hstack((np.ones(N_test).reshape(-1,1),PHI(d_test.x,m_i,h_i)))
         y = test_PHI @ weights
```

And now the normalized error of this prediction

```
In [66]: errorsTest = np.sqrt(np.sum((d_test.t - y)**2)/((N_test-1)*np.var(d_test.t)))
         errorsTest

Out[66]: 0.5234342003825635
```

Much better if we plot everything
Test data in black, Red data are the predictions and the blue line is the underlying function

```
In [67]: ax = d_test.plot.scatter(x='x',y='t',figsize=(8,8),c='black');
         ax.plot(np.linspace(-5,5), myf(np.linspace(-5,5)),'b-',linewidth=3);

         ax.plot(d_test.x,y,'ro');
```



Currently we are in the era of neural networks applications and there are very good neural network libraries for python (and other languages) like Tensorflow/Keras (from Google), pytorch (from Facebook) or MxNet (from Amazon) amonng others.

They are more complex but are prepared for large quantities of data using GPU training and are used for many applications in artificial intelligence like computer vision and natural language understanding