Project4

April 11, 2022

Implementating, evaluating and comparing on algorithms: Fairness Beyond Disparate Treatment & Disparate Impact: Learning Classification without Disparate Mistreatment (DM and DM-sen) and Handling Conditional Discrimination (LM and LPS)

Team members:

- Sarah Kurihara
- Varchasvi Vedula
- Wenhui Fang
- Krista Zhang
- Sharon Meng

1 Setup

1.1 Import essential packages

[5]:
```
import warnings

import time,sys
import zipfile #When running in jupyter notebook, delete this code
import random
import pandas as pd
import numpy as np
import tensorflow as tf
from google.colab import drive #When running in jupyter notebook, delete this␣
↪code
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from tensorflow import keras
from keras.layers import Dense, Input
from tensorflow.keras import Model
import scipy.stats as ss

warnings.filterwarnings('ignore')
```

1.2 Load Database and Data Preprocessing

[4]:
```
#When running in jupyter notebook, delete this code chunk
#!mkdir data
#drive.mount('/content/drive')
```

[6]:
```
start=time.time()
#data = pd.read_csv('drive/MyDrive/5243Project4/compas-scores-two-years.
↪csv')#When running in jupyter notebook, delete this code
data = pd.read_csv('./data/compas-scores-two-years.csv') #When running in␣
↪jupyter notebook, use this code
#print(data.shape)
# filter out groups other than African-American and Caucasian and set them as␣
↪0-1
data = data[(data['race']=='African-American') | (data['race']=='Caucasian')]
data['race'].loc[data['race']=='Caucasian'] = 1
data['race'].loc[data['race']=='African-American'] = 0
#print(data.shape)

nan = (data.isnull().sum()/len(data))
nan = nan[nan > 0.15].sort_values()
nan_var = list(nan.index)
data = data.drop(columns=nan_var)

data['c_jail_in'] = pd.to_datetime(data['c_jail_in'])
```

```
data['c_jail_out'] = pd.to_datetime(data['c_jail_out'])
data['los'] = np.log((data['c_jail_out']-data['c_jail_in']).
 ↪astype('timedelta64[h]')+1)#use log hours
data['in_custody'] = pd.to_datetime(data['in_custody'])
data['out_custody'] = pd.to_datetime(data['out_custody'])
data['custody'] = np.log((data['out_custody']-data['in_custody']).
 ↪astype('timedelta64[h]')+1)
data['lasts'] = np.log(data['end']-data['start']+1)
data['c_days_from_compas'] = np.log(data['c_days_from_compas']+1)

#filter out useless variables including high correlation and string type
useless_var =␣
 ↪['id','name','first','last','compas_screening_date','dob','age_cat','days_b_screening_arres
             ␣
 ↪'c_jail_in','c_jail_out','c_case_number','c_charge_desc','is_recid',
               'type_of_assessment','screening_date','v_type_of_assessment',
             ␣
 ↪'v_screening_date','in_custody','out_custody','score_text','v_score_text',
               'decile_score.1','v_decile_score','priors_count.1','start','end']
data = data.drop(columns=useless_var)
data = data[data['los']!=float('-inf')]
data = data[data['custody']!=float('-inf')]
data = data[data['lasts']!=float('-inf')]

#one hot encoding on several features:sex,c_charge_degree
data['sex'].loc[data['sex']=='Male']= 1
data['sex'].loc[data['sex']=='Female']= 0
data['c_charge_degree'].loc[data['c_charge_degree']=='M']= 1
data['c_charge_degree'].loc[data['c_charge_degree']=='F']= 0
#data.to_csv('./data/compas_preproc.csv',index=False,header=True)
del nan_var, useless_var

#data =␣
 ↪data[['age','race','sex','decile_score','priors_count','los','c_charge_degree','two_year_re
data = data.dropna()#6150*23->5730*16
#print(data.shape)
print(data.head(5))

X = data.drop(columns='two_year_recid')
features = list(X.columns)

X.index = range(data.shape[0])
#As age, priors_count, los are continuous variables, we can scale them
X_cont = X[['age', 'juv_fel_count', 'decile_score', 'juv_misd_count',␣
 ↪'juv_other_count', 'priors_count', 'c_days_from_compas', 'los', 'custody',␣
 ↪'lasts']]
```

```
X_cate = X[['sex', 'race', 'c_charge_degree', 'is_violent_recid', 'event']]
X_cont = pd.DataFrame(StandardScaler().fit_transform(X_cont),columns=['age',␣
↪'juv_fel_count', 'decile_score', 'juv_misd_count', 'juv_other_count',␣
↪'priors_count', 'c_days_from_compas', 'los', 'custody', 'lasts'])
#X_cont = X[['age', 'decile_score', 'priors_count', 'los']]
#X_cate = X[['sex', 'race', 'c_charge_degree']]
#X_cont = pd.DataFrame(StandardScaler().fit_transform(X_cont),columns=['age',␣
↪'decile_score', 'priors_count', 'los'])

X_df = pd.concat([X_cate,X_cont],axis=1)
#X['decile_score'] = X['decile_score']/10
y_df = data["two_year_recid"]
# convert class label 0 to -1 so as to add sign in distance
#y[y==0] = -1
features = list(X.columns)

X = np.asarray(X_df).astype('float32')
y = np.asarray(y_df).astype('float32')

del X_cate,X_cont
```

```
   sex  age  race  juv_fel_count  decile_score  juv_misd_count  juv_other_count  \
1    1   34     0              0             3               0                0
2    1   24     0              0             4               0                1
6    1   41     1              0             6               0                0
8    0   39     1              0             1               0                0
9    1   21     1              0             3               0                0

   priors_count  c_days_from_compas c_charge_degree  is_violent_recid  event  \
1             0            0.693147               0                 1      1
2             4            0.693147               0                 0      0
6            14            0.693147               0                 0      1
8             0            0.693147               1                 0      0
9             1            5.733341               0                 1      1

   two_year_recid       los   custody     lasts
1               1  5.488938  5.484797  5.017280
2               1  3.295837  0.000000  4.158883
6               1  5.023881  6.070738  3.583519
8               0  4.262680  4.290459  6.614726
9               1  3.178054  3.218876  6.061457
```

1.3 Data Splitting

[8]:
```
#Use 5:1:1 as the ratio of train:val:test
# Fro A4
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=800,␣
↪random_state=3)
X_train, X_val, y_train, y_val =␣
↪train_test_split(X_train,y_train,test_size=800,random_state=3)

train_size = len(y_train)
train_idx_AA = np.array(range(train_size))[X_train[:,1]==0.0]
train_idx_C = np.array(range(train_size))[X_train[:,1]==1.0]
test_size = len(y_test)
test_idx_AA = np.array(range(test_size))[X_test[:,1]==0.0]
test_idx_C = np.array(range(test_size))[X_test[:,1]==1.0]
print('\n',"#"*80,'\n',' '*20," Split up Train-Validation-Test sets for A4␣
↪",'\n',"#"*80,'\n')
print(" X_train size: ", X_train.shape, ",      y_train size: ", y_train.shape,␣
↪'\n',
      "X_validation size: ", X_val.shape, ",  y_validation size: ", y_val.
↪shape, '\n',
      "X_test size: ", X_test.shape, ',        y_test size: ',y_test.shape)
print(" X_train_AA size: ", X_train[train_idx_AA].shape, ",   X_train_C size:␣
↪", X_train[train_idx_C].shape, '\n',
      "X_test_AA size: ", X_test[test_idx_AA].shape, ',     X_test_C size:␣
↪',X_test[test_idx_C].shape, '\n',
      "Ratio:", X_train[train_idx_AA].shape[0]/X_train[train_idx_C].
↪shape[0],X_test[test_idx_AA].shape[0]/X_test[test_idx_C].shape[0])
print('\n',"#"*80)
###########################################################################################
# For A6
X_train_df, X_test_df, y_train_df, y_test_df = train_test_split(X_df, y_df,␣
↪test_size=800, random_state=3)
X_train_df, X_val_df, y_train_df, y_val_df =␣
↪train_test_split(X_train_df,y_train_df,test_size=800,random_state=3)

X_train_df.race = abs(X_train_df.race-1)
X_test_df.race = abs(X_test_df.race-1)
X_val_df.race = abs(X_val_df.race-1)
X_train_df.c_charge_degree = abs(X_train_df.c_charge_degree-1)
X_test_df.c_charge_degree = abs(X_test_df.c_charge_degree-1)
X_val_df.c_charge_degree = abs(X_val_df.c_charge_degree-1)
train = pd.concat([X_train_df.reset_index(drop=True), y_train_df.
↪reset_index(drop=True)], axis = 1)
test = pd.concat([X_test_df.reset_index(drop=True), y_test_df.
↪reset_index(drop=True)], axis = 1)
print('\n',"#"*80,'\n',' '*20," Split up Train-Validation-Test sets for␣
↪A6",'\n',"#"*80,'\n')
print(" X_train size: ", X_train_df.shape, ",      y_train size: ", y_train_df.
↪shape, '\n',
```

```
      "X_validation size: ", X_val_df.shape, ",  y_validation size: ", y_val_df.
↪shape, '\n',
      "X_test size: ", X_test_df.shape, ',        y_test size: ',y_test_df.
↪shape)
print('\n',"#"*80)
```

```
################################################################################
                       Split up Train-Validation-Test sets for A4
################################################################################

 X_train size:  (4130, 15) ,      y_train size:  (4130,)
 X_validation size:  (800, 15) ,  y_validation size:  (800,)
 X_test size:  (800, 15) ,        y_test size:  (800,)
 X_train_AA size:  (2472, 15) ,   X_train_C size:  (1658, 15)
 X_test_AA size:  (478, 15) ,     X_test_C size:  (322, 15)
 Ratio: 1.490952955367913 1.484472049689441

################################################################################

################################################################################
                       Split up Train-Validation-Test sets for A6
################################################################################

 X_train size:  (4130, 15) ,      y_train size:  (4130,)
 X_validation size:  (800, 15) ,  y_validation size:  (800,)
 X_test size:  (800, 15) ,        y_test size:  (800,)

################################################################################
```

2 Handling Conditional Discrimination (LM and LPS)

[9]:
```
# X_ALL = all predictors, S = sensitive attributes, E = explanatory attribute,␣
↪Y = response
S = "race"
E = "c_charge_degree"
Y = "two_year_recid"

X_ALL = ['age', 'juv_fel_count', 'decile_score', 'juv_misd_count',
         'juv_other_count', 'priors_count', 'c_days_from_compas', 'los',␣
↪'custody', 'lasts',
         'sex', 'race', 'c_charge_degree', 'is_violent_recid', 'event']

X_train_6 = train[X_ALL]
y_train_6 = train[Y]
```

```
X_test_6 = test[X_ALL]
y_test_6 = test[Y]
```

2.1 Baseline Model

We use a logistic regression classifier.

[10]:
```
clf = LogisticRegression(random_state=0).fit(X_train_6, y_train_6)
clf.score(X_test_6, y_test_6)
```

[10]: 0.93

2.2 A6 Algorithms

A6 uses the following setup:

- There is a sensitive attribute S. We want to avoid discrimination between different values of S. In our case, S is race, which takes values in {African American (1), Caucasian (0)}
- There is an explanatory variable E, which is correlated with S. Hence, simply removing S won't fix bias. In our case, we assume it is c_charge_degree.

2.2.1 Create functions used in pseudocode

[11]:
```
# Creates a list of partitions: 1 for each unique value of e

# X is the full dataset (in our case, train)
# e is the
def PARTITION(X):
    partitions = list()

    for e_i in np.unique(X[E]):
        partitions.append(X[X[E]==e_i])

    return partitions
```

[12]:
```
# Delta function returns the number of observations (i.e. people) who are␣
 ↪incorrectly classified
# based on theoretical probabilities of reciding, calculated as the average␣
 ↪rate of reciding
# for each explanatory varaible (in our case, type of crime comittied,␣
 ↪c_charge_degree)

def DELTA(X, X_ei, s_i):

    # Gi is the number of observations for each race
    # Don't we need to pass S as a parameter for this function?
    Gi = sum(X_ei[S] == s_i)

```

```
    # X_ei_si is the dataset that contains the observations for each race
    X_ei_si = X_ei[X_ei[S] == s_i]

    # P_denom is the number of people in group
    # P_num is number of observations who recid
    P_denom = X_ei_si.shape[0]
    P_num = sum(X_ei_si[Y] == 1)

    # P is the probability of reciding for one race
    # It is calculated by taking number of people who recid in each group
    # dividied by total number of people in that group
    P = P_num/P_denom

    # All other observations (for the other group)
    X_ei_not_si = X_ei[X_ei[S] != s_i]

    # The probability of reciding for the other group (same calculation as␣
↪above)
    Ps_2 = sum(X_ei_not_si[Y] == 1)/X_ei_not_si.shape[0]

    # Ps is P*, which is the theoretical true probability of reciding
    # Calculated by the average
    Ps = (P+Ps_2)/2

    # Calcualte the number of incorrectly classified people
    d = int(round(Gi * abs(P - Ps)))

    return(d)
```

2.2.2 Local Massaging

[13]:
```
relabeled_X_ei = list()

for X_ei in PARTITION(train):
    X_ei_copy = X_ei.copy()

    ranker_model = LogisticRegression(random_state=0).fit(X_ei[X_ALL], X_ei[Y])

    afam_index = [i for (i, v) in zip(list(range(X_ei.shape[0])), list(X_ei[S]␣
↪== 1)) if v]
    afam = X_ei[X_ei[S] == 1].copy()
    delta_afam = DELTA(train, X_ei, 1)
    afam_predicted_1_index = [afam_index[v] for v in np.squeeze(np.
↪where(ranker_model.predict(afam[X_ALL]) == 1))]
    afam_predicted_1_index_Y1 = [i for (i,v) in zip(afam_predicted_1_index,␣
↪X_ei.iloc[afam_predicted_1_index][Y]) if v==1]
    afam_predicted_1 = X_ei.iloc[afam_predicted_1_index_Y1]
```

```
    afam_ranks = (ss.rankdata(ranker_model.
↪decision_function(afam_predicted_1[X_ALL]))-1).astype(int)
    afam_tochange = [i for (i, v) in zip(list(range(len(afam_ranks))),␣
↪afam_ranks < delta_afam) if v]
    afam_tochange_idx = [afam_predicted_1_index_Y1[v] for v in afam_tochange]

    cauca_index = [i for (i, v) in zip(list(range(X_ei.shape[0])), list(X_ei[S]␣
↪== 0)) if v]
    cauca = X_ei[X_ei[S] == 0].copy()
    delta_cauca = DELTA(train, X_ei, 0)
    cauca_predicted_0_index = [cauca_index[v] for v in np.squeeze(np.
↪where(ranker_model.predict(cauca[X_ALL]) == 0))]
    cauca_predicted_0_index_Y0 = [i for (i,v) in zip(cauca_predicted_0_index,␣
↪X_ei.iloc[cauca_predicted_0_index][Y]) if v==0]
    cauca_predicted_0 = X_ei.iloc[cauca_predicted_0_index_Y0]

    cauca_ranks = (ss.rankdata(-ranker_model.
↪decision_function(cauca_predicted_0[X_ALL]))-1).astype(int)
    cauca_tochange = [i for (i, v) in zip(list(range(len(cauca_ranks))),␣
↪cauca_ranks < delta_cauca) if v]
    cauca_tochange_idx = [cauca_predicted_0_index_Y0[v] for v in cauca_tochange]

    for i in afam_tochange_idx:
        X_ei_copy.loc[X_ei_copy.index[i], Y] = 0
    for i in cauca_tochange_idx:
        X_ei_copy.loc[X_ei_copy.index[i], Y] = 1

    relabeled_X_ei.append(X_ei_copy)

    print("DELTA(African American) = ", delta_afam, "African Americans changed␣
↪from 1 to 0")
    print("DELTA(Caucasian) = ", delta_cauca, "Caucasians changed from 0 to 1")

local_massaging = pd.concat(relabeled_X_ei)
```

```
DELTA(African American) =  46 African Americans changed from 1 to 0
DELTA(Caucasian) =  39 Caucasians changed from 0 to 1
DELTA(African American) =  90 African Americans changed from 1 to 0
DELTA(Caucasian) =  54 Caucasians changed from 0 to 1
```

[14]:
```
lm_X_train = local_massaging[X_ALL]
lm_Y_train = local_massaging[Y]
```

[15]:
```
clf = LogisticRegression(random_state=0).fit(lm_X_train, lm_Y_train)
clf.score(X_test_6[X_ALL], y_test_6)
```

[15]: 0.915

[16]:
```
# Total number changed values should be sum of all DELTAs shown above
res = [1 for i, j in zip(train.sort_index()["two_year_recid"], pd.
↪DataFrame(lm_Y_train).sort_index()["two_year_recid"]) if i != j]
sum(res)
```

[16]: 229

2.2.3 Local Preferential Sampling

In this algorithm, we take in a dataset (train) and return a modified dataset of same size.

[17]:
```
pd.options.mode.chained_assignment = None
recomp_train = pd.DataFrame()

# for each partition (explanatory variable)
for X_ei in PARTITION(train):

    print("start partition")
    X_ei_copy = X_ei.copy()
    print("X_ei shape:", X_ei_copy.shape)

    # learn a ranker Hi : Xi -> Yi
    ranker_model = LogisticRegression(random_state=0).fit(X_ei[X_ALL], X_ei[Y])

    # Calculate half delta (AA: S_i = 1, AA: S_i = 0)
    half_delta_afam = DELTA(train, X_ei, 1) // 2
    half_delta_cauc = DELTA(train, X_ei, 0) // 2
    print("Half Delta(AA):", half_delta_afam)
    print("Half Delta(Cauc):", half_delta_cauc)

    # store indicies
    afam_index = [i for (i, v) in zip(list(range(X_ei.shape[0])), list(X_ei[S]␣
↪== 1)) if v]
    c_index = [i for (i, v) in zip(list(range(X_ei.shape[0])), list(X_ei[S] ==␣
↪0)) if v]
    print("Total AAs:", len(afam_index))
    print("Total Cs:", len(c_index))

    # get subset of data to work with
    afam = X_ei[X_ei[S] == 1].copy()
    c = X_ei[X_ei[S] == 0].copy()
    print("afam dataset shape:", afam.shape)
    print("c dataset shape:", c.shape)

    # rank AA
```

```
    afam.reset_index(drop=True, inplace=True)
    rank = pd.DataFrame(ranker_model.decision_function(afam[X_ALL]), columns =␣
↪['rank'])
    afam_with_rank = pd.concat([afam, rank], axis=1)

    # rank C
    c.reset_index(drop=True, inplace=True)
    rank = pd.DataFrame(ranker_model.decision_function(c[X_ALL]), columns =␣
↪['rank'])
    c_with_rank = pd.concat([c, rank], axis=1)

    # sort values, reset indices
    afam_with_rank = afam_with_rank.sort_values(['rank'])
    afam_with_rank.reset_index(drop = True, inplace = True)
    c_with_rank = c_with_rank.sort_values(['rank'])
    c_with_rank.reset_index(drop = True, inplace = True)

    ######## Modify AA data - find rows to delete/duplicate; decision boundary␣
↪is 0 #####
    recid = sum(afam_with_rank['rank'] > 0)
    no_recid = sum(afam_with_rank['rank'] < 0)
    total = len(afam_with_rank)

    # make copy of recids and no_recids
    # compas = compas[compas['days_b_screening_arrest'] >= -30]
    cleaned_recid = afam_with_rank[afam_with_rank['rank'] > 0]
    cleaned_no_recid = afam_with_rank[afam_with_rank['rank'] < 0]

    # delete first 1/2 delta values from recid
    N = half_delta_afam
    print("N:", N)
    print("rows in cleaned_recid before:", cleaned_recid.shape)
    cleaned_recid.drop(index=cleaned_recid.index[:N], axis=0, inplace=True)
    print("rows in cleaned_recid after:", cleaned_recid.shape)

    # flip order, then duplicate first 1/2 delta values from no_recid
    #print("cleaned_no_recid before:", cleaned_no_recid)
    cleaned_no_recid = cleaned_no_recid.sort_values(by='rank', ascending=False)
    #print("cleaned_no_recid after:", cleaned_no_recid)
    print("N:", N)
    print("rows in cleaned_no_recid before:", cleaned_no_recid.shape)
    cleaned_no_recid = cleaned_no_recid.append(cleaned_no_recid[0:N])
    print("rows in cleaned_no_recid after:", cleaned_no_recid.shape)

    # combine
    total_AA = pd.concat([cleaned_recid, cleaned_no_recid])
    print("size of final A:", total_AA.shape)
```

```python
########## Modify C data ############
# Find rows to delete/duplicate; decision boundary is 0; opposite code as␣
↪above
recid = sum(c_with_rank['rank'] < 0)
no_recid = sum(c_with_rank['rank'] > 0)
total = len(c_with_rank)

# make copy of recids and no_recids
cleaned_recid = c_with_rank[c_with_rank['rank'] < 0]
cleaned_no_recid = c_with_rank[c_with_rank['rank'] > 0]

# delete first 1/2 delta values from recid
M = half_delta_cauc
print("M:", M)
print("rows in cleaned_recid before:", cleaned_recid.shape)
cleaned_recid.drop(index=cleaned_recid.index[:M], axis=0, inplace=True)
print("rows in cleaned_recid after:", cleaned_recid.shape)

# flip order, then duplicate first 1/2 delta values from no_recid
#print("cleaned_no_recid before:", cleaned_no_recid)
cleaned_no_recid = cleaned_no_recid.sort_values(by='rank', ascending=False)
#print("cleaned_no_recid after:", cleaned_no_recid)
print("M:", M)
print("rows in cleaned_no_recid before:", cleaned_no_recid.shape)
cleaned_no_recid = cleaned_no_recid.append(cleaned_no_recid[0:M])
print("rows in cleaned_no_recid after:", cleaned_no_recid.shape)

# combine
total_C = pd.concat([cleaned_recid, cleaned_no_recid])
print("size of final C:", total_C.shape)
print("end partition")

# combine both datasets
recomp_train = recomp_train.append(total_AA)
recomp_train = recomp_train.append(total_C)
recomp_train = recomp_train.drop('rank', axis=1)

print("size of train:", train.shape)
print("size of recomp:", recomp_train.shape)
```

```
start partition
X_ei shape: (1386, 16)
Half Delta(AA): 23
Half Delta(Cauc): 19
Total AAs: 750
Total Cs: 636
```

```
afam dataset shape: (750, 16)
c dataset shape: (636, 16)
N: 23
rows in cleaned_recid before: (355, 17)
rows in cleaned_recid after: (332, 17)
N: 23
rows in cleaned_no_recid before: (395, 17)
rows in cleaned_no_recid after: (418, 17)
size of final A: (750, 17)
M: 19
rows in cleaned_recid before: (415, 17)
rows in cleaned_recid after: (396, 17)
M: 19
rows in cleaned_no_recid before: (221, 17)
rows in cleaned_no_recid after: (240, 17)
size of final C: (636, 17)
end partition
size of train: (4130, 16)
size of recomp: (1386, 16)
start partition
X_ei shape: (2744, 16)
Half Delta(AA): 45
Half Delta(Cauc): 27
Total AAs: 1722
Total Cs: 1022
afam dataset shape: (1722, 16)
c dataset shape: (1022, 16)
N: 45
rows in cleaned_recid before: (983, 17)
rows in cleaned_recid after: (938, 17)
N: 45
rows in cleaned_no_recid before: (739, 17)
rows in cleaned_no_recid after: (784, 17)
size of final A: (1722, 17)
M: 27
rows in cleaned_recid before: (561, 17)
rows in cleaned_recid after: (534, 17)
M: 27
rows in cleaned_no_recid before: (461, 17)
rows in cleaned_no_recid after: (488, 17)
size of final C: (1022, 17)
end partition
size of train: (4130, 16)
size of recomp: (4130, 16)
```

2.3 Evaluation

Notation: P_c stands for probability based on the classifier's predictions.

2.3.1 Metrics Used:

Parity or D_all Parity is defined as the difference is positive prediction rates in the two race groups. Paper 6 also calls this D_all, which stands for all discrimination. Fairness calls for Parity being close to 0.

Parity = |P_c(recid = 1 | race = African American) - P_c(recid = 1 | race = Caucasian)

Calibration Calibration is defined as the difference in accuracies between the two race groups. Fairness calls for Calibration being close to 0.

Calibration = |P_c(recid predicted correctly | race = African American) - P_c(recid predicted correctly | race = Caucasian)

Equality of Odds Equality of odds is achieved when the difference in positive prediction rates is equal for the two race groups. Fairness calls for the following value to be close to 0 for both y in {0,1}.

D_Odds = P_c(recid.hat = 1 | race = African American, recid = y) - P_c(recid.hat = 1 | race = Caucasian, recid = y)

[18]:
```
# X must include the sensitive feature
def PARITY(X, Y_PRED):
    s = X[S]

    afam = X[X[S] == 1]
    num_afam = sum(Y_PRED[X[S] == 1])
    den_afam = afam.shape[0]

    cauca = X[X[S] == 0]
    num_cauca = sum(Y_PRED[X[S] == 0])
    den_cauca = cauca.shape[0]

    print("P_c(recid = 1 | race = African American) =", num_afam/den_afam)
    print("P_c(recid = 1 | race = Caucasian) =", num_cauca/den_cauca)
    parity = abs(num_afam/den_afam - num_cauca/den_cauca)
    print("Parity =", parity)

    return(parity)
```

[19]:
```
# X must include S
def CALIBRATION(X, Y_TRUE, Y_PRED):

    afam = X[X[S] == 1]
    Y_TRUE_afam = Y_TRUE[X[S] == 1]
    num_afam = sum([1 for (i, v) in zip(Y_TRUE_afam, Y_PRED[X[S]==1]) if i ==␣
↪v])
    den_afam = afam.shape[0]
```

```
    cauca = X[X[S] == 0]
    Y_TRUE_cauca = Y_TRUE[X[S] == 0]
    num_cauca = sum([1 for (i, v) in zip(Y_TRUE_cauca, Y_PRED[X[S]==0]) if i ==␣
↪v])
    den_cauca = cauca.shape[0]

    print("P_c(recid predicted correctly | race = African American) =",␣
↪num_afam/den_afam)
    print("P_c(recid predicted correctly | race = Caucasian) =", num_cauca/
↪den_cauca)
    calibration = abs(num_afam/den_afam - num_cauca/den_cauca)
    print("Calibration =", calibration)
```

[20]:
```
def EQUALITY_OF_ODDS(X, Y_TRUE, Y_PRED):

    # S = afam, Y = 0
    X_afam_0 = X[np.logical_and(X[S]==1, Y_TRUE == 0)]
    Y_PRED_afam_0 = Y_PRED[np.logical_and(X[S]==1, Y_TRUE == 0)]
    num_afam_0 = sum([1 for i in Y_PRED_afam_0 if i == 1])
    denom_afam_0 = X_afam_0.shape[0]
    P_afam_0 = num_afam_0/denom_afam_0

    # S = afam, Y = 1
    X_afam_1 = X[np.logical_and(X[S]==1, Y_TRUE == 1)]
    Y_PRED_afam_1 = Y_PRED[np.logical_and(X[S]==1, Y_TRUE == 1)]
    num_afam_1 = sum([1 for i in Y_PRED_afam_1 if i == 1])
    denom_afam_1 = X_afam_1.shape[0]
    P_afam_1 = num_afam_1/denom_afam_1

    # S = cauca, Y = 0
    X_cauca_0 = X[np.logical_and(X[S]==0, Y_TRUE == 0)]
    Y_PRED_cauca_0 = Y_PRED[np.logical_and(X[S]==0, Y_TRUE == 0)]
    num_cauca_0 = sum([1 for i in Y_PRED_cauca_0 if i == 1])
    denom_cauca_0 = X_cauca_0.shape[0]
    P_cauca_0 = num_cauca_0/denom_cauca_0

    # S = cauca, Y = 1
    X_cauca_1 = X[np.logical_and(X[S]==0, Y_TRUE == 1)]
    Y_PRED_cauca_1 = Y_PRED[np.logical_and(X[S]==0, Y_TRUE == 1)]
    num_cauca_1 = sum([1 for i in Y_PRED_cauca_1 if i == 1])
    denom_cauca_1 = X_cauca_1.shape[0]
    P_cauca_1 = num_cauca_1/denom_cauca_1

    print("For recid = 0:\n")
    print("P_c(recid.hat = 1 | race = African American, recid = 0) = ",␣
↪P_afam_0)
    print("P_c(recid.hat = 1 | race = Caucasian, recid = 0) = ", P_cauca_0)
```

```
    print("Difference in odds of true recid = 0 is  = D_FPR =", abs(P_afam_0 -␣
↪P_cauca_0))
    print("\n\n")
    print("For recid = 1:\n")
    print("P_c(recid.hat = 1 | race = African American, recid = 1) = ",␣
↪P_afam_1)
    print("P_c(recid.hat = 1 | race = Caucasian, recid = 1) = ", P_cauca_1)
    print("Difference in odds of true recid = 1 is = D_TPR =", abs(P_afam_1 -␣
↪P_cauca_1))
```

[21]:
```
def D_FNR(X, Y_TRUE, Y_PRED):
    # S = afam, Y = 1
    X_afam_1 = X[np.logical_and(X[S]==1, Y_TRUE == 1)]
    Y_PRED_afam_1 = Y_PRED[np.logical_and(X[S]==1, Y_TRUE == 1)]
    num_afam_1 = sum([1 for i in Y_PRED_afam_1 if i == 0])
    denom_afam_1 = X_afam_1.shape[0]
    P_afam_1 = num_afam_1/denom_afam_1

    # S = cauca, Y = 1
    X_cauca_1 = X[np.logical_and(X[S]==0, Y_TRUE == 1)]
    Y_PRED_cauca_1 = Y_PRED[np.logical_and(X[S]==0, Y_TRUE == 1)]
    num_cauca_1 = sum([1 for i in Y_PRED_cauca_1 if i == 0])
    denom_cauca_1 = X_cauca_1.shape[0]
    P_cauca_1 = num_cauca_1/denom_cauca_1

    print("Difference in False Negative Rates")

    print("For recid = 1:\n")
    print("P_c(recid.hat = 0 | race = African American, recid = 1) = ",␣
↪P_afam_1)
    print("P_c(recid.hat = 0 | race = Caucasian, recid = 1) = ", P_cauca_1)

    print("D_FNR =", abs(P_afam_1 - P_cauca_1))
```

2.3.2 Baseline Evaluation

[22]:
```
clf = LogisticRegression(random_state=0).fit(X_train_6, y_train_6)
baseline_pred = clf.predict(X_test_6[X_ALL])
clf.score(X_test_6, y_test_6)
```

[22]: 0.93

[23]:
```
print(classification_report(y_test_6, clf.predict(X_test_6[X_ALL])))
```

```
              precision    recall  f1-score   support

           0       0.93      0.93      0.93       417
           1       0.93      0.93      0.93       383
```

```
    accuracy                           0.93       800
   macro avg       0.93      0.93      0.93       800
weighted avg       0.93      0.93      0.93       800
```

[24]:
```
# Parity

PARITY(X_test_6, baseline_pred)
```

```
P_c(recid = 1 | race = African American) = 0.5376569037656904
P_c(recid = 1 | race = Caucasian) = 0.391304347826087
Parity = 0.14635255593960345
```

[24]:
```
0.14635255593960345
```

[25]:
```
# Calibration

CALIBRATION(X_test_6, y_test_6, baseline_pred)
```

```
P_c(recid predicted correctly | race = African American) = 0.9372384937238494
P_c(recid predicted correctly | race = Caucasian) = 0.9192546583850931
Calibration = 0.01798383533875625
```

[26]:
```
# Equality of Odds

EQUALITY_OF_ODDS(X_test_6, y_test_6, baseline_pred)
```

```
For recid = 0:

P_c(recid.hat = 1 | race = African American, recid = 0) =  0.06787330316742081
P_c(recid.hat = 1 | race = Caucasian, recid = 0) =  0.0663265306122449
Difference in odds of true recid = 0 is  = D_FPR = 0.001546772555175907



For recid = 1:

P_c(recid.hat = 1 | race = African American, recid = 1) =  0.9416342412451362
P_c(recid.hat = 1 | race = Caucasian, recid = 1) =  0.8968253968253969
Difference in odds of true recid = 1 is = D_TPR = 0.04480884441973931
```

[27]:
```
# D_FNR

D_FNR(X_test_6, y_test_6, baseline_pred)
```

```
Difference in False Negative Rates
For recid = 1:

P_c(recid.hat = 0 | race = African American, recid = 1) =  0.058365758754863814
```

```
P_c(recid.hat = 0 | race = Caucasian, recid = 1) =  0.10317460317460317
D_FNR = 0.044808844419739355
```

2.3.3 Local Massaging Evaluation

[28]:
```python
clf = LogisticRegression(random_state=0).fit(lm_X_train, lm_Y_train)
lm_pred = clf.predict(X_test_6[X_ALL])
clf.score(X_test_6[X_ALL], y_test_6)
```

[28]:
```
0.915
```

[29]:
```python
print(classification_report(y_test_6, clf.predict(X_test_6[X_ALL])))
```

```
              precision    recall  f1-score   support

           0       0.91      0.93      0.92       417
           1       0.92      0.90      0.91       383

    accuracy                           0.92       800
   macro avg       0.92      0.91      0.91       800
weighted avg       0.92      0.92      0.91       800
```

[30]:
```python
# Parity

PARITY(X_test_6, lm_pred)
```

```
P_c(recid = 1 | race = African American) = 0.49581589958158995
P_c(recid = 1 | race = Caucasian) = 0.422360248447205
Parity = 0.07345565113438496
```

[30]:
```
0.07345565113438496
```

[31]:
```python
# Calibration

CALIBRATION(X_test_6, y_test_6, lm_pred)
```

```
P_c(recid predicted correctly | race = African American) = 0.9163179916317992
P_c(recid predicted correctly | race = Caucasian) = 0.9130434782608695
Calibration = 0.0032745133709296548
```

[32]:
```python
# Equality of Odds

EQUALITY_OF_ODDS(X_test_6, y_test_6, lm_pred)
```

```
For recid = 0:

P_c(recid.hat = 1 | race = African American, recid = 0) =  0.04524886877828054
P_c(recid.hat = 1 | race = Caucasian, recid = 0) =  0.09693877551020408
Difference in odds of true recid = 0 is  = D_FPR = 0.051689906731923536
```

```
For recid = 1:

P_c(recid.hat = 1 | race = African American, recid = 1) =  0.8832684824902723
P_c(recid.hat = 1 | race = Caucasian, recid = 1) =  0.9285714285714286
Difference in odds of true recid = 1 is = D_TPR = 0.04530294608115626
```

[33]:
```
# D_FNR

D_FNR(X_test_6, y_test_6, lm_pred)
```

```
Difference in False Negative Rates
For recid = 1:

P_c(recid.hat = 0 | race = African American, recid = 1) =  0.11673151750972763
P_c(recid.hat = 0 | race = Caucasian, recid = 1) =  0.07142857142857142
D_FNR = 0.045302946081156203
```

2.3.4 Local Preferential Sampling Evaluation

[34]:
```
recomp_X_train = recomp_train[X_ALL]
recomp_Y_train = recomp_train[Y]
print("size of recomp_X_train:", recomp_X_train.shape)
print("size of recomp_Y_train:", recomp_Y_train.shape)

clf_LPS = LogisticRegression(random_state=0).fit(recomp_X_train, recomp_Y_train)
LPS_pred = clf_LPS.predict(X_test_6[X_ALL])
clf_LPS.score(X_test_6[X_ALL], y_test_6)
```

```
size of recomp_X_train: (4130, 15)
size of recomp_Y_train: (4130,)
```

[34]: 0.9275

[35]:
```
print(classification_report(y_test_6, clf_LPS.predict(X_test_6[X_ALL])))
```

```
              precision    recall  f1-score   support

           0       0.93      0.93      0.93       417
           1       0.92      0.93      0.92       383

    accuracy                           0.93       800
   macro avg       0.93      0.93      0.93       800
weighted avg       0.93      0.93      0.93       800
```

[36]:
```
PARITY(X_test_6, LPS_pred)
```

```
P_c(recid = 1 | race = African American) = 0.5418410041841004
P_c(recid = 1 | race = Caucasian) = 0.391304347826087
Parity = 0.15053665635801344
```

[36]:
```
0.15053665635801344
```

[37]:
```
CALIBRATION(X_test_6, y_test_6, LPS_pred)
```

```
P_c(recid predicted correctly | race = African American) = 0.9330543933054394
P_c(recid predicted correctly | race = Caucasian) = 0.9192546583850931
Calibration = 0.013799734920346252
```

[38]:
```
EQUALITY_OF_ODDS(X_test_6, y_test_6, LPS_pred)
```

```
For recid = 0:

P_c(recid.hat = 1 | race = African American, recid = 0) =  0.07692307692307693
P_c(recid.hat = 1 | race = Caucasian, recid = 0) =  0.0663265306122449
Difference in odds of true recid = 0 is  = D_FPR = 0.010596546310832025



For recid = 1:

P_c(recid.hat = 1 | race = African American, recid = 1) =  0.9416342412451362
P_c(recid.hat = 1 | race = Caucasian, recid = 1) =  0.8968253968253969
Difference in odds of true recid = 1 is = D_TPR = 0.04480884441973931
```

[39]:
```
# D_FNR

D_FNR(X_test_6, y_test_6, LPS_pred)
```

```
Difference in False Negative Rates
For recid = 1:

P_c(recid.hat = 0 | race = African American, recid = 1) =  0.058365758754863814
P_c(recid.hat = 0 | race = Caucasian, recid = 1) =  0.10317460317460317
D_FNR = 0.044808844419739355
```

3 Fairness Beyond Disparate Treatment & Disparate Impact: Learning Classification without Disparate Mistreatment (DM and DM-sen)

3.1 Baseline Model And Evaluation (Using Neural Network)

[40]:
```python
def base_nn_model(X_in,y_in,X_val,y_val):
    feature = Input(X_in.shape[1],)
    y = Dense(2,"softmax")(feature)
    model = Model(feature,y)

    adam = tf.keras.optimizers.Adam(0.001)
    loss = keras.losses.BinaryCrossentropy(from_logits=True)
    metric = [tf.keras.metrics.BinaryAccuracy()]
    #,tf.keras.metrics.FalsePositives() tf.keras.metrics.FalseNegatives()
    model.compile(optimizer=adam, loss=loss, metrics=metric)
    model.fit(X_in,tf.
 ↪one_hot(y_in,2),epochs=10,batch_size=10,validation_data=(X_val,tf.
 ↪one_hot(y_val,2)))
    return model

def evaluation(model,X,y):
    y_pred = model.predict(X)
    y_pred = np.argmax(np.round(y_pred), axis=1)
    y_pred_AA, y_test_AA = y_pred[test_idx_AA], y[test_idx_AA]
    y_pred_C, y_test_C = y_pred[test_idx_C], y[test_idx_C]
    acc = model.evaluate(X, tf.one_hot(y,2))[1]
    FPR_all = sum(y_pred[y==0]==1)/len(y[y==0])
    FNR_all = sum(y_pred[y==1]==0)/len(y[y==1])
    FPR_AA = sum(y_pred_AA[y_test_AA==0]==1)/len(y_test_AA[y_test_AA==0])
    FNR_AA = sum(y_pred_AA[y_test_AA==1]==0)/len(y_test_AA[y_test_AA==1])
    FPR_C = sum(y_pred_C[y_test_C==0]==1)/len(y_test_C[y_test_C==0])
    FNR_C = sum(y_pred_C[y_test_C==1]==0)/len(y_test_C[y_test_C==1])
    pred_p_AA, pred_p_C = np.mean(y_pred_AA==1), np.mean(y_pred_C==1)
    acc_AA, acc_C = np.mean(y_pred_AA == y_test_AA), np.mean(y_pred_C ==␣
 ↪y_test_C)
    print('\n',"#"*80)
    print('The accuracy of baseline model NN is: %3f.'%(acc))
    print('The False Positive Rate for overall population is: %3f.'%FPR_all)
    print('The False Negative Rate for overall population is: %3f.'%FNR_all)
    print("Specifically:")
    print('Parity Check: The rate of positive estimate for African American and␣
 ↪Caucasian are %3f and %3f, and D_par=%3f.
 ↪'%(pred_p_AA,pred_p_C,pred_p_AA-pred_p_C))
    print('Calibration Check: The rate of correct estimate for African American␣
 ↪and Caucasian are %3f and %3f, and D_cal=%3f.'%(acc_AA,acc_C,acc_AA-acc_C))
```

```
    print('The False Positive Rate for African American and Caucasian are %3f␣
↪and %3f, and D_FPR=%3f.'%(FPR_AA,FPR_C,FPR_AA-FPR_C))
    print('The False Negative Rate for African American and Caucasian are %3f␣
↪and %3f, and D_FNR=%3f.'%(FNR_AA,FNR_C,FNR_AA-FNR_C))
    print('\n',"#"*80)
```

[41]:
```
#If we don't drop 'race' in the X_train and X_test:
NN1 = base_nn_model(X_train,y_train,X_val,y_val)
evaluation(NN1,X_test,y_test)
#If we drop 'race' in the X_train and X_test:
NN2 = base_nn_model(np.delete(X_train,0,1),y_train,np.delete(X_val,0,1),y_val)
evaluation(NN2,np.delete(X_test,1,1),y_test)
```

```
Epoch 1/10
413/413 [==============================] - 2s 4ms/step - loss: 0.6104 -
binary_accuracy: 0.7247 - val_loss: 0.4960 - val_binary_accuracy: 0.8037
Epoch 2/10
413/413 [==============================] - 2s 5ms/step - loss: 0.4546 -
binary_accuracy: 0.8264 - val_loss: 0.4063 - val_binary_accuracy: 0.8550
Epoch 3/10
413/413 [==============================] - 1s 3ms/step - loss: 0.3818 -
binary_accuracy: 0.8680 - val_loss: 0.3522 - val_binary_accuracy: 0.8813
Epoch 4/10
413/413 [==============================] - 1s 2ms/step - loss: 0.3353 -
binary_accuracy: 0.8881 - val_loss: 0.3146 - val_binary_accuracy: 0.8925
Epoch 5/10
413/413 [==============================] - 1s 2ms/step - loss: 0.3025 -
binary_accuracy: 0.8998 - val_loss: 0.2871 - val_binary_accuracy: 0.8988
Epoch 6/10
413/413 [==============================] - 1s 2ms/step - loss: 0.2780 -
binary_accuracy: 0.9075 - val_loss: 0.2665 - val_binary_accuracy: 0.9075
Epoch 7/10
413/413 [==============================] - 1s 2ms/step - loss: 0.2594 -
binary_accuracy: 0.9133 - val_loss: 0.2505 - val_binary_accuracy: 0.9087
Epoch 8/10
413/413 [==============================] - 1s 2ms/step - loss: 0.2451 -
binary_accuracy: 0.9160 - val_loss: 0.2380 - val_binary_accuracy: 0.9100
Epoch 9/10
413/413 [==============================] - 1s 2ms/step - loss: 0.2337 -
binary_accuracy: 0.9179 - val_loss: 0.2275 - val_binary_accuracy: 0.9112
Epoch 10/10
413/413 [==============================] - 1s 2ms/step - loss: 0.2246 -
binary_accuracy: 0.9201 - val_loss: 0.2192 - val_binary_accuracy: 0.9150
25/25 [==============================] - 0s 2ms/step - loss: 0.2076 -
binary_accuracy: 0.9300

################################################################################
The accuracy of baseline model NN is: 0.930000.
```

```
The False Positive Rate for overall population is: 0.083933.
The False Negative Rate for overall population is: 0.054830.
Specifically:
Parity Check: The rate of positive estimate for African American and Caucasian
are 0.562762 and 0.397516, and D_par=0.165246.
Calibration Check: The rate of correct estimate for African American and
Caucasian are 0.928870 and 0.931677, and D_cal=-0.002807.
The False Positive Rate for African American and Caucasian are 0.104072 and
0.061224, and D_FPR=0.042848.
The False Negative Rate for African American and Caucasian are 0.042802 and
0.079365, and D_FNR=-0.036564.

################################################################################
Epoch 1/10
413/413 [==============================] - 2s 3ms/step - loss: 0.6205 -
binary_accuracy: 0.7332 - val_loss: 0.5167 - val_binary_accuracy: 0.7975
Epoch 2/10
413/413 [==============================] - 1s 2ms/step - loss: 0.4597 -
binary_accuracy: 0.8312 - val_loss: 0.4146 - val_binary_accuracy: 0.8525
Epoch 3/10
413/413 [==============================] - 1s 2ms/step - loss: 0.3842 -
binary_accuracy: 0.8678 - val_loss: 0.3559 - val_binary_accuracy: 0.8725
Epoch 4/10
413/413 [==============================] - 1s 2ms/step - loss: 0.3368 -
binary_accuracy: 0.8874 - val_loss: 0.3165 - val_binary_accuracy: 0.8975
Epoch 5/10
413/413 [==============================] - 1s 2ms/step - loss: 0.3038 -
binary_accuracy: 0.8990 - val_loss: 0.2883 - val_binary_accuracy: 0.9050
Epoch 6/10
413/413 [==============================] - 1s 2ms/step - loss: 0.2796 -
binary_accuracy: 0.9097 - val_loss: 0.2673 - val_binary_accuracy: 0.9125
Epoch 7/10
413/413 [==============================] - 1s 2ms/step - loss: 0.2613 -
binary_accuracy: 0.9148 - val_loss: 0.2512 - val_binary_accuracy: 0.9162
Epoch 8/10
413/413 [==============================] - 1s 2ms/step - loss: 0.2470 -
binary_accuracy: 0.9165 - val_loss: 0.2383 - val_binary_accuracy: 0.9162
Epoch 9/10
413/413 [==============================] - 1s 2ms/step - loss: 0.2356 -
binary_accuracy: 0.9167 - val_loss: 0.2278 - val_binary_accuracy: 0.9175
Epoch 10/10
413/413 [==============================] - 1s 2ms/step - loss: 0.2264 -
binary_accuracy: 0.9196 - val_loss: 0.2195 - val_binary_accuracy: 0.9212
25/25 [==============================] - 0s 2ms/step - loss: 0.2076 -
binary_accuracy: 0.9275

################################################################################
The accuracy of baseline model NN is: 0.927500.
```

```
The False Positive Rate for overall population is: 0.079137.
The False Negative Rate for overall population is: 0.065274.
Specifically:
Parity Check: The rate of positive estimate for African American and Caucasian
are 0.554393 and 0.391304, and D_par=0.163089.
Calibration Check: The rate of correct estimate for African American and
Caucasian are 0.924686 and 0.931677, and D_cal=-0.006991.
The False Positive Rate for African American and Caucasian are 0.099548 and
0.056122, and D_FPR=0.043425.
The False Negative Rate for African American and Caucasian are 0.054475 and
0.087302, and D_FNR=-0.032827.

################################################################################
```

3.2 Baseline Model with customized constraint and loss

[42]:
```python
def DFR(model,X,y,type):
    '''
    type: str in ['dfnr','dfpr','both']
    '''
    if type!='dfnr' and type!='dfpr' and type!='both':
        return None
    size = len(y)
    idx_AA = np.array(range(size))[X[:,1]==0.0]
    idx_C = np.array(range(size))[X[:,1]==1.0]
    y_pred = model.predict(X)
    y_pred = np.argmax(np.round(y_pred), axis=1)
    y_pred_AA, y_AA = y_pred[idx_AA], y[idx_AA]
    y_pred_C, y_C = y_pred[idx_C], y[idx_C]
    FPR_AA = sum(y_pred_AA[y_AA==0]==1)/len(y_AA[y_AA==0])
    FNR_AA = sum(y_pred_AA[y_AA==1]==0)/len(y_AA[y_AA==1])
    FPR_C = sum(y_pred_C[y_C==0]==1)/len(y_C[y_C==0])
    FNR_C = sum(y_pred_C[y_C==1]==0)/len(y_C[y_C==1])
    dfnr = FNR_AA-FNR_C
    dfpr = FPR_AA-FPR_C
    if type=='dfnr':
        return dfnr
    elif type=='dfpr':
        return dfpr
    else:
        return dfnr,dfpr
dfnr,dfpr = DFR(NN1,X_test,y_test,'both')
print(dfnr,dfpr)
```

```
-0.0365635229448459 0.04284790839412689
```

[43]:
```python
def new_training_groups(model, X_train, y_train):
    '''
```

```
    X       n*d
    model.predict(X)    n*2
    y       n,
    delta   c,
    dfr   "dfpr","dfnr","both"
    '''

    #split training sets according to sensitive variable
    X_train_AA = X_train[np.array(X_train[:,1] == 0.0)]
    y_train_AA = y_train[np.array(X_train[:,1] == 0.0)]
    X_train_C = X_train[np.array(X_train[:,1] == 1.0)]
    y_train_C = y_train[np.array(X_train[:,1] == 1.0)]
    #get the ones with wrong prediction in discriminated group
    dn,dp = DFR(model,X_train,y_train,type="both")
    if dp>0: d = 0
    else: d = 1

    if d == 0:
        #take penalized trainers
        y_pred_AA = np.argmax(model.predict(X_train_AA),axis = 1)
        y_diff_AA = y_train_AA-y_pred_AA
        X_train_penalized = X_train_AA[y_diff_AA != 0.0]
        y_train_penalized = y_train_AA[y_diff_AA != 0.0]
        # safe trainers
        X_train_clean = X_train_AA[y_diff_AA == 0.0]
        y_train_clean = y_train_AA[y_diff_AA == 0.0]
        #make new
        X_train_clean = np.concatenate((X_train_clean,X_train_C),axis=0)
        y_train_clean = np.concatenate((y_train_clean,y_train_C),axis=0)

    else:
        #reverse the steps above for train set 1
        y_pred_C = np.argmax(model.predict(X_train_C),axis = 1)
        y_diff_C = y_train_C-y_pred_C
        X_train_penalized = X_train_C[y_diff_C != 0.0]
        y_train_penalized = y_train_C[y_diff_C != 0.0]
        # safe trainers
        X_train_clean = X_train_C[y_diff_C == 0.0]
        y_train_clean = y_train_C[y_diff_C == 0.0]
        #make new
        X_train_clean = np.concatenate((X_train_clean,X_train_AA),axis=0)
        y_train_clean = np.concatenate((y_train_clean,y_train_AA),axis=0)

    #X_train_penalized = tf.convert_to_tensor(X_train_penalized, dtype=tf.
↪float32)
    #X_train_safe = tf.convert_to_tensor(X_train_safe, dtype=tf.float32)
```

```
    return X_train_clean , y_train_clean , X_train_penalized ,␣
↪y_train_penalized, dn, dp
```

**Note: When few features are used in the model, the model and its prediction are unstable. Even though features are added, it sometimes ends the loop when D(fpr)>0.05.

[44]:
```
## initialization
np.random.seed(7777)
model =  base_nn_model(X_train, y_train, X_val, y_val)

#initialized C and delta
C = 1
delta = 0.2

# new training groups
X_ts, y_ts, X_tp, y_tp, dn, dp = new_training_groups(model, X_train, y_train)
feature = Input(X_train.shape[1],)
y = Dense(2,"softmax")(feature)
mod_loop = Model(feature,y)
adam = tf.keras.optimizers.Adam(0.001)
loss = keras.losses.BinaryCrossentropy(from_logits=True)
metric = [tf.keras.metrics.BinaryAccuracy()]

def penal_loss(y_true,y_pred):
  return loss(tf.one_hot(y_tp,2), mod_loop(X_tp))

def clean_loss(y_true,y_pred):
  return loss(tf.one_hot(y_ts,2), mod_loop(X_ts))

count = 0
# start while loop
while (count==0 or count%2==1 or abs(dp)>0.05) and count<20:
    # or examine dn, here count%2==1 used to control accuracy in case the␣
↪accuracy is below 0.5 and one of the overall fpr/fnr will be close to 1
    C = C+delta
    #print('Count:%d'%count)
    mod_loop.compile(optimizer=adam,loss=[penal_loss,␣
↪clean_loss],loss_weights=[C,1],metrics=metric)
    mod_loop.fit(X_train, tf.one_hot(y_train,2), epochs=10,␣
↪validation_data=(X_val,tf.one_hot(y_val,2)))
    X_ts, y_ts, X_tp, y_tp, dn, dp = new_training_groups(mod_loop, X_train,␣
↪y_train)
    #dp = DFR(model_in_loop,X_test,y_test,'dfpr')
    count+=1
```

```
Epoch 1/10
413/413 [==============================] - 1s 2ms/step - loss: 0.6254 -
binary_accuracy: 0.7535 - val_loss: 0.5199 - val_binary_accuracy: 0.8275
```

```
Epoch 2/10
413/413 [==============================] - 1s 2ms/step - loss: 0.4576 -
binary_accuracy: 0.8661 - val_loss: 0.4080 - val_binary_accuracy: 0.8750
Epoch 3/10
413/413 [==============================] - 1s 2ms/step - loss: 0.3746 -
binary_accuracy: 0.8864 - val_loss: 0.3449 - val_binary_accuracy: 0.8963
Epoch 4/10
413/413 [==============================] - 1s 2ms/step - loss: 0.3237 -
binary_accuracy: 0.9015 - val_loss: 0.3037 - val_binary_accuracy: 0.9000
Epoch 5/10
413/413 [==============================] - 1s 2ms/step - loss: 0.2893 -
binary_accuracy: 0.9102 - val_loss: 0.2746 - val_binary_accuracy: 0.9075
Epoch 6/10
413/413 [==============================] - 1s 2ms/step - loss: 0.2650 -
binary_accuracy: 0.9157 - val_loss: 0.2539 - val_binary_accuracy: 0.9125
Epoch 7/10
413/413 [==============================] - 1s 2ms/step - loss: 0.2472 -
binary_accuracy: 0.9182 - val_loss: 0.2384 - val_binary_accuracy: 0.9137
Epoch 8/10
413/413 [==============================] - 1s 2ms/step - loss: 0.2339 -
binary_accuracy: 0.9206 - val_loss: 0.2265 - val_binary_accuracy: 0.9162
Epoch 9/10
413/413 [==============================] - 1s 2ms/step - loss: 0.2236 -
binary_accuracy: 0.9208 - val_loss: 0.2173 - val_binary_accuracy: 0.9162
Epoch 10/10
413/413 [==============================] - 1s 2ms/step - loss: 0.2156 -
binary_accuracy: 0.9232 - val_loss: 0.2102 - val_binary_accuracy: 0.9162
Epoch 1/10
130/130 [==============================] - 1s 3ms/step - loss: 0.9071 -
binary_accuracy: 0.4160 - val_loss: 0.8304 - val_binary_accuracy: 0.3862
Epoch 2/10
130/130 [==============================] - 0s 2ms/step - loss: 0.7811 -
binary_accuracy: 0.3630 - val_loss: 0.7392 - val_binary_accuracy: 0.3475
Epoch 3/10
130/130 [==============================] - 0s 2ms/step - loss: 0.7104 -
binary_accuracy: 0.3397 - val_loss: 0.6851 - val_binary_accuracy: 0.3250
Epoch 4/10
130/130 [==============================] - 0s 2ms/step - loss: 0.6661 -
binary_accuracy: 0.3143 - val_loss: 0.6485 - val_binary_accuracy: 0.2850
Epoch 5/10
130/130 [==============================] - 0s 2ms/step - loss: 0.6341 -
binary_accuracy: 0.2833 - val_loss: 0.6203 - val_binary_accuracy: 0.2488
Epoch 6/10
130/130 [==============================] - 0s 2ms/step - loss: 0.6083 -
binary_accuracy: 0.2470 - val_loss: 0.5966 - val_binary_accuracy: 0.2188
Epoch 7/10
130/130 [==============================] - 0s 2ms/step - loss: 0.5861 -
binary_accuracy: 0.2136 - val_loss: 0.5758 - val_binary_accuracy: 0.1975
```

```
Epoch 8/10
130/130 [==============================] - 0s 2ms/step - loss: 0.5664 -
binary_accuracy: 0.1966 - val_loss: 0.5571 - val_binary_accuracy: 0.1875
Epoch 9/10
130/130 [==============================] - 0s 2ms/step - loss: 0.5486 -
binary_accuracy: 0.1801 - val_loss: 0.5402 - val_binary_accuracy: 0.1713
Epoch 10/10
130/130 [==============================] - 0s 2ms/step - loss: 0.5324 -
binary_accuracy: 0.1695 - val_loss: 0.5245 - val_binary_accuracy: 0.1562
Epoch 1/10
130/130 [==============================] - 1s 3ms/step - loss: 1.7477 -
binary_accuracy: 0.2481 - val_loss: 1.3499 - val_binary_accuracy: 0.3738
Epoch 2/10
130/130 [==============================] - 0s 2ms/step - loss: 1.1777 -
binary_accuracy: 0.4649 - val_loss: 1.0434 - val_binary_accuracy: 0.5213
Epoch 3/10
130/130 [==============================] - 0s 2ms/step - loss: 0.9578 -
binary_accuracy: 0.5772 - val_loss: 0.8824 - val_binary_accuracy: 0.6025
Epoch 4/10
130/130 [==============================] - 0s 2ms/step - loss: 0.8256 -
binary_accuracy: 0.6380 - val_loss: 0.7730 - val_binary_accuracy: 0.6413
Epoch 5/10
130/130 [==============================] - 0s 2ms/step - loss: 0.7302 -
binary_accuracy: 0.6726 - val_loss: 0.6896 - val_binary_accuracy: 0.6837
Epoch 6/10
130/130 [==============================] - 0s 2ms/step - loss: 0.6554 -
binary_accuracy: 0.7002 - val_loss: 0.6224 - val_binary_accuracy: 0.7125
Epoch 7/10
130/130 [==============================] - 0s 2ms/step - loss: 0.5941 -
binary_accuracy: 0.7262 - val_loss: 0.5666 - val_binary_accuracy: 0.7325
Epoch 8/10
130/130 [==============================] - 0s 2ms/step - loss: 0.5427 -
binary_accuracy: 0.7453 - val_loss: 0.5193 - val_binary_accuracy: 0.7550
Epoch 9/10
130/130 [==============================] - 0s 2ms/step - loss: 0.4987 -
binary_accuracy: 0.7656 - val_loss: 0.4785 - val_binary_accuracy: 0.7713
Epoch 10/10
130/130 [==============================] - 0s 2ms/step - loss: 0.4607 -
binary_accuracy: 0.7816 - val_loss: 0.4431 - val_binary_accuracy: 0.7825
Epoch 1/10
130/130 [==============================] - 1s 3ms/step - loss: 1.4945 -
binary_accuracy: 0.8306 - val_loss: 1.0354 - val_binary_accuracy: 0.8587
Epoch 2/10
130/130 [==============================] - 0s 2ms/step - loss: 0.8704 -
binary_accuracy: 0.8119 - val_loss: 0.7487 - val_binary_accuracy: 0.7375
Epoch 3/10
130/130 [==============================] - 0s 2ms/step - loss: 0.6773 -
binary_accuracy: 0.6806 - val_loss: 0.6167 - val_binary_accuracy: 0.6288
```

```
Epoch 4/10
130/130 [==============================] - 0s 2ms/step - loss: 0.5740 -
binary_accuracy: 0.6034 - val_loss: 0.5355 - val_binary_accuracy: 0.5838
Epoch 5/10
130/130 [==============================] - 0s 2ms/step - loss: 0.5059 -
binary_accuracy: 0.5538 - val_loss: 0.4783 - val_binary_accuracy: 0.5437
Epoch 6/10
130/130 [==============================] - 0s 2ms/step - loss: 0.4560 -
binary_accuracy: 0.5211 - val_loss: 0.4349 - val_binary_accuracy: 0.5225
Epoch 7/10
130/130 [==============================] - 0s 2ms/step - loss: 0.4172 -
binary_accuracy: 0.4990 - val_loss: 0.4002 - val_binary_accuracy: 0.5075
Epoch 8/10
130/130 [==============================] - 0s 2ms/step - loss: 0.3856 -
binary_accuracy: 0.4831 - val_loss: 0.3715 - val_binary_accuracy: 0.4938
Epoch 9/10
130/130 [==============================] - 0s 2ms/step - loss: 0.3592 -
binary_accuracy: 0.4676 - val_loss: 0.3472 - val_binary_accuracy: 0.4850
Epoch 10/10
130/130 [==============================] - 0s 2ms/step - loss: 0.3366 -
binary_accuracy: 0.4547 - val_loss: 0.3262 - val_binary_accuracy: 0.4787
Epoch 1/10
130/130 [==============================] - 1s 3ms/step - loss: 1.9943 -
binary_accuracy: 0.5830 - val_loss: 1.0849 - val_binary_accuracy: 0.7212
Epoch 2/10
130/130 [==============================] - 0s 2ms/step - loss: 0.8227 -
binary_accuracy: 0.7588 - val_loss: 0.6399 - val_binary_accuracy: 0.7912
Epoch 3/10
130/130 [==============================] - 0s 2ms/step - loss: 0.5459 -
binary_accuracy: 0.7966 - val_loss: 0.4707 - val_binary_accuracy: 0.8150
Epoch 4/10
130/130 [==============================] - 0s 2ms/step - loss: 0.4228 -
binary_accuracy: 0.8259 - val_loss: 0.3815 - val_binary_accuracy: 0.8338
Epoch 5/10
130/130 [==============================] - 0s 2ms/step - loss: 0.3520 -
binary_accuracy: 0.8329 - val_loss: 0.3253 - val_binary_accuracy: 0.8562
Epoch 6/10
130/130 [==============================] - 0s 2ms/step - loss: 0.3049 -
binary_accuracy: 0.8395 - val_loss: 0.2861 - val_binary_accuracy: 0.8562
Epoch 7/10
130/130 [==============================] - 0s 2ms/step - loss: 0.2710 -
binary_accuracy: 0.8407 - val_loss: 0.2567 - val_binary_accuracy: 0.8587
Epoch 8/10
130/130 [==============================] - 0s 2ms/step - loss: 0.2450 -
binary_accuracy: 0.8390 - val_loss: 0.2338 - val_binary_accuracy: 0.8550
Epoch 9/10
130/130 [==============================] - 0s 2ms/step - loss: 0.2244 -
binary_accuracy: 0.8378 - val_loss: 0.2153 - val_binary_accuracy: 0.8512
```

```
Epoch 10/10
130/130 [==============================] - 0s 2ms/step - loss: 0.2075 -
binary_accuracy: 0.8361 - val_loss: 0.1999 - val_binary_accuracy: 0.8500
```

[45]:
```
#Evaluation
print(DFR(mod_loop,X_test,y_test,type='both'))
evaluation(mod_loop,X_test,y_test)
```

```
(-0.06858748687542465, 0.07519161510758149)
25/25 [==============================] - 0s 1ms/step - loss: 0.1999 -
binary_accuracy: 0.8375

################################################################################
The accuracy of baseline model NN is: 0.837500.
The False Positive Rate for overall population is: 0.055156.
The False Negative Rate for overall population is: 0.279373.
Specifically:
Parity Check: The rate of positive estimate for African American and Caucasian
are 0.441423 and 0.273292, and D_par=0.168131.
Calibration Check: The rate of correct estimate for African American and
Caucasian are 0.820084 and 0.863354, and D_cal=-0.043270.
The False Positive Rate for African American and Caucasian are 0.090498 and
0.015306, and D_FPR=0.075192.
The False Negative Rate for African American and Caucasian are 0.256809 and
0.325397, and D_FNR=-0.068587.

################################################################################
```

[45]:
```
##################################################################################################
####################### IF NEED TO MODIFY CODES, USE THIS CHUNK TO RUN ABOVE OR␣
↪RUN BELOW ###########################
##################################################################################################
```

3.3 Implementation of DM_{sen} & DM algorithm

3.3.1 Implementation on DM_{sen}

We first implement DM_{sen} as we won't do anything to the dataset:

[46]:
```
#Use 4:1 as the ratio of train:test
y = data.two_year_recid
y = np.asarray(y).astype('float32')

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=0)
train_size = len(y_train)
train_idx_AA = np.array(range(train_size))[X_train[:,1]==0.0]
train_idx_C = np.array(range(train_size))[X_train[:,1]==1.0]
test_size = len(y_test)
```

```
test_idx_AA = np.array(range(test_size))[X_test[:,1]==0.0]
test_idx_C = np.array(range(test_size))[X_test[:,1]==1.0]

X_train1 = np.hstack((np.ones(X_train.shape[0]).reshape(X_train.shape[0],1),␣
 ↪X_train))
X_train1_AA = X_train1[train_idx_AA]
X_train1_C = X_train1[train_idx_C]
y_train_AA = y_train[train_idx_AA]
y_train_C = y_train[train_idx_C]
X_test1 = np.hstack((np.ones(X_test.shape[0]).reshape(X_test.shape[0],1),␣
 ↪X_test))
X_test1_AA = X_test1[test_idx_AA]
X_test1_C = X_test1[test_idx_C]
y_test_AA = y_test[test_idx_AA]
y_test_C = y_test[test_idx_C]

print('\n',"#"*80,'\n',' '*20," Split up Train-Test sets ",'\n',"#"*80,'\n')
print(" X_train size: ", X_train1.shape, ",      y_train size: ", y_train.
 ↪shape, '\n',
      "X_test size: ", X_test1.shape, ',       y_test size: ',y_test.shape)
print(" X_train_AA size: ", X_train1[train_idx_AA].shape, ",   X_train_C size:␣
 ↪", X_train1[train_idx_C].shape, '\n',
      "X_test_AA size: ", X_test1[test_idx_AA].shape, ',     X_test_C size:␣
 ↪',X_test1[test_idx_C].shape, '\n',
      "Ratio:", X_train1[train_idx_AA].shape[0]/X_train1[train_idx_C].
 ↪shape[0],X_test1[test_idx_AA].shape[0]/X_test1[test_idx_C].shape[0])
print('\n',"#"*80)
```

```
################################################################################
                       Split up Train-Test sets
################################################################################

 X_train size:  (4584, 16) ,      y_train size:  (4584,)
 X_test size:  (1146, 16) ,       y_test size:  (1146,)
 X_train_AA size:  (2745, 16) ,   X_train_C size:  (1839, 16)
 X_test_AA size:  (688, 16) ,     X_test_C size:  (458, 16)
 Ratio: 1.4926590538336053 1.502183406113537

################################################################################
```

(Note: From the paper, loss function is modified in logistic regression)

[48]:
```
#pip install dccp #when running in jupyter notebook, delete this chunk
```

[49]:
```
###PLEASE DON'T MOVE THIS CHUNK TO 1.1 IMPORT ESSENTIAL PACKAGES
import dccp
import cvxpy as cvx
```

```python
from cvxpy import *
```

[50]:
```python
def lossfunc(X,theta,y_true):
    # This function returns the log loss.
    y_true= 2*y_true - 1 #{0,1}->{-1,1}
    log_loss = sum(logistic(multiply(-y_true, X*theta)))
    return log_loss


np.random.seed(5243)
theta = cvx.Variable(X_train1.shape[1])
theta.value = np.random.rand(theta.shape[0])

tau, mu, EPS = 0.005, 1.5, 1
Prob1 = cvx.Problem(Minimize(lossfunc(X_train1,theta,y_train)),[]) # No␣
↪constraints


print(dccp.is_dccp(Prob1))
#print(theta.value)
#[0.5591043  0.9994264  0.57031546 0.43833912 0.08453454 0.05043884
# 0.91119515 0.16423428 0.3034639  0.41950956 0.85237613 0.4244003
# 0.96147514 0.26277008 0.02849745 0.61075812]
result = Prob1.solve(method='dccp', tau=tau, mu=mu, tau_max=1e10, verbose=True)␣
↪#Here changes the theta.value, result avoids the output
#print(theta.value)
#[-2.02760707  0.12980288  0.12303316 -0.27410271  1.71950929  4.75786739
# -0.27591511  0.02399848  0.45356176  0.05861524  0.04602311  0.27193313
# -0.04650322  0.15181987 -0.45112505 -2.24001541]
```

True

[51]:
```python
def predict(X,theta):
    #y:{-1,1}->{0,1}
    d = np.dot(X,theta)
    y_pred = (np.sign(d) + 1)/2
    return y_pred

theta_star = theta.value
y_pred = predict(X_test1, theta_star)
```

[52]:
```python
def evaluation_DM(X,y,y_pred):
    size = X.shape[0]
    idx_AA = np.array(range(size))[X[:,1]==0.0]
    idx_C = np.array(range(size))[X[:,1]==1.0]
    y_pred_AA, y_test_AA = y_pred[idx_AA], y[idx_AA]
    y_pred_C, y_test_C = y_pred[idx_C], y[idx_C]
```

```
    FPR_all = np.sum(y_pred[y==0]==1)/len(y[y==0])
    FNR_all = np.sum(y_pred[y==1]==0)/len(y[y==1])
    FPR_AA = np.sum(y_pred_AA[y_test_AA==0]==1)/len(y_test_AA[y_test_AA==0])
    FNR_AA = np.sum(y_pred_AA[y_test_AA==1]==0)/len(y_test_AA[y_test_AA==1])
    FPR_C = np.sum(y_pred_C[y_test_C==0]==1)/len(y_test_C[y_test_C==0])
    FNR_C = np.sum(y_pred_C[y_test_C==1]==0)/len(y_test_C[y_test_C==1])
    pred_p_AA, pred_p_C = np.mean(y_pred_AA==1), np.mean(y_pred_C==1)
    acc = np.sum(y_pred == y)/len(y)
    acc_AA, acc_C = np.mean(y_pred_AA == y_test_AA), np.mean(y_pred_C ==␣
 ↪y_test_C)
    print('\n',"#"*80)
    print('The accuracy of baseline model LR is: %3f.'%(acc))
    print('The False Positive Rate for overall population is: %3f.'%FPR_all)
    print('The False Negative Rate for overall population is: %3f.'%FNR_all)
    print("Specifically:")
    print('Parity Check: The rate of positive estimate for African American and␣
 ↪Caucasian are %3f and %3f, and D_par=%3f.
 ↪'%(pred_p_AA,pred_p_C,pred_p_AA-pred_p_C))
    print('Calibration Check: The rate of correct estimate for African American␣
 ↪and Caucasian are %3f and %3f, and D_cal=%3f.'%(acc_AA,acc_C,acc_AA-acc_C))
    print('The False Positive Rate for African American and Caucasian are %3f␣
 ↪and %3f, and D_FPR=%3f.'%(FPR_AA,FPR_C,FPR_AA-FPR_C))
    print('The False Negative Rate for African American and Caucasian are %3f␣
 ↪and %3f, and D_FNR=%3f.'%(FNR_AA,FNR_C,FNR_AA-FNR_C))
    print('\n',"#"*80)
#print(y_pred.shape)
evaluation_DM(X_test1,y_test,y_pred)
```

```
################################################################################
The accuracy of baseline model LR is: 0.924084.
The False Positive Rate for overall population is: 0.084956.
The False Negative Rate for overall population is: 0.067126.
Specifically:
Parity Check: The rate of positive estimate for African American and Caucasian
are 0.423729 and 0.538462, and D_par=-0.114733.
Calibration Check: The rate of correct estimate for African American and
Caucasian are 0.932203 and 0.921978, and D_cal=0.010225.
The False Positive Rate for African American and Caucasian are 0.077465 and
0.087470, and D_FPR=-0.010006.
The False Negative Rate for African American and Caucasian are 0.053191 and
0.069815, and D_FNR=-0.016624.

################################################################################
```

If we do not put constaints on the loss function, the accuracy of the logistic regression model is around 92%, while the FPR, FNR are around 0.05.

Then, we put the constraint in the following model:

[53]:
```
# Constaints on loss function
np.random.seed(5243)
theta1 = cvx.Variable(X_train1.shape[1])
theta1.value = np.random.rand(theta.shape[0])

tau, mu, EPS = 0.5, 1.6, 1e-4

def g_theta(y,X,theta):
    y = 2*y - 1
    d = matmul(X,theta)
    y_d = multiply(y,d)
    return minimum(np.zeros_like(y_d),y_d)

c = 0.05
N0 = X_train1_AA.shape[0]
N1 = X_train1_C.shape[0]
N = X_train1.shape[0]
print(N,N0,N1)


Prob2 = cvx.Problem(Minimize(lossfunc(X_train1,theta1,y_train)),
                 [N0/N*sum(g_theta(y_train_C,X_train1_C,theta1)) <= c + N1/
 ↪N*sum(g_theta(y_train_AA, X_train1_AA,theta1)),
                  N0/N*sum(g_theta(y_train_C,X_train1_C,theta1)) >= N1/
 ↪N*sum(g_theta(y_train_AA, X_train1_AA,theta1)) - c]) # With constraints
print(dccp.is_dccp(Prob2))
result1 = Prob2.solve(method='dccp', tau=tau, mu=mu, tau_max=1e10, verbose=True)
#g_theta(y_train,X_train1, theta.value).value.shape
#constraint()
#X_train.
#pd.DataFrame(X_train)
#pd.DataFrame(x_train1)
```

```
4584 2745 1839
True
```

[54]:
```
y_pred1 = predict(X_test1, theta1.value)
evaluation_DM(X_test1,y_test,y_pred1)
```

```
################################################################################
The accuracy of baseline model LR is: 0.924084.
The False Positive Rate for overall population is: 0.088496.
The False Negative Rate for overall population is: 0.063683.
Specifically:
Parity Check: The rate of positive estimate for African American and Caucasian
```

```
are 0.427966 and 0.541758, and D_par=-0.113792.
Calibration Check: The rate of correct estimate for African American and
Caucasian are 0.927966 and 0.923077, and D_cal=0.004889.
The False Positive Rate for African American and Caucasian are 0.084507 and
0.089835, and D_FPR=-0.005327.
The False Negative Rate for African American and Caucasian are 0.053191 and
0.065708, and D_FNR=-0.012517.

################################################################################
```

From above result, we may see the DM_{sen} algorithm slightly drops the D_{FPR} to around -0.005, which is very close to 0.

3.3.2 implementation on DM

[55]:
```
X_train1_sen = np.delete(X_train1,2,1)
X_train1_AA_sen = X_train1_sen[train_idx_AA]
X_train1_C_sen = X_train1_sen[train_idx_C]
X_test1_sen = np.delete(X_test1,2,1)
X_test1_AA_sen = X_test1_sen[test_idx_AA]
X_test1_C_sen = X_test1_sen[test_idx_C]

print('\n',"#"*80,'\n',' '*20," Split up Train-Test sets ",'\n',"#"*80,'\n')
print(" X_train1_sen size: ", X_train1_sen.shape, ",      y_train size: ",␣
↪y_train.shape, '\n',
      "X_test1_sen size: ", X_test1_sen.shape, ',       y_test size: ',y_test.
↪shape)
print(" X_train1_AA_sen size: ", X_train1_AA_sen.shape, ",   X_train1_C_sen␣
↪size: ", X_train1_C_sen.shape, '\n',
      "X_test1_AA_sen size: ", X_test1_AA_sen.shape, ',     X_test1_C_sen size:␣
↪',X_test1_C_sen.shape, '\n',
      "Ratio:", X_train1_AA_sen.shape[0]/X_train1_C_sen.shape[0],X_test1_AA_sen.
↪shape[0]/X_test1_C_sen.shape[0])
print('\n',"#"*80)
```

```
################################################################################
                        Split up Train-Test sets
################################################################################

 X_train1_sen size:  (4584, 15) ,      y_train size:  (4584,)
 X_test1_sen size:  (1146, 15) ,       y_test size:  (1146,)
 X_train1_AA_sen size:  (2745, 15) ,   X_train1_C_sen size:  (1839, 15)
 X_test1_AA_sen size:  (688, 15) ,     X_test1_C_sen size:  (458, 15)
 Ratio: 1.4926590538336053 1.502183406113537

################################################################################
```

[56]:
```
# Constaints on loss function
np.random.seed(5243)
theta2 = cvx.Variable(X_train1_sen.shape[1])
theta2.value = np.random.rand(theta2.shape[0])

tau, mu, EPS = 0.5, 1.6, 1e-4
c = 0.05
N0 = X_train1_AA_sen.shape[0]
N1 = X_train1_C_sen.shape[0]
N = X_train1_sen.shape[0]
print(N,N0,N1)


Prob2 = cvx.Problem(Minimize(lossfunc(X_train1_sen,theta2,y_train)),
                  [N0/N*sum(g_theta(y_train_C,X_train1_C_sen,theta2)) <= c + N1/
 ↪N*sum(g_theta(y_train_AA,X_train1_AA_sen,theta2)),
                   N0/N*sum(g_theta(y_train_C,X_train1_C_sen,theta2)) >= N1/
 ↪N*sum(g_theta(y_train_AA,X_train1_AA_sen,theta2)) - c]) # With constraints
print(dccp.is_dccp(Prob2))
result1 = Prob2.solve(method='dccp', tau=tau, mu=mu, tau_max=1e10, verbose=True)
```

```
4584 2745 1839
True
```

[58]:
```
y_pred2 = predict(X_test1_sen, theta2.value)
evaluation_DM(X_test1_sen,y_test,y_pred2)
```

```
################################################################################
The accuracy of baseline model LR is: 0.925829.
The False Positive Rate for overall population is: 0.084956.
The False Negative Rate for overall population is: 0.063683.
Specifically:
Parity Check: The rate of positive estimate for African American and Caucasian
are 0.423729 and 0.540659, and D_par=-0.116931.
Calibration Check: The rate of correct estimate for African American and
Caucasian are 0.932203 and 0.924176, and D_cal=0.008028.
The False Positive Rate for African American and Caucasian are 0.077465 and
0.087470, and D_FPR=-0.010006.
The False Negative Rate for African American and Caucasian are 0.053191 and
0.065708, and D_FNR=-0.012517.

################################################################################
```

[59]:
```
end = time.time()
print('The running time of overall algorithm is: %3fs.'%(end-start)) # around␣
 ↪100 seconds
```

```
The running time of overall algorithm is: 416.239410s.
```

We may see DM algorithm drops D_{FNR}. But from the above results, it's hard for us to see which one is perfect and how DM_{sen} violates the disparate treatment. Overall speaking, this algorithm has an impact on controlling the difference in FPR and FNR, but the effect deserves further study as when few features are in the model, both two algorithms seem to have no effect on controlling our target.