0. Introduction of the Problem:

Machine Learning Fairness:

Machine Learning classifiers are increasingly being used to assit decision makings, however, it is quite possible that the classifier makes decisions for people belonging to different social groups with different misclassification rates (e.g. misclassification rates for females are higher than males), thereby placing these groups at an unfair advantage. In this problem, we implement the methods introduced by *Fairness Beyond Disparate Treatment & Disparate Impact: Learning Classification without Disparate Mistreatment* to avoid such unfairness. To evaluate the performance of the classifiers implemented, accuracy and calibration are used as two main metrics.

DM model explained :

Since there is a lot of fairness notion and that only one notion can be tackled at once, we should elaborate on the main notion we are tackling here. It is the disparate mistreatment we are trying to avoid. This notion of fairness can be partitioned into several sub-notions:

- Overall Missclassification (OMR):
$$P[\hat{y} \neq y | z = 0] = P[\hat{y} \neq y | z = 1]$$
- False Negative Rate (FNR):
$$P[\hat{y} \neq y | z = 0, y = 0] = P[\hat{y} \neq y | z = 1, y = 0]$$
- False Positive Rate (FPR):
$$P[\hat{y} \neq y | z = 0, y = 1] = P[\hat{y} \neq y | z = 1, y = 1]$$

where $z \in \{0, 1\}$ corresponds to our sensitive feature and $y \in \{0, 1\}$ is the target.

Suppose now we want to tackle the Dipsarate Mistreatment in the FNR. Let $l_w(X, y)$ be the loss of our classifier where X corresponds to the selected features, than we would like to solve the following problem :

$$\begin{aligned} \min_{w \in \mathcal{R}^d} \quad & l_w(X, y) \\ \text{subject to} \quad & P[\hat{y}_w \neq y | z = 0, y = 0] - P[\hat{y}_w \neq y | z = 1, y = 0] \leq c \\ & P[\hat{y}_w \neq y | z = 0, y = 0] - P[\hat{y}_w \neq y | z = 1, y = 0] \geq -c \end{aligned}$$

But as explain [Zafar et al (https://arxiv.org/pdf/1610.08452.pdf)](https://arxiv.org/pdf/1610.08452.pdf) in their paper, this problem can not be solved easily and instead they proceed to estimate the covariance between the sensitive feature z and the signed distance $g_w(X)$ between the feature vectors of misclassified users and the classifier decision boundary as a way to tackle disparate mistreatment :

$$Cov(z, g_w(X)) \approx \frac{1}{N} \sum_{y,X,z \in \mathcal{D}} (z - \bar{z}) g_w(y, X)$$

The signed distance $g_w(X)$ between the feature vectors of misclassified users and the classifier decision boundary depends on the disparate mistreatment you want to tackle as well as the boundary decision of your classifier :

- For instance for OMR :

$$g_w^{OMR}(y, X) = \min(0, (2y-1)d_w(X))$$

- For FNR :

$$g_w^{FNR}(y, X) = \min(0, (1-y)(2y-1)d_w(X))$$

- For FPR :

$$g_w^{FPR}(y, X) = \min(0, y(2y-1)d_w(X))$$

For linear model, the boundary decision is $d_w(X) = Xw$ resulting that $g_w(y, X)$ is a concave function. This information is crucial because it will allow us to solve the DM problem using Convex-Concave solver. Indeed the resulting problem can be formulated as the following in spliting z into $z = 0$ and $z = 1$:

$$\begin{aligned} \min_{w \in \mathcal{R}^d} \quad & l_w(X, y) \\ \text{subject to} \quad & -\frac{N_1}{N} \textstyle\sum_{X,y \in \mathcal{D}_0} g_w(X, y) + \frac{N_0}{N} \sum_{X,y \in \mathcal{D}_1} g_w(X, y) \leq c \\ & -\frac{N_1}{N} \textstyle\sum_{X,y \in \mathcal{D}_0} g_w(X, y) + \frac{N_0}{N} \sum_{X,y \in \mathcal{D}_1} g_w(X, y) \geq -c \end{aligned}$$

This problem is Convex-Concave and can be solved using [DCCP (https://arxiv.org/pdf/2106.00772.pdf)](https://arxiv.org/pdf/2106.00772.pdf). This DCCP solver was implemented but a more robust version using the package DCCP developed in the paper is also used. Finally, for a logistic regression we end up solving the following Convex-Concave optimization problem:

$$\begin{aligned} \min_{w \in \mathcal{R}^d} \quad & \frac{1}{N} \textstyle\sum_{X,y \in \mathcal{D}} \log(1 + e^{Xw}) - yXw \\ \text{subject to} \quad & -\frac{N_1}{N} \textstyle\sum_{X,y \in \mathcal{D}_0} g_w(X, y) + \frac{N_0}{N} \sum_{X,y \in \mathcal{D}_1} g_w(X, y) \leq c \\ & -\frac{N_1}{N} \textstyle\sum_{X,y \in \mathcal{D}_0} g_w(X, y) + \frac{N_0}{N} \sum_{X,y \in \mathcal{D}_1} g_w(X, y) \geq -c \end{aligned}$$

Loading the used packages

In [2]:
```
#Loading the desired packages
!pip install dccp
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import cvxpy as cp
import dccp
import matplotlib.pyplot as plt
from IPython.core.display import display, HTML
import warnings
warnings.filterwarnings('ignore')
```

```
Requirement already satisfied: dccp in /usr/local/lib/python3.7/dist-packages (1.0.4)
Requirement already satisfied: cvxpy>=0.3.5 in /usr/local/lib/python3.7/dist-packages
(from dccp) (1.0.31)
Requirement already satisfied: osqp>=0.4.1 in /usr/local/lib/python3.7/dist-packages (f
rom cvxpy>=0.3.5->dccp) (0.6.2.post0)
Requirement already satisfied: scipy>=1.1.0 in /usr/local/lib/python3.7/dist-packages
(from cvxpy>=0.3.5->dccp) (1.4.1)
Requirement already satisfied: multiprocess in /usr/local/lib/python3.7/dist-packages
(from cvxpy>=0.3.5->dccp) (0.70.12.2)
Requirement already satisfied: numpy>=1.15 in /usr/local/lib/python3.7/dist-packages (f
rom cvxpy>=0.3.5->dccp) (1.21.5)
Requirement already satisfied: ecos>=2 in /usr/local/lib/python3.7/dist-packages (from
cvxpy>=0.3.5->dccp) (2.0.10)
Requirement already satisfied: scs>=1.1.3 in /usr/local/lib/python3.7/dist-packages (fr
om cvxpy>=0.3.5->dccp) (3.2.0)
Requirement already satisfied: qdldl in /usr/local/lib/python3.7/dist-packages (from os
qp>=0.4.1->cvxpy>=0.3.5->dccp) (0.1.5.post0)
Requirement already satisfied: dill>=0.3.4 in /usr/local/lib/python3.7/dist-packages (f
rom multiprocess->cvxpy>=0.3.5->dccp) (0.3.4)
```

In [ ]:
```
!git init
!git pull https://clement-micol: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@github.com/TZstat
```

1. Data Processing:

Loading the data:

In [6]:
```
data = pd.read_csv("/content/data/compas-scores-two-years.csv")
data
```

Out[6]:

	id	name	first	last	compas_screening_date	sex	dob	age	age_cat
0	1	miguel hernandez	miguel	hernandez	2013-08-14	Male	1947-04-18	69	Greater than 45
1	3	kevon dixon	kevon	dixon	2013-01-27	Male	1982-01-22	34	25 - 45
2	4	ed philo	ed	philo	2013-04-14	Male	1991-05-14	24	Less than 25
3	5	marcu brown	marcu	brown	2013-01-13	Male	1993-01-21	23	Less than 25
4	6	bouthy pierrelouis	bouthy	pierrelouis	2013-03-26	Male	1973-01-22	43	25 - 45
...	...	...	...	...	...	...	...	...	...
7209	10996	steven butler	steven	butler	2013-11-23	Male	1992-07-17	23	Less than 25
7210	10997	malcolm simmons	malcolm	simmons	2014-02-01	Male	1993-03-25	23	Less than 25
7211	10999	winston gregory	winston	gregory	2014-01-14	Male	1958-10-01	57	Greater than 45
7212	11000	farrah jean	farrah	jean	2014-03-09	Female	1982-11-17	33	25 - 45
7213	11001	florencia sanmartin	florencia	sanmartin	2014-06-30	Female	1992-12-18	23	Less than 25

7214 rows × 53 columns

Selecting the relevant features :

In [7]:
```
# Select only Caucasian and African-American as our sensitive feature
data = data[data["race"].isin(["Caucasian","African-American"])]

#Calculate length of stay
data["length_stay"] = pd.to_datetime(data["c_jail_out"]) - pd.to_datetime(data['c_jail_in'
data["length_stay"] = data["length_stay"].apply(lambda x: x.days)
data = data.drop(columns = ["c_jail_in","c_jail_out"])
data['length_stay'] = data["length_stay"].apply(lambda x: 0 if x <= 7 else x)
data['length_stay'] = data["length_stay"].apply(lambda x: 1 if 7< x <= 90 else x)
data['length_stay'] = data["length_stay"].apply(lambda x: 2 if 90 < x <= 180 else x)
data['length_stay'] = data["length_stay"].apply(lambda x: 3 if x > 180 else x)
```

In [8]:
```
# Select the features to predict the target y
X = data[["sex","age_cat","priors_count.1","c_charge_degree","length_stay","race"]]
y = data["two_year_recid"]

# Encode the categorical_features
for categorical_feature in ["sex","age_cat","c_charge_degree","length_stay","race"]:
  categorical_variable = pd.get_dummies(X[categorical_feature]).iloc[:,0]
  X = pd.concat([X,categorical_variable],axis=1)
  X = X.drop(categorical_feature,axis=1)
  X = X.rename(columns={list(X)[-1]:categorical_feature})
```

Constructing our training and test set:

In [ ]:
```
# Construct the training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1/5, random_state=42)
```

2. Implementing the Fairness model :

Defining the logistic loss:

In [ ]:
```
from sklearn.utils.extmath import weighted_mode

def logistic_loss(X,y,weight):
  '''
      Compute the logistic loss using the data set [X,y]

      Parameters
      ----------
      X : arrays
        Training features
      y : arrays
        Training target
      weight : array of size num_features
        The weight of our logistic regression model
  '''
  return 1/len(X)*cp.sum(cp.logistic(X @ weight)-cp.multiply(y,X @ weight))
```

Implementing the DM model ([Fairness Beyond Disparate Treatment & Disparate Impact: Learning Classification without Disparate Mistreatment (https://arxiv.org/pdf/1610.08452.pdf)](https://arxiv.org/pdf/1610.08452.pdf)) :

In [ ]:
```
class boundary_decision:
  '''
      This class compute the boundary decision as
      defined in the paper for different decision

      ...

      Parameters :
      ------------

      decision : str
          name of the missclassification measure tackled
          choice in ("OMR","FPR","FNR")
  '''

  def __init__(self, decision="OMR"):
    self.decision = decision

  def __call__(self,X,y, weight):
    '''
    Return the boundary decision (g_theta in the paper) according to the
    missclassification measure we want to tackled

    Parameters :
    ------------
    X : arrays
        Training features
    y : arrays
        Training target
    weight : array of size num_features
        The weight of our logistic regression model
    '''
    if self.decision == "OMR":
      return cp.minimum(0, cp.multiply((2*y-1),(X @ weight)))
    if self.decision == "FPR":
      return cp.minimum(0, cp.multiply((1-y)*(2*y-1),(X @ weight)))
    if self.decision == "FNR":
      return cp.minimum(0, cp.multiply(y*(2*y-1),(X @ weight)))

  def convexified(self,X,y, weight, new_weight):
    '''
    Return the linearized boundary decision around an input weight according to the
    missclassification measure we want to tackled

    Parameters :
    ------------
    X : arrays
        Training features
    y : arrays
        Training target
    weight : array of size num_features
        The weight around which the boundary decision is
        linearized
    new_weight : array of size num_features
        The weight in which we want to evaluate the
        convexified boundary decision
```

```
        '''
        if self.decision == "OMR":
          dirac = (2*y-1)*(X @ weight)<=0
          return self(X,y,weight)+cp.multiply(y*dirac,(X @ (new_weight-weight)))

        if self.decision == "FPR":
          dirac = (1-y)*(2*y-1)*(X @ weight)<=0
          return self(X,y,weight)+cp.multiply(dirac*(1-y)*(2*y-1),X @ (new_weight-weight))

        if self.decision == "FNR":
          dirac = y*(2*y-1)*(X @ weight) <=0
          return self(X,y,weight)+ cp.multiply(dirac*y*(2*y-1),X @ (new_weight-weight))
```

In [ ]:
```
class DM_DMsen:
    '''
        This class compute a default solver
        to get train a classifier without disparate treatment.
        Only logistic classifier was implemented yet but new implementation
        can be seen by changing the loss.

        ...

        Parameters :
        ------------


        X : arrays
            Training features
        y : arrays
            Training target
        weight : array of size num_features
            Initial weight of our logistic regression model
        method : str
            name of the missclassification measure tackled
            choice in ("OMR","FPR","FNR")
        algo : str
            choice of the algorithm to use between ("DM" and "DM-sen")
        validation : bool
            choice of spliting the training set into
            train and validation to evaluate the training
            (Only useful when running solver build from scratch)

        Hyperparameters :
        ------------
        c : float
            Lower-upper bound of the constraint in the DM formulation
        tau : float
            Initial penalization constant of the slack variables in the CCP solver
        mu : float
            At each iteration of the solver, we increase tau by tau*mu

        Methods :
        ---------
        solve_subproblem :
            This function is call by the CCP solver build from scratch.
            Solve the subproblem as defined in the CCP solver.
        solve :
            CCP solver build from scratch based on the proposed algorithm
        solve_DCCP :
            Solve the DM formulation using the DCCP package
        predict :
            Compute the prediction of our logistic regression model using
            the resulting weights.
        accuracy :
            Return the accuracy of our model by computing
            the average number of TPR and TNR over the
            data set [X_test, y_test]
    '''


    def __init__(self, X, y, c=0.1, tau=0.1, mu=10, algo = "DM", method="OMR", validation
```

```
        if algo == "DM-sen":
          self.X = X
        elif algo == "DM":
          self.X = X[:, 0:5]
        self.y = y
        self.c = cp.Parameter(value=c)
        if weight is None:
          self.weight = np.zeros(self.X.shape[1])
        else :
          self.weight = weight
        self.tau = tau
        self.history = {"loss": [], "accuracy": [], "DFNR": [], "DFPR": []}
        self.weight_k = cp.Variable(self.X.shape[1], value = self.weight)
        self.z = X[:,-1]
        if method == "FNR+FPR":
          self.si = cp.Variable(4,nonneg=True)
          self.g = boundary_decision("FPR")
          self.g_second = boundary_decision("FNR")
        else :
          self.si = cp.Variable(2,nonneg=True)
          self.g = boundary_decision(method)
        self.mu = mu
        self.algo = algo
        self.val = validation

      def solve_subproblem(self):
        '''
        This function is call by the CCP solver build from scratch.
        Solve the subproblem as defined in the CCP solver.
        '''
        N = len(self.z)
        N1 = sum(self.z)
        N0 = N-N1
        constraints = [
                 -N1/N*cp.sum(cp.multiply(self.z==0,self.g(self.X,self.y,self.weight_k)))+ N
                 -N0/N*cp.sum(cp.multiply(self.z==1,self.g(self.X,self.y,self.weight_k)))+ N
        ]
        try :
          constraints.extend([
                             -N1/N*cp.sum(cp.multiply(self.z==0,self.g_second(self.X,self.y
                             -N0/N*cp.sum(cp.multiply(self.z==1,self.g_second(self.X,self.y
                             ])
        except AttributeError :
          pass
        loss = logistic_loss(self.X,self.y,self.weight_k)
        obj = cp.Minimize(loss+self.tau*cp.sum(self.si))
        prob = cp.Problem(obj,constraints=constraints)
        res = prob.solve(warm_start = True)
        self.history["loss"].append(res)
        self.history["accuracy"].append(self.accuracy(self.X,self.y))
        self.history["DFPR"].append(DFPR(self,self.X,self.y,self.z))
        self.history["DFNR"].append(DFNR(self,self.X,self.y,self.z))

      def predict(self,X_test):
        '''
        Compute the prediction of our logistic regression model using
        the resulting weights.
```

```
        Parameters :
        ------------
        X_test : array
            New set of features on which we make our prediction
        '''
        if self.algo == "DM":
          X_test = X_test[:, 0:5]
        prob = np.exp(X_test @ self.weight_k.value)/(1+np.exp(X_test @ self.weight_k.value))
        return np.vectorize(lambda p: int(p>=0.5))(prob)

    def accuracy(self, X_val, y_val):
        '''
        Return the accuracy of our model by computing
        the average number of TPR and TNR over the
        data set [X_val, y_val]

        Parameters :
        -------------
        X_val : array
            Set of features on which we want to evaluate the accuracy
            of our model
        y_val : array
            The target we are trying to reach when evaluating our model
        '''
        y_hat  = self.predict(X_val)
        return np.sum(y_hat==y_val)/len(y_val)


    def solve(self,T):
        '''
        CCP solver build from scratch based on the proposed algorithm

        Parameters :
        -------------
        T : int
            Number of iteration the solver is run.
            Should we replace it by a termination criterion ?
        '''
        for i in range(T):
          # We split the data into a training and validation set at each step
          if self.val :
            self.X, X_val, self.y, y_val, self.z, z_val = train_test_split(self.X,self.y,sel
          self.solve_subproblem()
          self.tau = np.minimum(self.tau*self.mu,1)
          self.weight = self.weight_k.value
          if self.val :
            print("epoch {} - norm(s) {:.3f} - accuracy {:.3f} - val_accuracy {:.3f} - val_DI
            self.X = np.concatenate([self.X,X_val])
            self.y = np.concatenate([self.y,y_val])
            self.z = np.concatenate([self.z,z_val])

    def solve_DCCP(self):
        '''
        Solve the DM formulation using the DCCP package
        '''
        if self.val:
```

```
            self.X, X_val, self.y, y_val, self.z, z_val = train_test_split(self.X,self.y,self.
        N = len(self.z)
        N1 = sum(self.z)
        N0 = N-N1
        constraints = [
                 -N1/N*cp.sum(cp.multiply(self.z==0,self.g(self.X,self.y,self.weight_k)))<=
                 -N0/N*cp.sum(cp.multiply(self.z==1,self.g(self.X,self.y,self.weight_k)))<=
                 ]
        try :
          constraints.extend([
                             -N1/N*cp.sum(cp.multiply(self.z==0,self.g_second(self.X,self.y
                             -N0/N*cp.sum(cp.multiply(self.z==1,self.g_second(self.X,self.y
                             ])
        except AttributeError :
          pass
        loss = logistic_loss(self.X,self.y,self.weight_k)
        obj = cp.Minimize(loss)
        prob = cp.Problem(obj,constraints=constraints)
        result = prob.solve(method='dccp', warm_start=True)
        if self.val:
          print(prob.status)
          print("accuracy {:.3f} - val_accuracy {:.3f} - val_DFPR {:.3f} - val_DFNR {:.3f}".
          self.X = np.concatenate([self.X,X_val])
          self.y = np.concatenate([self.y,y_val])
          self.z = np.concatenate([self.z,z_val])
```

Evaluation :

Computing the metrics to evaluate our model :

In [ ]:
```
def DFPR(model, X_val, y_val, z_val):
    '''
    Compute the Difference of False Positive
    Rate in the validation set [X_val,y_val,z_val]
    of the model.

    Parameters :
    -------------
    model : the predicting model
    It should have a predict method!

    X_val : array
    The set of feature to evaluate the DFPR

    y_val : array
    The target to evaluate the DFPR

    z_val : array
    The sensitive feature to evaluate the DFPR
    '''
    X_val_0 = X_val[(z_val==0)&(y_val==0)]
    y_hat_0 = model.predict(X_val_0)
    y_val_0 = y_val[(z_val==0)&(y_val==0)]
    p1 = np.sum(y_hat_0!=0)/len(y_hat_0)

    X_val_1 = X_val[(z_val==1)&(y_val==0)]
    y_hat_1 = model.predict(X_val_1)
    y_val_1 = y_val[(z_val==1)&(y_val==0)]
    p2 = np.sum(y_hat_1!=0)/len(y_hat_1)

    return p1-p2

def DFNR(model, X_val, y_val, z_val):
    '''
    Compute the Difference of False Negative
    Rate in the validation set [X_val,y_val,z_val]
    of the model.

    Parameters :
    -------------
    model : the predicting model
    It should have a predict method!

    X_val : array
    The set of feature to evaluate the DFPR

    y_val : array
    The target to evaluate the DFPR

    z_val : array
    The sensitive feature to evaluate the DFPR
    '''
    X_val_0 = X_val[(z_val==0)&(y_val==1)]
    y_hat_0 = model.predict(X_val_0)
    y_val_0 = y_val[(z_val==0)&(y_val==1)]
    p1 = np.sum(y_hat_0!=1)/len(y_hat_0)
```

```
    X_val_1 = X_val[(z_val==1)&(y_val==1)]
    y_hat_1 = model.predict(X_val_1)
    y_val_1 = y_val[(z_val==1)&(y_val==1)]
    p2 = np.sum(y_hat_1!=1)/len(y_hat_1)

    return p1-p2

def calibration(model, X_val, y_val, z_val):
    '''
    Compute the calibration in the validation set [X_val,y_val,z_val]
    of the model.

    Parameters :
    -------------
    model : the predicting model
    It should have an accuracy/score method!

    X_val : array
    The set of feature to evaluate the DFPR

    y_val : array
    The target to evaluate the DFPR

    z_val : array
    The sensitive feature to evaluate the DFPR
    '''
    X_val_0 = X_val[z_val==0]
    y_val_0 = y_val[z_val==0]
    try :
      p1 = model.accuracy(X_val_0,y_val_0)
    except AttributeError:
      p1 = model.score(X_val_0,y_val_0)

    X_val_1 = X_val[z_val==1]
    y_val_1 = y_val[z_val==1]
    try :
      p2 = model.accuracy(X_val_1,y_val_1)
    except AttributeError:
      p2 = model.score(X_val_1,y_val_1)

    return p1-p2
```

Evaluation of the DM/DM-sen algorithm :

For the baseline regression :

In [ ]:
```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(random_state=42).fit(X_train,y_train)
score_logistic = {"accuracy" : model.score(X_test,y_test),
                  "DFNR" :  DFNR(model, X_test,y_test, X_test.iloc[:,-1]),
                  "DFPR" : DFPR(model,X_test,y_test,X_test.iloc[:,-1]),
                  "calibration" : calibration(model,X_test,y_test,X_test.iloc[:,-1])
}
print(pd.Series(score_logistic))
```

```
accuracy       0.627642
DFNR           0.366570
DFPR          -0.258211
calibration    0.030596
dtype: float64
```

In [ ]:
```
# Calculate the covariance of the unconstrained classifier
# to get an upper bound of the hy
d = 2*model.predict_proba(X_train)[:,1]-1
z = X_train.iloc[:,-1].to_numpy()
z_hat = np.mean(z)
print("Covariance OMR : ",np.mean((z-z_hat)*np.minimum(0, (2*y_train.to_numpy()-1)*d)))
print("Covariance FNR : ",np.mean((z-z_hat)*np.minimum(0, (1-y_train.to_numpy())*(2*y_trair
print("Covariance FPR : ",np.mean((z-z_hat)*np.minimum(0, (y_train.to_numpy())*(2*y_train.t
```

```
Covariance OMR :  0.004048143425567252
Covariance FNR :  -0.006192004580140546
Covariance FPR :  0.010240148005707798
```

For the OMR constraints :

In [ ]:
```
score_OMR = {}
solver = DM_DMsen(X_train.to_numpy(),y_train.to_numpy(), method="OMR", c=0.002)
solver.solve_DCCP()
score_OMR["acc"] = [solver.accuracy(X_test.to_numpy(),y_test.to_numpy())]
score_OMR["DFNR"] = [DFNR(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_numpy()[
score_OMR["DFPR"] = [DFPR(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_numpy()[
score_OMR["cal"] = [calibration(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_nu
solver = DM_DMsen(X_train.to_numpy(),y_train.to_numpy(), method="OMR", algo="DM-sen", c=0.
solver.solve_DCCP()
score_OMR["acc"].append(solver.accuracy(X_test.to_numpy(),y_test.to_numpy()))
score_OMR["DFNR"].append(DFNR(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_nump
score_OMR["DFPR"].append(DFPR(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_nump
score_OMR["cal"].append(calibration(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.t
```

In [ ]:
```
pd.DataFrame(score_OMR,index=["DM","DM-sen"])
```

Out[14]:

	acc	DFNR	DFPR	cal
DM	0.619512	0.165646	-0.089024	0.003722
DM-sen	0.627642	0.232764	-0.219117	0.050854

For the FNR constraints

In [ ]:
```
score_FNR = {}
solver = DM_DMsen(X_train.to_numpy(),y_train.to_numpy(), method="FNR", c=0.005)
solver.solve_DCCP()
score_FNR["acc"] = [solver.accuracy(X_test.to_numpy(),y_test.to_numpy())]
score_FNR["DFNR"] = [DFNR(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_numpy()[
score_FNR["DFPR"] = [DFPR(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_numpy()[
score_FNR["cal"] = [calibration(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_nu
solver = DM_DMsen(X_train.to_numpy(),y_train.to_numpy(), method="FNR", algo="DM-sen", c=0.
       -0.2137754 ]))
solver.solve_DCCP()
score_FNR["acc"].append(solver.accuracy(X_test.to_numpy(),y_test.to_numpy()))
score_FNR["DFNR"].append(DFNR(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_nump
score_FNR["DFPR"].append(DFPR(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_nump
score_FNR["cal"].append(calibration(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.t
```

In [ ]:
```
pd.DataFrame(score_FNR,index=["DM","DM-sen"])
```

Out[27]:

	acc	DFNR	DFPR	cal
DM	0.532520	0.040324	-0.140562	-0.032624
DM-sen	0.621951	0.124087	-0.076437	0.019887

For the FPR constraints

In [ ]:
```
score_FPR = {}
solver = DM_DMsen(X_train.to_numpy(),y_train.to_numpy(), method="FPR", c=0.01)
solver.solve_DCCP()
score_FPR["acc"] = [solver.accuracy(X_test.to_numpy(),y_test.to_numpy())]
score_FPR["DFNR"] = [DFNR(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_numpy()[
score_FPR["DFPR"] = [DFPR(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_numpy()[
score_FPR["cal"] = [calibration(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_nu
solver = DM_DMsen(X_train.to_numpy(),y_train.to_numpy(), method="FPR", algo="DM-sen", c=0.
       -0.2137754 ]))
solver.solve_DCCP()
score_FPR["acc"].append(solver.accuracy(X_test.to_numpy(),y_test.to_numpy()))
score_FPR["DFNR"].append(DFNR(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_nump
score_FPR["DFPR"].append(DFPR(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_nump
score_FPR["cal"].append(calibration(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.t
```

In [ ]:
```
pd.DataFrame(score_FPR,index=["DM","DM-sen"])
```

Out[29]:

	acc	DFNR	DFPR	cal
DM	0.563415	0.034928	-0.036200	0.114738
DM-sen	0.614634	0.012860	-0.000709	0.025413

For the FNR+FPR constraints :

In [ ]:
```
score_FNPR = {}
solver = DM_DMsen(X_train.to_numpy(),y_train.to_numpy(), method="FNR+FPR", c=0.005)
solver.solve_DCCP()
score_FNPR["acc"] = [solver.accuracy(X_test.to_numpy(),y_test.to_numpy())]
score_FNPR["DFNR"] = [DFNR(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_numpy()
score_FNPR["DFPR"] = [DFPR(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_numpy()
score_FNPR["cal"] = [calibration(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_n
solver = DM_DMsen(X_train.to_numpy(),y_train.to_numpy(), method="FNR+FPR", algo="DM-sen",
solver.solve_DCCP()
score_FNPR["acc"].append(solver.accuracy(X_test.to_numpy(),y_test.to_numpy()))
score_FNPR["DFNR"].append(DFNR(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_num
score_FNPR["DFPR"].append(DFPR(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.to_num
score_FNPR["cal"].append(calibration(solver, X_test.to_numpy(), y_test.to_numpy(), X_test.
```

In [ ]:
```
pd.DataFrame(score_FNPR,index=["DM","DM-sen"])
```

Out[56]:

	acc	DFNR	DFPR	cal
DM	0.560976	0.027116	-0.036200	0.118830
DM-sen	0.612195	0.007651	0.008757	0.022753

Summary of our results :

	Baseline logisitc	DM				DM-sen		
		OMR	FNR	FPR	FNR+FPR	OMR	FNR	FP
Accuracy	0.627642	0.619512	0.532520	0.563415	0.560976	0.627642	0.621951	0.61463
DFNR	0.366570	0.165646	0.040324	0.034928	0.027116	0.232764	0.124087	0.01286
DFPR	-0.258211	-0.089024	-0.140562	-0.036200	-0.036200	-0.219117	-0.076437	-0.0007C
Calibration	0.030596	0.003722	-0.032624	0.114738	0.118830	0.050854	0.019887	0.02541