discret_gradient

February 2, 2023

1 Le gradient et le discret

Les méthodes d'optimisation à base de gradient s'appuie sur une fonction d'erreur dérivable qu'on devrait appliquer de préférence sur des variables aléatoires réelles. Ce notebook explore quelques idées.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

1.1 Un petit problème simple

On utilise le jeu de données *iris* disponible dans scikit-learn.

[2]:
```
from sklearn import datasets

iris = datasets.load_iris()
X = iris.data[:, :2]  # we only take the first two features.
Y = iris.target
```

On cale une régression logistique. On ne distingue pas apprentissage et test car ce n'est pas le propos de ce notebook.

[3]:
```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(multi_class="ovr", solver="liblinear")
clf.fit(X, Y)
```

[3]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr',
          n_jobs=None, penalty='l2', random_state=None, solver='liblinear',
          tol=0.0001, verbose=0, warm_start=False)
```

Puis on calcule la matrice de confusion.

[4]:
```
from sklearn.metrics import confusion_matrix
pred = clf.predict(X)
confusion_matrix(Y, pred)
```

[4]:
```
array([[49,  1,  0],
       [ 2, 21, 27],
       [ 1,  4, 45]], dtype=int64)
```

1.2 Multiplication des observations

Le paramètre `multi_class='ovr'` stipule que le modèle cache en fait l'estimation de 3 régressions logistiques binaire. Essayons de n'en faire qu'une seule en ajouter le label `Y` aux variables. Soit un couple $(X_i \in \mathbb{R}^d, Y_i \in \mathbb{N})$ qui correspond à une observation pour un problème multi-classe. Comme il y a C classes, on multiplie cette ligne par le nombre de classes C pour obtenir :

$$\forall c \in [1, ..., C], \left\{ \begin{array}{l} X'_i = (X_{i,1}, ..., X_{i,d}, Y_{i,1}, ..., Y_{i,C}) \\ Y'_i = \mathbb{1}_{Y_i=c} \\ Y_{i,k} = \mathbb{1}_{c=k} \end{array} \right.$$

Voyons ce que cela donne sur un exemple :

[5]:
```
import numpy
import pandas

def multiplie(X, Y, classes=None):
    if classes is None:
        classes = numpy.unique(Y)
    XS = []
    YS = []
    for i in classes:
        X2 = numpy.zeros((X.shape[0], 3))
        X2[:,i] = 1
        Yb = Y == i
        XS.append(numpy.hstack([X, X2]))
        Yb = Yb.reshape((len(Yb), 1))
        YS.append(Yb)

    Xext = numpy.vstack(XS)
    Yext = numpy.vstack(YS)
    return Xext, Yext

x, y = multiplie(X[:1,:], Y[:1], [0, 1, 2])
df = pandas.DataFrame(numpy.hstack([x, y]))
df.columns = ["X1", "X2", "Y0", "Y1", "Y2", "Y'"]
df
```

[5]:

	X1	X2	Y0	Y1	Y2	Y'
0	5.1	3.5	1.0	0.0	0.0	1.0
1	5.1	3.5	0.0	1.0	0.0	0.0
2	5.1	3.5	0.0	0.0	1.0	0.0

Trois colonnes ont été ajoutées côté X, la ligne a été multipliée 3 fois, la dernière colonne est Y qui ne vaut 1 que lorsque le 1 est au bon endroit dans une des colonnes ajoutées. Le problème de classification qui été de prédire la bonne classe devient : est-ce la classe à prédire est k ? On applique cela sur toutes les lignes de la base et cela donne :

[6]:
```
Xext, Yext = multiplie(X, Y)
numpy.hstack([Xext, Yext])
df = pandas.DataFrame(numpy.hstack([Xext, Yext]))
df.columns = ["X1", "X2", "Y0", "Y1", "Y2", "Y'"]
df.iloc[numpy.random.permutation(df.index), :].head(n=10)
```

[6]:

	X1	X2	Y0	Y1	Y2	Y'
414	6.7	3.3	0.0	0.0	1.0	1.0
125	5.5	2.5	0.0	0.0	1.0	0.0

```
394  6.7  3.1  0.0  0.0  1.0  0.0
411  6.0  2.2  1.0  0.0  0.0  0.0
95   7.6  3.0  0.0  0.0  1.0  1.0
64   5.8  2.6  0.0  0.0  1.0  0.0
309  5.0  3.4  0.0  0.0  1.0  0.0
7    6.7  3.0  0.0  1.0  0.0  0.0
182  6.1  2.8  1.0  0.0  0.0  0.0
49   4.7  3.2  0.0  1.0  0.0  0.0
```

[7]:
```
from sklearn.ensemble import GradientBoostingClassifier
clf = GradientBoostingClassifier()
clf.fit(Xext, Yext.ravel())
```

[7]:
```
GradientBoostingClassifier(criterion='friedman_mse', init=None,
              learning_rate=0.1, loss='deviance', max_depth=3,
              max_features=None, max_leaf_nodes=None,
              min_impurity_decrease=0.0, min_impurity_split=None,
              min_samples_leaf=1, min_samples_split=2,
              min_weight_fraction_leaf=0.0, n_estimators=100,
              n_iter_no_change=None, presort='auto', random_state=None,
              subsample=1.0, tol=0.0001, validation_fraction=0.1,
              verbose=0, warm_start=False)
```

[8]:
```
pred = clf.predict(Xext)
confusion_matrix(Yext, pred)
```

[8]:
```
array([[278,  22],
       [ 25, 125]], dtype=int64)
```

1.3 Introduire du bruit

Un des problèmes de cette méthode est qu'on ajoute une variable binaire pour un problème résolu à l'aide d'une optimisation à base de gradient. C'est moyen. Pas de problème, changeons un peu la donne.

[9]:
```
def multiplie_bruit(X, Y, classes=None):
    if classes is None:
        classes = numpy.unique(Y)
    XS = []
    YS = []
    for i in classes:
        # X2 = numpy.random.randn((X.shape[0]* 3)).reshape(X.shape[0], 3) * 0.1
        X2 = numpy.random.random((X.shape[0], 3)) * 0.2
        X2[:,i] += 1
        Yb = Y == i
        XS.append(numpy.hstack([X, X2]))
        Yb = Yb.reshape((len(Yb), 1))
        YS.append(Yb)

    Xext = numpy.vstack(XS)
    Yext = numpy.vstack(YS)
    return Xext, Yext

x, y = multiplie_bruit(X[:1,:], Y[:1], [0, 1, 2])
df = pandas.DataFrame(numpy.hstack([x, y]))
```

```
df.columns = ["X1", "X2", "Y0", "Y1", "Y2", "Y'"]
df
```

[9]:

	X1	X2	Y0	Y1	Y2	Y'
0	5.1	3.5	1.107461	0.166893	0.018765	1.0
1	5.1	3.5	0.162464	1.187359	0.187721	0.0
2	5.1	3.5	0.086876	0.178472	1.179201	0.0

Le problème est le même qu'avant excepté les variables Y_i qui sont maintenant réel. Au lieu d'être nul, on prend une valeur $Y_i < 0.4$.

[10]:
```
Xextb, Yextb = multiplie_bruit(X, Y)
df = pandas.DataFrame(numpy.hstack([Xextb, Yextb]))
df.columns = ["X1", "X2", "Y0", "Y1", "Y2", "Y'"]
df.iloc[numpy.random.permutation(df.index), :].head(n=10)
```

[10]:

	X1	X2	Y0	Y1	Y2	Y'
295	5.5	2.6	0.197643	1.199976	0.180766	1.0
46	5.2	3.4	0.178395	0.190600	1.159765	0.0
187	6.7	3.1	0.188947	1.093288	0.139723	1.0
210	6.9	3.1	0.095428	0.182643	1.037533	1.0
29	5.5	3.5	1.131419	0.077241	0.177483	1.0
315	6.4	3.2	0.099738	0.197291	1.035431	1.0
152	5.8	2.7	0.069061	0.045325	1.061221	0.0
168	6.5	2.8	0.093164	1.177413	0.095890	1.0
348	6.9	3.1	1.094184	0.196944	0.083975	0.0
261	6.3	2.8	0.197558	0.080273	1.009379	1.0

[11]:
```
from sklearn.ensemble import GradientBoostingClassifier
clfb = GradientBoostingClassifier()
clfb.fit(Xextb, Yextb.ravel())
```

[11]:
```
GradientBoostingClassifier(criterion='friedman_mse', init=None,
              learning_rate=0.1, loss='deviance', max_depth=3,
              max_features=None, max_leaf_nodes=None,
              min_impurity_decrease=0.0, min_impurity_split=None,
              min_samples_leaf=1, min_samples_split=2,
              min_weight_fraction_leaf=0.0, n_estimators=100,
              n_iter_no_change=None, presort='auto', random_state=None,
              subsample=1.0, tol=0.0001, validation_fraction=0.1,
              verbose=0, warm_start=False)
```

[12]:
```
predb = clfb.predict(Xextb)
confusion_matrix(Yextb, predb)
```

[12]:
```
array([[299,   1],
       [ 10, 140]], dtype=int64)
```

C'est un petit peu mieux.

1.4 Comparaisons de plusieurs modèles

On cherche maintenant à comparer le gain en introduisant du bruit pour différents modèles.

[13]:
```
def error(model, x, y):
    p = model.predict(x)
    cm = confusion_matrix(y, p)
    return (cm[1,0] + cm[0,1]) / cm.sum()

def comparaison(model, X, Y):

    if isinstance(model, tuple):
        clf = model[0](**model[1])
        clfb = model[0](**model[1])
        model = model[0]
    else:
        clf = model()
        clfb = model()

    Xext, Yext = multiplie(X, Y)
    clf.fit(Xext, Yext.ravel())
    err = error(clf, Xext, Yext)

    Xextb, Yextb = multiplie_bruit(X, Y)
    clfb.fit(Xextb, Yextb.ravel())
    errb = error(clfb, Xextb, Yextb)
    return dict(model=model.__name__, err1=err, err2=errb)

from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier, ExtraTreeClassifier
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier,␣
 ↪AdaBoostClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier, RadiusNeighborsClassifier
from xgboost import XGBClassifier

models = [(LogisticRegression, dict(multi_class="ovr", solver="liblinear")),
          GradientBoostingClassifier,
          (RandomForestClassifier, dict(n_estimators=20)),
          DecisionTreeClassifier,
          ExtraTreeClassifier,
          XGBClassifier,
          (ExtraTreesClassifier, dict(n_estimators=20)),
          (MLPClassifier, dict(activation="logistic")),
          GaussianNB, KNeighborsClassifier,
          (AdaBoostClassifier,␣
 ↪dict(base_estimator=LogisticRegression(multi_class="ovr", solver="liblinear"),
                                    algorithm="SAMME"))]

res = [comparaison(model, X, Y) for model in models]
df = pandas.DataFrame(res)
df.sort_values("model")
```

[13]:

	err1	err2	model
10	0.333333	0.333333	AdaBoostClassifier
3	0.048889	0.000000	DecisionTreeClassifier

```
4   0.048889  0.000000         ExtraTreeClassifier
6   0.048889  0.000000        ExtraTreesClassifier
8   0.333333  0.333333                  GaussianNB
1   0.104444  0.044444  GradientBoostingClassifier
9   0.104444  0.091111        KNeighborsClassifier
0   0.333333  0.333333          LogisticRegression
7   0.333333  0.333333               MLPClassifier
2   0.053333  0.002222      RandomForestClassifier
5   0.333333  0.053333                XGBClassifier
```

err1 correspond à Y_0, Y_1, Y_2 binaire, *err2* aux mêmes variables mais avec un peu de bruit. L'ajout ne semble pas faire décroître la performance et l'améliore dans certains cas. C'est une piste à suivre. Reste à savoir si les modèles n'apprennent pas le bruit.

1.5 Avec une ACP

On peut faire varier le nombre de composantes, j'en ai gardé qu'une. L'ACP est appliquée après avoir ajouté les variables binaires ou binaires bruitées. Le résultat est sans équivoque. Aucun modèle ne parvient à apprendre sans l'ajout de bruit.

[14]:
```
from sklearn.decomposition import PCA

def comparaison_ACP(model, X, Y):

    if isinstance(model, tuple):
        clf = model[0](**model[1])
        clfb = model[0](**model[1])
        model = model[0]
    else:
        clf = model()
        clfb = model()

    axes = 1
    solver = "full"
    Xext, Yext = multiplie(X, Y)
    Xext = PCA(n_components=axes, svd_solver=solver).fit_transform(Xext)
    clf.fit(Xext, Yext.ravel())
    err = error(clf, Xext, Yext)

    Xextb, Yextb = multiplie_bruit(X, Y)
    Xextb = PCA(n_components=axes, svd_solver=solver).fit_transform(Xextb)
    clfb.fit(Xextb, Yextb.ravel())
    errb = error(clfb, Xextb, Yextb)
    return dict(modelACP=model.__name__, errACP1=err, errACP2=errb)

res = [comparaison_ACP(model, X, Y) for model in models]
dfb = pandas.DataFrame(res)
pandas.concat([ df.sort_values("model"), dfb.sort_values("modelACP")], axis=1)
```

[14]:
```
        err1      err2                      model   errACP1   errACP2  \
10  0.333333  0.333333          AdaBoostClassifier  0.333333  0.333333
3   0.048889  0.000000      DecisionTreeClassifier  0.333333  0.000000
4   0.048889  0.000000         ExtraTreeClassifier  0.333333  0.000000
6   0.048889  0.000000        ExtraTreesClassifier  0.333333  0.000000
```

```
8   0.333333  0.333333                 GaussianNB  0.333333  0.333333
1   0.104444  0.044444  GradientBoostingClassifier  0.333333  0.224444
9   0.104444  0.091111       KNeighborsClassifier  0.335556  0.340000
0   0.333333  0.333333         LogisticRegression  0.333333  0.333333
7   0.333333  0.333333              MLPClassifier  0.333333  0.333333
2   0.053333  0.002222     RandomForestClassifier  0.333333  0.024444
5   0.333333  0.053333              XGBClassifier  0.333333  0.315556

                    modelACP
10         AdaBoostClassifier
3      DecisionTreeClassifier
4         ExtraTreeClassifier
6        ExtraTreesClassifier
8                  GaussianNB
1   GradientBoostingClassifier
9        KNeighborsClassifier
0          LogisticRegression
7               MLPClassifier
2      RandomForestClassifier
5               XGBClassifier
```

1.6 Base d'apprentissage et de test

Cette fois-ci, on s'intéresse à la qualité des frontières que les modèles trouvent en vérifiant sur une base de test que l'apprentissage s'est bien passé.

[15]:
```
from sklearn.model_selection import train_test_split

def comparaison_train_test(models, X, Y, mbruit=multiplie_bruit, acp=None):

    axes = acp
    solver = "full"

    ind = numpy.random.permutation(numpy.arange(X.shape[0]))
    X = X[ind,:]
    Y = Y[ind]
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=1./3)

    res = []
    for model in models:

        if isinstance(model, tuple):
            clf = model[0](**model[1])
            clfb = model[0](**model[1])
            model = model[0]
        else:
            clf = model()
            clfb = model()

        Xext_train, Yext_train = multiplie(X_train, Y_train)
        Xext_test, Yext_test = multiplie(X_test, Y_test)
        if acp:
            Xext_train_ = Xext_train
```

```
            Xext_test_ = Xext_test
            acp_model = PCA(n_components=axes, svd_solver=solver).fit(Xext_train)
            Xext_train = acp_model.transform(Xext_train)
            Xext_test = acp_model.transform(Xext_test)
        clf.fit(Xext_train, Yext_train.ravel())

        err_train = error(clf, Xext_train, Yext_train)
        err_test = error(clf, Xext_test, Yext_test)

        Xextb_train, Yextb_train = mbruit(X_train, Y_train)
        Xextb_test, Yextb_test = mbruit(X_test, Y_test)
        if acp:
            acp_model = PCA(n_components=axes, svd_solver=solver).fit(Xextb_train)
            Xextb_train = acp_model.transform(Xextb_train)
            Xextb_test = acp_model.transform(Xextb_test)
            Xext_train = acp_model.transform(Xext_train_)
            Xext_test = acp_model.transform(Xext_test_)
        clfb.fit(Xextb_train, Yextb_train.ravel())

        errb_train = error(clfb, Xextb_train, Yextb_train)
        errb_train_clean = error(clfb, Xext_train, Yext_train)
        errb_test = error(clfb, Xextb_test, Yextb_test)
        errb_test_clean = error(clfb, Xext_test, Yext_test)

        res.append(dict(modelTT=model.__name__, err_train=err_train,␣
↪err2_train=errb_train,
                err_test=err_test, err2_test=errb_test,␣
↪err2b_test_clean=errb_test_clean,
                err2b_train_clean=errb_train_clean))

    dfb = pandas.DataFrame(res)
    dfb = dfb[["modelTT", "err_train", "err2_train", "err2b_train_clean", "err_test",␣
↪"err2_test", "err2b_test_clean"]]
    dfb = dfb.sort_values("modelTT")
    return dfb

dfb = comparaison_train_test(models, X, Y)
dfb
```

[15]:

	modelTT	err_train	err2_train	err2b_train_clean	err_test	err2_test	err2b_test_clean
10	AdaBoostClassifier	0.333333	0.333333	0.333333	0.333333	0.333333	0.333333
3	DecisionTreeClassifier	0.026667	0.000000	0.226667	0.333333	0.333333	0.333333
4	ExtraTreeClassifier	0.026667	0.000000	0.253333	0.333333	0.333333	0.333333
6	ExtraTreesClassifier	0.026667	0.000000	0.140000	0.333333	0.333333	0.333333
8	GaussianNB	0.333333	0.333333	0.333333	0.333333	0.333333	0.333333
1	GradientBoostingClassifier	0.080000	0.013333	0.176667	0.333333	0.333333	0.333333
9	KNeighborsClassifier	0.070000	0.076667	0.073333	0.333333	0.333333	0.333333
0	LogisticRegression	0.333333	0.333333	0.333333	0.333333	0.333333	0.333333
7	MLPClassifier	0.333333	0.333333	0.333333	0.333333	0.333333	0.333333
2	RandomForestClassifier	0.026667	0.000000	0.156667	0.333333	0.333333	0.333333
5	XGBClassifier	0.106667	0.036667	0.333333	0.333333	0.333333	0.333333

```
10  0.333333   0.333333          0.333333
3   0.206667   0.273333          0.313333
4   0.213333   0.253333          0.273333
6   0.200000   0.213333          0.220000
8   0.333333   0.333333          0.333333
1   0.186667   0.246667          0.240000
9   0.160000   0.160000          0.166667
0   0.333333   0.333333          0.333333
7   0.333333   0.333333          0.333333
2   0.206667   0.266667          0.213333
5   0.193333   0.280000          0.346667
```

Les colonnes *err2b_train_clean* et *err2b_test_clean* sont les erreurs obtenues par des modèles appris sur des colonnes bruitées et testées sur des colonnes non bruitées ce qui est le véritable test. On s'aperçoit que les performances sont très dégradées sur la base d'test. Une raison est que le bruit choisi ajouté n'est pas centré. Corrigeons cela.

[16]:
```
def multiplie_bruit_centree(X, Y, classes=None):
    if classes is None:
        classes = numpy.unique(Y)
    XS = []
    YS = []
    for i in classes:
        # X2 = numpy.random.randn((X.shape[0]* 3)).reshape(X.shape[0], 3) * 0.1
        X2 = numpy.random.random((X.shape[0], 3)) * 0.2 - 0.1
        X2[:,i] += 1
        Yb = Y == i
        XS.append(numpy.hstack([X, X2]))
        Yb = Yb.reshape((len(Yb), 1))
        YS.append(Yb)

    Xext = numpy.vstack(XS)
    Yext = numpy.vstack(YS)
    return Xext, Yext

dfb = comparaison_train_test(models, X, Y, mbruit=multiplie_bruit_centree, acp=None)
dfb
```

[16]:
```
                       modelTT  err_train  err2_train  err2b_train_clean  \
10          AdaBoostClassifier   0.333333    0.333333           0.333333
3       DecisionTreeClassifier   0.033333    0.000000           0.143333
4          ExtraTreeClassifier   0.033333    0.000000           0.143333
6         ExtraTreesClassifier   0.033333    0.000000           0.123333
8                   GaussianNB   0.333333    0.333333           0.333333
1   GradientBoostingClassifier   0.083333    0.013333           0.203333
9         KNeighborsClassifier   0.106667    0.106667           0.100000
0           LogisticRegression   0.333333    0.333333           0.333333
7                MLPClassifier   0.333333    0.333333           0.333333
2       RandomForestClassifier   0.040000    0.000000           0.176667
5                XGBClassifier   0.080000    0.063333           0.170000

    err_test  err2_test  err2b_test_clean
10  0.333333   0.333333          0.333333
3   0.193333   0.273333          0.206667
```

```
4   0.226667   0.233333          0.180000
6   0.200000   0.213333          0.193333
8   0.333333   0.333333          0.333333
1   0.193333   0.226667          0.280000
9   0.180000   0.180000          0.193333
0   0.333333   0.333333          0.333333
7   0.333333   0.333333          0.333333
2   0.206667   0.240000          0.253333
5   0.186667   0.220000          0.240000
```

C'est mieux mais on en conclut que dans la plupart des cas, la meilleure performance sur la base d'apprentissage avec le bruit ajouté est due au fait que les modèles apprennent par coeur. Sur la base de test, les performances ne sont pas meilleures. Une erreur de 33% signifie que la réponse du classifieur est constante. On multiplie les exemples.

[17]:
```
def multiplie_bruit_centree_duplique(X, Y, classes=None):
    if classes is None:
        classes = numpy.unique(Y)
    XS = []
    YS = []
    for i in classes:

        for k in range(0,5):
            #X2 = numpy.random.randn((X.shape[0]* 3)).reshape(X.shape[0], 3) * 0.3
            X2 = numpy.random.random((X.shape[0], 3)) * 0.8 - 0.4
            X2[:,i] += 1
            Yb = Y == i
            XS.append(numpy.hstack([X, X2]))
            Yb = Yb.reshape((len(Yb), 1))
            YS.append(Yb)

    Xext = numpy.vstack(XS)
    Yext = numpy.vstack(YS)
    return Xext, Yext

dfb = comparaison_train_test(models, X, Y, mbruit=multiplie_bruit_centree_duplique,␣
 ↪acp=None)
dfb
```

[17]:
```
                      modelTT  err_train  err2_train  err2b_train_clean  \
10          AdaBoostClassifier   0.333333    0.333333           0.333333
3       DecisionTreeClassifier   0.040000    0.000000           0.120000
4          ExtraTreeClassifier   0.040000    0.000000           0.073333
6         ExtraTreesClassifier   0.040000    0.000000           0.066667
8                   GaussianNB   0.333333    0.333333           0.333333
1   GradientBoostingClassifier   0.086667    0.087333           0.166667
9         KNeighborsClassifier   0.110000    0.094667           0.106667
0           LogisticRegression   0.333333    0.333333           0.333333
7                MLPClassifier   0.333333    0.333333           0.333333
2       RandomForestClassifier   0.046667    0.000667           0.090000
5                XGBClassifier   0.123333    0.108667           0.173333

    err_test  err2_test  err2b_test_clean
10  0.333333   0.333333          0.333333
```

```
3   0.180000   0.209333          0.240000
4   0.213333   0.232000          0.220000
6   0.213333   0.168000          0.160000
8   0.333333   0.333333          0.333333
1   0.173333   0.192000          0.186667
9   0.113333   0.158667          0.153333
0   0.333333   0.333333          0.333333
7   0.333333   0.333333          0.333333
2   0.160000   0.188000          0.226667
5   0.153333   0.204000          0.193333
```

Cela fonctionne un peu mieux le fait d'ajouter du hasard ne permet pas d'obtenir des gains significatifs à part pour le modèle SVC.

[18]:
```
def multiplie_bruit_centree_duplique_rebalance(X, Y, classes=None):
    if classes is None:
        classes = numpy.unique(Y)
    XS = []
    YS = []
    for i in classes:

        X2 = numpy.random.random((X.shape[0], 3)) * 0.8 - 0.4
        X2[:,i] += 1  # * ((i % 2) * 2 - 1)
        Yb = Y == i
        XS.append(numpy.hstack([X, X2]))
        Yb = Yb.reshape((len(Yb), 1))
        YS.append(Yb)


    Xext = numpy.vstack(XS)
    Yext = numpy.vstack(YS)
    return Xext, Yext

dfb = comparaison_train_test(models, X, Y,␣
 ↪mbruit=multiplie_bruit_centree_duplique_rebalance)
dfb
```

[18]:
```
                       modelTT  err_train  err2_train  err2b_train_clean  \
10          AdaBoostClassifier   0.333333    0.333333           0.333333
3       DecisionTreeClassifier   0.033333    0.000000           0.143333
4          ExtraTreeClassifier   0.033333    0.000000           0.246667
6         ExtraTreesClassifier   0.033333    0.000000           0.143333
8                   GaussianNB   0.333333    0.333333           0.333333
1   GradientBoostingClassifier   0.090000    0.013333           0.133333
9         KNeighborsClassifier   0.103333    0.110000           0.123333
0           LogisticRegression   0.333333    0.333333           0.333333
7                MLPClassifier   0.333333    0.333333           0.333333
2       RandomForestClassifier   0.040000    0.000000           0.146667
5                XGBClassifier   0.100000    0.033333           0.210000

    err_test  err2_test  err2b_test_clean
10  0.333333   0.333333          0.333333
3   0.200000   0.233333          0.193333
4   0.233333   0.320000          0.300000
```

6	0.206667	0.220000	0.180000
8	0.333333	0.333333	0.333333
1	0.220000	0.206667	0.186667
9	0.206667	0.180000	0.186667
0	0.333333	0.333333	0.333333
7	0.333333	0.333333	0.333333
2	0.180000	0.266667	0.173333
5	0.206667	0.240000	0.246667

1.7 Petite explication

Dans tout le notebook, le score de la régression logistique est nul. Elle ne parvient pas à apprendre tout simplement parce que le problème choisi n'est pas linéaire séparable. S'il l'était, cela voudrait dire que le problème suivant l'est aussi.

[19]:
```
M = numpy.zeros((9, 6))
Y = numpy.zeros((9, 1))
for i in range(0, 9):
    M[i, i//3] = 1
    M[i, i%3+3] = 1
    Y[i] = 1 if i//3 == i%3 else 0
M,Y
```

[19]:
```
(array([[1., 0., 0., 1., 0., 0.],
        [1., 0., 0., 0., 1., 0.],
        [1., 0., 0., 0., 0., 1.],
        [0., 1., 0., 1., 0., 0.],
        [0., 1., 0., 0., 1., 0.],
        [0., 1., 0., 0., 0., 1.],
        [0., 0., 1., 1., 0., 0.],
        [0., 0., 1., 0., 1., 0.],
        [0., 0., 1., 0., 0., 1.]]), array([[1.],
        [0.],
        [0.],
        [0.],
        [1.],
        [0.],
        [0.],
        [0.],
        [1.]]))
```

[20]:
```
clf = LogisticRegression(multi_class="ovr", solver="liblinear")
clf.fit(M, Y.ravel())
```

[20]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr',
          n_jobs=None, penalty='l2', random_state=None, solver='liblinear',
          tol=0.0001, verbose=0, warm_start=False)
```

[21]:
```
clf.predict(M)
```

[21]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

A revisiter.

[22]: