P.1. mCLESS

Note: Some **machine learning** algorithms are considered as **black boxes**, because

- the models are sufficiently complex and
- they are not straightforwardly interpretable to humans.

Lack of interpretability in predictive models can undermine trust in those models, especially in health care, in which so many decisions are – literally – life and death issues [57].

Project Objectives

- Develop a family of **interpretable** machine learning algorithms.
  - We will develop algorithms involving least-squares formulation.
  - The family is called the *Multi-Class Least Error Square Sum* (**mCLESS**).
- Compare with traditional methods, using public domain datasets.

P.1.1. Review: Simple classifiers

The **Perceptron** [62] (or Adaline) is the simplest artificial neuron that makes decisions for datasets of two classes by *weighting up evidence*.

- Inputs: feature values $\mathbf{x} = [x_1, x_2, \cdots, x_d]$
- Weight vector and bias: $\mathbf{w} = [w_1, w_2, \cdots, w_d]^T$, w_0
- Net input:
$$z = w_0 + w_1\,x_1 + w_2\,x_2 + \cdots + w_d\,x_d \tag{P.1.1}$$
- *Activation*:
$$\phi(z) = \begin{cases} 1, & \text{if } z \ge \theta \\ 0, & \text{otherwise}, \end{cases} \tag{P.1.2}$$
where θ is a threshold. When the logistic sigmoid function is chosen for the **activation function**, i.e., $\phi(z) = 1/(1+e^{-z})$, the resulting classifier is called the **Logistic Regression**.

Remark P.1. Note that the net input in (P.1.1) represents a **hyperplane** in $\mathbb{R}^d$.

- More complex neural networks can be built, stacking the simple artificial neurons as building blocks.
- Machine learning (ML) is to train weights from datasets of an arbitrary number of classes.
  - The weights must be trained in such a way that *data points in a class are heavily weighted by the corresponding part of weights.*
- The **activation function** is incorporated in order
  (a) **to keep the net input restricted to a certain limit** as per our requirement and, more importantly,
  (b) **to add nonlinearity** to the network.

P.1.2. The mCLESS classifier

Here we present a new classifier which is based on a least-squares formulation and able to classify datasets having arbitrary numbers of classes. Its nonlinear expansion will also be suggested.

Two-layer Neural Networks

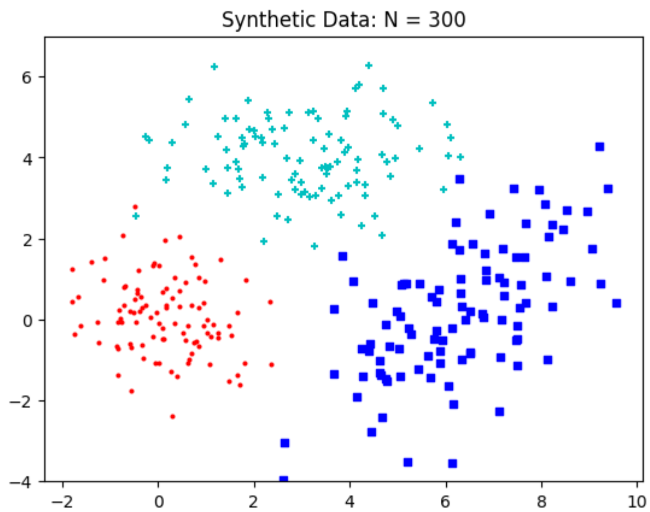

Figure P.1: A synthetic data of three classes.

- In order to describe the proposed algorithm effectively, we exemplify a synthetic data of three classes, as shown in Figure P.1, in which each class has 100 points.
- A point in the c-th class is expressed as

$$\mathbf{x}^{(c)} = [x_1^{(c)}, x_2^{(c)}] = [x_1, x_2, c] \quad c = 0, 1, 2,$$

  where the number in () in the superscript denotes the class that the point belongs.
- Let's consider an artificial neural network of the identity activation and no hidden layer, for simplicity.

A set of weights can be trained in a way that ***points in a class are heavily weighted by the corresponding part of weights***, i.e.,

$$w_0^{(j)} + w_1^{(j)} x_1^{(i)} + w_2^{(j)} x_2^{(i)} = \delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{P.1.3}$$

where δ_{ij} is called the Kronecker delta and $w_0^{(j)}$ is a bias for the class j.

- Thus, for neural networks which classify a dataset of C classes with points in $\mathbb{R}^d$, the weights to be trained must have dimensions $(d+1) \times C$.
- The weights can be computed by the least-squares method.
- We will call the algorithm the *Multi-Class Least Error Square Sum* (**mCLESS**).

Training in the mCLESS

- **Dataset**: We express the dataset $\{X, \mathbf{y}\}$ used for Figure P.1 by

$$X = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ \vdots & \vdots \\ x_{N1} & x_{N2} \end{bmatrix} \in \mathbb{R}^{N\times 2}, \quad \mathbf{y} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{bmatrix}, \tag{P.1.4}$$

  where $c_i \in \{0, 1, 2\}$, the class number.

- **The algebraic system**: It can be formulated using (P.1.3).

  – Define the **information matrix**:

$$A = \begin{bmatrix} 1 & x_{11} & x_{12} \\ 1 & x_{21} & x_{22} \\ & \vdots & \\ 1 & x_{N1} & x_{N2} \end{bmatrix} \in \mathbb{R}^{N\times 3}. \tag{P.1.5}$$

  Note. The information matrix can be made using

```
A = np.column_stack((np.ones([N,]),X))
```

– The **weight matrix** to be learned is:

$$W = [\mathbf{w}^{(0)}, \mathbf{w}^{(1)}, \mathbf{w}^{(2)}] = \begin{bmatrix} w_0^{(0)} & w_0^{(1)} & w_0^{(2)} \\ w_1^{(0)} & w_1^{(1)} & w_1^{(2)} \\ w_2^{(0)} & w_2^{(1)} & w_2^{(2)} \end{bmatrix}, \tag{P.1.6}$$

where the j-th column weights heavily points in the j-th class.

– Define the **source matrix**:

$$B = [\delta_{c_i,j}] \in \mathbb{R}^{N\times 3}. \tag{P.1.7}$$

For example, if the i-th point is in Class 0, then the i-th row of B is $[1, 0, 0]$.

- Then the **multi-column least-squares** (MC-LS) problem reads

$$\widehat{W} = \arg\min_W ||AW - B||^2, \tag{P.1.8}$$

which can be solved by the **method of normal equations**:

$$(A^T A)\,\widehat{W} = A^T B, \qquad A^T A \in \mathbb{R}^{3\times 3}. \tag{P.1.9}$$

- **The output of training**: The weight matrix $\widehat{W}$.

Note: The normal matrix $A^T A$ is occasionally singular, particularly for small datasets. In the case, the MC-LS problem can be solved using the **singular value decomposition (SVD)**.

Prediction in the mCLESS

The prediction step in the mCLESS is quite simple:

(a) Let $[x_1, x_2]$ be a new point.

(b) Compute

$$[1, x_1, x_2]\,\widehat{W} = [p_0, p_1, p_2], \qquad \widehat{W} \in \mathbb{R}^{3\times 3}. \tag{P.1.10}$$

Note. Ideally, if the point $[x_1, x_2]$ is in class j, then p_j is near 1, while others would be near 0. Thus p_j is the largest.

(c) Decide the class c:

$$\texttt{c} = \texttt{np.argmax}([p_0, p_1, p_2], \texttt{axis} = 1). \tag{P.1.11}$$

Experiment **P.2. mCLESS, with a Synthetic Dataset**

- As a preprocessing, the dataset X is scaled column-wisely so that the maximum value in each column is 1 in modulus.
- The training is carried out with randomly selected 70% the dataset.
- The output of training, $\widehat{W}$, represents three sets of parallel lines.
  - Let $[w_0^{(j)}, w_1^{(j)}, w_2^{(j)}]^T$ be the j-th column of $\widehat{W}$. Define $L_j(x_1, x_2)$ as
$$L_j(x_1, x_2) = w_0^{(j)} + w_1^{(j)} x_1 + w_2^{(j)} x_2, \quad j = 0, 1, 2. \tag{P.1.12}$$
  - Figure P.2 depicts $L_j(x_1, x_2) = 0$ and $L_j(x_1, x_2) = 1$ superposed on the training set.
- It follows from (P.1.11) that the mCLESS can be viewed as an **one-versus-rest (OVR)** classifier; see Section 3.2.3.

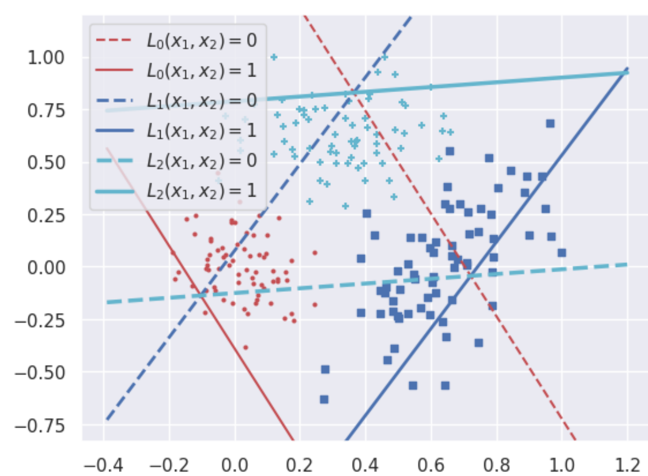

Figure P.2: Lines represented by the weight vectors. mCLESS is interpretable!

The whole algorithm (training-prediction) is run 100 times, with randomly splitting the dataset into 70:30 parts respectively for training and prediction; which results in 98.37% and 0.00171 sec for the average accuracy and e-time. The used is a laptop of an Intel Core i7-10750H CPU at 2.60GHz.

P.1.3. Feature expansion

Remark P.3. Nonlinear mCLESS

- The mCLESS so far is a **linear classifier**.
- As for other classifiers, its nonlinear expansion begins with a data transformation, more precisely, **feature expansion**.
- For example, the **Support Vector Machine (SVM)** replaces the dot product of feature vectors (point) with the result of a kernel function applied to the feature vectors, in the construction of the Gram matrix:

  $$\mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) \approx \sigma(\mathbf{x}_i) \cdot \sigma(\mathbf{x}_j),$$

  where σ is a function for feature expansion.

  – Thus, without an explicit expansion of feature vectors, the SVM can incorporate the effect of data transformation effectively. Such a technique is called the **kernel trick**. See Section 5.3.5.
- However, the **mCLESS** does not incorporate dot products between points.

  – As a result, we must *perform feature expansion without a kernel trick*, which results in an augmented normal matrix, expanded in both column and row directions.

Feature Expansion for mCLESS

- A feature expansion is expressed as

$$\begin{cases} \mathbf{x} = [x_1, x_2, \cdots, x_d] \\ \mathbf{w} = [w_0, w_1, \cdots, w_d]^T \end{cases} \Rightarrow \begin{cases} \widetilde{\mathbf{x}} = [x_1, x_2, \cdots, x_d, \sigma(\mathbf{x})] \\ \widetilde{\mathbf{w}} = [w_0, w_1, \cdots, w_d, w_{d+1}]^T \end{cases} \tag{P.1.13}$$

  where $\sigma()$ is a **feature function** of x.

- Then, the expanded weights must be trained to satisfy

$$[1, \widetilde{\mathbf{x}}^{(i)}]\, \widetilde{\mathbf{w}}^{(j)} = w_0^{(j)} + w_1^{(j)} x_1^{(i)} + \cdots + w_d^{(j)} x_d^{(i)} + w_{d+1}^{(j)} \sigma(\mathbf{x}^{(i)}) = \delta_{ij}, \tag{P.1.14}$$

  for all points in the dataset. Compare the equation with (P.1.3).

- The corresponding expanded information and weight matrices read

$$\widetilde{A} = \left[\begin{array}{ccccc:c} 1 & x_{11} & x_{12} & \cdots & x_{1d} & \sigma(\mathbf{x}_1) \\ 1 & x_{21} & x_{22} & \cdots & x_{2d} & \sigma(\mathbf{x}_2) \\ & \vdots & & \ddots & & \vdots \\ 1 & x_{N1} & x_{N2} & \cdots & x_{Nd} & \sigma(\mathbf{x}_N) \end{array}\right], \quad \widetilde{W} = \left[\begin{array}{cccc} w_0^{(0)} & w_0^{(1)} & \cdots & w_0^{(C-1)} \\ w_1^{(0)} & w_1^{(1)} & \cdots & w_1^{(C-1)} \\ \vdots & & \ddots & \vdots \\ w_d^{(0)} & w_d^{(1)} & \cdots & w_d^{(C-1)} \\ \hdashline w_{d+1}^{(0)} & w_{d+1}^{(1)} & \cdots & w_{d+1}^{(C-1)} \end{array}\right], \tag{P.1.15}$$

  where $\widetilde{A} \in \mathbb{R}^{N\times(d+2)}$, $\widetilde{W} \in \mathbb{R}^{(d+2)\times C}$, and C is the number of classes.

- Feature expansion can be performed multiple times. When α features are added, the optimal weight matrix $\widehat{W} \in \mathbb{R}^{(d+1+\alpha)\times C}$ is the least-squares solution of

$$(\widetilde{A}^T \widetilde{A})\, \widehat{W} = \widetilde{A}^T B, \tag{P.1.16}$$

  where $\widetilde{A}^T \widetilde{A} \in \mathbb{R}^{(d+1+\alpha)\times(d+1+\alpha)}$ and B is the same as in (P.1.7).

Remark P.4. Various feature functions $\sigma()$ can be considered. Here we will focus on the **feature function** of the form

$$\sigma(\mathbf{x}) = \|\mathbf{x} - \mathbf{p}\|, \tag{P.1.17}$$

the Euclidean distance between x and a prescribed point p.
Now, the question is: "*How can we find* p?"

Generation of the Synthetic Data

synthetic_data.py

```
import numpy as np
import matplotlib.pyplot as plt
from GLOBAL_VARIABLES import *

def generate_data(n,scale,theta):
    # Normally distributed around the origin
    x = np.random.normal(0,1, n); y = np.random.normal(0,1, n)
    P = np.vstack((x, y)).T
    # Transform
    sx,sy = scale
    S = np.array([[sx,0],[0,sy]])
    c,s = np.cos(theta), np.sin(theta)
    R = np.array([[c,-s],[s,c]]).T #T, due to right multiplication
    return P.dot(S).dot(R)

def synthetic_data():
    N=0
    plt.figure()
    for i in range(N_CLASS):
        scale = SCALE[i]; theta = THETA[i]; N+=N_D1
        D1 = generate_data(N_D1,scale,theta) +TRANS[i]
        D1 = np.column_stack((D1,i*np.ones([N_D1,1])))
        if i==0: DATA = D1
        else:    DATA = np.row_stack((DATA,D1))
        plt.scatter(D1[:,0],D1[:,1],s=15,c=COLOR[i],marker=MARKER[i])

    np.savetxt(DAT_FILENAME,DATA,delimiter=',',fmt=FORMAT)
    print('   saved: %s' %(DAT_FILENAME))

    #xmin,xmax = np.min(DATA[:,0]), np.max(DATA[:,0])
    ymin,ymax = np.min(DATA[:,1]), np.max(DATA[:,1])
    plt.ylim([int(ymin)-1,int(ymax)+1])

    plt.title('Synthetic Data: N = '+str(N))
    myfigsave(FIG_FILENAME)
    if __name__ == '__main__':
        plt.show(block=False); plt.pause(5)

if __name__ == '__main__':
    synthetic_data()
```

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

N_D1 = 100
FORMAT = '%.3f','%.3f','%d'

SCALE = [[1,1],[1,2],[1.5,1]]; TRANS = [[0,0],[6,0],[3,4]]
#SCALE = [[1,1],[1,1],[1,1]]; TRANS = [[0,0],[4,0],[8,0]]
THETA  = [0,-0.25*np.pi, 0]
COLOR  = ['r','b','c']
MARKER = ['.','s','+','*']
LINESTYLE = [['r--','r-'],['b--','b-'],['c--','c-']]

N_CLASS = len(SCALE)

DAT_FILENAME = 'synthetic.data'
FIG_FILENAME = 'synthetic-data.png'
FIG_INTERPRET = 'synthetic-data-interpret.png'

def myfigsave(figname):
    plt.savefig(figname,bbox_inches='tight')
    print('   saved: %s' %(figname))
```

What to do

1. Implement mCLESS:
   - **Training**. You should implement modules for each of (P.1.5) and (P.1.7). Then use `Xtrain` and `ytrain` to get A and B.
   - **Test**. Use the same module (implemented for A) to get `Atest` from `Xtest`. Then perform `P = (Atest)*`$\widehat{W}$ as in (P.1.10). Now, you can get the prediction using
     ```
     prediction = np.argmax(P,axis=1);
     ```
     which may be compared with `ytest` to obtain accuracy.
2. Use following datasets:
   - **Synthetic datasets**. Generate two different synthetic datasets:
     ① Use Line 7 in `GLOBAL_VARIABLES.py`
     ② Use Line 8 in `GLOBAL_VARIABLES.py`
   - **Real datasets**. Use public datasets such as `iris` and `wine`.
     To get the public datasets, you may use:
     ```
     from sklearn import datasets
     data_read1 = datasets.load_iris()
     data_read2 = datasets.load_wine()
     ```
3. Compare the performance of mCLESS with
   - `LogisticRegression(max_iter = 1000)`
   - `KNeighborsClassifier(5)`
   - `SVC(gamma=2, C=1)`
   - `RandomForestClassifier(max_depth=5, n_estimators=50, max_features=1)`

   `See Section 1.3.`
4. (**Optional for Undergraduate Students**) Add modules for feature expansion, as described on page 365.
   - For this, try to an **interpretable strategy** to find an effective point p such that the feature expansion with (P.1.17) improves accuracy.
   - Experiment Steps 1-3.
5. Report your experiments with the code and results.

You may start with the **machine learning modelcode** in Section 1.3; add your own modules.