TP 3 – AOS1

Bayesian logistic regression, Gaussian process classification— partial solution

1 Introduction

This practical session aims first at applying logistic regression with Bayesian regularization in a first step, and to briefly study Gaussian processes in a second step. You will need several libraries for this purpose.

```
import numpy as np
import pandas as pd
import scipy as sp
import scipy.stats as spst
import matplotlib.pyplot as plt
import seaborn as sns
```

2 Logistic regression

2.1 LR without regularization

Logistic regression is implemented in scikit-learn via the LogisticRegression class; it is available with the following command:

```
from sklearn.linear_model import LogisticRegression
```

The penalty argument is particularly important. By default, a ℓ_2 penalization term is used. You will also consider the ℓ_1 term, and no penalization (penalty="none").

After training, several attributes are available, among which coef_ contains the coefficients of the model, and intercept_ the intercept coefficient.

$\boxed{1}$ Consider the synthetic datasets provided [1]. Train a logistic regression model *without regularization*, and visualize the decision boundary using the appropriate function from utils.py.

```
Xy = pd.read_csv("data/SynthPara_n1000_p2.csv")
X = Xy.iloc[:, :-1]
y = Xy.iloc[:, -1]

cls = LogisticRegression(penalty="none")
cls.fit(X, y)

from utils import add_decision_boundary
sns.scatterplot(x="X1", y="X2", hue="z", data=Xy)
add_decision_boundary(cls)
plt.show()
```

[1]contained in the SynthCross_n1000_p2.csv, SynthPara_n1000_p2.csv and SynthPlus_n1000_p2.csv files

$\boxed{2}$ Using the `coef_` and `intercept_` attributes, get the expression of the decision boundary.

The logistic regression model writes as

$$\sigma^{-1}(p) = w_0 + w_1 x + w_2 y,$$

with w_0 = `cls.intercept_[1]`, w_1 = `cls.coef_[1]` and w_2 = `cls.coef_[2]`. On the `SynthPara_n1000_p2.csv`, we approximately retrieve the $y = x$ decision boundary.

$\boxed{3}$ Use the `levels` argument so as to display the level curves of the posterior probability distribution. You may for instance use the levels 0.1, 0.3, 0.5, 0.7 and 0.9.

```
sns.scatterplot(x="X1", y="X2", hue="z", data=Xy)
levels = [.1, .3, .5, .7, .9]
add_decision_boundary(cls, levels=levels)
```

Remark 1. If the `add_decision_boundary` does not work properly, you can use the following code to display the level curves for the model outputs:

```
Gx, Gy = np.mgrid[-15:16:200j, -15:16:200j]
grid = np.stack([Gx, Gy], axis=-1)

fG = LR_model.predict_proba(grid.reshape(-1, 2))[:, 1].reshape(*Gx.shape)

fig, ax = plt.subplots()
ax.scatter(X.X1, X.X2, c=(y=="A").astype(int))
CS = ax.contour(Gx, Gy, np.reshape(fG, Gx.shape))
ax.clabel(CS, inline=1, fontsize=10)
ax.set_title('Contour plot, $p(y^*=1|x^*,X,y)$')
```

2.2 LR with regularization

$\boxed{4}$ Sub-sample the dataset so as to keep only a (small) fraction of the original data (e.g., 1%). Train a logistic regression classifier *without regularization* on the subsampled data and display the results. Compare with the model trained *without regularization* on the whole dataset.

```
Xy_ = Xy.sample(frac=0.01)
X_ = Xy_.iloc[:, :-1]
y_ = Xy_.iloc[:, -1]

cls_ = LogisticRegression(penalty="none")
cls_.fit(X_, y_)

fig, ax = plt.subplots(ncols=2, sharex=True, sharey=True)

sns.scatterplot(x="X1", y="X2", hue="z", data=Xy, ax=ax[0])
levels = [.1, .3, .5, .7, .9]
add_decision_boundary(cls, levels=levels, ax=ax[0])

sns.scatterplot(x="X1", y="X2", hue="z", data=Xy_, ax=ax[1])
levels = [.1, .3, .5, .7, .9]
add_decision_boundary(cls_, levels=levels, ax=ax[1])
```

$\boxed{5}$ Do the same *with regularization*.

The model estimated is much more stable as can be observed when it is retrained over various small samples. The “transition” between the classes is also much smoother due to regularization.

```
cls = LogisticRegression(penalty="l2")
cls.fit(X, y)
cls_ = LogisticRegression(penalty="l2")
cls_.fit(X_, y_)

fig, ax = plt.subplots(ncols=2, sharex=True, sharey=True)

sns.scatterplot(x="X1", y="X2", hue="z", data=Xy, ax=ax[0])
levels = [.1, .3, .5, .7, .9]
add_decision_boundary(cls, levels=levels, ax=ax[0])

sns.scatterplot(x="X1", y="X2", hue="z", data=Xy_, ax=ax[1])
levels = [.1, .3, .5, .7, .9]
add_decision_boundary(cls_, levels=levels, ax=ax[1])
```

3 Gaussian process classification

This part aims at testing Gaussian process classification on synthetic data. You can import the `scikit-learn` Gaussian process classification implementation using

```
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import ConstantKernel as CK, RBF
```

$\boxed{6}$ Generate a 1D synthetic dataset with $n_1 = 20$ instances from class c_1 and $n_2 = 20$ instances from class c_2, with

$$X \underset{c_1}{\sim} \frac{1}{2}\mathcal{N}(-2,1) + \frac{1}{2}\mathcal{N}(2,1), \quad X \underset{c_2}{\sim} \mathcal{N}(0,1).$$

```
n11 = spst.binom(n=20,p=0.5).rvs()
n12 = 20-n11
Xtr = np.concatenate((spst.norm(-2,1).rvs(size=n11),
                      spst.norm(2,1).rvs(size=n12),
                      spst.norm(0,1).rvs(size=20)),
                     axis=0)
#ytr = 2+np.sum(Xtr, axis=1)
ytr = np.concatenate((np.repeat(1,20),
                      np.repeat(0,20)))

fig, ax = plt.subplots()
plt.plot(Xtr[ytr==0], ytr[ytr==0], 'bo')
plt.plot(Xtr[ytr==1], ytr[ytr==1], 'rx')
```

$\boxed{7}$ Train a Gaussian process classifier on these 1D data with the following code. Display the model outputs and interpret.

```
rbf = CK(1.0) * RBF(length_scale=1.0)
gpc = GaussianProcessClassifier(kernel=rbf)
```

The estimated posterior probabilities (more or less) follow the density of the data, as can be seen by plotting the model outputs using the previous code.
The Gaussian process classifier can thus be interpreted as a generalized logistic regression model based on estimated “memberships” to the classes.

```
rbf = CK(1.0) * RBF(length_scale=1.0)
gpc = GaussianProcessClassifier(kernel=rbf)

gpc.fit(Xtr.reshape(-1,1), ytr)

gpc.predict_proba(Xpl.reshape(-1, 1))[:,1]
plt.plot(Xpl, gpc.predict_proba(Xpl.reshape(-1, 1))[:,1],
         color='black', ls=':')
plt.xlabel('x')
plt.ylabel('$p(y_*=1|\mathbf{f})$')
plt.legend();
```

$\boxed{8}$ Generate a 2D dataset using the `make_moons` function from the `datasets` component of the `scikit-learn` library. Display the data obtained.

```
X, y = make_moons(200, noise=0.5, random_state=1)

plt.scatter(X[:, 0], X[:, 1], c=y)
```

$\boxed{9}$ Fit a Gaussian process classification model to the data. Display the results.

```
rbf = CK(1.0) * RBF(length_scale=1.0)
gpc = GaussianProcessClassifier(kernel=rbf)

gpc.fit(X, y)

Gx, Gy = np.mgrid[-4:4:200j, -4:4:200j]
grid = np.stack([Gx, Gy], axis=-1)

fG = gpc.predict_proba(grid.reshape(-1, 2))[:, 1].reshape(*Gx.shape)

fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], c=y)
CS = ax.contour(Gx, Gy, np.reshape(fG, Gx.shape))
ax.clabel(CS, inline=1, fontsize=10)
ax.set_title('Contour plot, $p(y^*=1|x^*,X,y)$')
```