Chapter P

Projects

Finally we add projects.

Contents of Projects

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 [17].

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. What is machine learning?

Definition P.1. Machine learning (ML)

- **ML algorithms** are algorithms that can learn from **data** (input) and produce **functions/models** (output).
- **Machine learning** is the science of getting machines to act, without functions/models being explicitly programmed to do so.

Example P.2. There are three different types of ML:

- **Supervised learning**: e.g., classification, regression
  - Labeled data
  - Direct feedback
  - Predict outcome/future
- **Unsupervised learning**: e.g., clustering
  - No labels
  - No feedback
  - Find hidden structure in data
- **Reinforcement learning**: e.g., chess engine
  - Decision process
  - Reward system
  - Learn series of actions

Note: The most popular type is **supervised learning**.

Supervised Learning

Assumption. Given a data set $\{(\mathrm{x}_i, y_i)\}$, where y_i are labels, there exists a relation $f : X \to Y$.

Supervised learning:

$$\begin{cases} \text{Given}: & \text{A } \textbf{training data } \{(\mathrm{x}_i, y_i) \mid i = 1, \cdots, N\} \\ \text{Find}: & \widehat{f} : X \to Y, \text{ a good approximation to } f \end{cases} \tag{P.1.1}$$

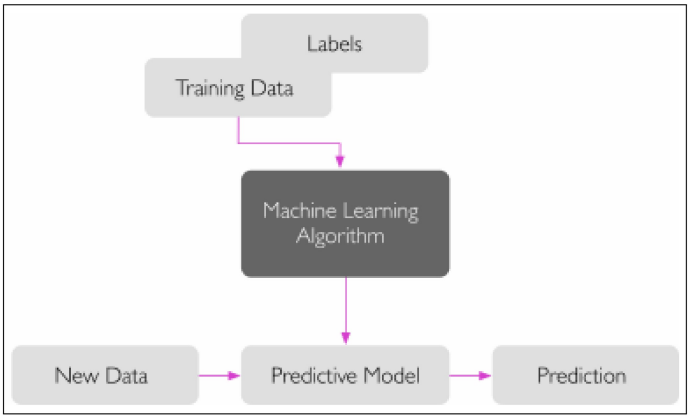

Figure P.1: Supervised learning and prediction.

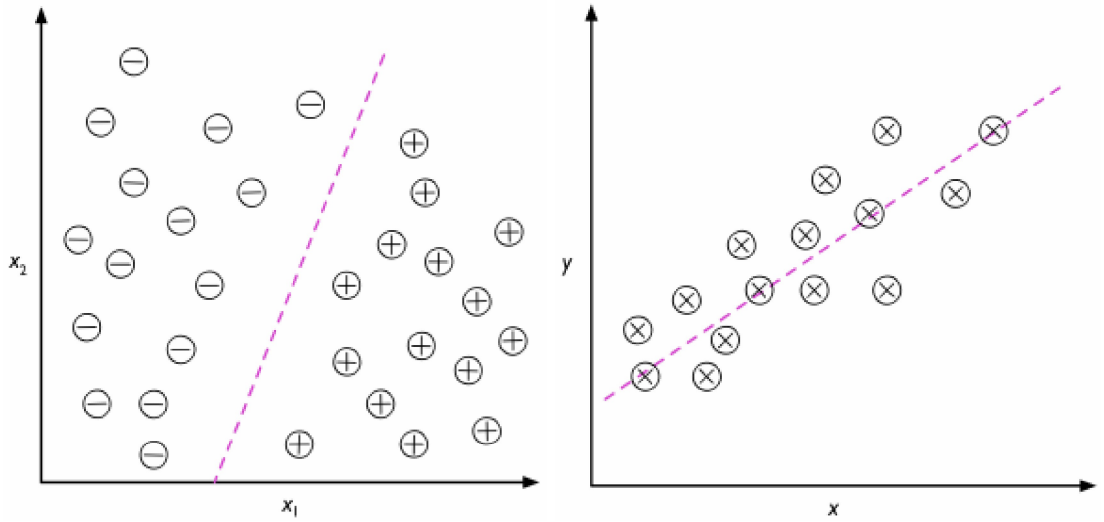

Figure P.2: Classification and regression.

P.1.2. Simple classifiers

The **Perceptron** [18] (or Adaline) is the simplest artificial neuron that makes decisions for datasets of two classes by *weighting up evidence*.

- Inputs: feature values $\mathrm{x} = [x_1, x_2, \cdots, x_d]$
- Weight vector and bias: $\mathrm{w} = [w_1, w_2, \cdots, w_d]^T$, w_0
- Net input:
$$z = w_0 + w_1\, x_1 + w_2\, x_2 + \cdots + w_d\, x_d \tag{P.1.2}$$
- *Activation*:
$$\phi(z) = \begin{cases} 1, & \text{if } z \geq \theta \\ 0, & \text{otherwise,} \end{cases} \tag{P.1.3}$$
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.3. Note that the net input in (P.1.2) 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.1.3. 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

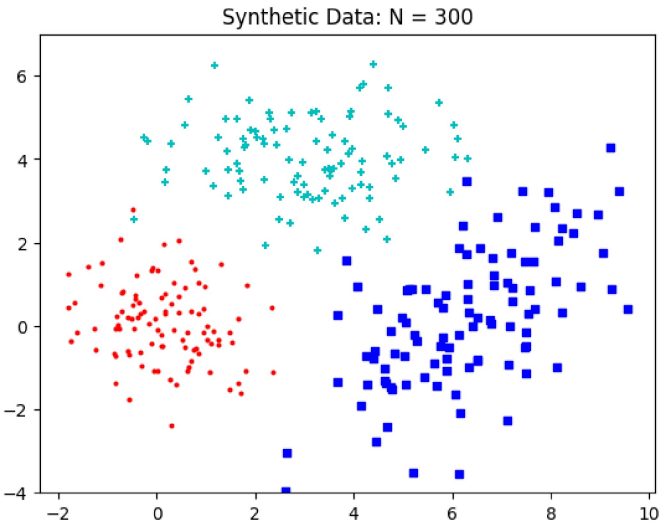

Figure P.3: A synthetic data of three classes.

- In order to describe the proposed algorithm effectively, we exemplify a synthetic data of three classes, as shown in Figure P.3, in which each class has 100 points.
- A point in the c-th class is expressed as

$$\mathbf{x}^{(c)} = [x_1^{(c)}, x_2^{(c)}] = [x_1, x_2, c] \quad c = 1, 2, 3,$$

  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.,

$$w_0^{(j)} + w_1^{(j)} x_1^{(i)} + w_2^{(j)} x_2^{(i)} = \delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{P.1.4}$$

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 in Figure P.3 by

$$X = \begin{bmatrix} x_{11} & x_{12} & c_1 \\ x_{21} & x_{22} & c_2 \\ & \vdots & \\ x_{N1} & x_{N2} & c_N \end{bmatrix} \in \mathbb{R}^{N\times 3}, \tag{P.1.5}$$

  where $c_i \in \{1, 2, 3\}$, the class number.

- **The algebraic system**: It can be formulated using (P.1.4).

  – Define the **information matrix**:

$$A = \begin{bmatrix} 1 & x_{11} & x_{12} \\ 1 & x_{21} & x_{22} \\ & \vdots & \\ 1 & x_{N1} & x_{N2} \end{bmatrix} \in \mathbb{R}^{N\times 3}. \tag{P.1.6}$$

  – Define the **weight matrix**:

$$W = [\mathbf{w}^{(1)}, \mathbf{w}^{(2)}, \mathbf{w}^{(3)}] = \begin{bmatrix} w_0^{(1)} & w_0^{(2)} & w_0^{(3)} \\ w_1^{(1)} & w_1^{(2)} & w_1^{(3)} \\ w_2^{(1)} & w_2^{(2)} & w_2^{(3)} \end{bmatrix}, \tag{P.1.7}$$

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

– Define the **source matrix**:

$$B = [\delta_{c_i,j}] \in \mathbb{R}^{N\times 3}. \tag{P.1.8}$$

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

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

$$\widehat{W} = \arg\min_{W} ||AW - B||^2, \tag{P.1.9}$$

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

$$(A^T A)\,\widehat{W} = A^T B, \qquad A^T A \in \mathbb{R}^{3\times 3}. \tag{P.1.10}$$

- **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

$$[1, x_1, x_2]\,\widehat{W} = [p_1, p_2, p_3], \qquad \widehat{W} \in \mathbb{R}^{3\times 3}. \tag{P.1.11}$$

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:

$$c = \arg\max([p_1, p_2, p_3]). \tag{P.1.12}$$

Remark P.4. The proposed classifier, mCLESS, is implemented in Python, in which indices begin with 0. 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

$$L_j(x_1, x_2) = w_0^{(j)} + w_1^{(j)} x_1 + w_2^{(j)} x_2, \quad j = 0, 1, 2. \tag{P.1.13}$$

- Figure P.4 depicts $L_j(x_1, x_2) = k$, $j = 0, 1, 2$, $k = 0, 1$, with the training set.
- It follows from (P.1.12) that the mCLESS can be viewed as an one-versus-rest (OVR) classifier.

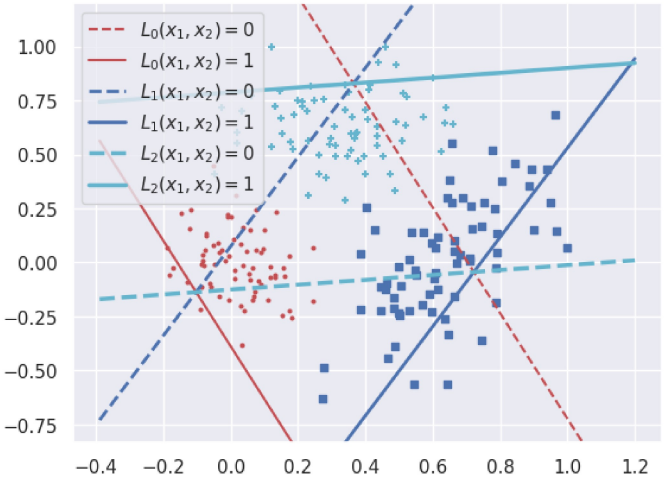

Figure P.4: Lines represented by the weight vectors. mCLESS is interpretable!

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.1.4. Feature expansion

- 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**.

Note: However, the mCLESS does not incorporate dot products between points. Thus 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

$$\begin{cases} \mathbf{x} = [x_1, x_2, \cdots, x_d] \\ \mathbf{w} = [w_0, w_1, \cdots, w_d]^T \end{cases} \Rightarrow \begin{cases} \widetilde{\mathbf{x}} = [x_1, x_2, \cdots, x_d, \sigma(\mathbf{x})] \\ \widetilde{\mathbf{w}} = [w_0, w_1, \cdots, w_d, w_{d+1}]^T \end{cases} \tag{P.1.14}$$

  where $\sigma()$ is a nonlinear feature function of $\mathbf{x}$.

- Then, the expanded weights must be trained to satisfy

$$[1, \widetilde{\mathbf{x}}^{(i)}]\, \widetilde{\mathbf{w}}^{(j)} = w_0^{(j)} + w_1^{(j)} x_1^{(i)} + \cdots + w_d^{(j)} x_d^{(i)} + w_{d+1}^{(j)} \sigma(\mathbf{x}^{(i)}) = \delta_{ij}, \tag{P.1.15}$$

  for all points in the dataset. Compare the equation with (P.1.4).

- The corresponding expanded information and weight matrices read

$$\widetilde{A} = \left[\begin{array}{ccccc|c} 1 & x_{11} & x_{12} & \cdots & x_{1d} & \sigma(\mathbf{x}_1) \\ 1 & x_{21} & x_{22} & \cdots & x_{2d} & \sigma(\mathbf{x}_2) \\ & \vdots & & \ddots & & \vdots \\ 1 & x_{N1} & x_{N2} & \cdots & x_{Nd} & \sigma(\mathbf{x}_N) \end{array}\right] \qquad \widetilde{W} = \left[\begin{array}{cccc} w_0^{(1)} & w_0^{(2)} & \cdots & w_0^{(C)} \\ w_1^{(1)} & w_1^{(2)} & \cdots & w_1^{(C)} \\ \vdots & & \ddots & \vdots \\ w_d^{(1)} & w_d^{(2)} & \cdots & w_d^{(C)} \\ \hline w_{d+1}^{(1)} & w_{d+1}^{(2)} & \cdots & w_{d+1}^{(C)} \end{array}\right], \tag{P.1.16}$$

  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

$$(\widetilde{A}^T \widetilde{A})\, \widehat{W} = \widetilde{A}^T B, \tag{P.1.17}$$

  where $\widetilde{A}^T \widetilde{A} \in \mathbb{R}^{(d+1+\alpha) \times (d+1+\alpha)}$ and B is the same as in (P.1.8).

Remark P.5. Various feature functions $\sigma()$ can be considered. Here we will focus on the feature function of the form

$$\sigma(\mathbf{x}) = \|\mathbf{x} - \mathbf{p}\|, \tag{P.1.18}$$

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

What to do

1. Implement mCLESS.
   - **Training**. You should implement modules for each of (P.1.6) and (P.1.8). 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.1.11). Now, you can get the prediction using
     ```
     prediction = np.argmax(P,axis=1);
     ```
     which may be compared with `ytest` to obtain accuracy.
2. Also, add modules for feature expansion, as described on page 341.
   - For this, try to an **interpretable strategy** to find an effective point p such that the feature expansion with (P.1.18) improves accuracy.
3. Report your experiments with the code and results.

You may start with the following; add your own modules.

Machine_Learning_Model.py

```
import numpy as np;    import pandas as pd
import seaborn as sbn; import matplotlib.pyplot as plt
import time
from sklearn.model_selection import train_test_split
from sklearn import datasets; #print(dir(datasets))
np.set_printoptions(suppress=True)

#=====================================================================
# DATA: Read & Preprocessing
# load_iris, load_wine, load_breast_cancer, ...
#=====================================================================
data_read = datasets.load_iris();  #print(data_read.keys())

X = data_read.data
y = data_read.target
datafile = data_read.filename
targets  = data_read.target_names
features = data_read.feature_names

print('X.shape=',X.shape, 'y.shape=',y.shape)

#---------------------------------------------------------------------
```

```
# SETTING
#---------------------------------------------------------------------
N,d = X.shape; labelset=set(y)
nclass=len(labelset);
print('N,d,nclass=',N,d,nclass)

rtrain = 0.7e0; run = 100
rtest  = 1-rtrain

#=====================================================================
# CLASSIFICATION
#=====================================================================
btime = time.time()
Acc = np.zeros([run,1])
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(5)

for it in range(run):
    Xtrain, Xtest, ytrain, ytest = train_test_split(
        X, y, test_size=rtest, random_state=it, stratify = y)
    clf.fit(Xtrain, ytrain);
    Acc[it] = clf.score(Xtest, ytest)

#---------------------------------------------
# Print: Accuracy && E-time
#---------------------------------------------
etime = time.time()-btime
print('   %s: Acc.(mean,std) = (%.2f,%.2f)%%; Average E-time= %.5f'
          %(datafile,np.mean(Acc)*100,np.std(Acc)*100,etime/run))

#=====================================================================
# Scikit-learn Classifiers, for Comparisions
#=====================================================================
#exec(open("sklearn_classifiers.py").read())
```

sklearn_classifiers.py

```
#=====================================================================
# Required: X, y, datafile
print('========= Scikit-learn Classifiers, for Comparisions =========')
#=====================================================================
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.inspection import DecisionBoundaryDisplay

#-----------------------------------------------
names = [
    "Logistic Regr",
    "KNeighbors-7 ",
    "Linear SVM   ",
    "RBF SVM      ",
    "Random Forest",
    "Deep-NN      ",
    "AdaBoost     ",
    "Naive Bayes  ",
    "QDA          ",
    "Gaussian Proc",
]

#-----------------------------------------------
classifiers = [
    LogisticRegression(max_iter = 1000),
    KNeighborsClassifier(7),
    SVC(kernel="linear", C=0.5),
    SVC(gamma=2, C=1),
    RandomForestClassifier(max_depth=5, n_estimators=50, max_features=1),
    MLPClassifier(alpha=1, max_iter=1000),
    AdaBoostClassifier(),
    GaussianNB(),
    QuadraticDiscriminantAnalysis(),
```

```
    GaussianProcessClassifier(),
]

#------------------------------------------------
acc_max=0
for name, clf in zip(names, classifiers):
    Acc = np.zeros([run,1])
    btime = time.time()

    for it in range(run):
        Xtrain, Xtest, ytrain, ytest = train_test_split(
            X, y, test_size=rtest, random_state=it, stratify = y)

        clf.fit(Xtrain, ytrain);
        Acc[it] = clf.score(Xtest, ytest)

    etime = time.time()-btime
    accmean = np.mean(Acc)*100
    print('%s: %s: Acc.(mean,std) = (%.2f,%.2f)%%; E-time= %.5f'
          %(datafile,name,accmean,np.std(Acc)*100,etime/run))
    if accmean>acc_max:
        acc_max= accmean; algname = name
print('sklearn classifiers max: %s= %.2f' %(algname,acc_max))
```