TP 3 – AOS1

Regularized logistic regression
Corrigé

1 Introduction

The purpose of the practical session is to program and use (binary) logistic regression, and then use two kinds of Bayesian regularization in order to improve its results.

2 Binary logistic regression

2.1 Implementation

We will begin with implementing binary logistic regression. The algorithm used for training will be the Newton-Raphson algorithm presented in the course.
You can compare the results with those obtained via the `scikit-learn` implementation, using the following instructions (mind the `penalty` argument):

```
from sklearn.linear_model import LogisticRegression as SklearnLogisticRegression
sk_cls = SklearnLogisticRegression(penalty="none")
sk_cls.fit(X, y)
sk_cls.coef_
sk_cls.intercept_
```

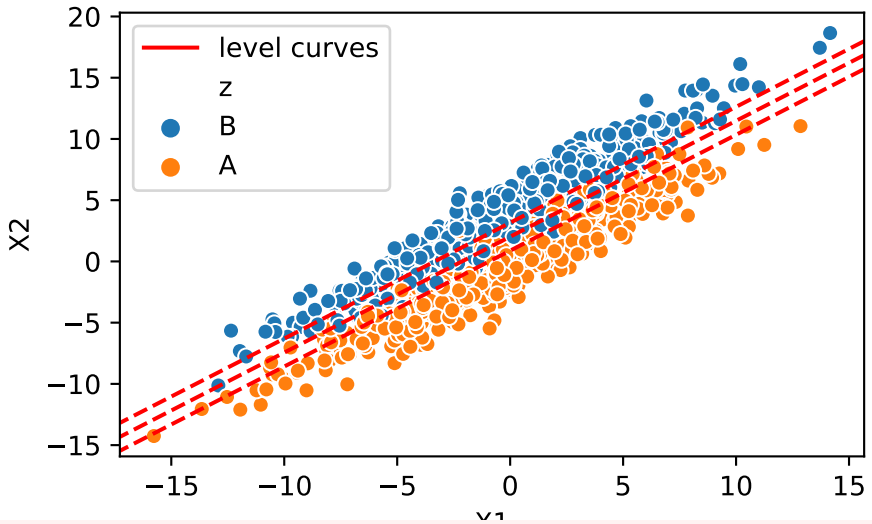
(1) Fill-in the missing parts in the `src/logistic_regression.py` file provided.

In [1]:
```python
import numpy as np
from sklearn.base import BaseEstimator
from sklearn.linear_model._base import LinearClassifierMixin
from sklearn.utils import check_X_y

# logit function
from scipy.special import expit

# Inherit from `LinearClassifierMixin` which manages the prediction part
# we must define `coef_`, `intercept_` and `classes_` appropriately
# during training
class LogisticRegression(BaseEstimator, LinearClassifierMixin):
    def __init__(self, max_iter=1000, tol=1e-5, fit_intercept=True):
        self.max_iter = max_iter
        self.tol = tol
        self.fit_intercept = fit_intercept

    def fit(self, X, y):
        # Check that `X` and `y` are appropriately defined: e.g.
        # transform DataFrame or Pandas Series into a Numpy array
        X, y = check_X_y(X, y)
        # Specify (convert) classes as 0/1 integers
        self.classes_, y = np.unique(y, return_inverse=True)

        # Intercept
        p = X.shape[1]
        if self.fit_intercept:
            p += 1
            X = np.column_stack((np.ones(X.shape[0]), X))

        it = 1
        step = self.tol + 1
        beta = np.zeros(p)

        while np.linalg.norm(step) > self.tol and it < self.max_iter:
            # Compute posterior probabilities with the logit function
            pi = expit(X @ beta)

            # Gradient vector
            grad = X.T @ (y - pi)
            # Hessian matrix
            W = np.diag(pi * (1 - pi))
            hessian = -X.T @ W @ X
            # Step
            step = -np.linalg.solve(hessian, grad)

            # Parameter update
            beta += step

            it += 1

        # Parameter storage
        # In order for the `predict` function to work, `coef_` must be
        # a line matrix, `intercept_` must an array; also mind the
        # `fit_intercept` parameter
        self.coef_ = beta[None, 1:] if self.fit_intercept else beta[None,
        ↪  :]
        self.intercept_ = beta[[0]] if self.fit_intercept else
        ↪  np.array([0])
```

(2) Train the logistic regression model on the data contained in the `SynthPara_n1000_p2.csv` dataset and compare the model obtained to the one obtained via the `scikit-learn` function. The decision boundary can be plotted using the `add_decision_boundary` function.

You may use the following code for this purpose.

```
In [2]: from sklearn.linear_model import LogisticRegression as
        ↪ SklearnLogisticRegression
        sk_cls = SklearnLogisticRegression(penalty="none")

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

        sk_cls.fit(X, y)
        sk_cls.coef_
Out [2]: array([[-1.82244181,  1.92618178]])
In [3]: sk_cls.intercept_
Out [3]: array([-3.8899488])
In [4]: cls = LogisticRegression()
        cls.fit(X, y)

        np.isclose(sk_cls.coef_, cls.coef_)
Out [4]: array([[ True,  True]])
In [5]: np.isclose(sk_cls.intercept_, cls.intercept_)
Out [5]: array([ True])
In [6]: sns.scatterplot(x="X1", y="X2", hue="z", data=Xy)
        from scipy.special import logit
        levels = logit(np.array([.1, .5, .9]))
        add_decision_boundary(cls, levels=levels, label="level curves")
        plt.show()
```



2.2 Polynomial logistic regression

The logistic regression model inherently provides linear decision boundaries. Its extension to nonlinear classification problems is however straightforward. The principle is to extend the original dataset onto a nonlinear space, where the decision boundary is (supposedly) linear. This generalization of logistic regression comes however at a price: the number of parameters to be estimated is indeed higher.

Here, we will explore a strategy based on the polynomial expansion of the input variables in order to introduce more flexibility in logistic regression. For instance, assume that $\boldsymbol{X} = (X_1, X_2, X_3)$, then mapping the instances into a second-order polynomial space feature would lead to define a new feature vector

$$\tilde{\boldsymbol{X}} = \left(X^1, X^2, X^3, X^1X^2, X^1X^3, X^2X^3, (X^1)^2, (X^2)^2, (X^3)^2\right).$$

Polynomial (and thus quadratic) expansions are already implemented in `scikit-learn` through the model `PolynomialFeatures`:

```
from sklearn.preprocessing import PolynomialFeatures
```

This class admits the polynomial degree of the expansion as input: for instance, the third-degree polynomial expansion of the feature vector can be obtained using the following code.

```
poly = PolynomialFeatures(degree=3)
poly.fit_transform(X)
```

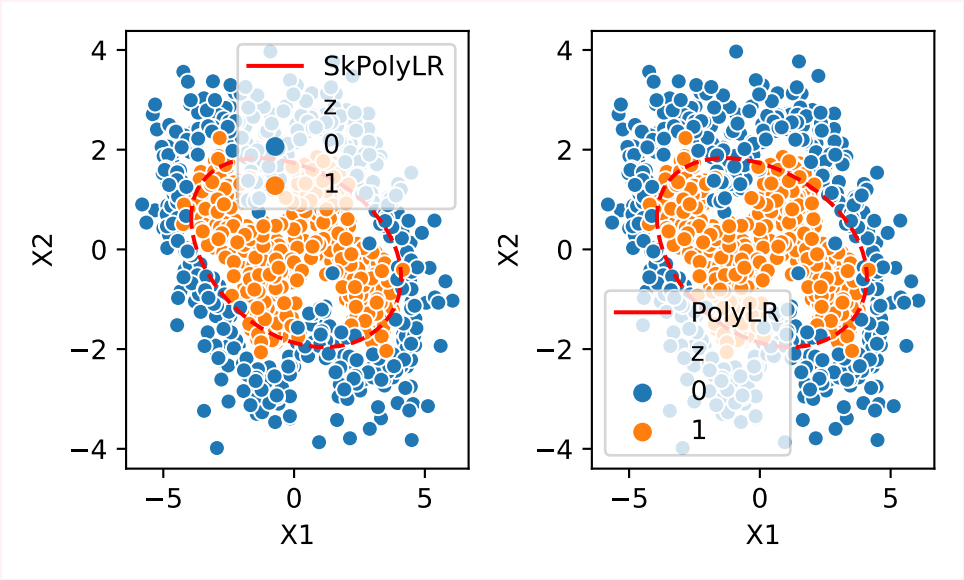
The polynomial transform can be composed with other processings by creating a pipeline.

```
from sklearn.pipeline import make_pipeline
poly = PolynomialFeatures(degree=2, include_bias=False)
cls = LogisticRegression()
pipe = make_pipeline(poly, cls)
```

(3) Using both your implementation and the `scikit-learn` version, make pipelines so as to compute the d-order polynomial expansion of the data in the `SynthNLin_n1000.csv` and learn a (non-penalized) logistic regression model on the expanded data, for increasing degrees $d = 1, 2, \ldots, 10$. Plot the decision boundaries and compare. What do you notice as the degree increases ?

Both logistic regression models can be trained and compared using the following code.

In [7]:
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

Xy =
↪  pd.read_csv("/Users/quostben/Documents/Travail/Cours/AOS1/A2020/TP3_ML/data/SynthNLin
X = Xy.iloc[:, :-1]
y = Xy.iloc[:, -1]

poly = PolynomialFeatures(degree=3, include_bias=False)

cls_skl = SklearnLogisticRegression(max_iter=1000, solver='newton-cg',
↪  penalty='none')
pipe_skl = make_pipeline(poly, cls_skl)
pipe_skl.fit(X, y)
fig, axs = plt.subplots(1, 2, tight_layout=True)

sns.scatterplot(x="X1", y="X2", hue="z", data=Xy, ax=axs[0])
add_decision_boundary(pipe_skl, label="SkPolyLR", ax=axs[0])

cls = LogisticRegression()
pipe = make_pipeline(poly, cls)
pipe.fit(X, y)
sns.scatterplot(x="X1", y="X2", hue="z", data=Xy, ax=axs[1])
add_decision_boundary(pipe, label="PolyLR", ax=axs[1])

fig
```



We can see that as d increases, the decision boundary becomes smoother and better adapts to the data. The function developped fails at some point for numerical reasons; the `scikit-learn` function is more robust, and using other solvers in order to maximize the likelihood might help dealing with high degrees. Nevertheless, we can notice some overfitting at some point, which suggests that using a penalization term might help regularizing the boundary.

(4) Using the `scikit-learn` logistic regression implementation only, make pipelines so as to compute the d-order polynomial expansion of the same data (for increasing degrees $d = 1, 2, \ldots, 8$) and learn three logistic regression models on the expanded data:

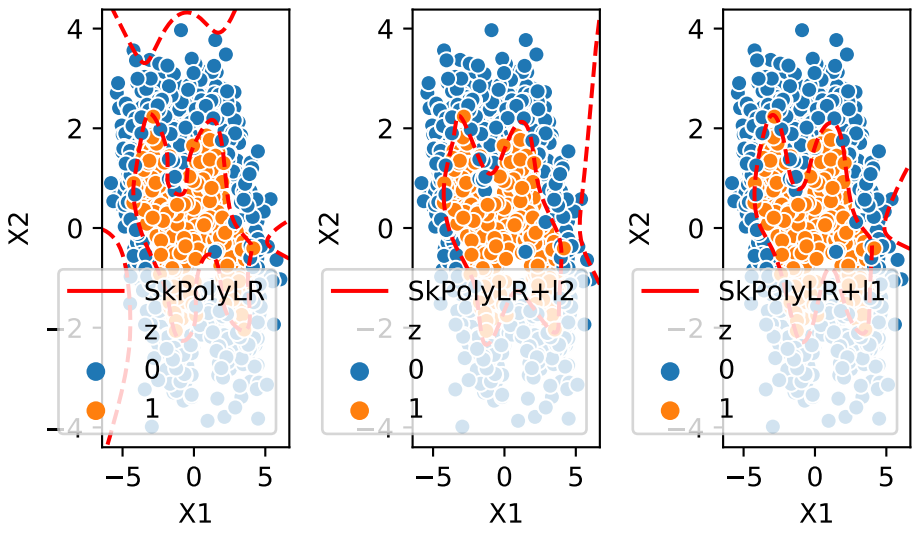
- a non-penalized one,

- a ℓ_2-penalized one,
- a ℓ_1-penalized one (you will require to change the solver to `liblinear`).

Again, plot the decision boundaries and compare. Also analyze the coefficients of the model (which can be accessed, e.g. for the non-penalized model learnt previously, via `pipe_skl[1].coef_`). What do you notice ?

The models can be compared using the following code.

In [8]:
```
fig, axs = plt.subplots(1, 3, tight_layout=True)

poly = PolynomialFeatures(degree=7, include_bias=False)

cls_skl = SklearnLogisticRegression(max_iter=1000, solver='newton-cg',
↪  penalty='none')
pipe_skl = make_pipeline(poly, cls_skl)
pipe_skl.fit(X, y)
sns.scatterplot(x="X1", y="X2", hue="z", data=Xy, ax=axs[0])
add_decision_boundary(pipe_skl, label="SkPolyLR", ax=axs[0])

cls_skl = SklearnLogisticRegression(max_iter=1000, solver='liblinear',
↪  penalty='l2')
pipe_skl_l2 = make_pipeline(poly, cls_skl)
pipe_skl_l2.fit(X, y)
sns.scatterplot(x="X1", y="X2", hue="z", data=Xy, ax=axs[1])
add_decision_boundary(pipe_skl_l2, label="SkPolyLR+l2", ax=axs[1])

cls_skl = SklearnLogisticRegression(max_iter=1000, solver='liblinear',
↪  penalty='l1')
pipe_skl_l1 = make_pipeline(poly, cls_skl)
pipe_skl_l1.fit(X, y)
sns.scatterplot(x="X1", y="X2", hue="z", data=Xy, ax=axs[2])
add_decision_boundary(pipe_skl_l1, label="SkPolyLR+l1", ax=axs[2])

fig
```



We can see that as d increases, using a penalization term regularizes the decision boundaries, possibly resulting in some of the training points falling on the "wrong" side. In addition, the analysis of the coefficients shows that the ℓ_1 regularization (especially for high degrees) has some coefficients being put to zero, hence resulting in a more parsimonious model.