main

December 8, 2021

0.1 Import

[1]:
```
import numpy as np
from numpy import mean, std
import scipy.optimize as optim
import pandas as pd
import math
from sklearn.metrics import accuracy_score
from tabulate import tabulate
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import KFold, cross_val_score
import time
```

0.2 Data Preprocessing

[2]:
```
# import data
data=pd.read_csv('../data/compas-scores-two-years.csv')
# filter groups other than African-American and Caucasian
data = data[(data['race']=='African-American') | (data['race']=='Caucasian')]
# data['race'].loc[data['race']=='African-American']= 1
# data['race'].loc[data['race']=='Caucasian']= 0
data = data.replace({'race': 'Caucasian'}, 1)
data = data.replace({'race': 'African-American'}, 0)
```

We first deleted some independent variables such as names, dates, and variables with lots of missing values or NA values.

[3]:
```
data=data.drop(columns=['id', 'name', 'first',␣
 ↪'last','compas_screening_date','dob','age','c_jail_in', 'c_jail_out',␣
 ↪'c_case_number','c_offense_date','c_charge_desc',␣
 ↪'c_arrest_date','r_charge_desc',
'r_case_number','r_charge_desc','r_offense_date', 'r_jail_in',␣
 ↪'r_jail_out','violent_recid','vr_case_number',
'vr_offense_date', 'vr_charge_desc',␣
 ↪'screening_date','v_screening_date','in_custody','out_custody','r_charge_degree','r_days_fr
```

```
↪'vr_charge_degree','type_of_assessment','v_type_of_assessment' ])
```

[4]:
```
data.shape
```

[4]: (6150, 22)

[5]:
```
# drop NA values
data=data.dropna()
```

[6]:
```
# create dummy variables for categorical variables
data['sex'].loc[data['sex']=='Male']= 1
data['sex'].loc[data['sex']=='Female']= 0
data['age_cat'].loc[data['age_cat']=='25 - 45']= 'B'
data['age_cat'].loc[data['age_cat']=='Greater than 45']= 'C'
data['age_cat'].loc[data['age_cat']=='Less than 25']= 'A'
data.loc[data['age_cat']=='A', 'age_cat1'] = 1
data.loc[data['age_cat']!='A', 'age_cat1'] = 0
data.loc[data['age_cat']=='B', 'age_cat2'] = 1
data.loc[data['age_cat']!='B', 'age_cat2'] = 0
data['c_charge_degree'].loc[data['c_charge_degree']=='M']= 1
data['c_charge_degree'].loc[data['c_charge_degree']=='F']= 0
data['v_score_text'].loc[data['v_score_text']=='High']= 'A'
data['v_score_text'].loc[data['v_score_text']=='Low']= 'C'
data['v_score_text'].loc[data['v_score_text']=='Medium']= 'B'
data.loc[data['v_score_text']=='A', 'v_score_text1'] = 1
data.loc[data['v_score_text']!='A', 'v_score_text1'] = 0
data.loc[data['v_score_text']=='B', 'v_score_text2'] = 1
data.loc[data['v_score_text']!='B', 'v_score_text2'] = 0

data['score_text'].loc[data['score_text']=='High']= 'A'
data['score_text'].loc[data['score_text']=='Low']= 'B'
data['score_text'].loc[data['score_text']=='Medium']= 'C'
data.loc[data['score_text']=='A', 'score_text1'] = 1
data.loc[data['score_text']!='A', 'score_text1'] = 0
data.loc[data['score_text']=='B', 'score_text2'] = 1
data.loc[data['score_text']!='B', 'score_text2'] = 0
```

```
/anaconda3/lib/python3.7/site-packages/pandas/core/indexing.py:189:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/indexing.html#indexing-view-versus-copy
  self._setitem_with_indexer(indexer, value)
```

[7]:
```python
# drop original categorical variables
data=data.drop(columns=['age_cat','v_score_text','score_text'])
```

[8]:
```python
# checking that there is no NaN values in the current dataset
new_table_content = [['predictor variable', '# of unique values', '# of NaN␣
↪values']]
for item in data.columns:
    new_table_content.append([item, len(data[item].unique()), sum(data[item].
↪isna() == True)])
```

[9]:
```python
print(tabulate(new_table_content, headers='firstrow', tablefmt='fancy_grid',␣
↪showindex=range(1,26)))
```

	predictor variable	# of unique values	# of NaN values
1	sex	2	0
2	race	2	0
3	juv_fel_count	10	0
4	decile_score	10	0
5	juv_misd_count	10	0
6	juv_other_count	8	0
7	priors_count	37	0
8	days_b_screening_arrest	381	0
9	c_days_from_compas	342	0
10	c_charge_degree	2	0
11	is_recid	2	0
12	is_violent_recid	2	0
13	decile_score.1	10	0
14	v_decile_score	10	0
15	priors_count.1	37	0
16	start	221	0

17	end	1087	0
18	event	2	0
19	two_year_recid	2	0
20	age_cat1	2	0
21	age_cat2	2	0
22	v_score_text1	2	0
23	v_score_text2	2	0
24	score_text1	2	0
25	score_text2	2	0

[10]:
```
data.head()
```

[10]:
```
   sex  race  juv_fel_count  decile_score  juv_misd_count  juv_other_count  \
1    1     0              0             3               0                0
2    1     0              0             4               0                1
6    1     1              0             6               0                0
8    0     1              0             1               0                0
9    1     1              0             3               0                0

   priors_count  days_b_screening_arrest  c_days_from_compas  c_charge_degree  \
1             0                     -1.0                 1.0                0
2             4                     -1.0                 1.0                0
6            14                     -1.0                 1.0                0
8             0                     -1.0                 1.0                1
9             1                    428.0               308.0                0

     …     start  end  event  two_year_recid  age_cat1  age_cat2  \
1    …         9  159      1               1       0.0       1.0
2    …         0   63      0               1       1.0       0.0
6    …         5   40      1               1       0.0       1.0
8    …         2  747      0               0       0.0       1.0
9    …         0  428      1               1       1.0       0.0

   v_score_text1  v_score_text2  score_text1  score_text2
1            0.0            0.0          0.0          1.0
2            0.0            0.0          0.0          1.0
6            0.0            0.0          0.0          0.0
```

```
8              0.0          0.0          0.0          1.0
9              0.0          1.0          0.0          1.0

[5 rows x 25 columns]
```

0.3 Data Split

Split the dataset into training, validation and testing set (0.6, 0.2, 0.2)

[11]:
```
data_train, data_test = train_test_split(data, test_size=0.2, random_state=1)
data_train, data_val= train_test_split(data_train, test_size=0.25,␣
↪random_state=1)
```

[12]:
```
data_train=pd.concat([data_train.sex,data_train.race,data_train.
↪juv_fel_count,data_train.decile_score,data_train.two_year_recid],axis=1)
data_train=data_train.iloc[:20]
data_val=pd.concat([data_val.sex,data_val.race,data_val.juv_fel_count,data_val.
↪decile_score,data_val.two_year_recid],axis=1)
data_test=pd.concat([data_test.sex,data_test.race,data_test.
↪juv_fel_count,data_test.decile_score,data_test.two_year_recid],axis=1)
```

[13]:
```
data_train.head()
```

[13]:

	sex	race	juv_fel_count	decile_score	two_year_recid
246	1	1	0	1	0
3883	1	0	0	1	0
6410	1	0	0	2	0
437	1	1	0	10	1
3497	0	1	0	10	1

0.4 Learning Fair Representations

The following functions are helper functions for LFR

[14]:
```
# distance - d(x_n, v_k, alpha)
# this function returns a distance matrix of a shape NxK
def distance(X, v, alpha):
    # X is the dataset
    # v is a list of vectors, where each vector v_k has a length of D
    # alpha is a list of weights, each alpha_i is the weight for some feature
    N = X.shape[0]
    D = X.shape[1]
    K = len(v)

    # initialize the distance matrix
```

```python
    res = np.zeros((N, K))

    # calculate distances
    for n in range(N):
        for k in range(K):
            for d in range(D):
                res[n, k] += alpha[d]*(X.iloc[n][d] - v[k, d])**2

    return res
```

[15]:
```python
# M_nk = P(Z=k|x_n), which is the probablity that x_n maps to v_k
# this function returns a M_nk matrix with shape NxK
def M_nk(dist, k):
    # dist is the distance matrix
    # K is the number of v_k
    N = dist.shape[0]
    K = dist.shape[1]
    M_nk = np.zeros((N, K))
    # initialize the M_nk matrix
    expo_res = np.zeros((N, K))

    # calculate M_nk
    for n in range(N):
        deno = 0
        for k in range(K):
            expo_res[n, k] = math.exp((-1)*dist[n, k])
            deno += expo_res[n, k]
        for k in range(K):
            M_nk[n, k] = expo_res[n, k] / deno
    return M_nk
```

[16]:
```python
# this function returns a list of M_k
def M_k(X, M_nk, k):
    # X is the dataset
    # M_nk is the M_nk matrix
    # K is the number of v_k
    N = X.shape[0]
    K = M_nk.shape[1]
    M_k = np.zeros(K)

    # calculate M_k
    for k in range(K):
        for n in range(N):
            M_k[k] += M_nk[n, k]
        M_k[k] = M_k[k]/N
    return M_k
```

[17]:
```python
# this function returns the reconstruction of x_n of a shape NxD and L_x
def x_n_hat(X, M_nk, v):
    # X is the dataset
    # M_nk is the M_nk matrix
    # v is a list of vectors, where each vector v_k has a length of D
    N = M_nk.shape[0]
    D = X.shape[1]
    K = M_nk.shape[1]
    x_n_hat = np.zeros((N, D))
    L_x = 0

    # calculate x_n
    for n in range(N):
        for d in range(D):
            for k in range(K):
                x_n_hat[n, d] += M_nk[n, k]*v[k, d]
        # calculate L_x
        L_x += (X.iloc[n][d] - x_n_hat[n, d])**2

    return x_n_hat, L_x
```

[18]:
```python
# this function calculates the list of prediction for y_n and L_y
def y_n_hat(M_nk, w, y):
    # M_nk is the M_nk matrix
    # w is a list of weights of length K
    # y is the corresponding list of labels for y_n
    N = M_nk.shape[0]
    K = M_nk.shape[1]
    y_n_hat = np.zeros(N)
    L_y = 0

    # calculate prediction for y_n
    for n in range(N):
        for k in range(K):
            y_n_hat[n] += M_nk[n, k]*w[k]
        # calculate L_y
        L_y += (-1)*y.iloc[n]*np.log(y_n_hat[n]) - (1 - y.iloc[n])*np.log(1 -␣
 ↪y_n_hat[n])

    return y_n_hat, L_y
```

[19]:
```python
# this functions returns the metric function we want to minimize
def L(param, sen_df, nsen_df, sen_y, nonsen_y, K, A_z, A_x, A_y):
    # param is the list of parameters
    # sen_df is the sensitive dataset
    # nsen_df is the nonsensitive dataset
    # sen_y is the list of labels for sensitive dataset
```

```
    # nonsen_y is the list of labels for nonsensitive dataset
    # K, A_z, A_x and A_y are hyperparameters, the values are decided by the␣
↪users

    sen_N, sen_D = sen_df.shape
    nsen_N, nsen_D = nsen_df.shape

    # form parameters in correct forms
    alpha_sen = param[:sen_D]
    alpha_nsen = param[sen_D : 2 * sen_D]
    w = param[2 * sen_D : (2 * sen_D) + K]
    v = np.matrix(param[(2 * sen_D) + K:]).reshape((K, sen_D))

    # calculate the distance matrix
    dist_sen = distance(sen_df, v, alpha_sen)
    dist_nsen = distance(nsen_df, v, alpha_nsen)

    # calculate the M_nk matrix
    M_nk_sen = M_nk(dist_sen, K)
    M_nk_nsen = M_nk(dist_nsen, K)

    # calculate the M_k matrix
    M_k_sen = M_k(sen_df, M_nk_sen, K)
    M_k_nsen = M_k(nsen_df, M_nk_nsen, K)

    # calculate L_z
    L_z = 0
    for k in range(K):
        L_z += abs(M_k_sen[k] - M_k_nsen[k])

    # calculate x_n_hat and L_x
    x_n_hat_sen, L_x_sen = x_n_hat(sen_df, M_nk_sen, v)
    x_n_hat_nsen, L_x_nsen = x_n_hat(nsen_df, M_nk_nsen, v)
    L_x = L_x_sen + L_x_nsen

    # calculate y_n_hat and L_y
    y_hat_sen, L_y_sen = y_n_hat(M_nk_sen, w, sen_y)
    y_hat_nsen, L_y_nsen = y_n_hat(M_nk_nsen, w, nonsen_y)
    L_y = L_y_sen + L_y_nsen

    # the function we want to minimize
    metric = A_z*L_z + A_x*L_x + A_y*L_y

    return metric
```

[20]:
```
# this function defines the threshold for y_n_hat to be 0 or 1
def predic_threshold(preds):
```

```python
    for i in range(len(preds)):
        if preds[i] >= 0.5:
            preds[i] = 1
        else:
            preds[i] = 0
    return preds
```

[21]:
```python
# this function calculate y_n_hat by using the best parameters
def cal_pred(params, D, K, sen_dt, nsen_dt, sen_label, nsen_label):
    # form parameters in new forms
    best_alpha_sen = params[:D]
    best_alpha_nsen = params[D : 2 * D]
    best_w = params[2 * D : (2 * D) + K]
    best_v = np.matrix(params[(2 * D) + K:]).reshape((K, D))

    # calculate the distance matrix
    best_dist_sen = distance(sen_dt, best_v, best_alpha_sen)
    best_dist_nsen = distance(nsen_dt, best_v, best_alpha_nsen)

    # calculate the M_nk matrix
    best_M_nk_sen = M_nk(best_dist_sen, K)
    best_M_nk_nsen = M_nk(best_dist_nsen, K)

    # calculate the y_n_hat matrix
    y_hat_sen, L_y_sen = y_n_hat(best_M_nk_sen, best_w, sen_label)
    y_hat_nsen, L_y_nsen = y_n_hat(best_M_nk_nsen, best_w, nsen_label)

    return y_hat_sen, y_hat_nsen
```

[22]:
```python
# this function calculates the total accuracy, seperate accuracy for
# sensitive group and nonsensitive group and calibration
def cal_calibr(y_pred_sen, y_pred_nsen, y_sen_label, y_nsen_label):
    converted_y_hat_sen = predic_threshold(y_pred_sen)
    converted_y_hat_nsen = predic_threshold(y_pred_nsen)

    y_pred_sen = pd.DataFrame(converted_y_hat_sen)
    y_pred_nsen = pd.DataFrame(converted_y_hat_nsen)

    # calculate the accuracy
    acc_sen = accuracy_score(y_sen_label, y_pred_sen)
    acc_nsen = accuracy_score(y_nsen_label, y_pred_nsen)

    all_labels = y_sen_label.append(y_nsen_label)
    all_preds = y_pred_sen.append(y_pred_nsen)
    total_accuracy = accuracy_score(all_preds, all_labels)

    print("The accuracy for the entire dataset is: ", total_accuracy)
```

```python
    print("The accuracy for African-American group is: ", acc_sen)
    print("The accuracy for Caucasian group is: ", acc_nsen)
    print("The calibration is: ", abs(acc_sen-acc_nsen))
```

[23]:
```python
# the main LFR function which returns the best parameters and the results for
# training and validation accuracy
def LFR(training_data, val_data, y_name, sen_variable_name, K, A_z, A_x, A_y):
    # divide the training set into sensitive & nonsensitive group
    sen_training = training_data[training_data[sen_variable_name]==0]
    nsen_training = training_data[training_data[sen_variable_name]==1]

    # divide the validation set into sensitive & nonsensitive group
    sen_val = val_data[val_data[sen_variable_name]==0]
    nsen_val = val_data[val_data[sen_variable_name]==1]

    # remove sensitive variable in the sensitive training and validation group
    sen_training=sen_training.drop(columns=[sen_variable_name])
    sen_val=sen_val.drop(columns=[sen_variable_name])

    # remove sensitive variable in the nonsensitive training and validation␣
 ↪group
    nsen_training = nsen_training.drop(columns=[sen_variable_name])
    nsen_val = nsen_val.drop(columns=[sen_variable_name])

    # assign y labels for sensitive training group
    y_sen_training = sen_training[y_name]
    sen_training = sen_training.drop(columns=[y_name])

    # assign y labels for sensitive validation group
    y_sen_val = sen_val[y_name]
    sen_val = sen_val.drop(columns=[y_name])

    # assign y labels for nonsensitive training group
    y_nsen_training = nsen_training[y_name]
    nsen_training = nsen_training.drop(columns=[y_name])

    # assign y labels for nonsensitive validation group
    y_nsen_val = nsen_val[y_name]
    nsen_val = nsen_val.drop(columns=[y_name])

    # pick random values for parameters as an initial guess
    # note that since alpha and w are weights
    # they are between 0 and 1 and sum up to 1
    alpha_sen_1=np.random.random_sample((sen_training.shape[1],))
    alpha_nsen_1=np.random.random_sample((nsen_training.shape[1],))
    alpha_sen=alpha_sen_1/sum(alpha_sen_1)
    alpha_nsen=alpha_nsen_1/sum(alpha_nsen_1)
```

```
    w_1=np.random.random_sample((K,))
    w=w_1/sum(w_1)
    v=np.random.random((K, sen_training.shape[1]))

    # reform the parameters
    initial = []
    initial.extend(alpha_sen)
    initial.extend(alpha_nsen)
    initial.extend(w)

    for item in v:
        initial.extend(item)
    initial = np.array(initial)

    # the boundary of the parameters
    bound=[]

    # as mentioned before, alpha and w are between 0 and 1 and sum up to 1
    for d in range(sen_training.shape[1]):
        bound.append((0, 1))

    for d in range(nsen_training.shape[1]):
        bound.append((0, 1))

    for k in range(K):
        bound.append((0, 1))

    # other parameters does not have constriants
    for k in range(K):
        for d in range(sen_training.shape[1]):
            bound.append((None, None))

    # minimize the metric by parameters alpha, w and v
    para, min_L, d = optim.fmin_l_bfgs_b(L, x0=initial, epsilon=1e-5,
                                          args=(sen_training, nsen_training,␣
↪y_sen_training,
                                                y_nsen_training, K, A_z, A_x,␣
↪A_y),
                                          bounds = bound, approx_grad=True,
                                          maxfun=150000, maxiter=150000)

    # predict y_n_hat for the training set
    y_hat_sen_tr, y_hat_nsen_tr = cal_pred(para, sen_training.shape[1], K,␣
↪sen_training,
                  nsen_training, y_sen_training, y_nsen_training)

    print("For the training set:")
```

```python
    cal_calibr(y_hat_sen_tr, y_hat_nsen_tr, y_sen_training, y_nsen_training)

    print("+++++++++++++++++++++++++++")
    print("For the validation set:")

    # predict y_n_hat for the validation set
    y_hat_sen_val, y_hat_nsen_val = cal_pred(para, sen_val.shape[1], K, sen_val,
             nsen_val, y_sen_val, y_nsen_val)
    print("For the validation set:")
    cal_calibr(y_hat_sen_val, y_hat_nsen_val, y_sen_val, y_nsen_val)

    return  para
```

In the IFR algorithm, we found out that there are lots of distance calculations. In order to make IFR more efficient, we implemented it in a way that it calculates all the distances just once. This reduces a lot of calculations.

During the process of training, we found out that this algorithm is very inefficient. It took more than two hours to train on the entire training set and we still can't get the results. Our laptops crashed several times. We've tried the free GPU of Google Colab, but it still did not work. The GPU setting automatically switched back to CPU setting while we were training our model. Due to the limit of our devices, we decide to make the training set 20 rows and 3 columns to show that the implemented IFR model does work.

Note that we set K=10 since the number of samples are 20 we almost value the L_x, L_y and L_z almost equally, but we do want accurate labels more

[46]:
```python
# call LFR function to train the model
start = time.time()
para_test = LFR(data_train, data_val, 'two_year_recid', 'race', 10, 0.3, 0.3, 0.
 ↪4)
end = time.time()
print( f"Total training time: {end-start}")
```

```
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:16: RuntimeWarning:
invalid value encountered in log
  app.launch_new_instance()
```

```
For the training set:
The accuracy for the entire dataset is:  0.85
The accuracy for African-American group is:  0.8333333333333334
The accuracy for Caucasian group is:  0.875
The calibration is:  0.04166666666666663
++++++++++++++++++++++++++++
For the validation set:
For the validation set:
The accuracy for the entire dataset is:  0.6179205409974641
The accuracy for African-American group is:  0.6167832167832168
```

```
The accuracy for Caucasian group is:  0.6196581196581197
The calibration is:  0.002874902874902885
Total training time: 29.193768978118896
```

[47]:
```
# Testing result
sen_test = data_test[data_test['race']==0]
nsen_test = data_test[data_test['race']==1]

sen_test=sen_test.drop(columns=['race'])
nsen_test = nsen_test.drop(columns=['race'])

y_sen_test = sen_test['two_year_recid']
sen_test = sen_test.drop(columns=['two_year_recid'])

y_nsen_test = nsen_test['two_year_recid']
nsen_test = nsen_test.drop(columns=['two_year_recid'])

start = time.time()
y_hat_sen_test, y_hat_nsen_test = cal_pred(para_test, sen_test.shape[1], 10,␣
 ↪sen_test,
             nsen_test, y_sen_test, y_nsen_test)
end = time.time()
print( f"Testing time: {end-start}")
cal_calibr(y_hat_sen_test, y_hat_nsen_test, y_sen_test, y_nsen_test)
```

```
Testing time: 4.0875022411346436
The accuracy for the entire dataset is:  0.6635672020287405
The accuracy for African-American group is:  0.6494413407821229
The accuracy for Caucasian group is:  0.6852248394004282
The calibration is:  0.03578349861830532
```

As we can see from the result, the IFR algorithm is quite powerful since it achieves a test result of around 0.6 with using only 20 rows and 3 columns of the training data.

[ ]:
```

```

0.5 Handling Conditional Probability

0.6 1.Local massaging

[26]:
```
x = data.drop(['two_year_recid'],1)
x = pd.get_dummies(x)
y = data['two_year_recid']
```

Firstly, Learn a ranker: Logistic regression and compute posterior probability

```
[27]: x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 1/7)
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size =␣
↪1/6)
```

```
[28]: model = LogisticRegression()
model.fit(x_train, y_train)
#Calculate Posterior probability and then rank
predicted_prob = model.predict_proba(x)
#Selecting right Col for class_1: crisis
pred_crisis = predicted_prob[:,1]
#Updating features in X
x['pred_crisis'] = pred_crisis
x['two_year_recid'] = y
```

```
[30]: #race: Caucasian = 0; African-American = 1
x_c = x[x['race'] == 0]
x_a = x[x['race'] == 1]
```

```
[31]: #Algo 4: subroutine DELTA(race)
G_c = x_c.shape[0]
G_a = x_a.shape[0]
print(G_c, G_a)
```

```
3537 2378
```

```
[32]: #To those Caucasian whose class is 1 (predicted_crisis > 0.5)
p_c_c = x_c[x_c['pred_crisis']>0.5].shape[0]/G_c
#To those African-American whose class is 1 (predicted_crisis > 0.5)
p_c_a = x_a[x_a['pred_crisis'] > 0.5].shape[0]/G_a
p_star_c = (p_c_c+p_c_a)/2

#To calculate DELTA(Caucasian)
delta_c = G_c*abs(p_c_c - p_star_c)
#To calculate DELTA(African-American)
delta_a = G_a*abs(p_c_a - p_star_c)
print(delta_c, delta_a)
```

```
224.91274179983176 151.21359909527848
```

```
[33]: delta_c = 206
delta_a = 139
#We want to relabel 206 Caucasian by lables from - to +
#We want to relabel 139 African-American by labels from + to -
```

```
[34]: x_a_1 = x_a[x_a['two_year_recid'] == 1]
x_a_sorted = x_a_1.sort_values(by = 'pred_crisis', ascending = False)
x_a_sorted = x_a_sorted[x_a_sorted['pred_crisis']>0.5]
```

```
#We want to relabel the last 163 African-American by labels from + to -
x_a_sorted[-139:]['two_year_recid'] = [0]*139
x_c_1 = x_c[x_c['two_year_recid'] == 0]
x_c_sorted = x_c_1.sort_values(by = 'pred_crisis', ascending = False)
x_c_sorted = x_c_sorted[x_c_sorted['pred_crisis']<0.5]
x_c_sorted[:206]['two_year_recid'] = [1]*206
```

```
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:6:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/indexing.html#indexing-view-versus-copy

/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:10:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/indexing.html#indexing-view-versus-copy
  # Remove the CWD from sys.path while we load stuff.
```

[35]:
```
# Updating on original X:
cond_1 = (x['two_year_recid']==0) & (x['pred_crisis']<0.5) & (x['race'] == 0)
cond_2 = (x['two_year_recid']==1) & (x['pred_crisis']>0.5) & (x['race'] == 1)
x[cond_1] = x_c_sorted
x[cond_2] =x_a_sorted
new_x = x.drop(['two_year_recid','pred_crisis'],1)
new_y = x['two_year_recid']
```

[36]:
```
#Inputing modified data into Logistic Regression Model
```

[37]:
```
x_train, x_test, y_train, y_test = train_test_split(new_x, new_y, test_size = 1/
 ↪7)
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size =␣
 ↪1/6)
```

[38]:
```
model_new = LogisticRegression()
start = time.time()
model_new.fit(x_train, y_train)
#10-cross-fold-validation
cv = KFold(n_splits=10, random_state=1, shuffle=True)
scores = cross_val_score(model_new, new_x, new_y, scoring='accuracy', cv=cv,␣
 ↪n_jobs=-1)
```

```
end = time.time()
print('Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
print( f"Testing time: {end-start}")
new_data=pd.concat([new_x,new_y],axis=1)
new_sen=new_data[new_data['race']==0]
new_nsen=new_data[new_data['race']==1]
new_sen_y=new_sen['two_year_recid']
new_sen_x=new_sen.drop(columns=['two_year_recid'])
new_nsen_y=new_nsen['two_year_recid']
new_nsen_x=new_nsen.drop(columns=['two_year_recid'])
score_sen=cross_val_score(model_new, new_sen_x, new_sen_y, scoring='accuracy',␣
 ↪cv=cv, n_jobs=-1)
score_nsen=cross_val_score(model_new, new_nsen_x, new_nsen_y,␣
 ↪scoring='accuracy', cv=cv, n_jobs=-1)
calib_1=abs(mean(score_sen)-mean(score_nsen))
print('Calibration: ', calib_1)
```

```
Accuracy: 0.973 (0.006)
Testing time: 0.7535979747772217
Calibration:  0.009318281071454937
```

0.7 2. Local preferential Sampling

From local massaging, we know that delta_c = 206, delta_a = 139 We want to at first delete 0.5*206 Caucasian - and duplicate 0.5*246 Caucasian +

Also,a we want to delete 0.5*139 African-American + and duplicate 0.5*163 African-American -

[39]:
```
x_a_sorted_pos = x_a_sorted
#crisis = 1; no crisis = 0
x_a_0 = x_a[x_a['two_year_recid'] == 0]
x_a_sorted = x_a_0.sort_values(by = 'pred_crisis', ascending = False)
x_a_sorted_neg = x_a_sorted[x_a_sorted['pred_crisis']<0.5]
```

[40]:
```
#we want to delete 70 African-American + and duplicate 82 African-American -
x_a_sorted = x_a_1.sort_values(by = 'pred_crisis', ascending = False)
x_a_sorted_pos = x_a_sorted[x_a_sorted['pred_crisis']>0.5]


x_a_sorted_pos_new = x_a_sorted_pos[:-70]
frame_a = [x_a_sorted_pos_new, x_a_sorted_neg[:70]]
new_df_a = pd.concat(frame_a)
new_df_a.tail(100)
```

[40]:
```
      sex  race  juv_fel_count  decile_score  juv_misd_count  juv_other_count  \
1062    1     1              0             3               0                0
223     1     1              0             3               0                0
```

6256	1	1	0	9	0	0
5090	1	1	0	2	0	0
532	1	1	0	1	0	0
6024	1	1	0	3	0	0
3873	1	1	0	2	0	0
2109	1	1	0	9	0	0
4536	1	1	0	7	0	0
2163	1	1	0	5	0	0
6084	1	1	0	4	0	1
6960	1	1	0	5	0	0
4571	1	1	0	3	0	0
3663	1	1	0	10	0	0
5157	0	1	0	7	0	0
6971	1	1	0	1	0	0
2994	1	1	0	8	0	0
6115	1	1	0	9	1	0
4719	1	1	0	8	0	0
3876	1	1	0	9	0	0
2002	1	1	0	1	0	0
315	1	1	0	1	0	0
6402	1	1	0	2	0	0
1780	0	1	0	1	0	0
3946	1	1	0	8	0	0
486	0	1	0	9	0	1
5866	1	1	0	4	1	1
1602	0	1	0	7	0	0
5626	1	1	0	5	0	1
5241	1	1	0	7	0	0
…	…	…	…	…	…	…
620	0	1	0	10	0	0
2973	0	1	0	1	0	0
35	1	1	0	8	0	2
6239	1	1	0	10	0	0
4922	1	1	0	8	0	0
672	0	1	0	3	0	0
1968	1	1	0	1	0	0
3925	1	1	0	1	0	0
317	0	1	0	2	0	0
3618	0	1	0	6	0	0
1363	0	1	0	2	0	0
2096	0	1	0	8	0	0
2629	1	1	0	6	0	0
3016	1	1	0	1	0	0
28	0	1	0	3	0	0
6910	1	1	0	6	0	0
4827	0	1	0	3	0	0
3595	1	1	0	9	0	0

3148	1	1	0	2	0	0
5956	1	1	0	2	0	0
7173	1	1	0	3	0	0
959	1	1	0	1	0	0
3707	0	1	0	3	0	0
6021	1	1	0	2	0	0
4177	1	1	0	6	0	1
2062	1	1	0	3	0	0
6814	1	1	0	6	0	0
5565	1	1	0	1	0	0
4952	1	1	0	1	0	0
1614	1	1	0	8	0	2

	priors_count	days_b_screening_arrest	c_days_from_compas	\
1062	1	-1.0	1.0	
223	4	0.0	0.0	
6256	3	-39.0	39.0	
5090	8	-1.0	1.0	
532	1	-7.0	7.0	
6024	0	-1.0	1.0	
3873	1	-81.0	81.0	
2109	9	-1.0	1.0	
4536	3	-1.0	1.0	
2163	5	-1.0	1.0	
6084	4	-1.0	1.0	
6960	4	-1.0	1.0	
4571	0	500.0	1.0	
3663	3	0.0	1.0	
5157	0	-2.0	2.0	
6971	4	-1.0	2.0	
2994	9	0.0	0.0	
6115	4	-1.0	1.0	
4719	0	-1.0	2.0	
3876	0	-1.0	1.0	
2002	1	-1.0	1.0	
315	0	-1.0	1.0	
6402	0	-1.0	1.0	
1780	0	-1.0	1.0	
3946	8	-259.0	259.0	
486	6	0.0	1.0	
5866	3	0.0	1.0	
1602	5	-1.0	1.0	
5626	5	-22.0	22.0	
5241	18	248.0	24.0	
…	…	…	…	
620	3	-1.0	1.0	
2973	1	-3.0	3.0	

35	6	81.0	382.0
6239	1	-1.0	61.0
4922	4	-8.0	8.0
672	2	26.0	400.0
1968	2	-11.0	11.0
3925	0	0.0	0.0
317	0	-3.0	3.0
3618	0	-1.0	1.0
1363	0	-2.0	3.0
2096	0	-1.0	1.0
2629	0	-3.0	3.0
3016	0	-1.0	1.0
28	3	53.0	95.0
6910	5	84.0	1.0
4827	2	-3.0	3.0
3595	2	-50.0	50.0
3148	3	-1.0	1.0
5956	2	-12.0	12.0
7173	7	35.0	62.0
959	2	-64.0	64.0
3707	4	-11.0	11.0
6021	6	-26.0	27.0
4177	1	-1.0	1.0
2062	0	0.0	1.0
6814	7	-1.0	1.0
5565	2	-44.0	45.0
4952	5	-1.0	1.0
1614	9	112.0	297.0

	c_charge_degree	…	end	event	age_cat1	age_cat2	\
1062	0	…	528	1	0.0	0.0	
223	0	…	516	1	0.0	1.0	
6256	1	…	547	1	0.0	1.0	
5090	1	…	512	1	0.0	0.0	
532	0	…	565	1	0.0	1.0	
6024	0	…	211	0	0.0	1.0	
3873	1	…	208	0	0.0	1.0	
2109	0	…	277	0	0.0	0.0	
4536	0	…	320	0	1.0	0.0	
2163	0	…	589	1	0.0	1.0	
6084	0	…	537	1	1.0	0.0	
6960	0	…	269	0	0.0	1.0	
4571	0	…	500	1	0.0	1.0	
3663	1	…	223	0	1.0	0.0	
5157	0	…	241	0	1.0	0.0	
6971	1	…	544	1	0.0	0.0	
2994	0	…	369	0	0.0	1.0	

```
6115                   1       …         578      1        1.0        0.0
4719                   0       …         296      0        0.0        1.0
3876                   0       …         578      1        1.0        0.0
2002                   0       …         615      1        0.0        1.0
315                    1       …         570      1        0.0        1.0
6402                   1       …         558      1        0.0        0.0
1780                   0       …         582      1        0.0        0.0
3946                   0       …         640      1        0.0        1.0
486                    0       …         654      1        1.0        0.0
5866                   1       …         548      1        1.0        0.0
1602                   1       …         613      1        0.0        0.0
5626                   0       …         632      1        1.0        0.0
5241                   0       …         248      0        0.0        0.0
…                …       …         …     …       …       …
620                    0       …          38      0        1.0        0.0
2973                   0       …          57      0        0.0        1.0
35                     1       …          81      0        0.0        1.0
6239                   1       …          12      0        0.0        1.0
4922                   1       …        1048      1        0.0        0.0
672                    0       …          26      0        0.0        0.0
1968                   0       …          36      0        0.0        0.0
3925                   1       …          21      0        0.0        0.0
317                    0       …          57      0        0.0        1.0
3618                   0       …          88      0        0.0        1.0
1363                   1       …          47      0        0.0        1.0
2096                   0       …         110      0        1.0        0.0
2629                   0       …          58      0        0.0        1.0
3016                   1       …          30      0        0.0        0.0
28                     0       …          53      0        0.0        1.0
6910                   0       …          84      0        0.0        1.0
4827                   1       …          44      0        0.0        1.0
3595                   0       …          71      0        0.0        1.0
3148                   0       …          52      0        0.0        0.0
5956                   1       …          34      0        0.0        1.0
7173                   0       …          35      0        0.0        0.0
959                    0       …          84      0        0.0        1.0
3707                   0       …          52      0        0.0        0.0
6021                   0       …          53      0        0.0        0.0
4177                   0       …          82      0        1.0        0.0
2062                   0       …          53      0        0.0        1.0
6814                   1       …         111      0        0.0        1.0
5565                   1       …          56      0        0.0        0.0
4952                   1       …          56      0        0.0        0.0
1614                   1       …         112      0        0.0        1.0

      v_score_text1  v_score_text2  score_text1  score_text2  pred_crisis  \
1062            0.0            0.0          0.0          1.0     0.920598
```

223	0.0	0.0	0.0	1.0	0.919782
6256	0.0	1.0	1.0	0.0	0.918197
5090	0.0	0.0	0.0	1.0	0.918109
532	0.0	0.0	0.0	1.0	0.917972
6024	0.0	0.0	0.0	1.0	0.917605
3873	0.0	0.0	0.0	1.0	0.915765
2109	0.0	0.0	1.0	0.0	0.913530
4536	0.0	1.0	0.0	0.0	0.913264
2163	0.0	0.0	0.0	0.0	0.912949
6084	0.0	0.0	0.0	1.0	0.912108
6960	0.0	0.0	0.0	0.0	0.910975
4571	0.0	0.0	0.0	1.0	0.910532
3663	0.0	1.0	1.0	0.0	0.908983
5157	0.0	1.0	0.0	0.0	0.907969
6971	0.0	0.0	0.0	1.0	0.907498
2994	0.0	0.0	1.0	0.0	0.906288
6115	1.0	0.0	1.0	0.0	0.905994
4719	0.0	1.0	1.0	0.0	0.904888
3876	0.0	1.0	1.0	0.0	0.904350
2002	0.0	0.0	0.0	1.0	0.902964
315	0.0	0.0	0.0	1.0	0.899240
6402	0.0	0.0	0.0	1.0	0.898493
1780	0.0	0.0	0.0	1.0	0.896330
3946	0.0	0.0	1.0	0.0	0.893870
486	0.0	0.0	1.0	0.0	0.890469
5866	0.0	1.0	0.0	1.0	0.888078
1602	0.0	0.0	0.0	0.0	0.885948
5626	0.0	1.0	0.0	0.0	0.885549
5241	0.0	0.0	0.0	0.0	0.884333
…	…	…	…	…	…
620	1.0	0.0	1.0	0.0	0.101269
2973	0.0	0.0	0.0	1.0	0.099335
35	1.0	0.0	1.0	0.0	0.096513
6239	1.0	0.0	1.0	0.0	0.094312
4922	0.0	0.0	1.0	0.0	0.094140
672	0.0	0.0	0.0	1.0	0.092624
1968	0.0	0.0	0.0	1.0	0.091656
3925	0.0	0.0	0.0	1.0	0.090194
317	0.0	0.0	0.0	1.0	0.090043
3618	0.0	0.0	0.0	0.0	0.087042
1363	0.0	0.0	0.0	1.0	0.085262
2096	0.0	1.0	1.0	0.0	0.083311
2629	0.0	1.0	0.0	0.0	0.083153
3016	0.0	0.0	0.0	1.0	0.082801
28	0.0	0.0	0.0	1.0	0.082319
6910	0.0	0.0	0.0	0.0	0.082072
4827	0.0	0.0	0.0	1.0	0.079779

3595	0.0	1.0	1.0	0.0	0.079578
3148	0.0	0.0	0.0	1.0	0.076142
5956	0.0	0.0	0.0	1.0	0.075215
7173	0.0	0.0	0.0	1.0	0.074493
959	0.0	0.0	0.0	1.0	0.074110
3707	0.0	0.0	0.0	1.0	0.072919
6021	0.0	0.0	0.0	1.0	0.072833
4177	0.0	1.0	0.0	0.0	0.071065
2062	0.0	0.0	0.0	1.0	0.070910
6814	0.0	1.0	0.0	0.0	0.068795
5565	0.0	0.0	0.0	1.0	0.068220
4952	0.0	0.0	0.0	1.0	0.067319
1614	0.0	1.0	1.0	0.0	0.065628

	two_year_recid
1062	1
223	1
6256	1
5090	1
532	1
6024	1
3873	1
2109	1
4536	1
2163	1
6084	1
6960	1
4571	1
3663	1
5157	1
6971	1
2994	1
6115	1
4719	1
3876	1
2002	1
315	1
6402	1
1780	1
3946	1
486	1
5866	1
1602	1
5626	1
5241	1
…	…
620	0

```
2973              0
35                0
6239              0
4922              0
672               0
1968              0
3925              0
317               0
3618              0
1363              0
2096              0
2629              0
3016              0
28                0
6910              0
4827              0
3595              0
3148              0
5956              0
7173              0
959               0
3707              0
6021              0
4177              0
2062              0
6814              0
5565              0
4952              0
1614              0

[100 rows x 26 columns]
```

[41]:
```
x_copy = x
cond_a = (x['two_year_recid']==1) & (x['pred_crisis']>0.5) & (x['race'] == 1)
x_copy[cond_a] = new_df_a
```

[42]:
```
#We want to delete 103 Caucasian - and duplicate 103 Caucasian +
x_c_sorted_neg = x_c_sorted
x_c_1 = x_c[x_c['two_year_recid'] == 1]
x_c_sorted_1 = x_c_1.sort_values(by = 'pred_crisis', ascending = False)
x_c_sorted_pos = x_c_sorted_1[x_c_sorted_1['pred_crisis']>0.5]
x_c_sorted_pos.head(124)
```

[42]:
```
      sex  race  juv_fel_count  decile_score  juv_misd_count  juv_other_count  \
5746    1     0              0             9               2                6
117     1     0              0             8              12                2
46      1     0              1             8               0                4
```

3394	1	0	0	7	4	4
3141	0	0	0	10	3	2
1549	1	0	0	9	1	3
2073	1	0	0	6	2	1
4522	1	0	0	9	0	3
1456	1	0	0	9	5	0
4434	1	0	0	8	1	1
5467	0	0	0	6	0	3
3752	0	0	0	10	2	0
5798	1	0	0	8	2	3
7150	1	0	0	5	0	1
1759	1	0	0	8	2	1
1102	0	0	0	5	0	0
1122	1	0	0	6	0	0
5618	1	0	0	5	0	3
2221	0	0	0	8	0	3
3106	1	0	0	9	13	1
32	1	0	0	8	0	0
2573	0	0	0	9	0	0
6465	1	0	0	9	1	1
1773	1	0	0	1	0	1
2416	1	0	0	7	1	2
3601	1	0	0	5	0	0
3869	1	0	0	8	0	0
2183	1	0	0	8	0	0
2621	1	0	0	5	0	0
7080	1	0	1	5	0	1
…	…	…	…	…	…	…
2857	1	0	0	5	1	2
3520	0	0	0	9	0	0
1316	1	0	0	7	0	0
61	1	0	1	10	1	2
4924	1	0	0	8	0	0
2593	1	0	0	8	0	0
5287	1	0	0	8	0	0
84	1	0	1	10	6	1
613	1	0	0	5	0	0
4603	1	0	0	10	1	0
3562	1	0	0	9	0	0
278	1	0	0	10	1	1
5223	1	0	0	6	0	0
4309	0	0	0	2	0	0
653	1	0	0	7	0	0
952	1	0	0	6	0	0
7103	1	0	2	9	3	0
628	1	0	0	9	1	1
7172	1	0	0	9	0	0

6813	1	0	0	6	1	0
2069	1	0	0	6	0	0
3853	1	0	0	2	0	0
4649	1	0	0	5	0	0
1688	1	0	0	8	1	0
4951	1	0	0	8	0	0
3338	1	0	0	9	1	0
3807	1	0	0	1	0	0
1805	1	0	0	8	0	0
1374	1	0	0	7	0	0
6317	1	0	0	10	2	2

	priors_count	days_b_screening_arrest	c_days_from_compas	\
5746	8	30.0	234.0	
117	28	-1.0	1.0	
46	13	-1.0	1.0	
3394	9	-1.0	1.0	
3141	10	0.0	0.0	
1549	14	0.0	1.0	
2073	13	0.0	0.0	
4522	18	-1.0	1.0	
1456	12	0.0	0.0	
4434	8	-1.0	1.0	
5467	18	-85.0	84.0	
3752	22	-1.0	1.0	
5798	11	-57.0	57.0	
7150	11	-1.0	1.0	
1759	7	0.0	0.0	
1102	3	-1.0	1.0	
1122	9	-1.0	1.0	
5618	4	-1.0	1.0	
2221	9	-1.0	1.0	
3106	21	0.0	0.0	
32	4	-1.0	1.0	
2573	5	-1.0	1.0	
6465	8	0.0	0.0	
1773	3	-1.0	1.0	
2416	5	-1.0	1.0	
3601	3	-1.0	1.0	
3869	4	-1.0	1.0	
2183	14	-1.0	1.0	
2621	0	0.0	0.0	
7080	1	3.0	22.0	
…	…	…	…	
2857	1	0.0	1.0	
3520	1	-1.0	1.0	
1316	0	-1.0	1.0	

61	15	-1.0	1.0
4924	0	-1.0	1.0
2593	14	-1.0	1.0
5287	15	-1.0	1.0
84	14	-1.0	1.0
613	4	-1.0	1.0
4603	6	-1.0	1.0
3562	28	-55.0	53.0
278	3	-1.0	1.0
5223	14	0.0	0.0
4309	2	-1.0	1.0
653	13	-1.0	1.0
952	2	-1.0	1.0
7103	16	28.0	88.0
628	4	-1.0	1.0
7172	4	-1.0	1178.0
6813	3	-1.0	1.0
2069	1	-1.0	1.0
3853	0	-1.0	1.0
4649	10	-20.0	20.0
1688	4	-27.0	27.0
4951	6	-1.0	1.0
3338	2	-16.0	16.0
3807	0	-3.0	4.0
1805	9	-1.0	1.0
1374	0	0.0	1.0
6317	13	0.0	0.0

	c_charge_degree	…	end	event	age_cat1	age_cat2	\
5746	0	…	2	1	1.0	0.0	
117	0	…	83	1	0.0	1.0	
46	0	…	9	1	0.0	1.0	
3394	0	…	59	1	0.0	1.0	
3141	0	…	2	1	0.0	1.0	
1549	0	…	9	1	0.0	1.0	
2073	0	…	22	1	0.0	1.0	
4522	0	…	11	1	0.0	1.0	
1456	1	…	7	1	1.0	0.0	
4434	0	…	10	1	0.0	1.0	
5467	0	…	56	1	0.0	0.0	
3752	0	…	21	1	0.0	1.0	
5798	0	…	52	1	0.0	1.0	
7150	0	…	21	1	0.0	1.0	
1759	0	…	5	1	0.0	1.0	
1102	0	…	14	1	1.0	0.0	
1122	0	…	2	1	0.0	1.0	
5618	0	…	29	1	1.0	0.0	

2221	1	…	32	1	0.0	1.0
3106	0	…	254	1	0.0	1.0
32	0	…	6	1	0.0	1.0
2573	0	…	8	1	0.0	1.0
6465	0	…	3	1	1.0	0.0
1773	0	…	8	1	0.0	1.0
2416	0	…	16	1	0.0	1.0
3601	0	…	17	1	0.0	1.0
3869	0	…	2	1	0.0	1.0
2183	0	…	22	1	0.0	1.0
2621	0	…	21	1	0.0	1.0
7080	0	…	3	1	1.0	0.0
…	…	…	…	…	…	…
2857	1	…	59	1	1.0	0.0
3520	0	…	15	1	1.0	0.0
1316	0	…	28	1	0.0	1.0
61	0	…	44	1	0.0	1.0
4924	0	…	5	1	1.0	0.0
2593	0	…	40	1	0.0	0.0
5287	0	…	41	1	0.0	0.0
84	0	…	100	1	0.0	1.0
613	0	…	63	1	0.0	1.0
4603	0	…	18	1	0.0	1.0
3562	1	…	5	1	0.0	1.0
278	1	…	4	1	1.0	0.0
5223	0	…	32	1	0.0	1.0
4309	0	…	7	1	0.0	1.0
653	0	…	32	1	0.0	1.0
952	0	…	13	1	1.0	0.0
7103	1	…	28	1	0.0	1.0
628	0	…	42	1	1.0	0.0
7172	1	…	14	1	0.0	0.0
6813	1	…	8	1	0.0	1.0
2069	0	…	6	1	1.0	0.0
3853	0	…	5	1	0.0	1.0
4649	0	…	47	1	0.0	1.0
1688	0	…	43	1	1.0	0.0
4951	0	…	40	1	0.0	1.0
3338	0	…	15	1	1.0	0.0
3807	0	…	39	1	0.0	1.0
1805	0	…	21	1	0.0	1.0
1374	0	…	5	1	1.0	0.0
6317	0	…	72	1	0.0	1.0

	v_score_text1	v_score_text2	score_text1	score_text2	pred_crisis	\
5746	0.0	1.0	1.0	0.0	0.999919	
117	0.0	1.0	1.0	0.0	0.999903	

46	0.0	0.0	1.0	0.0	0.999866
3394	0.0	0.0	0.0	0.0	0.999835
3141	1.0	0.0	1.0	0.0	0.999809
1549	1.0	0.0	1.0	0.0	0.999797
2073	0.0	0.0	0.0	0.0	0.999769
4522	1.0	0.0	1.0	0.0	0.999764
1456	1.0	0.0	1.0	0.0	0.999758
4434	0.0	0.0	1.0	0.0	0.999756
5467	0.0	0.0	0.0	0.0	0.999742
3752	0.0	0.0	1.0	0.0	0.999740
5798	0.0	1.0	1.0	0.0	0.999728
7150	0.0	0.0	0.0	0.0	0.999727
1759	0.0	1.0	1.0	0.0	0.999726
1102	0.0	0.0	0.0	0.0	0.999720
1122	0.0	0.0	0.0	0.0	0.999712
5618	0.0	1.0	0.0	0.0	0.999712
2221	0.0	1.0	1.0	0.0	0.999698
3106	1.0	0.0	1.0	0.0	0.999697
32	0.0	0.0	1.0	0.0	0.999687
2573	0.0	0.0	1.0	0.0	0.999684
6465	0.0	1.0	1.0	0.0	0.999684
1773	0.0	0.0	0.0	1.0	0.999683
2416	0.0	1.0	0.0	0.0	0.999674
3601	0.0	0.0	0.0	0.0	0.999674
3869	0.0	0.0	1.0	0.0	0.999673
2183	0.0	0.0	1.0	0.0	0.999668
2621	0.0	0.0	0.0	0.0	0.999667
7080	1.0	0.0	0.0	0.0	0.999667
…	…	…	…	…	…
2857	1.0	0.0	0.0	0.0	0.999554
3520	0.0	1.0	1.0	0.0	0.999552
1316	0.0	0.0	0.0	0.0	0.999551
61	1.0	0.0	1.0	0.0	0.999550
4924	0.0	1.0	1.0	0.0	0.999549
2593	0.0	0.0	1.0	0.0	0.999549
5287	0.0	0.0	1.0	0.0	0.999548
84	1.0	0.0	1.0	0.0	0.999548
613	0.0	0.0	0.0	0.0	0.999548
4603	1.0	0.0	1.0	0.0	0.999548
3562	1.0	0.0	1.0	0.0	0.999546
278	0.0	1.0	1.0	0.0	0.999546
5223	0.0	0.0	0.0	0.0	0.999545
4309	0.0	0.0	0.0	1.0	0.999544
653	0.0	0.0	0.0	0.0	0.999542
952	0.0	1.0	0.0	0.0	0.999541
7103	0.0	1.0	1.0	0.0	0.999540
628	0.0	1.0	1.0	0.0	0.999538

7172	0.0	0.0	1.0	0.0	0.999537
6813	0.0	1.0	0.0	0.0	0.999537
2069	0.0	1.0	0.0	0.0	0.999534
3853	0.0	0.0	0.0	1.0	0.999534
4649	0.0	0.0	0.0	0.0	0.999532
1688	0.0	1.0	1.0	0.0	0.999531
4951	0.0	0.0	1.0	0.0	0.999529
3338	0.0	1.0	1.0	0.0	0.999528
3807	0.0	0.0	0.0	1.0	0.999527
1805	0.0	1.0	1.0	0.0	0.999527
1374	0.0	1.0	0.0	0.0	0.999525
6317	1.0	0.0	1.0	0.0	0.999524

	two_year_recid
5746	1
117	1
46	1
3394	1
3141	1
1549	1
2073	1
4522	1
1456	1
4434	1
5467	1
3752	1
5798	1
7150	1
1759	1
1102	1
1122	1
5618	1
2221	1
3106	1
32	1
2573	1
6465	1
1773	1
2416	1
3601	1
3869	1
2183	1
2621	1
7080	1
…	…
2857	1
3520	1

```
1316              1
61                1
4924              1
2593              1
5287              1
84                1
613               1
4603              1
3562              1
278               1
5223              1
4309              1
653               1
952               1
7103              1
628               1
7172              1
6813              1
2069              1
3853              1
4649              1
1688              1
4951              1
3338              1
3807              1
1805              1
1374              1
6317              1

[124 rows x 26 columns]
```

[43]:

```
x_c_sorted_neg_new = x_c_sorted_neg[-103:]
x_c_sorted_pos_new = x_c_sorted_pos[:103]
x_c_sorted_neg_new = x_c_sorted_neg.iloc[:-103,]
frame_2 = [x_c_sorted_neg_new, x_c_sorted_pos_new]
new_df_c = pd.concat(frame_2)
cond_c = (x['two_year_recid']==0) & (x['pred_crisis']<0.5) & (x['race'] == 0)
x_copy[cond_c] = new_df_c
```

With modified dataset, we then run a logistic Regression

[44]:

```
x_copy = x_copy.dropna()
new_x = x_copy.drop(['two_year_recid','pred_crisis'],1)
new_y = x_copy['two_year_recid']
x_train, x_test, y_train, y_test = train_test_split(new_x, new_y, test_size = 1/
 ↪7)
```

```
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size =␣
 ↪1/6)
```

[45]:
```
start = time.time()
model_new.fit(x_train, y_train)
#10-cross-fold-validation
cv = KFold(n_splits=10, random_state=1, shuffle=True)
end = time.time()
scores = cross_val_score(model_new, new_x, new_y, scoring='accuracy', cv=cv,␣
 ↪n_jobs=-1)
print('Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
print( f"Testing time: {end-start}")
new_data=pd.concat([new_x,new_y],axis=1)
new_sen=new_data[new_data['race']==0]
new_nsen=new_data[new_data['race']==1]
new_sen_y=new_sen['two_year_recid']
new_sen_x=new_sen.drop(columns=['two_year_recid'])
new_nsen_y=new_nsen['two_year_recid']
new_nsen_x=new_nsen.drop(columns=['two_year_recid'])
score_sen=cross_val_score(model_new, new_sen_x, new_sen_y, scoring='accuracy',␣
 ↪cv=cv, n_jobs=-1)
score_nsen=cross_val_score(model_new, new_nsen_x, new_nsen_y,␣
 ↪scoring='accuracy', cv=cv, n_jobs=-1)
calib_2=abs(mean(score_sen)-mean(score_nsen))
print('Calibration: ', calib_2)
```

```
Accuracy: 0.972 (0.007)
Testing time: 0.0608117580413818 36
Calibration:  0.010388366835863305
```

[ ]:

0.8 Conclusion

As we can see, LM and LPS are very efficient and have a relatively higher accuracy. For LFR, it is very slow and returns relatively bad results if we want to reduce the training time by just training it on a small training set.

[ ]: