Projet d'optimisation, Master Statistique et Big Data

Camille Moatti

Avril 2019

Abstract

Ce devoir, donné pour l'évaluation du cours d'optimisation, a pour objet l'implémentation d'un algorithme de régression ou de classification. Le processus d'entraînement se fera grâce à une descente de gradient. L'algorithme pourra être régularisé et les paramètres optimaux seront choisis en implémentant une validation croisée. Aucun package de *Machine Learning* ne sera utilisé, que ce soit pour l'algorithme de classification, pour les routines de *preprocessing* ou pour la validation croisée.

1 Introduction

Afin de mettre en pratique les enseignements du cours d'optimisation, nous allons évaluer une régression logistique. Afin de rendre le devoir plus complet:

- nous choisissons un jeu de données avec une variable cible ayant plus de deux modalités,
- nous voulons pouvoir régulariser lors de l'estimation de nos paramètres afin d'éviter le surapprentissage.

```
from collections import OrderedDict
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
from tqdm import *
```

Les dépendances sont limitées à `Numpy` pour l'algèbre linéaire et à `Pandas` pour la manipulation de tableaux. `tqdm` donne accès à une barre de progression.
L'ensemble des classes et des fonctions ont été écrites dans le cadre de ce devoir afin d'implémenter de manière simplifiée les algorithmes de Machine Learning. Les routines n'ont pas été testées avec d'autres jeux de données et n'ont pas vocation à être réutilisées.

2 Hypothèses

La fonction logistique est notée :

$$g(z) = \frac{1}{1 + e^{-z}}$$

avec

$$z = \theta x^t$$

où θ est un vecteur colonne contenant l'ensemble des paramètres à évaluer :

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ ... \\ \theta_n \end{bmatrix}$$

et x un vecteur ligne issue de la matrice de design X :

$$X = \begin{bmatrix} x_0^{(0)} & x_1^{(0)} \\ x_0^{(1)} & x_1^{(1)} \\ x_0^{(2)} & x_1^{(2)} \\ x_0^{(3)} & x_1^{(3)} \end{bmatrix}$$

La fonction de coût, vectorisée et sans régularisation est donnée par :

$$J(\theta) = \frac{1}{m}.(-y^T log(h) - (1-y)^T log(1-h))$$

h est le vecteur des prédictions $h = g(X.\theta)$
Le gradient de la fonction de coût sans régularisation est donné par :

$$\nabla J(\theta) = \frac{1}{m}.X^T.(h-y)$$

3 Les données

Nous utiliserons ce jeu de données : lien vers la page

Les attributs contiennent des mesures faites sur de l'écriture. A l'aide des 216 *features*, il faut prédire quel chiffre est écrit (de 1 à 10). Celui-ci est donné dans la colonne `class` .

```
df_raw = pd.read_csv("../data/dataset_12_mfeat-factors (1).csv")
df_raw.head()
```

Les huit premières colonnes et les cinq premières lignes sont reproduites ci-dessous :

	att1	att2	att3	att4	att5	att6	att7	att8
0	98	236	531	673	607	647	2	9
1	121	193	607	611	585	665	7	9
2	115	141	590	605	557	627	12	6
3	90	122	627	692	607	642	0	6
4	157	167	681	666	587	666	8	6

Nous isolons les *features* et la *target* dans deux variables :

```
y = df_raw['class']
df_raw = df_raw.iloc[:,:-1]
```

4 Les fonctions et classes

Afin de ne dépendre d'aucun package d'apprentissage, nous implémentons nous même les fonctions `StandardScaler` et `train_test_split`.Ces implémentations s'inspirent de l'API de `ScikitLearn` à la différence que ce sont des `DataFrame` ou des `Series` qui sont attendus en entrées :

```
class StandardScaler:
    def __init__(self):
        return None
    def fit(self, X: pd.DataFrame):
        self.mean = X.mean(0)
        self.std = X.std(0)
        return self
    def transform(self, X: pd.DataFrame) -> pd.DataFrame :
        X_new = (X.values.T - self.mean[:,np.newaxis]).T
        X_new = (X_new.T / self.std[:,np.newaxis]).T
        return pd.DataFrame(X_new, columns=X.columns)
```

La fonction `train_test_split` est très simplifiée. Afin de conserver un échantillonage stratifié, pour chaque classe à prédire, un nombre aléatoire de ligne est tiré à partir parmi les lignes appartenant à chaque classe. Le nomre de ligne à ajouter dans le jeu de test est trouvé en comptant le nombre de lignes de la classe à prédire multiplié par le ratio et en arrondissant à l'entier le plus proche. Les lignes prélevées ne sont pas replacées et ne peuvent donc pas être tirées deux fois :

```
def train_test_split(X: pd.DataFrame, y: pd.Series, ratio: float) -> pd.DataFrame:
    train_set = pd.DataFrame()
```

```
    test_set = pd.DataFrame()
    y_train_set = pd.Series()
    y_test_set = pd.Series()

    for target in y.unique():
        index_target = y[(y==target)].index.values
        number_of_lines = len(index_target)
        number_of_lines_to_draw = int(round(len(index_target) * ratio, 0))
        y_small = np.random.choice(index_target, size=number_of_lines_to_draw,
                                   replace=False)
        y_small = np.sort(y_small)
        y_big = np.setdiff1d(index_target, y_small, assume_unique=True)
        y_test_set = y_test_set.append(y[y_small], verify_integrity=True)
        y_train_set = y_train_set.append(y[y_big], verify_integrity=True)

        test_set = test_set.append(X.iloc[y_small,:], verify_integrity=True)
        train_set = train_set.append(X.iloc[y_big,:], verify_integrity=True)

    return train_set.reset_index(drop=True), test_set.reset_index(drop=True),
        y_train_set.reset_index(drop=True), y_test_set.reset_index(drop=True)
```

Nous pouvons désormais isoler un jeu de test :

```
X_train, X_test, y_train, y_test = train_test_split(df_raw, y, ratio=0.3333)
```

Nous normalisons les données. Le jeu de test est normalisé en utilisant les valeurs (moyenne et écart-type) du jeu de d'entraînement. Nous procédons comme cela afin de traiter les données du jeu de test comme de nouvelles observations complètement inconnues :

```
st_sc = StandardScaler()
X_train = st_sc.fit(X_train).transform(X_train)
X_test = st_sc.transform(X_test)
```

Dans le jeu de départ, les classes sont équilibrées (10% du jeu chacune). Cet équilibre est conservé dans les deux sous-ensembles grâce à la stratification :

```
y_train.value_counts() / len(y_train)
10   0.1
9    0.1
8    0.1
7    0.1
6    0.1
5    0.1
4    0.1
3    0.1
2    0.1
1    0.1
dtype: float64

y_test.value_counts() / len(y_test)
10   0.1
9    0.1
8    0.1
7    0.1
6    0.1
5    0.1
4    0.1
3    0.1
2    0.1
1    0.1
dtype: float64

len(y_train)
1330

len(X_train)
```

```
1330
```

```
len(y_test)
```
```
670
```

```
len(X_test)
```
```
670
```

5 L'algorithme de régression logistique avec descente de gradient

5.1 Sans régularisation

Nous implémentons une classe `LogisticRegression`. Pour commencer, cette régression logistique n'est pas régularisée. A l'instantiation de la classe, l'utilisateur renseigne le nombre d'itérations et le *learning rate*.
La méthode h renvoie $sigmoid(X.\theta)$ où $sigmoid(x) = \frac{1}{1+e^{-x}}$
La méthode `fit` optimise les coefficients de la régression grâce à une descente de gradient. La fonction de coût et la formule du gradient sont disponibles au début du devoir. L'implémentation garde l'évolution de la fonction de coût et stocke les valeurs dans un vecteur afin de pouvoir visualiser son évolution par la suite.

```
class LogisticRegression:
    def __init__(self, epochs=100, learning_rate=0.1, verbose = False):
        self.epochs = epochs
        self.lr = learning_rate
        self.verbose = verbose
        self.loss_array = np.empty(epochs)
        return None

    def _h(self, features, coefficients):
        x = features @ coefficients
        return 1.0/(1.0+np.exp(-x))

    def fit(self,features, target):
        features = features.values
        target = target.values

        coef = np.random.randn(features.shape[1])
        sum_error = 0.0
        if self.verbose == True:
            print("size of features matrix :", features.shape,
                "\nsize of the target matrix :", target.shape)
        for epoch in tqdm(range(self.epochs)):
            yhat = self._h(features, coef)
            loss = ((-target.T * np.log(yhat) -
                    (1 - target).T * np.log(1 - yhat)).mean())
            self.loss_array[epoch] = loss
            gradient = np.dot(features.T, (yhat - target)) / target.shape[0]
            coef -= self.lr * gradient
            try:
                if self.verbose == True and ((self.epochs % epoch) == 0 or
                                             epoch==self.epochs-1):
                    print("epoch :", epoch,":",
                          "loss function :", round(loss, 2))
                    print("gradient :", gradient)
            except ZeroDivisionError:
                pass
        self.coef = coef
        return self.coef

    def get_loss_evo(self):
        return self.loss_array

    def predict_proba(self, features):
        return self._h(features.values, self.coef)
```

```
    def predict(self, features):
        proba = self.predict_proba(features)
        predictions = np.round(proba, 0)
        return predictions
```

Les méthodes `predict_proba` et `predict` permettent de faciliter l'obtention de prédictions. Les prédictions sont 1 ou 0 ici.
Nous pouvons donc faire un premier test :

```
y_train_binary = y_train.copy()
y_test_binary = y_test.copy()

def transform_to_bin_class(y, target):
    target = 1
    y[y==target] = 'TARGET'
    y[y!='TARGET'] = 'OTHER'
    y[y=='TARGET'] = 1
    y[y!=1] = 0
    return y.astype(float)
```

Nous binarisons la colonne contenant les classes car nous ne pouvons que prédire deux classes pour l'instant. Ici, les 1 resteront donc 1 (c'est un hasard) et les autres chiffres seront remplacés par *0*.

```
y_train_binary = transform_to_bin_class(y_train_binary, 1)
y_test_binary = transform_to_bin_class(y_test_binary, 1)

lreg = LogisticRegression(epochs=30_000, verbose = False)

%time coef_ = lreg.fit(X_train, y_train_binary)
  0%|          | 0/30000 [00:00<?,
      ?it/s]/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:25: RuntimeWarning: divide
      by zero encountered in log
/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:25: RuntimeWarning: invalid value
    encountered in multiply
100%|| 30000/30000 [00:12<00:00, 2350.96it/s]

CPU times: user 19.1 s, sys: 5.83 s, total: 24.9 s
Wall time: 12.8 s
```

Après 30 000 itérations, nous pouvons utiliser notre modèle pour prédire si le chiffre à l'étude est un un ou un autre chiffre.

```
predictions = lreg.predict(X_test)
```

Nous pouvons calculer notre *accuracy* :

```
np.isclose(predictions, y_test_binary.values).sum()/len(y_test_binary)
0.8208955223880597
```

Nous pouvons noter ici que comme nous utilisons un arrondi, le seuil de prédiction est de 0.5. L'implémentation OvR (voir ci-dessous) va choisir le score maximum afin de faire un choix. Par exemple si, pour une observation dont la classe est 4, la régression entraînée pour reconnaître des 3 renvoie une probabilité (plus justement un score) de 0.8 et que celle entraînée pour reconnaître des 4 renvoie 0.90, la décision finale sera 4 mais la personne de disposant que de la première régression fera une prédiction erronée si on lui demande si l'observation est un 3 ou un autre chiffre.

Nous visualisons l'évolution de notre fonction de perte avec la mise à jour itérative des coefficients :

```
plt.plot(lreg.get_loss_evo());
```

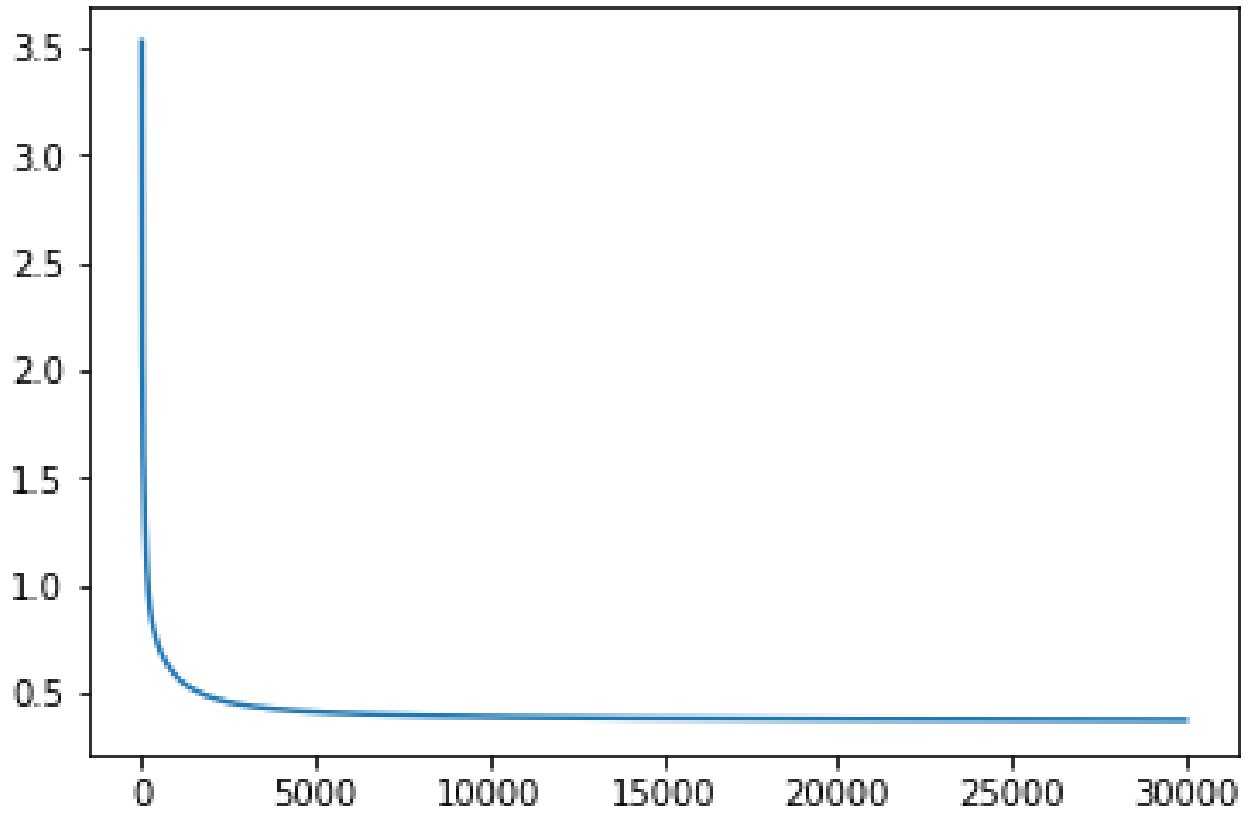

Figure 1: Evolution de la perte en fonction des itérations

Toutefois, notre problème comporte plus de deux classes à prédire. Une première approche serait d'utiliser une régression softmax. La seconde approche, celle retenue ici et utilisée par défaut dans `ScikitLearn` est la méthode OvR(One versus Rest). L'idée est de fitter autant de régressions logistiques qu'il ya de classes à prédire. Pour faire une prédiction nous prendrons la classe associée à la régression qui envoie la valeur la plus élevée. La classe ci-dessous s'appuie sur la classe précédente pour fonctionner. Elle collecte les régressions logistiques simples dans un dictionnaire.

```
class LogisticRegression_OVR:
    def __init__(self, epochs=100, learning_rate=0.1, verbose = False):
        self.epochs = epochs
        self.lr = learning_rate
        self.verbose = verbose
        return None
    def _transform_to_bin_class(self, y, target):
        y_ = y.copy()
        y_[y_==target] = 'TARGET'
        y_[y_!='TARGET'] = 'OTHER'
        y_[y_=='TARGET'] = 1
        y_[y_!=1] = 0
        return y_.astype(float)

    def fit(self,features, target):
        self.log_reg_collection = OrderedDict()
        self.classes = []
        for target_one in target.unique():
            new_y = self._transform_to_bin_class(target, target_one)
            self.log_reg_collection[target_one] = \
```

```
                LogisticRegression(epochs = self.epochs)
            self.log_reg_collection[target_one].fit(features, new_y)
            self.classes.append(target_one)
        return self.log_reg_collection

    def predict_proba(self, X_test):
        proba_list = []
        for i, j in self.log_reg_collection.items():
            proba_list.append(j.predict_proba(X_test))
        pred = np.array(proba_list).T
        return pred

    def predict(self, X_test):
        proba = self.predict_proba(X_test)
        predictions_raw = proba.argmax(1)
        predictions = np.array([self.classes[i] for i in predictions_raw])
        return predictions
```

Afin d'aller un peu plus vite, nous fixons le nombre d'itérations à 10 000 pour chaque régression.

```
log_reg_ovr = LogisticRegression_OVR(epochs=10_000, verbose = False)

regs = log_reg_ovr.fit(X_train, y_train)
```

La méthode `predict` nous renvoie désormais toutes les classes. Nous pouvons donc comparer à notre `y_test` et faire une matrice de confusion :

```
log_reg_ovr.predict(X_test)
```

```
    array([ 1, 1, 9, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
           1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
           1, 1, 1, 1, 9, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
           1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2,
           2, 5, 2, 2, 2, 2, 2, 2, 2, 5, 2, 2, 2, 5, 2, 2, 2,
           2, 2, 2, 5, 2, 2, 2, 2, 2, 2, 2, 2, 2, 5, 5, 2, 2,
           2, 6, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
           2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3,
           3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
           3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 8, 3,
           3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
           3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4,
           4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 7, 4, 4, 4, 4,
           4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
           4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 8, 4, 4, 4, 4,
           4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5,
           5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
           5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 5, 5, 5, 5,
           5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
           7, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6,
           7, 6, 6, 7, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
           6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
           6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
           6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7,
           7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
           7, 7, 7, 7, 7, 7, 7, 6, 7, 7, 7, 7, 7, 7, 7, 7, 7,
           7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
           7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8,
           8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
           8, 8, 8, 8, 8, 8, 8, 8, 2, 8, 8, 8, 8, 8, 8, 8, 8,
           8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 10, 8,
           8, 8, 8, 8, 8, 8, 8, 8, 10, 9, 9, 9, 9, 9, 1, 9, 9,
           9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
           9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
           9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 5, 9, 9, 9, 9, 9, 9,
           9, 9, 9, 9, 9, 9, 9, 9, 10, 10, 10, 10, 10, 10, 10, 2, 10,
          10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
          10, 10, 10, 10, 10, 10, 10, 10, 8, 10, 10, 10, 10, 10, 10, 10, 10,
```

```
        10, 10, 10, 10, 10, 8, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
        10, 10, 10, 10, 10, 10, 10])
```

Nous pouvons également calculer l'*accuracy* :

```
((log_reg_ovr.predict(X_test)) == y_test.values).sum() / len(y_test)
```

```
0.9626865671641791
```

```
def confusion_matrix(predicted: np.array, actual: np.array)-> pd.DataFrame:
    df = pd.DataFrame([predicted, actual], index=['y_Predicted','y_Actual']).T
    confusion_matrix = pd.crosstab(df['y_Actual'], df['y_Predicted'],
                            rownames=['Actual'], colnames=['Predicted'])
    return confusion_matrix

confusion_matrix(log_reg_ovr.predict(X_test), y_test.values)
```

Predicted Actual	1	2	3	4	5	6	7	8	9	10
1	65	0	0	0	0	0	0	0	2	0
2	0	60	0	0	6	1	0	0	0	0
3	0	0	66	0	0	0	0	1	0	0
4	0	0	0	65	0	0	1	1	0	0
5	0	0	0	0	65	1	1	0	0	0
6	0	0	0	0	0	65	2	0	0	0
7	0	0	0	0	0	1	66	0	0	0
8	0	1	0	0	0	0	0	64	0	2
9	1	0	0	0	1	0	0	0	65	0
10	0	1	0	0	0	0	0	2	0	64

5.2 Avec régularisation

Pour éviter le surapprentissage, la régularisation L2 est souvent utilisée. Elle doit être affinée grâce au paramètre λ qui contrôle la force de la régularisation. Avec un λ élevé, les coefficients auront plus de mal à s'ajuster aux données (le paramètre C de ScikitLearn a l'effet inverse).

La fonction de coût devient :

$$J(\theta) = \frac{1}{m}.(-y^T log(h) - (1-y)^T log(1-h)) + \frac{\lambda}{2.m}.\sum_{j=1}^{n} {\theta_j}^2$$

Le gradient devient :

$$\nabla J(\theta) = \frac{1}{m}.X^T.(h-y) + \frac{\lambda}{m}.\theta$$

```
class LogisticRegression_L2:
    def __init__(self, epochs=100, learning_rate=0.1, verbose = False):
        self.epochs = epochs
        self.lr = learning_rate
        self.verbose = verbose
        self.loss_array = np.empty(epochs)
        return None

    def _h(self, features, coefficients):
        x = features @ coefficients
        return 1.0/(1.0+np.exp(-x))

    def fit(self,features, target, lambda_):
        features = features.values
        target = target.values
        m = target.shape[0]

        coef = np.random.randn(features.shape[1])
```

```
        sum_error = 0.0
        if self.verbose == True:
            print("size of features matrix :", features.shape,
                  "\nsize of the target matrix :", target.shape)
        for epoch in tqdm(range(self.epochs)):
            yhat = self._h(features, coef)
            loss = ((-target.T * np.log(yhat) -
                   (1 - target).T * np.log(1 - yhat)).mean() +
                   lambda_/(2*m) * sum(coef**2))
            self.loss_array[epoch] = loss
            gradient = np.dot(features.T, (yhat - target)) / m + lambda_/m * coef
            coef -= self.lr * gradient
            try:
                if self.verbose == True and ((self.epochs % epoch) == 0 or
                                             epoch==self.epochs-1):
                    print("epoch :", epoch,":", "loss function :", round(loss, 2))
                    print("gradient :", gradient)
            except ZeroDivisionError:
                pass
        self.coef = coef
        return self.coef

    def get_loss_evo(self):
        return self.loss_array

    def predict_proba(self, features):
        return self._h(features.values, self.coef)

    def predict(self, features):
        proba = self.predict_proba(features)
        predictions = np.round(proba, 0)
        return predictions
```

```
lreg = LogisticRegression_L2(epochs=30_000, verbose = False)
%time coef_=lreg.fit(X_train, y_train_binary, lambda_=2)
predictions = lreg.predict(X_test)
```

```
  0%|          | 0/30000 [00:00<?,
    ?it/s]/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:26: RuntimeWarning:
    divide by zero encountered in log
100%|| 30000/30000 [00:14<00:00, 2113.56it/s]

CPU times: user 20.6 s, sys: 7 s, total: 27.6 s
Wall time: 14.2 s
```

```
(predictions == y_test_binary.values).sum() / len(y_test)
```

```
0.826865671641791
```

En suivant la même procédure que précédemment, nous implémentons une classe OvR qui ajoute une régularisation L2 à la régression logistique simple.

```
class LogisticRegression_OVR_L2:
    def __init__(self, epochs=100, learning_rate=0.1, verbose = False):
        self.epochs = epochs
        self.lr = learning_rate
        self.verbose = verbose
        return None
    def _transform_to_bin_class(self, y, target):
        y_ = y.copy()
        y_[y_==target] = 'TARGET'
```

```
        y_[y_!='TARGET'] = 'OTHER'
        y_[y_=='TARGET'] = 1
        y_[y_!=1] = 0
        return y_.astype(float)

    def fit(self,features, target, lambda_):
        self.log_reg_collection = OrderedDict()
        self.classes = []
        for target_one in target.unique():
            new_y = self._transform_to_bin_class(target, target_one)
            self.log_reg_collection[target_one] = \
                LogisticRegression_L2(epochs = self.epochs)
            self.log_reg_collection[target_one].fit(features, new_y, lambda_=lambda_)
            self.classes.append(target_one)
        return self.log_reg_collection

    def predict_proba(self, X_test):
        proba_list = []
        for i, j in self.log_reg_collection.items():
            proba_list.append(j.predict_proba(X_test))
        pred = np.array(proba_list).T
        return pred

    def predict(self, X_test):
        proba = self.predict_proba(X_test)
        predictions_raw = proba.argmax(1)
        predictions = np.array([self.classes[i] for i in predictions_raw])
        return predictions

log_reg_ovr = LogisticRegression_OVR_L2(epochs=10_000, verbose = False)

regs = log_reg_ovr.fit(X_train, y_train, lambda_=1)
```

Après le *fitting* de 10 régressions, nous pouvons évaluer l'*accuracy* sur le jeu de test :

```
((log_reg_ovr.predict(X_test)) == y_test.values).sum() / len(y_test)
```

```
0.9671641791044776
```

```
confusion_matrix(log_reg_ovr.predict(X_test), y_test.values)
```

Predicted Actual	1	2	3	4	5	6	7	8	9	10
1	65	0	0	0	0	0	0	0	2	0
2	0	62	0	0	4	1	0	0	0	0
3	0	0	66	0	0	0	0	1	0	0
4	0	0	0	64	0	0	1	2	0	0
5	0	0	0	0	65	1	1	0	0	0
6	0	0	0	0	0	65	2	0	0	0
7	0	0	0	0	0	1	66	0	0	0
8	0	0	0	0	0	0	0	65	0	2
9	0	0	0	0	1	0	0	0	66	0
10	0	1	0	0	0	0	0	2	0	64

La régularisation avec $\lambda = 2$ permet donc d'améliorer légèrement notre score sur le jeu de test. Nous obtenos un score de 0.9672 contre 0.9627 avant.

6 Validation croisée

Afin de choisir la valeur de λ de manière plus scientifique, nous implémentons une procédure de validation croisée.

L'idée principale est de réutiliser la fonction `train_test_split` définie précédemment en faisant varier le paramètre ratio afin de conserver le même noblre de lignes à travers les `folds`. Après avoir appelé la

méthode `fit` sur une instance de `KFold`, nous pouvons accéder aux *features* et à la colonne *target* pour chaque *folds* en utilisant cette syntaxe : `folds[index du fold][0 pour feautures/1 pour target]`.

```
class KFold:
    def __init__(self, n_splits=5):
        self.n_splits=5
        self.ratios = \
            [1/(f+1) for f in list(range(self.n_splits))[::-1]]
    def fit(self, X, y):
        self.X = X.copy()
        self.y = y.copy()

        self.folds = OrderedDict()
        for i, ratio in enumerate(self.ratios):
            train_set, test_set, y_train_set, y_test_set = \
                train_test_split(self.X, self.y, ratio)
            self.folds[i] = (test_set, y_test_set)
            self.X = train_set
            self.y = y_train_set
        return self.folds

def cross_val_score(model, X, y, cv=None, **kwargs):
    folds_dictionnary = cv.fit(X, y)
    scores = []
    for i in folds_dictionnary:
        X_test = folds_dictionnary[i][0]
        y_test = folds_dictionnary[i][1]
        X_train = pd.DataFrame()
        y_train = pd.Series()
        for j in folds_dictionnary:
            if i != j:
                X_train = X_train.append(folds_dictionnary[j][0],
                                         verify_integrity=False)
                y_train = y_train.append(folds_dictionnary[j][1],
                                         verify_integrity=False)
            else:
                pass
        model.fit(X_train, y_train, **kwargs)
        scores.append(((model.predict(X_test)) == y_test.values).sum() / len(y_test))
    return scores

five_folds = KFold(5)

folds = five_folds.fit(X_train, y_train)

folds[3][0].head()
```

Sortie cachée : le tableau des *features* pour le 4ème *fold*

Nous lançons donc la *cross validation* pour les 5 *folds*. L'*accuracy* est maximisé pour $\lambda = 3$ (0.9596). Nous recalculons ensuite la régression sur l'ensemble du jeu d'apprentissage en spécifiant $\lambda = 3$:

```
scores_dict = {}
for i in range(6):
    log_reg_ovr = LogisticRegression_OVR_L2(epochs=10_000, verbose = False)
    scores_dict[i] = \
        cross_val_score(log_reg_ovr, X_train, y_train, cv=five_folds, lambda_ = i)


print(sum(scores_dict[0])/5)
print(sum(scores_dict[1])/5)
print(sum(scores_dict[2])/5)
print(sum(scores_dict[3])/5)
print(sum(scores_dict[4])/5)
print(sum(scores_dict[5])/5)
```

```
    0.9562962962962963
    0.9556695156695156
    0.9572079772079771
    0.9592592592592594
    0.9563532763532765
    0.9556980056980056
```

```
log_reg_ovr = LogisticRegression_OVR_L2(epochs=10_000, verbose = False)
regs = log_reg_ovr.fit(X_train, y_train, lambda_=3)
```

```
((log_reg_ovr.predict(X_test)) == y_test.values).sum() / len(y_test)
```

```
    0.9686567164179104
```

Nous arrivons à un score de 0.969 et améliorons encore notre score obtenu avec $\lambda = 2$ (0.967).